DESIGN OF A MINIMAL TOOL SET FOR A SOFTWARE DEVELOPMENT ENVIRONMENT UNDER UNIX

by

Ruth A. Pfaffmann

B. S., Western Illinois University, 1967
B. A., North Central College, 1982

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

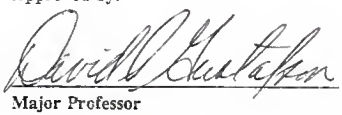
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:



Major Professor

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

The author would like to thank Dr. David A. Gustafson for serving as the Major Professor for this implementation project and Masters Report. The numerous hours he spent in discussions, reviewing the report, and providing constructive criticism are very much appreciated.

Thank you also goes to the members of my committee, Dr. Virgil E. Wallentine and Dr. Austin C. Melton, for their interest in the project and report.

My appreciation for the patience, support, and encouragement from my husband Jack can never be expressed enough. Our two daughters, Amy and Kim, have been a stimulus to my continuation even through rough times. The support of my friends and colleagues is also very much appreciated.

Without AT&T this project would not have been possible. The project was implemented on the UNIX Operating System, the report was edited with the VI screen editor, and formatted and printed using the MM Memorandum Macros. Last, but not least, without the existence of the AT&T Summer-On-Campus program, none of this would have been possible.

CHAPTER 1

INTRODUCTION

Software development is an expensive and time consuming proposition. Several authors have reported the need to control cost overruns [CHA81, CHE84, ROB83], the need to deliver software products in a timely fashion [BOE80, CHA81], and the need to manage resources [BOE80, ER84, OST81].

A software development environment can provide several benefits to the programming community. These benefits include: on time delivery of high quality systems, improved productivity of staff via simplified development activities, cost effective use of resources and project management, and traceability [HOW81, RID80, WAS80, WAS81]. A software development environment is proposed here. This environment will include a minimal set of tools that are to be built on top of and therefore extend the capabilities of UNIX®.

A software development environment may be defined as the technical methods, the management procedures, the computing equipment, the mode of computer use, the automated tools to support development, and the physical workspace [WAS81]. This definition is very broad and must be narrowed. The rest of this chapter will narrow this definition as well as review several software development environments in existence today.

Chapter two contains the main thrust of the paper: a proposal for a minimal set of tools to be used with the UNIX Operating System to aid in the implementation phase of a project. In addition, one of the proposed tools will be implemented. This tool is presented in Chapters three and four. Chapter five contains the conclusion and recommendations for extensions.

1.1 Development Life Cycle

In the late sixties the term "software engineering" was introduced [WAS80b]. The term suggests the need to follow an engineering type of discipline in the creation of computer software. One of the software engineering concepts is the development life cycle. The development life cycle describes the sequence of phases that comprise software development. Although the exact terminology used to define the software development life cycle often varies from author to author [HOW82, KER81, WAS80b], it is generally comprised of five phases.

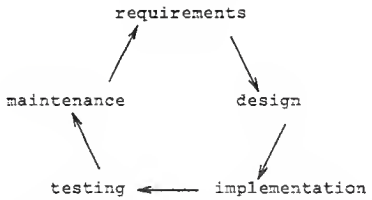


Figure 1. Development Life Cycle

The five phases include the ones shown in Figure 1: requirements, design, implementation, quality assurance (testing) and maintenance.

These same authors, [HOW82, KER81, WAS80b], also offer slight variations of what comprises the goals and outputs for each of these life cycle phases. The goals and outputs from each of the phases are listed in Figure 2.

Phase	Goal	Output
Requirements	Define the problem to be solved and a way to solve it	Requirements document* Problem statement Software specification Prototype user manual Test plans Project schedule
Design	Identify the system structure and algorithms to be used	Design document* (architectural and/or detailed) Data dictionary
Implementation	Produce an executable product	Source code* Object code* Budgets Schedules Test scripts
Testing	Produce a working product	Test data*
Maintenance	Fix problems and add enhancements	Trouble reports Change requests

Figure 2. Goals and Outputs of the Five Life Cycle Phases

Requirements is the first phase of the software life cycle. It is during this initial period that a problem is identified, studied, understood and possibly documented. After the problem is clearly understood, the requirements document is written. The requirements document includes a description of the functions that the software system is to provide. It may also include necessary development constraints, capacities, response times, mean time between failures, and development

* These outputs are almost without exception produced during the life cycle phase. The other outputs are sometimes produced during the phase, depending on the needs of the organization.

and operational costs [WAS80b].

The purpose of the second phase, design, is to identify the software system's structure and functionality. The output from the design phase is a design document. Some organizations prefer a single document comprised of two sections, one for the architectural design and one for the detailed design. Other organizations prefer two separate documents. An additional document that may be produced during the design phase is a data dictionary.

The implementation phase typically produces source and object code, thereby meeting the goal of an executable product. Some documentation may also be produced during the implementation phase. The documentation typically includes modified budgets, modified schedules, modified milestones and test scripts.

The goal of the testing phase is to produce a working software system. The main product from the testing phase is test data that shows whether or not the testing goal has been met.

The outputs from the maintenance phase are usually trouble reports and change requests. To achieve the goal for the maintenance phase, to fix problems and add enhancements, the trouble reports and change requests must be studied and evaluated. It is at this point that the life cycle becomes circular. Evaluation of trouble reports and change requests means that problems are studied and clearly understood. The next step is to write requirements that tell how to solve the problem. The assumption is being made therefore, that the maintenance phase is really repeated cycles of the other software life cycle phases [KER81].

Although some authors [HOW82, KER81, WAS80b] don't agree on the exact outputs from each phase of the life cycle, they do agree that the outputs need to be managed. Management of all the documentation, source code, budgets, schedules, milestones, trouble reports, change requests, etc. can be an incredible task.

Many problems can occur without control. The lack of information control can result in the information being lost through either misplacement or error. Project management is difficult at best if the appropriate project status information is not stored in a centralized location where it can be easily accessed. Second, without adequate controls, developers can potentially destroy each other's modifications to documents and/or source code. A third difficultly is the maintenance of a system when all the pieces are not located in a centralized location with adequate controls. Fourth, without control each project tends to develop its own methodology. Multiple methodologies make it difficult to transfer personnel without extensive retraining or to reuse tools that the projects have created [BOE84]. Finally, when a system is delivered, usually behind time and over budget, it is not very reliable. In the end, the software users are not very satisfied with the product. These problems make the need for a controlled development environment absolutely essential.

The trend over the past decade has been toward a fully integrated software development environment that covers the entire life cycle [JOH83]. In the mid-seventies, a few tools were used on mainframe machines. They gradually filtered down to the smaller machines. These tools were relocatable assemblers and linkers. In the late seventies, the majority of people doing programming had a computer science background. High level languages became commonplace. The tools of the late seventies included symbolic debuggers, screen editors, multi-user operating systems and cross assemblers. With the eighties came the overriding issue of how to support the multitude of software that is being developed.

It is anticipated that the design of the minimal tool set will provide a means by which software and all the associated information can be managed in a cost effective manner.

1.2 Support of the Implementation Phase

The proposed tool set will be targeted at the implementation phase because it is one of the most important and costly phases of the life cycle. During the implementation phase, a software development environment provides the surroundings that assist in the orderly evolution of a software system. A narrow definition of software development would encompass those activities which follow the requirements phase and result in a deliverable software system [RID80]. Recall, however, that the activities performed after delivery and during the maintenance phase are very similar to the implementation phase. The increasing ratio of maintenance to new development therefore makes the need for source code control and the managed integration of changes into a software system critical. As software systems evolve during the implementation and maintenance phases, it becomes increasingly important to support multiple versions (or releases) of both documentation and source code. An automated set of tools assistance is essential for managing all the pieces that make up a software system.

Although a development environment needs to support the life cycle, it does not necessarily need to provide automated support for all phases of the life cycle [STE87]. An environment should have an identifiable scope. It is for these reasons that tools which provide support during the implementation phase of the life cycle are crucial.

1.3 Attributes of an Environment

Although there are many attributes that make up a software development environment [CHE84, KER81, STE87], five will be examined. These five attributes are: environment category, type of architecture, incrementability, degree of integration, and tailorability.

Riddle identified three categories for software development environments: a toolbox, a development system and a development support system [RID80]. The distinguishing factor

between each of these categories is the development methodology used. Toolboxes generally don't reflect any particular methodology. They provide developers with a wide range of tools that may or may not be related. Developers can then use the tools in any way they desire. At the opposite end of the spectrum is a development system. Development systems support a single methodology. Normally, development systems are oriented toward the development of a particular type of software. Toolbox environments have less tool integration, less tool specialization, fewer complementary tools and more tool specialization than do development systems. Somewhere in the middle is the development support system. Support systems provide developers with a collection of complementary methods that are "suggestive, helpful and supportive" [RID80]. Support systems are preferred.

The architecture of a software development environment can either be open or closed. An open architecture means that new tools can be added to the environment with relative ease. The design of the basic structure of the environment remains the same. New tools can be added to enhance the capabilities of the environment without distracting from its ease of use or intended purpose.

It is important to be able to build incrementally upon the basic environment. New development needs are realized as older needs are satisfied. If an environment's basic architecture is open, new tools can be added incrementally as the new needs are identified.

An integrated environment is one in which the tools complement each other and work toward the common goal of making project management an easier and more cost effective task. The degree of integration can range from high to low. Delisle, Menicosy and Schwartz define a highly integrated environment as "a single tool that encapsulates the mechanisms used to implement the environment's functionality" [DEL84]. In a loosely integrated tool set each tool provides a specific function that works toward a common goal. It is more difficult to add new tools to a

highly integrated environment. It is desirable, therefore, that an environment have a loosely integrate set of tools.

Suppose that an organization has decided to begin using a programming environment for the first time. Also suppose that the organization has a choice of two environments. The architecture of the first environment is closed and it is not an incrementally integrated set of tools. This means it will have to be installed and used as a total unit. Using this first environment would mean that all development would have to stop during the conversion to the environment. The conversion process itself could be a large task, possibly requiring the abandonment or rework of existing code. The second environment has an open architecture and an incrementally integrated set of tools. It can be installed and used incrementally. With the installation of the second environment, development could proceed and any necessary conversion could take place in parallel with the usual development process. The "gulp factor" [KER81] of the first environment is far more likely to be more costly in terms of time and money than is the incremental structure of the second environment.

Finally, an environment needs to be tailorable to the specific needs of the project or projects that it is being used to support. If an environment is tailorable, its capabilities can be adapted to the specific needs of the project. In other words, there is a mechanism by which the environment can be easily altered so that it will behave differently. The behavior mechanism should be part of the project database rather than having project specific information built into the tools. The way in which the tools adapt themselves to the project specific information should be invisible to the user. If an environment is tailorable, it is more likely to be used by a variety of projects with the end result that more information can be shared across project boundaries, reuse of tools is higher and personnel will have more mobility between projects because they will not have to be retrained in the use of the tool set.

A tailorable environment should allow a large project to spread its development across several machines. This means that there will probably be some sort of networking capability available between the machines. However, the networking facilities between the machines may differ from project to project or even within a project. For example, a project may have it's development split across three machines, machines A, B, and C. The networking facility between machines A and B may be *uucp*. The networking facility between machines B and C and between C and A may be Ethernet. While the different networking facilities accomplish the same task (the transfer of files), the command lines to accomplish the task may be quite different. A tailorable environment should have information about these networking facilities stored in the project database. Then when the tools are executed, they can determine the appropriate command line for the networking facility being used. The user should not have to know about the available networking facilities. The same environment should allow a smaller project to have all development work done on a single machine.

A tailorable environment should also allow projects to administer their source in one centralized database or in several distributed databases. For example, even if a project is small enough to be administered all on one machine, it may be too large to all fit on one file system. It may therefore be desirable to administer each project component in it's own database with each database being located on a different file system. The tools will need to be able to locate the appropriate database to make additions, deletions, and updates as well as to do information retrieval. If the project database contains information about the location of each database and the components administered in each database, the tools should be able to access the appropriate database without the user having to supply the information.

Finally, a tailorable environment should allow projects to specify the exact methodology they wish to follow within the general frame work provided by the tool set. For example, the

administrators of one project may wish to put products through three levels of testing called component testing, integration testing, and system testing. The administrators for another project may feel that two level of testing is sufficient. If the number of testing levels and their names are stored in the project database, the tools should be able to track the status of the products through each phase of testing.

1.4 An Analysis of Existing Environments

There are many software development environment tool sets currently available. Figure 3 summarizes six environments.

Environment	Environment Category	Architecture	Incremental	Degree of Integration	Tailorable
Toolpack	Development System	Open	Yes	High	No
Joseph	Development Support Sys.	Open	Yes	Medium	No
SAMS	Development System	Open	Yes	High	No
SMSS	Development System	Open	Yes	Medium	No
UNIX (PWB)	Toolbox	Open	Yes	Low	No
ISTAR	Development Support Sys.	Open	Yes	Medium	Yes

Figure 3. Existing Environment Analysis Summary

The goal of the Toolpack environment is to create a feedback loop between developers and users so that the development and maintenance of software can be more effective [OST83]. The Toolpack target community is mathematical programmers who use Fortran. It can be classified as development system because it supports the goals and procedures (methodology) of this very

specialized group. Its architecture is open and tools can be added to it incrementally. However, adding tools to the Toolpack environment could be difficult because the tools are so highly integrated and have considerable knowledge about each others functions. Toolpack is made up of: a compiling/loading system, a Fortran-intelligent source code editor, a formatter that puts source in canonical form, a structurer to create Fortran loop constructs, a dynamic testing and validation aid, a static error detection and validation aid, a static portability checking aid, a document generation aid and a program transformer that assists in translating one dialect of Fortran to another. Toolpack is not tailorable, it was created specifically for use on one type of project.

The Joseph software development environment has the goal of being able to support development work from file creation through verification [RID83]. Its target community is the developers of an operating system for a high-performance, multiple-processor, scientific computer system at Cray Laboratories. Its architecture is open and it was specifically created with the concept of incremental development in mind because it was realized from the beginning that the total environment could not be created all at once. Although only a portion of Joseph was actually developed the result was a development support system with a medium degree of integration. Joseph is a development support system because it suggests a series of activities that can be used in the development of software. The exact sequence of activities can be modified to achieve different goals. The basic components of Joseph are: Pharaoh, a requirements definition set of tools; Oasis, design description tools; and Crypt, a database system that was an interface to project specific information. The underlying operating system was UNIX. There was no evidence that Joseph was tailorable to any other kind of software development projects.

The Software Automated Management System (SAMS) was developed to aid in controlling software development by the Jet Propulsion Laboratory of the California Institute of Technology [MOL80]. It was used to develop software for an orbiter spacecraft for launch from a space

shuttle. SAMS is classified as a development system because the size and complexity of the software being developed for the orbiter required the strong management control methodology provided by SAMS. SAMS's open architecture allowed the incremental development of tools which included an automated work breakdown structure processor for the planning, control and resource estimation, schedule generation processor to create software schedules based on user input, an information display process to aid in the design of input and output before a program is coded, a distribution list processor to aid in the timely dissemination of project information, an action item processor that gives the status of action items, a document/memo index processor that includes a database for the safe keeping of documents that were written using WORDSTAR, a software delivery processor that keeps the latest version in a central software library, a computer margin summary process that reports on computer resource usage, and a configuration management processor that keeps track of changes to the system. SAMS is not tailorable because it was developed with a specific purpose in mind and because its tools are highly integrated.

SMSS (Software Management Support System) was created by the Configuration Management organization of GTE Automatic Electric Laboratories to manage and control the software development effort [CHA81]. It can be classified as a development system because is supports a particular development philosophy (Structured Design and Jackson Design Methods) and is geared toward a particular language (GTD Pascal). SMSS's architecture is open and it can be added to incrementally. The tools are integrated, but no mention is made about tailorability. The basic pieces of SMSS are the source code control system and the Programmer's Workbench from UNIX.

The UNIX Operating System has been recognized as providing an abundance of utilities that create an effective working environment [FAR85, WAR841]. UNIX is incremental in nature, new tools are added to it with regularly without disrupting the development process. It includes

tools that are aimed at managing the evolution of software. Tools such as the Source Code Control System (SCCS) and *make* are a good foundation for an incremental software development environment. Its open architecture and low level of integration make a perfect toolbox from which to draw a foundation upon which to base a software development environment.

Finally, the ISTAR development environment is a development support system that is language independent [DOW86]. ISTAR is a development support system because it does not impose a particular way of doing software development. The tools are integrated but the architecture is open and new tools can be added incrementally. ISTAR views all software activities as contracts. Any work is considered a contract between the contractor, e.g. a programmer, and a client, e.g. a manager. ISTAR is made up of four main components: a database system, a user interface system, a communication system and a set of tools.

CHAPTER 2

PROPOSED MINIMAL TOOL SET

2.1 Proposed Environment Characteristics

The needs of software development projects determine the functions that the proposed tool set should be able to provide. Of the three categories of software development environments identified by Riddle [RID80], the development support system is the most preferred. Recall that it provides the developer with a helpful collection of methods rather than a restrictive set of rules. Other desirable characteristics of a software development environment are that it should have an open architecture so that new tools can be added to it incrementally. The tool set should be integrated but not so tightly that tools cannot function without each other. Finally, the software development environment should be tailorable so that the tool set can be reused by a number of different projects even though each project may be configured differently. It is the goal of the proposed tool set to contain as many of these characteristics as possible.

2.2 User Roles/Views/Needs Within an Environment

The purpose of a software development environment is to make the user more productive and thereby lower development costs. A user can have one of four different roles within a software development project: the developer, the software production personnel, the system integrator and tester or project manager. The people in each of these roles have different views of the development process and different needs that the tools must satisfy. To provide an effective development environment each of these roles must be examined to determine the duties associated with each of the roles, how the development process is viewed from within that role, and the kinds of functions the tool set should provide to satisfy the needs of the people in each role.

2.2.1 Developer

The duties of a developer include the writing of documentation (requirements, design, manuals, etc.), source code and test scripts. Quite often the activities of one developer may overlap with the activities of several other developers. In other words, developer A may be adding a new feature to an existing software system and developer B may be fixing a problem that was found in the same system. It is very possible that both developers need to modify the same files. If they both made private copies of the files and then worked only with their private copies, there is the possibility that one of the developers could overwrite the changes made by the other. For example, developer A makes a private copy of the file and puts his or her changes in the private copy. Developer B does the same thing. Developer A finishes first and copies the private file over the file in the public domain.[1] Then developer B finishes and copies his or her private file over the one in the public domain. At this point the public domain file contains only the changes made by developer B. Developer A's changes have been lost. The problem of possibly overwriting each others files grows rapidly with the number of people needing access to a particular file.

Overwriting of files is only part of the reason developers lose information. Sometimes a system is just not organized in a logical manner. A software development environment should provide developers with a means of logically organizing information. The kinds of information that developers need at their finger tips includes: a list of the changes that have been requested to the project component that they are responsible for, a list of the source files that are effected by a particular change, and the status of changes both requested and already being worked on.

1. Public domain is used here to refer to an area where the software system products are kept (official library). It is assumed that there is little or no control over the contents.

A software development environment should provide developers with a view of the development process that will allow them to easily complete the development task. This means it should help developers identify and prioritize work. It should also help them complete their own work without being concerned about the work of others. In particular, the environment should manage source files and change requests and provide a simplified way to construct the software product.

2.2.2 Software Production Personnel

Software production personnel are the people whose job it is to accept changed modules from developers, build them into the existing software system, and then give the software system to the testing organization. These people need to know which components are ready to be built into the system so that system integrators and testers can test the changes. To accomplish the build process, the production personnel need to be able to identify change requests that have been completed and the software that is ready for building. They also need to be able to identify the source files effected by the change requests.

Building of a software system for testing means that the production personnel need to be able to gather all the source files into a centralized location for compilation. This isn't a problem if the project is small enough that all of its development can take place on one machine. If, however, the project is so large that it must be distributed over several machines, the software development environment needs to provide a way to gather the source files from the distributed computers onto a centralized machine so the source can be built into a whole system. Once the system has been built, the source and resulting object files need to be re-distributed back to the other machines.

At this point the changes have been built into the software system but they have not yet passed the testing phase. This means that the software development environment needs to provide for an

intermediate repository to hold software source and object files that have been built but not yet tested.

2.2.3 System Integrator and Tester

System integrators and testers are responsible for putting together all the components of a software system and verifying that they have the functionality specified in the requirements. With development occurring in incremental steps, system integrators and testers need to be able to identify which components have had changes made to them so that they know what kind of testing needs to be done. They also need to know the status of the change and what particular portion of the software system was modified so that the appropriate testing can take place.

System integrators and testers need the ability to control which changes become a part of the official version[2] of the software system. When testing takes place, some components will pass all tests and others will not. Only those components that pass all tests should be allowed to become part of the official version. System integrators and testers also need to be able to control when changes are introduced into the system. The time when changes are introduced may not only have an impact on scheduling, but also on the functionality of the system. In other words, component X may be completed before component R but component X depends on output from component R so it can only be partially tested.

A software development environment needs to have a means for keeping track of the status of change requests. System integrators and testers need a software development environment that provides a reporting mechanism and/or a query mechanism. In addition, the system integrators and testers need to be able to specify which changes have passed the testing phase and are ready to

2. A project may have several libraries (repositories for source and object files). The official library will contain only the source and object that have been approved by system integrators and testers. When the project is completed, it is from the official library that the final product will be delivered to the customer.

become part of the official version of the software system being developed.

2.2.4 Project Management

It is the project manager's responsibility to control the development of a software system. Many different kinds of problems can arise. It is often the case that an entire system cannot be developed all at once because the task is too large or there aren't enough people available. Sometimes there isn't enough computing resources available to do the development in a centralized location. It is also possible that the requirements for the whole system are not known. As the development process proceeds changes may have to be made to software that has already been written. These changes could be the result of a change in the requirements or design or the result of problems found during testing.

Managers, therefore, need a software development environment that will allow a software system to be developed incrementally and will allow changes to the components of the system. The software development environment should allow a large software project to be broken apart and distributed over several machines. Smaller projects should be allowed to maintain all information in one centralized location.

Management of resources should also be aided by the software environment. A manager needs to be able to specify which developers are to work on a particular assignment. Managers also need the ability to monitor and adjust other resources such as machine usage, budgets and schedules.

Meetings are often held on project status. A software development environment should provide managers with the kinds of information needed at these meetings. For example, managers need to be aware of the changes being requested to the software system they are managing. They also need to know the status of each of the change requests.

2.2.5 Summary of User Roles/Views/Needs

Although the roles and tasks of the individuals working on the development of a software system vary, their views and needs overlap. As is indicated in Figures 4 and 5, everyone needs a global view of the project. Developers will do most of their work in a private area and won't use the global view as often as software production personnel, system integrators and testers or project managers, but they still need to be able to access files throughout the software system. Version control is only listed as a need for developers but it should be understood that the reliability of the system as a whole depends on it. Developers and software production personnel share the need for simplified software construction. The ability to retrieve information via reports or a query mechanism is very important no matter what the person's role may be in the software development process.

Role	View	Task	Needs
Developer	Private work area and global	Write software code and documentation to add new features and fix problems	Concurrent access without collision (version control) Individual change request status reports Simplified software construction

Figure 4. Summary of User Roles, Views and Needs - Part 1

Role	View	Task	Needs
Software Production Personnel	Global	Manage project libraries and build software system	Status of change request Files effected by change Simplified software construction Library mgmt. aids
System Integrators and Testers	Global	Verify results of system build and determine what becomes part of official library	Status of change requests Files effected by change Control of official library
Project Manager	Global	Control software development process and resources	Status of change requests Schedules Machine usage reports Budget reports

Figure 5. Summary of User Roles, Views and Needs - Part 2

2.3 Development Environment System Modules

The core of the proposed tool set will be made up of five basic parts. Each of these modules will provide a particular function, and may be developed and added to the tool set one at a time. The modules will be loosely coupled, meaning that they may communicate with each other, but

they will function equally well in a stand-alone environment. The only exception to this is the fifth module, the System Integration Module, which has a direct dependency on the existence of the Source Tracking Module and the Problem/Enhancement Tracking Module.

Because a development environment tool package cannot be created overnight, it is necessary to acquire the tools incrementally. The order in which they are described is the suggested order of acquisition. The order of acquisition recommendation is based upon the estimated amount of development effort required to invent the tool and the expected degree of benefit to be gained from using the tool.

2.3.1 Product Assembly Module (PAM)

The Product Assembly Module (PAM) provides the compile, link-edit and optional installation capabilities for creating a software product (an executable program) from source code files. In actuality, most of the product assembly tools are already provided by UNIX. These tools include, for example, the *make*, *cc* and *ld* commands.[3]

The *make* command provides a mechanism for keeping track of and re-creating software, thereby solving the problem of being able to reproduce a software product. The *make* command requires that the command sequence needed to create a product be placed in an input file called a makefile [UNIX85]. A makefile is a description file that contains dependencies[4] and rules (commands). It provides the *make* command with directions about how to create a particular product.

Maintenance of large makefiles is an error-prone process [FEL79]. Makefiles usually contain large command sequences written by users whose style and degree of sophistication vary. As

3. The assumption is being made that the reader is familiar with these commands as they are described in [UNIX85].
4. A file X depends on file Z if a change in file Z requires file X to be updated, recompiled or relinked.

people move in and out of an organization, maintenance of existing makefiles can quickly become a problem. Developers may not be aware of implied dependencies that are the result of an include statement[5] [WAL84].

Manually created makefiles can have any of several problem areas. Consider the following examples.

A product is made up of a component called COMP. COMP is made up of two files, file.c and defs.h. File.c includes the defs.h file and defs.h includes another file, global.h from the HEAD directory. The global.h file in the HEAD directory then includes the defs.h file from the COMP directory.

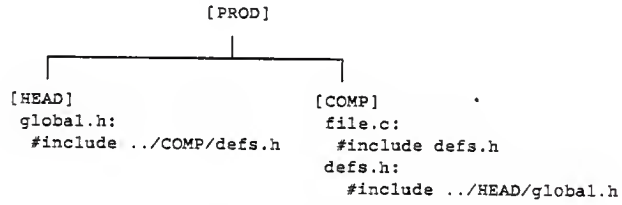


Figure 6. Manually Created Makefile Problems-Example 1

It can quickly be seen that a circular include has taken place. However, if a developer was not aware that the global.h file included the defs.h file, it may take a considerable amount of time and effort to find the problem.

Being aware of all the files needed to create a product is another problem with which the developer must be concerned. In example one, the developer may be very much aware that file.c

5. An include statement refers to a file whose contents are to take the place of the include line in the source file.

includes both the defs.h and global.h files. However, he or she may not know that the global.h file has include lines in it unless he or she actually scans the global.h file.

Finding the global.h file may be difficult if the developer is not familiar with the directory structure and how the project is organized. Unless one is familiar with a project, looking through all the files needed to create a product is a time consuming task that is often left undone.

Inexperienced developers may create incorrect targets[6]. The inexperienced developer may see that file.c includes defs.h. They will often create an incorrect target:dependency pair[7] such as, file.c:defs.h, where the correct target:dependency pair should be file.o:defs.h. It is the object file that depends on the included file, not the source file.

Yet another problem may occur with nested include files, even if they are not circularly included. Suppose that file.c in the COMP directory included defs.h from the same directory. The defs.h file then includes global.h from the HEAD directory, which includes const.h, which includes var.h, which includes header.h, which includes defines.h from the COMP directory, which includes defs.h.

There are two problems in the second example. First, the number of nested includes has become almost unmanageable. Secondly, one may think that another circular include has just been identified. However, this is not a circular include. The defs.h file has been included twice, once ftom the file.c file and once from the defines.h file.

A tool is needed that will aid users in the writing of makefiles. It is possible to provide such a tool as a part of the Product Assembly Module. The makefile writing aid could solve the

6. A target is the item being made. It may be the product or it may be some component of the product.

7. A target:dependency pair is declared in the makefile. It means that the names to the left of the colon depend on each of the names to the right of the colon [FEL79].

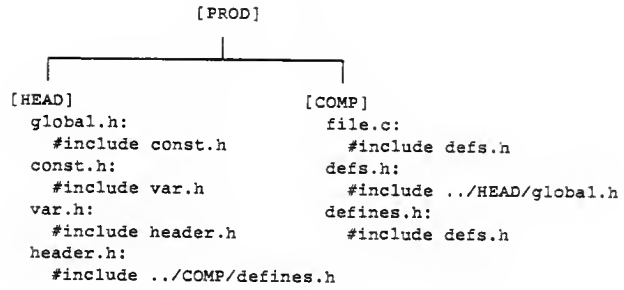


Figure 7. Manually Created Makefile Problems-Example 2

problems enumerated above by automatically generating dependencies. The makefile writing aid will be referred to as *mkaid*. The *mkaid* tool should be able to:

- identify when files have been circularly included,
- find ALL included files (nested includes),
- locate included files,
- generate correct and consistent dependencies,
- identify when the number of nested includes has become too large and unmanageable, and
- eliminate duplicate include files.

The Product Assembly Module (PAM) is the first module that should be acquired. PAM is a truly stand-alone module that can be implemented and installed without any of the other development environment modules. It is estimated that it can be implemented in a fairly short period of time, and the degree of benefit will be considerable. For these reasons PAM has been chosen as the module that will be implemented to accompany this paper.

Actually, most of PAM is already available through the UNIX operating system. The commands already included in UNIX are *make*, *cc* and *ld*. There is only one tool that will make up PAM. That tool is the makefile writing aid, *mkaid*. The *mkaid* command could be considered as an extension to the *make* command.

The *make* command expects to find its directions in a file called a makefile. Makefiles are usually created manually which is error-prone. The makefile writing aid can automate the process of creating makefiles. The benefits that can be achieved by using the makefile writing aid (*mkaid*) are:

- increased developer productivity through reduced developer effort in creating and maintaining makefiles (less time will be spent creating and maintaining makefiles so more time can be spent on other development activities),
- consistent, standardized, system generated makefiles (makefiles will be easier to read and understand because they will all have the same format), and
- complete and correct dependency generation (the tool will have built-in knowledge: that object files depend on included files, that included files can be located within a given directory structure, and that a depth of only a certain number of includes is allowed).

2.3.2 Source Tracking Module (STM)

One of the main problems in development and maintenance of a software project is how to manage changes [ER84]. The Source Tracking Module (STM) should be the next unit of the development environment to be acquired. It provides a repository for the safe keeping of source files and version control.

Controlling source files can solve several problems. The first problem is that source files are always changing. Changes may be made to fix bugs that have been found to add enhancements

and new functionality or to optimize critical functions. Another problem is the possibility of more than one person accessing a source file at approximately the same time. Then, depending on the sequence of events, possibly overwriting other's changes. Recall the example of this problem as developers experience it. If more than one release of a software product is currently being worked on, the changes may need to be made in every release. It is possible that one of the releases could inadvertently be forgotten. In addition, a considerable amount of space is needed to hold all the source files in each release. If developers are allowed to just make changes to source files, there is no tracking information that identifies what the change was, when the change was made, or who made the change.

Another problem is deciding which versions are difficult for a customer. If the software system has several releases being used by customers, it can be hard to figure out which release is causing difficulty for the customer. It is often difficult to tie a particular change request to a change that has been made to source files.

Finally, one change request may be directly dependent on another. For example, a change request (CR1) is made to fix a bug in file X. Another change request (CR2) is made to fix another bug in file X. If both CR1 and CR2 are used to modify the source for file X, the fixes may depend on each other. Therefore, when changes have been made using both change requests, they need to be identified as possibly dependent on each other. This means that they will have to be included in the same system build by the software production personnel.

The Source Tracking Module (STM) will be built on top of the UNIX Source Code Control System (SCCS) [ROC75]. SCCS by itself already provides much of the functionality needed to solve may of the problems listed above. It will conserve on space by storing the source file only

once and then adding deltas[8] [ROC75] that contain only the changes made. SCCS also keeps track of what a specific change consisted of, who made the change, and when the change was made. The particular release that a customer is using can be identified through the version number that SCCS assigns to each delta. SCCS also cootrols access to files so they are protected from unauthorized changes. This is done through the read, write and execute file modes [CHA81]. When a file has been "checked out" so that modifications can be made to it, SCCS locks the file in the database so no one else can check it out. The file lock is removed when the editing session is over and the user has indicated that they are done. This file locking prevents collisions in accessing source files.

Two problems that SCCS does not address, however, are the problems of being able to tie a particular change request to the actual delta for the change and the problem of being able to identify changes that are dependent upon each other. The purpose of STM is to extend the functionality of SCCS. STM should allow a change request to be identified. One usually thinks of the identifier, or change request, as being some meaningful number, but the exact implementation is not of concern here. The point is that the change request number should be an input to STM. This way developers can create their own change request number until the Problem/Enhancement Tracking Module (PET) has been developed. Once PET is developed it can pass the change request number to STM. Just as a safety check, however, STM should make sure that the change request number is unique. The change request number should be stored in the delta that contains the modifications required by the change request.

The problem of being able to identify changes that are dependent upon each other isn't quite as easy to solve. STM will have to be able to assign a status to the delta and thereby to the

8. A delta is equivalent to a change or an editing session [ROC75].

change request number, as it proceeds through the development process. For example, STM may assign the status of "change" to any delta and change request number whose effects have been added to the file but not yet approved by the system integrators and testers. The "tested" status could be assigned to deltas and change requests numbers that have been approved by system integrators and testers. Finally, the "integrated" status could be assigned to deltas and change requests that have been integrated into the official library by the software production personnel. Getting the statuses assigned means that there should be a set of commands available to the software production personnel so they can be executed as the changes work their way through the software process.

Once the statuses have been assigned to the deltas and change requests, they can be checked for dependencies. Any deltas and change requests that have the "change" or "tested" status AND touch the same source file should be declared as dependent upon each other until they both move to the "integrated" state. In the previous example, where CR1 and CR2 both have changes that are to be applied to file X, CR1 and CR2 are dependent on each other if their state is "change" or "tested". If either of the change requests were in the "integrated" state, they would not be dependent. When a change request's status becomes "integrated" the changes associated with it have been placed in the official library. Therefore, all additional changes will be applied on top of it rather than along side of it.

The advantages of using SCCS are well known and many systems have been built on top of it [BOE84, CHA81, ER84, FIL86]. The advantages of building STM on top of SCCS include the ability to tie a particular change request with a specific change or set of changes in source files and the ability to determine dependencies between change requests. Tying change requests to source file changes allows considerably more information to be available for reports. For example, information contained in SCCS about who made a change, when the change was made, etc.

becomes available when reports are generated about change requests. The advantage of being able to determine when change requests depend on each other is that there is less likely to be build errors or testing problems related to having an incomplete set of changes.

2.3.3 Environment Access Module (EAM)

The first phase of the Environment Access Module (EAM) should now be developed. This will be the smallest of the tools, but it is none-the-less important. The function of EAM is to connect the user with the rest of the tools. This module will need to change and grow as new tools are added to the development environment.

Several different users of the software development environment have been identified. As was discussed earlier, each of these users has a different task that is to be performed. Some of the tools in the proposed tool set will be used by everyone while other tools should only by used by certain people. For example, everyone should have access to. tools that generate reports or assemble products in their private work area. However, only the software production personnel should have access to tools that administer the project databases and libraries. Another potential area for difficulty for users is being able to locate the project's databases and libraries, especially if the project is distributed over several machines.

The function that EAM should perform is to connect the user with the appropriate databases and the appropriate set of tools. This should be done with as little input from the user as possible. The software development environment should have a database within it that contains information about bow the project is set up, how many machines the project is distributed across, what is on each machine, how many releases are being supported, the name of each release, etc. All the user should have to do is execute EAM, possibly from their .profile file, tell it the name of the project they are working on, and then answer any questions EAM may ask. EAM should only ask questions of the user when there is a choice that has to be made. For example, if the

project is currently supporting three releases, EAM will need to know which one the user wants to work on. If the project is only supporting one release, EAM should know that there is only one choice and not bother the user.

UNIX has an environmental variable called PATH that is used to identify all the locations to be searched for an executable command. EAM should set the users PATH according to who the user is and what their task is. In other words, software production personnel should get the database administrative tools in their path while developers should not.

EAM gives the user a view of the software development environment that is appropriate for his or her job. It helps prevent users from using the wrong set of tools and makes it easy to access the appropriate set of tools. It also connects the user to the particular project configuration that he or she is working on.

2.3.4 Problem/Enhancement Tracking Module (PET)

Since there is now a means to control source (STM) and a way to convert the source into object (PAM), one now needs a method of tracking the system alterations that are desired. This is the function of the Problem/Enhancement Tracking Module (PET). An alteration may be desired because a problem or a bug has been detected and needs to be corrected, or because an idea for an enhancement has been identified.

Managing change requests in an orderly fashion is just as difficult a task as managing source files. Manually managing change requests often leads to lost or misplaced information. Preparing reports by hand is a tedious and time consuming process. The desired contents of reports change often, making it a non-routine task. With information scattered all over, it is difficult to determine priorities and set up a useful schedule. It is difficult if not impossible to determine who, if anyone, is working on a particular problem. The connection between the actual change to

the source and the change request itself may never make it into the STM database. Finally, customers or users of a software system are never sure that the change they requested was even considered, let alone made it into the system.

The UNIX Programmer's Workbench (PWB) includes a change request system called Modification Request Control System (MRCS) [KNU76]. MRCS is an interactive system that tracks the flow of change requests through the development life cycle and provides management with status information. Like SCCS, MRCS solves most of the problems identified with respect to change request management. PET should, therefore, be an extension to MRCS.

It is desirable to continue having two separate database systems, one for source management and one for change request management. The basic reason for keeping the two separate is that it allows flexibility in a project's use of the software development environment. Some examples can make this point more clearly. Take the case of a large project that is supporting several releases, each in a separate instance of STM, and is spread over several machines. It may be desirable for that project to have one change request database that will feed the appropriate change request into the correct source database. It may even be possible that several projects that are somewhat like each other wish to share change request information but don't want to share source files. Another possibility is that a non-software organization, e.g. a hardware group, would like to be able to manage their change requests even though they don't have any source that needs administering.

The one thing that is missing is the link between the two database systems. The purpose of PET is to provide an automatic communication link between the Source Tracking Module (STM) and the change request system. PET should automatically assign an identifier (change request number) to a change request; determine what project, release, component, machine and the person that the request should be assigned to and then automatically notify both the person responsible for making the changes and the change request originator that the request has been

recorded in the database.

MRCS assigns a status to each change request as it moves though the software life cycle. Another extension that PET should provide is the automatic notification to the appropriate people each time the status changes. Much the same approach could be used with this problem as was used by SAMS (Software Automated Management System) [MOL80]. When the change request originator has entered the request into the system, a copy of the request should automatically be sent to the assigned programmer. The programmer should then study the problem and propose a way in which to solve it. The proposal should then be sent back to the originator and also to a change control board. The action taken by the board, approval or rejection, should be automatically sent back to the programmer. If the proposal was accepted by the board, PET should also automatically notify the appropriate instance of STM. If the proposal was rejected, the programmer should be allowed to make another proposal. Once the modifications to the source have been completed, the testing organization should be notified. When testing is complete, both the programmer and the originator should be notified that the change request has been successfully accepted into the official library for the software system.

There are several benefits that come from using PET. First, the project gains quick easy access to information stored in the database provided by MRCS. This information aids in determining schedules (both individual and project), provides reports of what problems/enhancements are being requested, identifies who is working on each problem/enhancement and aids in the reallocation of resources. The additional benefits provided by PET include the automatic notification to the appropriate people as the change request progresses through the software life cycle and the automatic tie between the change request and the STM databases. In other words, a programmer must have a valid change request number to be able to check a source file out of STM, make modifications which are attached to the change

request number and then put the modified file back into STM [CHA81].

2.3.5 System Integration Module (SIM)

The System Integration Module (SIM) has been left to last because it is the only unit of the development environment that depends upon the existence of the other units. SIM will allow creation of a software system by putting together several components of a project into one integrated software package. To accomplish its task it will need access to the PET and the STM modules.

A software project's source files are administered in STM. It's change requests are administered in PET. But how do all the source files for all the components that make up a software system get put together and compiled into an executable system, especially if the components are each being developed on different machines? The sudden problem of software not building together at the last moment before delivery is not a pleasant surprise. Another unpleasant surprise would be a system that didn't work together as a whole; this could easily happen if the system wasn't put together and tested as a unit. The logistics of such a problem are enormous. It is best to start with an example so that the problem can be understood. Consider the situation where a large project is made up of several components. The three components shown in Figure 8 are NAVIGATOR, CABIN and ARM.

The NAVIGATOR component deals with mobility of the UNDERWATER EXPLORER which includes a RADAR system for checking the contour of the ocean bottom and a DEPTH detector. The CABIN component has subcomponents of PRESSURE to keep the cabin pressure within a human tolerable range, TEMP to adjust the temperature inside the cabin, and AIR to make sure the diver doesn't run out of oxygen. The ARM component has two subcomponents, MOVEMENT and LIGHT. The MOVEMENT subcomponent includes directories for each kind of movement that the arm is likely to make. There are directories for upward movements,

downward movements, rotating movements, to pick up an object and to put down an object. Note that directory names are the capital letters contained in square brackets. The file names are in small letters. Some files may reside in directories that also have subdirectories under them.

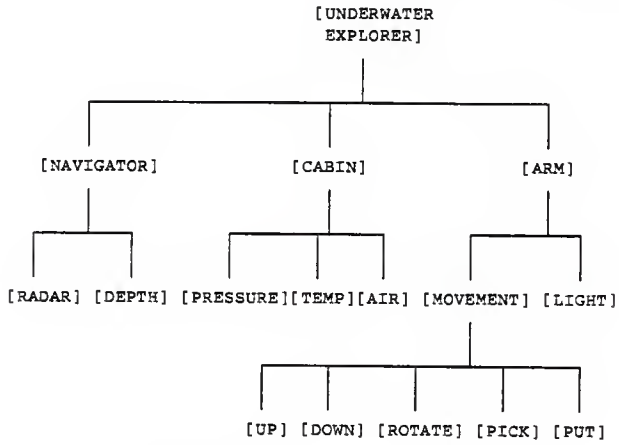


Figure 8. Sample of Project Directory Structure

In very large projects it is often desirable to split the development of components (NAVIGATOR, CABIN and ARM), across several machines so that no one machine's resources are over taxed. The sample project from Figure 8 can be organized on three machines as shown in Figure 9.[9] The development of the NAVIGATOR component will take place on machine A. The STM instance will be located there along with the group of developers who are working on it [CHA81]. Both the CABIN and ARM components will be located on machine B along with the

9. The PET database is not shown in Figure 9. PET could reside on any one of the machines or on a fourth machine.

STM instances and the developers working on them. Both instances are on the same machine because each of them is smaller than the NAVIGATOR component. The third machine, C, is used strictly for the intermediate and official library configuration. It is on machine C where all the source files from the NAVIGATOR, CABIN and ARM components will be brought together and compiled into an executable system. Once the system has been compiled in the intermediate library, it can be redistributed back to the development machines. This should be done so that developers can view the latest changes even though they aren't "official" yet. While the modified source and object are in the intermediate library, system integrators and testers can verify that the whole system, or any piece of it, works as advertised. When the system integrators and testers are satisfied with the results of the tests, the source and object can be merged into the official library. The distinguishing factor between the official and intermediate libraries is the the integrity of the software contained in them [CHA81]. The software in the intermediate library has not yet been tested whereas the software in the official library has been tested.

It can been seen from the sample project that the first problem is extracting the source from the STM instances. The library configuration function is done by the software production personnel. It would be preferable if they did not have to keep changing from one machine to another to get their job done. Logging in and out of several machines is time consuming. After the source files have been extracted from STM, the next problem is getting them sent to the library configuration machine. Once the source is on the library configuration machine, it can be compiled using the tools in PAM. After the software has successfully compiled, the developers need to be able to see the results of their changes or the changes that some one else made. This means that there are two problems: first, being able to distinguish intermediate (untested) changes from official (tested) changes and second, the problem of getting both the changed source and the object files distributed back onto the development machines.

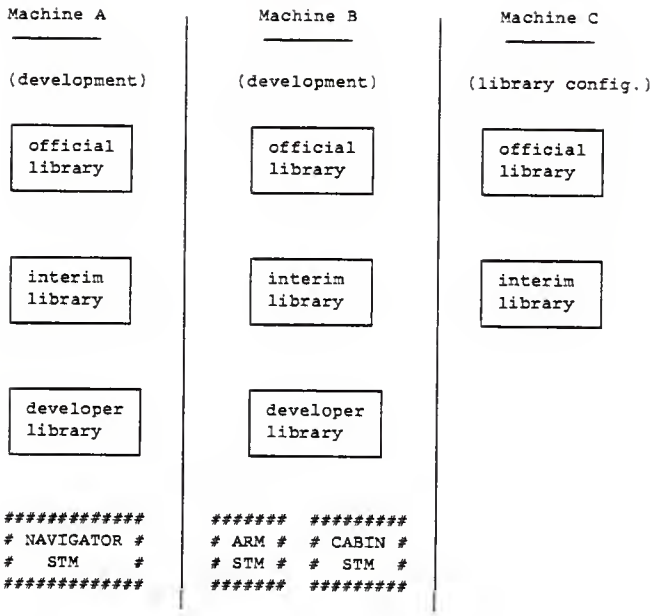


Figure 9. Sample of Project Directory Structure

It should be the function of the System Integration Module (SIM) to automate the process just described. SIM should be restricted to use by the software production personnel only. The intermediate and official libraries should not be writable by anyone else.

SIM should have read or read/write access to the project's databases. SIM should have read access to the database that contains information about the project's configuration. It will have to be able to determine the configuration of the project so that it can gather source files from the appropriate places and bring them to the library configuration machine for compilation. This database will also be used to determine which components should be redistributed to which

machines. SIM will need read access to each of the STM databases so it can extract source files from the STM databases (this is the gathering of source files). Once the source files have been built, it will have to be able to update the status of a change request to show where the change is in the software life cycle. For this activity it will need read/write access to the PET database. Most of the file movement from one machine to another can be accomplished via the UNIX cron facility. One tool on the library configuration machine can send a job to the development machines. The cron can be set up to look for the job every few minutes or so. The job would then be kicked off by the cron and the source files would be extracted from STM. Then the same process could take place again, only in reverse order, sending the source files to the library configuration machine. Again, the same type of thing can be done to redistribute the built source and object files back to the development machines. Another set of tools can be used to merge the intermediate and official libraries and to update the PET database.

The benefits of having SIM are many. It allows large software development projects to be split across several machines. Splitting of projects is often desirable because it conserves on machine resources, keeps all development for a particular component together in a neat package and still allows the entire system to be built together long before delivery time. Testing efforts can be improved too. Component testing can take place on the development machines while testing of the whole system can be done on the library configuration machine.

2.4 Functional Representation of Proposed Tool Set

The five modules of the environment tool set fit together into an integrated tool package. The functional representation of the tools package is presented in Figure 10. A tour of the functional diagram will describe when each tool is used as changes to the software system flow through the life cycle.

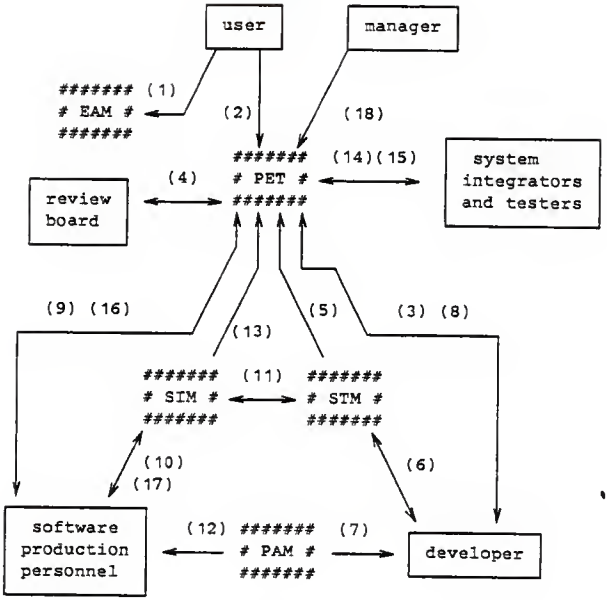


Figure 10. Functional Representation

A user has identified a need for a change. The need could be a problem with an existing software system, a new feature he or she thinks should be added to the system, or a need for a whole new system. The first thing ((1) in Figure 10) the user should do is logon to a machine where the tool set has been installed. The logging in process should automatically connect the user with the rest of the software development environment tools if the user has invoked it from his or her .profile file.

Once connected, the user can record the fact that he or she would like a change to be made (2). At this point the user is interacting directly with the PET module. The change request is

stored in PET's database, assigned a unique number, and assigned the state of "under investigation" [KNU76]. PET automatically notifies the developer (3) who is assigned to the component of the software system that needs to change. The developer then enters the requirements and design phases of the life cycle by studying the problem identified in the change request and finding a solution. When a solution has been decided upon, the developer records his or her proposal with PET. PET changes the change request's status to "waiting approval" in its database.

Once the request's status is "waiting approval", PET notifies (4) the review board that a solution has been proposed. The review board either rejects or accepts the proposed change. If the board rejects the change, the request's status is changed to "rejected", the developer is notified and allowed to make another proposal. If the board accepts the proposal, the request's status is changed to "being fixed" and STM is notified (5) that the change request is a valid one. STM will only allow source files to be checked out with a valid change request.

At this point the developer can use the change request number to check files out of STM (6), make his or her changes, and check the files back into STM. The developer is now working in the implementation phase of the life cycle. During this phase the developer makes use of the product assembly tools in PAM (7) to compile his or her changes. When the developer has finished, he or she notifies PET (8). PET changes the change request's status to "finished", and notifies the software production personnel (9) that the change request is ready to be included in a system build.

The software production personnel use the change request number to tell SIM (10) that all the source files touched by it should be gathered from STM (11) and brought to the library configuration machine. SIM interacts with STM though the cron facility to extract source files from the database and send them to the library configuration machine. The software production

personnel then use PAM (12) to build all the changes into the software system and place them in the intermediate library. They then change the status of the change request to "test" in PET (13). PET notifies the system integrators and testers (14) that the change is ready to enter the testing phase of its life cycle.

Upon satisfactory completion of the testing phase, system integrators and testers change the request's status to "release" in PET (15). PET then notifies the software production personnel (16) that SIM can be used to merge (17) the integrated library into the official library.

Although managers are not directly involved in the software life cycle, they can monitor the progress of changes through a report/query mechanism in PET (18).

It is important to note that this description of events is the usual situation when all goes well. It is possible, for example, that the developer will be able to get the change to build and test just fine in his or her private work area, but the change won't build with the rest of the system for the software production personnel. If at any point in the process something doesn't work, the software development environment should provide the ability to back up and try again.

The proposed tool set has most of the characteristics that a software development environment should have. The tools provide a development support system that suggests a certain set of activities, but does not limit its use to a particular methodology. The tool set has an open architecture to which new tools can be added incrementally. The overall degree of integration is medium. Four of the five tools (EAM, PET, STM, and PAM) are loosely integrated and can be used independent of the other tools. The fifth tool, SIM, is tightly integrated and requires the presence of the PET and STM databases. The proposed tool set also provides a development environment that is tailorable. Information can be retrieved from the project configuration database. This allows multiple projects to use the same set of tools and still configure the project databases according to the project's specific needs.

Using the proposed tool set could provide a project with several benefits. The tailorability of the development support system means that multiple projects can use the same tool set. The benefits associated with this are that the tools themselves are reused, personnel become mobile between projects without the need for retraining, and projects can share information about change requests. The ability to add new tools to the tool set means that the development environment can evolve with the development process and new capabilities can be added as they are identified. The programming community should gain the following benefits: on time delivery of a high quality software system, improved productivity of staff through simplified activities, and cost effective use of resources and management.

CHAPTER 3

MAKEFILE WRITING AID

Developer productivity can be improved through the use of an automated makefile writing aid (*mkaid*). Productivity increases because less time, knowledge and effort is required to produce a consistently complete and correct makefile. The *mkaid* command provides the capability to identify circularly included files, create correct target:dependency pairs, identify when too many nested includes have occurred to be manageable, and uniquely sort the list of included files.

The *mkaid* command should solve these problems by automatically generating makefiles with a minimal amount of input from the user. The *mkaid* command's analysis of included files should not allow circular includes or too many nested includes. Correct target:dependency pairs in a uniform format should be generated by *mkaid*. Also, *mkaid* should be able to identify and eliminate duplicate include files and be able to locate source and include files within a given directory structure.

3.1 Requirements

Besides solving these problems there are requirements that *mkaid* must satisfy. There are four types of requirements for the makefile writing aid (*mkaid*): general requirements, input requirements, output requirements and functionality requirements. These requirements can best be described in terms of an example.

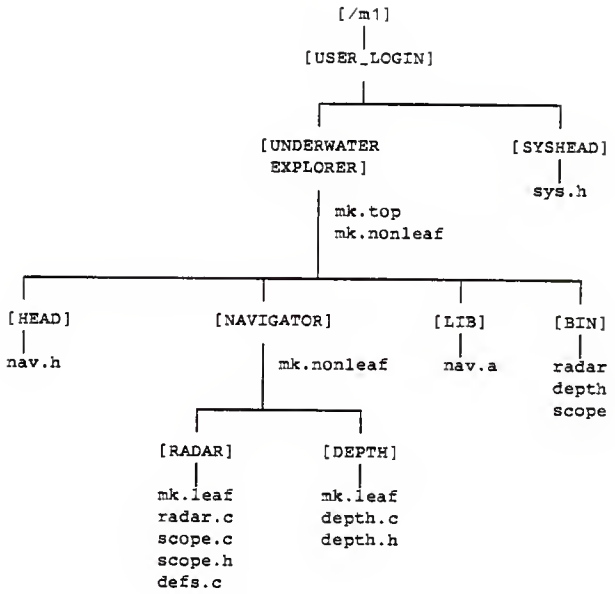


Figure 11. Sample Directory Structure

Figure 11 shows a typical directory structure. It will be used as an example. Notice that the directory structure contains the NAVIGATOR component along with its sub-directories, RADAR and DEPTH. Three more directories have been added to the structure. The HEAD directory contains header files used by the whole UNDERWATER EXPLORER project. The LIB and BIN directories hold finished products. The LIB holds libraries that have been made and BIN holds the executable products. Files are placed in the LIB and BIN directories by user supplied install instructions. Figure 11 is not explained in greater detail here because different sample situations will be explained and examined in the discussion of the requirements which

follow.

3.1.1 General Requirements

The general requirements include a list of functions that *mkaid* should provide to the user. These requirements pertain to the entire tool and are global in nature. The issues addressed in the general requirements include: the language for which *mkaid* will generate makefiles, the treatment of source files for a language other than the chosen language, and the condition of the file system when *mkaid* has finished executing.

The *make* command is a general tool that will compile any file according to the directions provided in a file called makefile. It is *mkaid*'s objective to create the makefile. Ideally, *mkaid* should be able to generate instructions that would tell *make* how to compile software written in any language. However, it is intended that *mkaid* should be added to incrementally. Therefore, this prototype of the *mkaid* tool will generate instructions only for the C language. The C language was chosen because the the *mkaid* tool is to aid development on the UNIX operating system and C is the preferred language in UNIX.

Since *mkaid* will generate instructions on how to compile C source files into object files, it will need a standard way of identifying the two kinds of files. Therefore, *mkaid* will assume that any file whose name ends with the suffix ".c" is a C source file. Likewise, any file whose name ends with the suffix ".o" will be assumed to be an object file. For any files that don't have a suffix, or for any files whose names end in a suffix other than ".c" or ".o", the user will have to specify the instructions he or she wishes the *make* command to use.

The last general requirement is concerned with the condition of the file system upon *mkaid*'s completion. In the event that any temporary files are created by *mkaid*, they should be removed whenever *mkaid* is exited. The temporary files should be removed whether *mkaid* exits

successfully or on an error condition.

3.1.2 Requirements on Inputs

The *mkaid* command should gather certain information from its input files. The information *mkaid* will need includes: the scope of the directory structure within which it is expected to work, any user defined macros and their values, the directory type of the current directory (i.e. leaf or nonleaf), the sub-directories that are to be made, and the products that are to be made.

3.1.2.1 Scope of Directory Structure

The *mkaid* command should be able to determine the scope of the directory structure within which it is expected to work. The directory that contains a file named "mk.top" will be assumed to be the root or top of the directory tree. It is important to note that the "mk.top" file may reside anywhere under the user's login. In the example shown in Figure 11 the directory UNDERWATER EXPLORER under USER_LOGIN will be considered as the top of the directory structure within which *mkaid* will work. This is true because it is in the directory UNDERWATER EXPLORER that the "mk.top" file is located. If it should happen that *mkaid* cannot find the "mk.top" somewhere between the current directory and the USER_LOGIN directory, it will be assumed that no "mk.top" is present. The *mkaid* command will then print an error message to standard error and processing will halt. The error message will be of the form:

ERROR: a mk.top input file cannot be located

The following alternatives where considered and rejected. First, the USER_LOGIN directory could always be considered the root or top of the directory tree. This alternative was rejected because it does not allow for the development of multiple projects under one login. For example, the explorer project in Figure 11 may be only one of several small projects being developed under

the USER_LOGIN directory. If the top of the directory structure was always the USER_LOGIN, all the projects would have to use the same directory structure and they would have to use the same macros (discussed later). This is obviously an undesirable situation. If only one project was being developed under the USER_LOGIN, all the components of the project would have the same set of restrictions.

A second alternative would be to require the user to supply the information on the command line. This alternative is undesirable because the user should not be asked for information that the tool can get for itself. Besides, there is a very good possibility that the user could introduce typing errors.

The last alternative that was considered and rejected was that the current directory would be considered the top of the tree. This alternative does not allow the definition of macros to be used throughout the directory structure. Macros would have to be redefined at every directory level, an undesirable situation.

Several alternatives were considered for the name of the file used to define the scope of the directory structure. The alternatives considered were: global, mk.top, top, and scope. Mk.top was chosen because it is descriptive. Otherwise, the choice was purely arbitrary.

3.1.2.2 Macro Capabilities

A macro capability is very important. A macro is a substitution mechanism whereby a variable is set to some value. Then every time the variable is encountered, it is substituted with the value. For example, if the variable X was set to 5 (X=5), then every time the variable $(X) is encountered, the number 5 will be used in its place.

Users should be able to define macros globally. The macros should be usable in the generation of all makefiles in the directory structure. Therefore, any user defined macro may be

placed in the "mk.top" file and will be used in the generation of makefiles throughout the directory structure. For example, the BIN directory in Figure 11 should be defined in the "mk.top" file as:

BIN=$(TOP)/bin

The TOP macro is a standard macro.[10] that is defined as a relative path from the current directory by *mkaid* when the scope of the directory structure is determined. The user defined BIN macro is therefore defined in terms of $(TOP). If a user would like to install a product in the bin directory, he or she may do so with a command similar to:

cp prod $(BIN)/prod

There is certain information that is necessary for *mkaid* and is most easily expressed in the form of a macro. This information includes the user defined directories that are to be searched for source and included files. These two "macro like" defines are INC and SEARCH. INC can be used to find files outside the scope of the directory structure as defined in the "mk.top" file. SEARCH can be used to find files within the scope of the directory structure but not in the current directory.

For example, assume that the product to be generated in the directory structure in Figure 11 needs to include the header file sys.h which resides in directory SYSHEAD. The scope of the tree stops at the NAVIGATOR directory because it contains the "mk.top" file as described in the above requirement. The SYSHEAD directory is outside the scope of *mkaid*. The *mkaid* command, therefore, will need a way to be able to find the sys.h file. The user can define INC in

10. Standard macros will be discussed in the section on functionality requirements.

the "mk.top" file. INC should be equivalent to the full path name to the directory or directories where the needed file can be found. In this case the macro definition could be:

INC = /m1/LOGIN_ID/SYSHEAD

Note: this is a full path name to the
SYSHEAD directory

In this example, *mkaid* will first search the current directory, then the directories specified by SEARCH, and then the directories specified by INC. In other words, the current directory, the $(TOP)/NAVIGATOR/RADAR and then /m1/LOGIN_ID/SYSHEAD.

Another useful "macro like" define is SEARCH. The SEARCH "macro like" define will aid the user in being able to use files that are located within the scope of *mkaid*, (under the UNDERWATER EXPLORER directory in the example), but not located in the current directory. For example, if the file defs.c is needed to create a product in the RADAR directory, the path to the NAVIGATOR/DEPTH directory (where nav.a resides) could be defined in terms of SEARCH. If SEARCH is defined as:

SEARCH=$(TOP)/NAVIGATOR/DEPTH

in either the "mk.top" or "mk.leaf" file. The user would then be able to simply refer to the needed file as a dependency.

radar: defs.c

The *mkaid* tool will first search[11] the current directory and then the directories as defined by SEARCH. In the unlikely event that there are two defs.c files, the first one found would be used. The order the directories appear in SEARCH list determines the order that the directories are searched.

Note that both the SEARCH and INC "macro like" defines may have more than one directory assigned to them. If more than one directory is defined, they should be a space separated list.

The advantages of defining macros and "macro like" defines in the "mk.top" file are that:

- the definitions can be decided upon and implemented by a project coordinator or planner,
- individual developers only need to know the macro name,
- the likelihood of misspellings or typing errors is reduced,
- the location of files can be changed without making it necessary to edit all or many of the *mkaid* input files,
- multiple developers have access to the same macro, thereby making generation of the final product a more uniform process.

It is necessary to note that it is acceptable for the "mk.top" file to be empty if the project does not have any macros or "macro like" defines. However, the empty "mk.top" file must still be present for *mkaid* to determine the scope of the directory structure.

3.1.2.3 Current Directory Type

The *mkaid* command should be able to determine what kind of a directory it is currently working in, a non-leaf or a leaf directory. This information will be determined by the name of an

11. The search algorithms will be discussed in the section on functionality requirements.

input file in each directory. Non-leaf directories will be expected to contain a file named "mk.nonleaf" and leaf directories will be expected to contain a file named "mk.leaf". Like the selection of the "mk.top" file name, the selection of the "mk.nonleaf" and "mk.leaf" file names was made because the names are descriptive. Otherwise the choice was purely arbitrary.

If it should happen that *mkaid* cannot find the input file, a "mk.nonleaf" or a "mk.leaf", it will issue an error message to the user and then stop execution. The error message will be printed to standard error and will be of the following form:

ERROR: no input file in dir_name

where dir_name is the name of the
directory missing the input file

This situation can best be illustrated with Figure 11 The "mk.nonleaf" in the UNDERWATER EXPLORER directory tells *mkaid* to process the two directories RADAR and DEPTH. If, however, either of the two directories should be missing the *mkaid* input file, "mk.leaf" an error message would be printed to standard error and processing would halt.

In somewhat the same fashion, it is possible that both a "mk.nonleaf" and a "mk.leaf" file could be inadvertently placed in the same directory. If *mkaid* should discover both a "mk.nonleaf" and a "mk.leaf" file in the same directory, it will issue an error message and stop execution. The error message will be sent to standard error and will be of the form:

ERROR: too many input files in dir_name

where dir_name is the name of the
directory containing both files

By not allowing a mixture of "mk.nonleaf" and "mk.leaf" files in the same directory, *mkaid* encourages top down structured design and implementation of a project.

3.1.2.4 Making of Sub-Directories

If *mkaid* is working in a non-leaf directory, it should be able to determine which sub-directories are to be made. This information will be taken from the PRODUCTS[12] line in the "mk.nonleaf" input file. The products line is the only required line in the "mk.nonleaf" file. It is not necessary that all sub-directories be listed just because they exist in the directory tree. The products line in the "mk.nonleaf" file in the UNDERWATER EXPLORER directory of Figure 11 could be:

PRODUCTS = HEAD NAVIGATOR

In this case the directories LIB and BIN would not be made, they would simply be ignored.

The *mkaid* command should be able to determine the order in which directories are to be made. Again, this information will be taken from the PRODUCTS line of the "mk.nonleaf" file. The order will be determined by the order in which the directories are listed on the PRODUCTS line. In the above example where PRODUCTS = HEAD NAVIGATOR, the HEAD directory will be made first and the NAVIGATOR directory will be made second.

3.1.2.5 Making of Executables

If *mkaid* is working in a leaf directory, it should be able to determine the list of products to be made. The list of products to be created will be determined by the PRODUCTS line in the "mk.leaf" file. In this case, the PRODUCTS line is used to determine the actual software product being created. As an example, the products line in the "mk.leaf" file in the RADAR directory in

12. The term PRODUCTS is used to refer to the item(s) being made. At the non-leaf level, the products are sub-directories. At the leaf level the products are the actual executables.

Figure 11 may list two products that are to be created. The products line could be:

PRODUCTS = radar scope

The *mkaid* command should be able to determine the order in which the software products are to be created. Again, this information will be taken from the order in which the products are listed on the PRODUCTS line in the "mk.leaf" file. In the above example where PRODUCTS = radar scope, the product, radar, will be created first and the product, scope, will be created second.

The *mkaid* command should be able to determine what makes up a software product. Extending the example from the two preceding requirements, the following information will need to be added by the user to the "mk.leaf" file in the RADAR directory.

radar: radar.o defs.c

scope: scope.o defs.c sys.h

This means that to make the radar executable, *mkaid* will expect to be able to find the file radar.o in the current directory, the RADAR directory. As will be discussed in the section on functionality requirements, *mkaid* will have information that tells it that a ".o" file is created from a ".c" file. The *mkaid* command will, therefore, know that it will not be able to find a ".o" file and will generate the rule to create the ".o" file from a ".c" file. The *mkaid* command will expect to find a ".c" file for every ".o" file, it will expect to find a file named radar.c in the above example. If *mkaid* cannot, an error message will be sent to standard error and processing will halt. The error message will be:

ERROR: cannot find file.c

To create the second product, scope, *mkaid* will expect to be able to find the file scope.c in the current directory, the DEPTH directory. The *mkaid* command will first look in the current directory for the file defs.c. Being unable to find defs.c in the DEPTH directory it will begin looking in directories as defined by the SEARCH macro. This means that *mkaid* will look in the RADAR directory for file defs.c. In addition, it will expect to be able to find the sys.h file in the USER_LOGIN/SYSHEAD directory, assuming that INC is defined the same as it was in the previous example.

At this point one could visualize *mkaid* as a black-box that accepts a certain set of inputs from the "mk.top", "mk.nonleaf", and "mk.leaf" files. A graphical version of *mkaid* is shown in Figure 12.

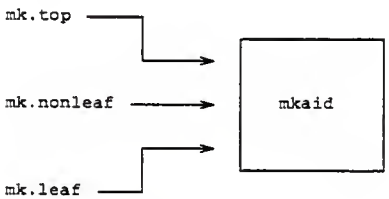


Figure 12. Inputs to mkaid

The following figures are samples of the three types of input files and their contents that *mkaid* will expect from the user.

```
# mk.top file for UNDERWATER EXPLORER

INC=/m1/LOGIN_ID/SYSHEAD
SEARCH=$(TOP)/NAVIGATOR/DEPTH
```

Figure 13. Sample mk.top Input File

```
# mk.nonlead file for NAVIGATOR

PRODUCTS = RADAR DEPTH
```

Figure 14. Sample mk.nonleaf Input File

```
# mk.leaf file for RADAR

PRODUCTS = radar scope

radar: radar.o defs.c
scope: scope.o defs.c sys.h
```

Figure 15. Sample mk.leaf Input File

Note that comment lines may be added to the *mkaid* input files with no ill effects. In fact, it is probably a good idea to use comments to identify each input file.

3.1.3 Requirements on Outputs

The *mkaid* command's goal is to produce a generated makefile that can be used as instructions to the *make* command. There are three basic requirements that *mkaid*'s output should meet: the output should have a uniform format, the output should be placed in a file named "makefile", and the output should be compatible with the *make* command.

3.1.3.1 Makefile Format

The *mkaid* command should standardize the format of makefiles. All resulting makefiles, whether for non-leaf or leaf level directories, should consist of four parts. The first part should be the list of all standard macros known internally to *mkaid* for the type of makefile being created, non-leaf level or leaf level. The second part of a makefile should be a verbatim listing of the "mk.top" file. Third, there should be a verbatim listing of the input file, either the "mk.nonleaf" or the "mk.leaf" file. The forth and final part of the resulting makefile should be the *mkaid* generated instructions that will be used by the *make* command to determine how the products are to be created.

3.1.3.2 Output File Name

The *make* command expects to get its instructions from a file that is named makefile, Makefile, s.makefile, or s.Makefile [UNIX85], in that order. However, the *make* command also has a -f option that will cause *make* to accept a file of any name as input. Therefore, *mkaid* will allow the user to specify the name of the resulting makefile. If the user does not specify a name, *mkaid* will place the generated results into a file called makefile. The file name, makefile, was decided upon because it is the first name that the *make* command looks for. The user can move the makefile to another name if he or she wishes to use one of the other three names or use the -f option on the *make* command line.

3.1.3.3 Compatibility with Make

Compatibility with the *make* command should be maintained. This means that the *make* command will accept and use the instructions in makefiles generated by *mkaid*.

The *mkaid* command can now be visualized as a black-box that accepts inputs as it did before and also creates an output of a specific format.

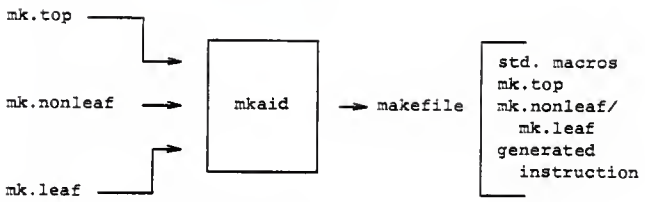


Figure 16. Formatted Output from mkaid

3.1.4 Functionality Requirements

The functionality requirements provide an understanding of what it is that *mkaid* will do with a given set of inputs to create a given output. There are four basic actions that *mkaid* needs to take. These include actions specific to non-leaf level directories, actions specific to leaf level directories, search algorithms to find either source files or included files, and the creation of standard macros.

3.1.4.1 Non-Leaf Directories

In non-leaf directories *mkaid* should: be able to generate rules to change directory (cd) into sub-directories in the appropriate order, be able to execute the UNIX *make* command in every directory that it changes to, and pass the environment to sub-directories via the MKAIDENV macro.[13]

The *mkaid* command's objective in non-leaf level directories is to generate instructions that will cause the *make* command to change (cd) into sub-directories in a user specified order. It is necessary for the user to be able to specify the order in which the sub-directories are made because the contents of one sub-directory may be dependent upon the contents of another sub-directory.

13. The MKAIDENV macro is described in the section on standard macros.

For example, if the HEAD directory of the UNDERWATER EXPLORER (Figure 11) creates a library that is installed in the LIB directory, and if the NAVIGATOR directory needs to access the library, it will be necessary for the HEAD directory to be made before the NAVIGATOR component. If it should happen the the NAVIGATOR component was made first, it would not be able to access the library (because the library hasn't been made yet) and the make would fail.

Once the *make* command has changed to a sub-directory, it should be able to recursively invoke itself in each of the sub-directories. It is necessary for the *make* command to be invoked recursively so that the entire directory tree can be made. The *mkaid* command can assure that the recursion will take place by placing a recursive command line into the resulting non-leaf level makefile.

Another objective is that the entire directory tree be made in such a way that all directories have access to the same set of macros. This objective can be met by passing the environment down from one directory to the next through the MKAIDENV macro.

3.1.4.2 Leaf Directories

In leaf level directories *mkaid*'s objectives are to be able to generate rules for creating object files, create a dependency list, identify nested includes, locate files according to a search algorithm, and identify possible circular includes or includes that are nested too deep.

The *mkaid* command should be able to generate the appropriate *ld* command line that will create an object file of the same name as the source file. In other words, if the source file is "radar.c" then the object file should be named "radar.o". This can be achieved by using the -o option to the *ld* command.

The *mkaid* command should be able to generate a dependency list. The dependency list should include all files that are #included in a C source file. The dependency list should include the

path, either relative or full path, to the included file. For example, if *mkaid* determined that the "radar.c" file includes the stdio.h file from /usr/include, the dependency list should include the following line:

radar.o: /usr/include/stdio.h

Every file that is included in a source file may also include another file. The *mkaid* command should, therefore, be able to recursively check through all included files for further includes. In addition, these nested included files should be identified as nested includes in the resulting makefile. The resulting dependency line for a nested include would be:

radar.o: /usr/include/sys/types.h #nested include

Finally, *mkaid* should be able to identify whether or not a circular include as taken place or if included files have been nested too deeply. This objective can be achieved if *mkaid* keeps track of the level of nested includes that have taken place. If the number of level exceeds seven,[14] *mkaid* should provide an error message that indicated that a problem with circular includes or too many nested includes has taken place.

3.1.4.3 Search Algorithms

The *make* command must be able to find any files that have been identified as dependencies. To prevent the user from having to know where every file is located, *mkaid* will use two different search algorithms to locate files. One algorithm will locate source files and the other will locate included files. The *mkaid* command's search algorithm for finding source files will be: to search

14. The selection of the number seven was chosen because that is about the limit of the number of activities that a human mind can keep track of at any one time.

the current directory first, and then to search the directories specified by the user in SEARCH. The *mkaid* command's search algorithm for finding included files will be: to search in the current directory first, then to search directories specified by SEARCH, then to search the directories specified by INC, and finally to search directories specified by UNIC. If UINC is not specified by the user, it will default to the /usr/include directory.

3.1.4.4 Standard Macros

The *mkaid* command will have standard macros. The following three figures are lists of the standard macros for non-leaf level makefiles, user defined macros, and leaf level makefiles. They include the macro names, how they will be defined internally to *mkaid* and what they will be used for.

macro	definition
MKFLAGS	A way to specify $MAKE flags from a high level directory.
AIDENV	A way to specify low level cc macros from a high level directory (e.g., AIDENV="CFLAGS=-g")

Figure 17. User Defined Macros

macro	definition	use
SEARCH	[user specified]	search path for source files and include files
INC	[user specified]	directories searched for included files
UINC	[user specified] defaults to /usr/include	specifies directories to be searched for included files
INC_LIST	[generated -I #include list]	list used by cc to define relative directory paths to #include files. Set to directories actually used from the $INC.
LIB_LIST	[user specified library list]	library list used by the loader.
CC	cc	the cc command
CFLAGS	-O	flags used for cc
CC_CMD	$(CC) -c $(CFLAGS)	combines CC and CFLAGS for makefile uniformity
CCLD_CMD	$(CC) $(LDFLAGS)	used to link-edit
LD	ld	the ld command
LDFLAGS		flags used for ld
LD_CMD	$(LD) $(LDFLAGS)	combines LD and LDFLAGS for makefile uniformity
DEFS		user specified cpp definitions
TARGET	all	set to $(PRODUCTS)

Figure 18. Leaf Level Macros

macro	definition	use
MK.TOP	Relative location of "mk.top" file from the current directory	The "mk.top" file contains macro definitions used globally within the directory structure
TOP	Relative path to the top of the directory struct.	Used to define scope of directory structure structure
MAKE_CMD	$(MKFLAGS) $(MKAIDENV) $(AIDENV)	The make command with associated parameters
MKAID	mkaid	The mkaid command
TARGET	all	List of $MAKE targets in the nonleaf makefile. Set to $PRODUCTS thereby forcing the make order.
MKAIDENV	"MKFLAGS=$(MKFLAGS)" "TARGET=$(TARGET)" "AIDENV=$(AIDENV)"	Used to pass make environment down from one directory to the next.

Figure 19. Nonleaf Level Macros

In actuality, two different kinds of makefiles will be generated by *mkaid*, either a high level makefile or a low level makefile. High level makefiles will reside in non-leaf directories while low level makefiles will reside in leaf directories. The inputs for a high level makefile will be the "mk.top" and the "mk.nonleaf" files. The inputs for the low level makefile will be the "mk.top" and the "mk.leaf" files. A somewhat more concise representation is therefore provided in the Figure 20.

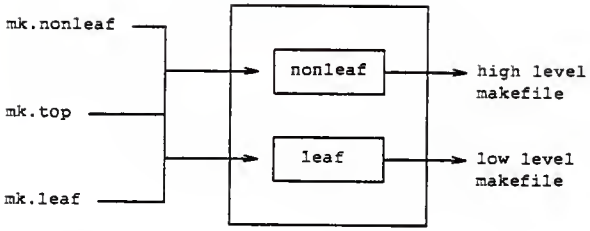


Figure 20. Low Level and High Level Makefiles

3.2 Conclusion

The above requirements have thus far provided information about the inputs to *mkaid*, about the functionality of *mkaid*, and about the output of *mkaid*. It is intended that *mkaid* will aid the user in the writing of makefiles for the development and maintenance of a project. It will aid the user by requiring only a minimal amount of input, locating files for the user, and by creating a file called "makefile" in the user's current directory.

CHAPTER 4

DESIGN

The requirements have provided information about the inputs to *mkaid*, about the functionality of *mkaid*, and about the output of *mkaid*. The design will be concerned only with the internal workings of the *mkaid* command. First, the high level or architectural design will break *mkaid* down into modules. Each module will provide a specific function. Then the low level or detailed design will provide an explanation of the algorithm to be used in each function.

In very general terms, *mkaid* will have to perform three basic functions to create either a high or low level makefile. These functions are:

1. find and read the input files to determine which sub-directories or products are to be made and the order in which they are to be made,
2. generate the targets and the rules to make each sub-directory or product, and
3. write the resulting makefile.

To accomplish these three basic operations, *mkaid* will be made up of five modules or functions, each of which has a specific task to perform. The architecture of *mkaid* is presented pictorially in Figure 21.

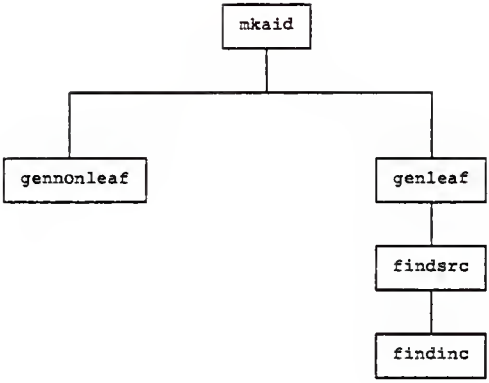


Figure 21. Architectural Design of mkaid

The language used for an implementation has an effect on the architectural design. Since *mkaid* is a prototype it will be implemented in shell. The definition of global variables is difficult in shell. For example, module A can set a variable to a value, say X=5. When module A finishes executing the X variable vanishes. If module B then begins executing, it will not be able to access the variable X unless it resets it. As a result, if a global variable is needed to perform more than one function, it is generally easier to include those functions in one module (that is, if they aren't too large).

The *mkaid* command contains a very important global variable. The global variable is the type of the current directory (non-leaf or leaf). The current directory type determines whether the resulting makefile will contain instructions to make sub-directories or executable products. It therefore determines the specific kind of instructions that are placed in the makefile as well as the specific set of standard macros.

The operational flow of *mkaid* is determined by a series of calls to modules which make up the whole of the *mkaid* command. The **mkaid** module which determines whether the *gennonleaf* or *genleaf* module is to be executed. The first information that *mkaid* needs is scope of the directory structure and the relationship between the current directory and the top of the directory structure. Once the top of the directory structure has been determined, *mkaid* needs to know the type of makefile it is expected to generate, a leaf level or non-leaf level makefile. This information is determined by the name of the input file in the current directory, a "mk.nonleaf" or "mk.leaf". If the generated makefile is for a non-leaf directory, then **gennonleaf** is called to generate the instructions for a non-leaf makefile. If the generated makefile is for a leaf level directory, then **genleaf** is called to generate the instructions for a leaf level makefile. The **genleaf** command will call the **findsrc** module to implement the search algorithm to locate source files, check for files included in the source file and then call *findinc*. The *findinc* command will locate included files. If an included file includes another file (a nested include), *findinc* will recursively call itself. Once all the information has been gathered and the *gennonleaf* or *genleaf* modules have returned to *mkaid*, the resulting makefile will be printed.

The *mkaid* modules will create three temporary files during the construction of the makefile. All three of these files will be created in the user's current directory and will be removed by *mkaid* when the program has completed. The first of the three temporary files will be named "tmpmake" and will be created by either the **gennonleaf** or **genleaf** module, depending on which type of makefile is being generated. If it is created by **gennonleaf** it will contain information about the target directories to be made and the rules to be used to make the directories. If the temporary file, "tmpmake", is created by **genleaf**, it will contain information about what files are to be included to make the intermediary target (.o files) and the rule to be used to create the target.

The second temporary file will be named "deplist" and is only created if *mkaid* is working in a

leaf level directory. This temporary file will contain the target:dependency pairs for one ".o" file at a time. When *mkaid* has found all the target:dependency pairs for a particular ".o" file, the "deplist" file will be uniquely sorted and appended to the "tmpmake".

The final temporary file will be named "inclist" and will again only be created if *mkaid* is working in a leaf level directory. The "inclist" file will contain a list of directories in which include files were actually found. In other words, the user may define several directories in SEARCH, INC and UINC, but only a couple of the directories may actually be needed to locate the included files. The "inclist" file will contain a list of the directories that are actually needed. If more than one include file happened to be located in the same directory, that directory would show up in "inclist" multiple times. Therefore, "inclist" will be uniquely sorted before its contents are added to the resulting makefile.

A somewhat more detailed description of each of the modules can provide better insight into *mkaid*'s functionality.[15] The *mkaid* module is the main module and it is invoked by the user. There is only one optional parameter, a user specified name for the resulting makefile. If the user does not specify a name, *mkaid* will place the resulting makefile in a file named "makefile". The other inputs that *mkaid* expects to be able to find are a "mk.top" file and either a "mk.nonleaf" or mk.leaf" file. The first action performed by *mkaid* is to locate the "mk.top" file and set the TOP macro (relative path to the directory containing the "mk.top" file) and the MK.TOP macro (relative path to the "mk.top" file).

Next *mkaid* determines the type of directory it is working in (leaf or non-leaf). This information is determined by the presence of a "mk.nonleaf" or "mk.leaf" file. If neither file or

15. A shortened version of the detailed design is provided in Appendix A.

both files exist in the same directory, an error message is printed, temporary files are removed, and processing stops. If a "mk.nonleaf" file is in the current directory, *mkaid* will call gennonleaf to generate make instructions. If a "mk.leaf" file is in the current directory, *mkaid* will call genleaf to generate make instructions. When gennonleaf or genleaf returns to *mkaid*, the appropriate standard macros, "mk.top" file, "mk.nonleaf" or "mk.leaf" file and generated instructions are placed in the resulting makefile.

The gennonleaf module is called if the current directory type is non-leaf. The gennonleaf function is a very short module. It extracts the PRODUCTS line from the "mk.nonleaf" file, and then initializes the temporary file "tmpfile" with an all target that depends on all the products from the PRODUCTS line. Next it checks to make sure that each directory actually resides under the current directory. If the directory does not exist, an error message is written and processing stops. If the directory does exist, gennonleaf generates the appropriate target:dependency pairs and make instructions and places them in "tmpfile".

If *mkaid* determines that the current directory is a leaf level directory, it calls genleaf. The genleaf function extracts the PRODUCTS line from the "mk.leaf" file. Then for each product listed on the PRODUCTS line, it checks to make sure that the product is listed as a target in the "mk.leaf" file. If it isn't, an error message is printed and processing stops. If the product is listed as a target in the "mk.leaf" file, all of the target's dependencies are checked to determine if any of them are ".o" files. Every ".o" file name is then turned into a ".c" file name and passed as a parameter to findsrc.

The findsrc function takes the ".c" file name passed to it by genleaf as input. The findsrc function implements the search algorithm to locate source files by looking for the ".c" file in the current directory and then in each of the directories listed in SEARCH. SEARCH can be set in either the "mk.top" file or the "mk.leaf" file. If SEARCH should happen to be set in both files,

the one from the "mk.leaf" file takes precedence. After the ".c" file has been located, findsrc extract any #include lines from the ".c" file. It then calls findinc with the list of include files that it extracted from the ".c" file.

The findinc function is a recursive module that implements the search algorithm to locate include files. The first thing it does is to check the number of times it has been called. This check fulfills two requirements: it checks for circular includes and it makes sure that the number of nested includes is not too deep. Next, findinc determines all the directories in which it should look for an include file by concatenating the current directory with the contents of SEARCH, INC and UINC. Then, findinc will look for the included file in each of the directories. When the included file has been located, findinc will extract any #include files from it and recursively call itself. If an included file cannot be located, findinc will print an error message, remove any temporary files and stop processing.

The design of *mkaid* is specialized for an implementation in shell. The design is modularized so that each of the units is as self contained as possible. The setting and passing of global variables in shell limited the modularization of the mkaid module, but it is still small enough to be easily understood and maintained.

CHAPTER 5

IMPLEMENTATION, CONCLUSIONS AND EXTENSIONS

The *mkaid* command was implemented on a VAX as a prototype tool to aid in the writing of makefiles at Kansas State University. The *mkaid* tool has a twofold purpose: it is the beginning of the proposed minimal tool set that provides a software development environment and it is a tool that can be used by Kansas State University students in the administration of their team projects. The total implementation consists of approximately 400 lines of shell code that executes under the UNIX Operating System.

The *mkaid* command enforces good software engineering principles, but is still a very flexible tool. Good software engineering principles are encouraged because *mkaid* assumes a structured top-down design of a project. That is, a project is assumed to be made up of components; components are assumed to be made up of sub-components, etc.; and it is only at the lowest leaf levels of the directory structure that the actual executable products are created. Although *mkaid* encourages the decomposition of a project in a structured manner, it is still flexible. The users have the ability to create and define their own macros. In addition, the users can gain access to files located anywhere in the directory structure via the SEARCH, INC, and UINC "macro like" defines.

There are at least three possible extensions to *mkaid*. The *mkaid* command's performance can be enhanced by converting it from shell to the C programming language. In addition to executing faster, *mkaid* could be more modularized if it was written in the C language because C provides a better facility for setting and passing global information. The *mkaid* command could also be extended to generate makefiles that will provide the *make* command with instructions to build

software products from source code written in Pascal, Lex, Yacc, Assembly language, etc. Lastly, a very useful extension would be for *mkaid* to recursively descend the directory tree generating makefiles in all appropriate sub-directories.

While the proposed software development environment tool set does not cover the entire software development life cycle, it does cover a large portion of it by supporting the implementation phase. If the maintenance phase is considered to be repeated cycles of the software life cycle phases, the proposed tool set provides an even better coverage.

Each of the five tool modules that make up the software development environment solves a particular set of problems. The total environment, however, solves problems that are more global in nature. Accurate up-to-date information can help project managers to make well-informed decisions about resource allocation, scheduling, and costs. Developers can become more productive because less of their time is spent reworking the same kinds of problems multiple times or hunting around for misplaced information. Software production personnel have less confusion and fewer errors in the building of a software system because much of the process is automated. System integrators and testers can easily identify the pieces of a software system that need testing. In other words, the proposed software development environment can provide a means by which software and all the associated information can be managed in a cost effective manner.

There are several extensions that can be made to the proposed software development environment. The first is a user interface. The interface should provide a consistent link between the user and the development environment tool set. Next a project management tracking system could be implemented. Such a tracking system could include a calendar manager, a milestone watchdog that notifies users when target dates are getting close, and a project critical path generator that recalculates the critical path to answer "what if" questions. An automated testing module would be of value to both developers and system integrators and testers. The testing

module should be able to automatically execute test sets during low machine usage hours and compare the results with the results from previous runs. The result comparisons could be extremely valuable in regression testing.

Finally, the proposed software development environment could be extended to work on individual work stations. Work stations are becoming more and more popular. However, they present a whole new set of project management issues. As an example, if several developers are working on individual work stations, the issue of how the pieces are integrated into a system becomes more difficult.

REFERENCES

[BOE80] Boehm, B.W., "Software and Its Impact: A Quantitative Assessment", Tutorial on Software Design Techniques, 3rd. ed., IEEE Computer Society, Long Beach, California, 1980, pp.5-16.

[BOE84] Boehm, B., "A Software Development Environment for Improving Productivity", Computer, Vol. 17, No. 6, June 1984, pp. 30-44.

[CHA81] Chauza, E.J. and Fortune, L.E., "GTD-5 EAX Software Management Support System", GTE Automatic Electric World-Wid Communications Journal, Vol. 19, No. 3, May-June 1981, pp. 90-96.

[CHE84] Chesi, M., Dameri, E., Franceschi, M.P., Gatti, M.G., Simonelli, C., "ISDE : An Interactive Software Development Environment", ACM SIGPLAN Notices, Vol. 19, No. 5, May 1984, pp. 81-88.

[DEL84] Delisle, N.M., Menicosy, D.E., and Schwartz, M.D., "Viewing a Programming Environment as a Single Tool", ACM SIGPLAN Notices, Vol. 19, No. 5, May 1984, pp. 49-56.

[DOW86] Dowson, M., "ISTAR - An Integrated Project Support Environment", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN, Vol. 22, No. 1, January 1987, pp. 27-33.

[ER84] Er, M.C., "Managing Source Code", Journal of Systems Management, Vol. 35, No. 10, October 1984, pp. 12-14.

[FAR85] Farrell, K., "Subjective impressions of a non-naive Unix and C beginner", Microprocessors and Microsystems, Vol. 9, No. 7, September 1985, pp. 333-336.

[FEL79] Feldman, S.I., "Make - A Program for Maintaining Computer Programs", Software-Practice and Experience, Vol. 9, No. 4, April 1979, pp. 255-265.

[FIL86] Filipshi, A., and Dion, L.C., "Project source file management under the UNIX operating system", National Computer Conference, AFIPS, Vol. 55, 1986, pp. 267-271.

[HOW81] Howden, W., "Contemporary Software Development Environments", SIGSOFT Software Engineering Notices, Vol. 6, No. 4, August 1981, pp. 6-15.

[HOW82] Howden, W.E., "Contemporary Software Development Environments", Communications of the ACM, Vol. 25, No. 5, May 1982, pp.318-329.

[JOH83] Johnson, D., "Development Tools Move Into High-Level Programming Environments", Digital Design, Vol. 13, No. 7, July 1983, pp. 90-92.

[KER81] Kernighan, B.W., and Mashey, J.R., "The Unix Programming Environment", Computer, Vol. 14, No. 4, April 1981, pp. 12-24.

[KNU76] Knudsen, D.B, Berofsk, A. and Satz, L.R., "A Modification Request Control System", Proceedings 2nd. International Conference on Software Engineering, San

Francisco, California, October 1976, pp. 187-192.

[MOL80] Molko, P.M., "SAMS: Addressing Managers' Needs", Proceedings 4th International Conference on Computer Software and Applications, Chicago, Illinois, 1980, pp. 691-697.

[OST81] Osterweil, L., "Software Environment Research: Directions for the Next Five Years", Computer, Vol. 14, No.4, April 1981, pp. 35-43.

[OST83] Osterweil, L.J., "Toolpack -- An Experimental Software Development Environment Research Project", IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, November 1983, pp. 673-685.

[RID80] Riddle, W.E., "Panel: Software Development Environments", Proceeding 4th. International Conference on Computer Software and Applications, Chicago, Illinois, 1980, pp. 220-224.

[RID83] Riddle, W.E., "The Evolutionary Approach to Building the Joseph Software Development Environment", SOFTFAIR. A Conference on Software Development Tools, Techniques, and Alternatives. Proceedings, Washington D. C., 1983, pp. 317-325.

[ROB83] Reese, R., Sheppard, S., "A Software Development Environment for Simulation Programming", IEEE, Vol. 2, New York, December 1983, pp. 419-426.

[ROC75] Rochking, M.J., "The Source Code Control System", IEEE Transaction on Software Engineering, Vol. SE-1, No. 4, December 1975, pp. 364-370.

[STE87] Stenning, V., "On the Role of an Environment", Proceedings 9th International Conference on Software Engineering, Monterey, California, April 1987, pp. 30-34.

[UNIX85] AT&T, UNIX System V User's Manual, October 1985.

[WAL84] Walden, K., "Automatic Generation of Make Dependencies", Software-Practice and Experience, Vol. 14, No. 6, June 1984, pp. 575-585.

[WAR84] Warren, C., "Utility software streamlines program development and operation", Mini-Micro Systems, Vol. 17, No. 12, October 1984, pp. 113-114.

[WAS80a] Wasserman, A.I., "A Strategy for Improving Software Development Practices", Tutorial on Software Design Techniques 3rd. Edition, IEEE Computer Society, Long Beach, California, 1980, pp. 440-444.

[WAS80b] Wasserman, A.I., "Software Tools and the User Software Engineering Project", Software Development Tools, 1980, pp. 93-113.

[WAS81] Wasserman, A.I., "Automated Development Environments", Computer, Vol. 14, No. 4, April 1981, pp. 7-10.

mkaid

Called by: user

Synopsis: mkaid [file_name]

Inputs: optional file name on command line,
also expects to be able to find a
"mk.top" file and either a "mk.nonleaf"
or "mk.leaf" file

Function: Determines flow the of the program, calls
the other functions.
Determines the current directory and the
login directory locations. It searches
backward from the current directory to
the login directory looking for a
"mk.top" file. Then either sets the TOP
and MK.TOP macros if the "mk.top" file is
found, or prints and error message and
exists if one is not found.
Determines type of directory it is
working in, leaf or nonleaf, by the type
of input file present in the directory.
It writes an error message if no input
file is present or if both types of input
files are present, and then exits.

Calls: **gennonleaf** if the current directory
is a non-leaf directory to generate non-
leaf level make instructions

genleaf if the current directory
is a leaf level directory to generate
a leaf level make instructions

Temp. files : tmpfile - read regardless of
the directory type to gather
target:dependency pair information

inclist - read only when the directory
type is nonleaf to gather
information about what directories
are used to locate include files

deplist - ready only when the
directory type is nonleaf to gather
information about the
target:dependency pair for each
intermediate (".o" file) target

Outputs: a generated makefile

Exit condition: exits (error) if there are too many
input files, if there aren't any
input files, or if the scope of the
directory structure cannot be
determined exits (success) if
everything went well and a makefile
was generated

gennonleaf

Called by: mkaid if the directory type is nonleaf

Synopsis: gennonleaf

Inputs: none

Function: generates targets and rules to make sub-directories

Calls: none

Temp. files: tmpfile - contains information about how to make sub-directories

Outputs: none

Exit condition: exits (error) if a directory listed on the PRODUCTS line in the mk.nonleaf does not exist under the current directory

exits (success) if it created the temporary file "tmpfile"

genleaf

Called by: mkaid if the directory type is leaf

Synopsis: genleaf $MK.TOP

Inputs: $MK.TOP - relative path name to "mk.top" file

Function: It generates targets and rules to make products and puts the targets and rules in the makefile.

Calls: The **findsrc** module to locate source or included files.

Temp. files: tmpfile - contains information about how to make intermediary targets (".o" files)

Outputs: tmpfile

Exit condition: exits (error) if a product is not specified as a target in the "mk.leaf" file

exits (sucess) if everything went well

findsrc

Called by: genleaf

Synopsis: findsrc SDOT_C SMK.TOP

Inputs: SDOT_C - ".c" file name

SMK.TOP - relavtive path the "mk.top" file

Function: searchs for the specified source file according to the search algorithm and extracts include file names from the souce file

Calls: **findinc** with the name of included files

Temp. files: deplist - contains information about the ".o":".c" target:dependency pair

Outputs: deplist

Exit conditions: exits (error) if a ".c" file cannot be located

exits (success) if everything went well

findinc

Called by: findsrc

Synopsis: findinc $INC $DOT_O $MK.TOP

Inputs: $INC - include file name

$DOT_O - ".o" file name

$MK.TOP - relative path to "mk.top" file

Function: searchs for the specified include file according to the search algorithm for include files

Calls: **findinc** recursively with the name of included files to determine the location of included files

Temp. files: inclist - contains information about the files to be included in the creation of an intermediary target (".o" files)

Outputs: inclist

Exit conditions: exits (error) if an include file cannot be located

exits (success) if everything went well

NAME

mkaid – a makefile writing aid

SYNOPSIS

mkaid [output file name]

DESCRIPTION

The *mkaid* command is a software development tool which expands a minimum specification of how to produce a product from its component parts into a makefile suitable for use by *make*. The *mkaid* command utilizes the UNIX directory structure to support a directory hierarchy. Non-leaf level directories are defined as those in the hierarchy which are control points and contain makefiles which change directory to subdirectories, then invoke *make*. Leaf level directories contain makefiles which produce a product, e.g., executables, etc.

Non-leaf level directories contain a file, named "mk.nonleaf", which includes at least a macro definition:

PRODUCTS = [list of subdirectories]

The *mkaid* command converts the list of subdirectories into *make* rules which change directory to the subdirectories, and invoke *make*. A different input file name may be optionally specified.

Leaf level directories are where objects, etc., are built; the specification of how to produce the product uses *make* syntax, and is kept in a file by the name: "mk.nonleaf." The resulting makefile is written in the current directory and takes on the following form:

standard/generated macro definitions

a project-supplied file named "mk.top"
which defines the scope of the development
directory structure, and provides project-
dependent macros.

the user-supplied "mk.leaf" file

generated rules and dependencies for
building each major product

The *mkaid* command generates a complete makefile by adding rules and dependencies to the supplied "mk.leaf" file. Rules for building object files are generated according to the suffix rules below:

From	To	Rule
file.c	file.o	$(CC_CMD) file.c

All source and header files are located and scanned for #include dependencies. These dependencies and nested dependencies are included in the generated makefile.

The *mkaid* command recognizes #include statements by the '#include' in column one. Comment characters and any lines without a '#' in column one are ignored.

Source files (.c) are searched for in the current directory. The current directory is augmented by the search path defined by a the macro "SEARCH," which is a blank separated ordered list of full pathnames. Included files are searched for in the same directories as source files plus the directories defined by the INC and UINC macros.

A code of zero is returned by *mkaid* if the makefile is generated correctly. Errors are indicated by a error message and a non-zero return code.

EXAMPLE

The following is an example of a "mk.leaf" file that makes the product "product" from the object files "x.o". The rules and dependencies for making the .o files from the appropriate source files (e.g. x.c) are provided by *mkaid* in the generated makefile.

```
PRODUCTS = product

PROD_OBJ = 2
    x.o

product:  $(PROD_OBJ)
    $(CCLD_CMD) product $(PROD_OBJ)
```

The following command may be used to generate the makefile in the current directory:

```
mkaid
```

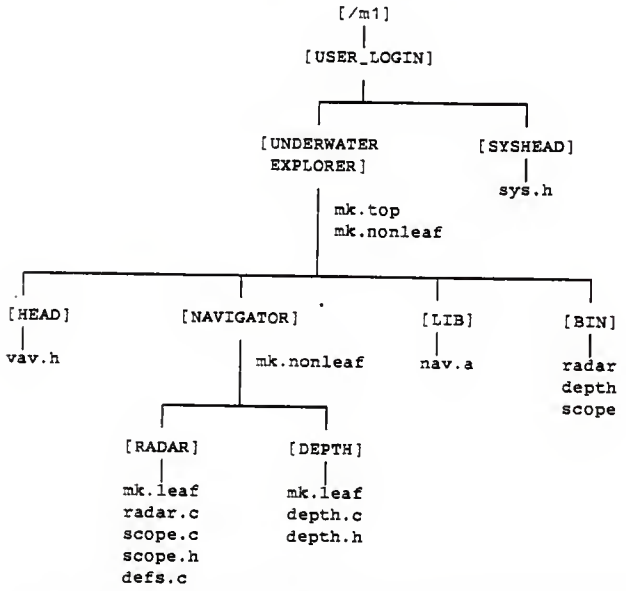
FILES

"mk.top", "mk.nonleaf", "mk.leaf"

SEE ALSO

make(1)

Using *mkaid* can be very easy if a few simple rules are followed. The directory structure for the Underwater Explorer project, which was presented in Chapter 3, will be used as the sample project in this user manual. The directory structure is shown below.



The first rule to be followed is that a "mk.top" file must be present in the directory structure. In the Underwater Explorer project, the "mk.top" file resides directly under the UNDERWATER_EXPLORER directory. The "mk.top" file may contain any macros that the project wishes to be defined throughout the directory structure or it may be empty. The "mk.top" file in the example has two macros defined, SEARCH and INC. Both macros are defined as full path names to directories. These directories will be searched for either source or included files.

```
#
# mk.top for underwater project
#

SEARCH=/usre/att/pfaff/include \
        /usre/att/pfaff/underwater/head
INC=/usre/att/pfaff/underwater\
        /lib /usre/att/pfaff/syshead
```

The second rule to be followed is that each non-leaf directory must contain a "mk.nonleaf" file. The "mk.nonleaf" file must contain at least one line; the PRODUCTS line. The PRODUCTS line should be a space separated list of directories that are to be made. The "mk.nonleaf" file for the UNDERWATER directory is shown below. Notice that only one sub-directory, NAVIGATOR, is to be made.

```
# mk.nonleaf for underwater

PRODUCTS= navigator
```

At this point, the required files are present so that a makefile can be created in the UNERWATER_EXPLORER directory. To create the makefile, the user only needs to type:

```
mkaid
```

As a result, a makefile will be placed in the current directory. The resulting makefile is shown below.

```
###########
#
#       Standard Macros
#
###########

TOP = /usre/att/pfaff/underwater
MK_TOP = /usre/att/pfaff/underwater/mk.top
MAKE_CMD = make $(MKFLAGS) $(MKAIDENV) $(AIDENV)
MKAIDENV = $(MKFLAGS) $(TARGET) $(AIDENV)
TARGET = all

all:


###########
#
#       mk.top
#
###########

#
# mk.top for underwater project
#

SEARCH=/usre/att/pfaff/include \
        /usre/att/pfaff/underwater/head
INC=/usre/att/pfaff/underwater/lib \
        /usre/att/pfaff/syshead


###########
#
#       mk.nonleaf
#
###########

# mk.nonleaf for underwater

PRODUCTS= navigator


###########
#
```

```
#       Instructions for making sub-directories
#
##########

all:    navigator

navigator:      dnavigator
dnavigator:
        @d=`pwd`; \
        echo "Making $$d/navigator"; \
        cd navigator; exec $(MAKE_CMD) $(TARGET)
```

The same rules apply for each non-leaf level directory. A "mk.nonleaf" files must be present. It must contain a PRODUCTS line which names the directories to be made. The "mk.nonleaf" file for the NAVIGATOR directory is shown below. Notice that two sub-directories, RADAR and DEPTH, are to be made.

```
#
# mk.nonleaf for navigator
#

PRODUCTS = radar depth
```

```
##########
#
#       Standard Macros
#
##########

TOP = ..
MK_TOP = ../mk.top
MAKE_CMD = make $(MKFLAGS) $(MKAIDENV) $(AIDENV)
MKAIDENV = $(MKFLAGS) $(TARGET) $(AIDENV)
TARGET = all

all:


##########
#
#       mk.top
#
##########

#
# mk.top for underwater project
#

SEARCH=/usre/att/pfaff/include \
        /usre/att/pfaff/underwater/head
INC=/usre/att/pfaff/underwater/lib \
        /usre/att/pfaff/syshead


##########
#
#       mk.nonleaf
#
##########

#
# mk.nonleaf for navigator
#

PRODUCTS = radar depth
```

```
##########
#
#       Instructions for making sub-directories
#
##########

all:      radar depth

radar:  dradar
dradar:
        @d='pwd'; \
        echo "Making $$d/radar"; \
        cd radar; exec $(MAKE_CMD) $(TARGET)

depth:  ddepth
ddepth:
        @d='pwd'; \
        echo "Making $$d/depth"; \
        cd depth; exec $(MAKE_CMD) $(TARGET)
```

The leaf level directories will also need to have makefiles created in them. The *mkaid* input file in a leaf level directory is a "mk.leaf" file. The "mk.leaf" file also has a PRODUCTS line, however, this time the items listed on the products line are the actual executable programs that are to be created. In addition, each executable listed on the products line will also have to be listed as a target in the "mk.leaf" file. The sample "mk.leaf" file for the RADAR directory is shown below.

```
#
# mk.leaf for radar
#

PRODUCTS=radar scope

radar:  radar.o
        $(CCLD_CMD) radar radar.o

scope:  scope.o
        $(CCLD_CMD) scope scope.o
```

The user can create the resulting makefile by simply executing *mkaid* as it was invoked in non-leaf directories.

```
mkaid
```

The resulting leaf level makefile is shown below.

```
##########
#
#       Standard Macros
#
##########

TOP = ../..
MK_TOP = ../../mk.top
CC = cc
CFLAGS = -O
DEFS =
CC_CMD = $(CC) -c $(CFLAGS)
CCLD_CMD = $(CC) $(LDFLAGS)
LD = ld
LDFLAGS = -o
LD_CMD = $(LD) $(LDFLAGS)
TARGET = all


INC_LIST        =\
        -I/usr/include\
        -I/usre/att/pfaff/syshead/sys.h\
        -I/usre/att/pfaff/underwater/head/nav.h\
        -I/usre/att/pfaff/underwater/navigator/radar

all:
install:  all


##########
#
#       mk.top
#
##########

#
# mk.top for underwater project
#

SEARCH=/usre/att/pfaff/include \
        /usre/att/pfaff/underwater/head
INC=/usre/att/pfaff/underwater/lib \
        /usre/att/pfaff/syshead
```

```
##########
#
#       mk.leaf
#
##########

#
# mk.leaf for radar
#

PRODUCTS=radar scope

radar:  radar.o
        $(CCLD_CMD) radar radar.o

scope:  scope.o
        $(CCLD_CMD) scope scope.o


##########
#
#       Instructions for making executables
#
##########

all:    radar scope

radar.o: /usr/include/stdio.h
radar.o: /usre/att/pfaff/syshead/sys.h
radar.o: /usre/att/pfaff/underwater/navigator\
                /radar/defs.c
radar.o: radar.c
        $(CC_CMD) radar.c

scope.o:/usr/include/stdio.h
scope.o:/usre/att/pfaff/underwater/head/nav.h
scope.o:/usre/att/pfaff/underwater/navigator\
                /radar/scope.h
scope.o:scope.c
        $(CC_CMD) scope.c
```

Following a few rules and the sample *mkaid* input files should provide enough information so

that *mkaid* can be a useful software development tool.

```
#######################
#
# Name:  mkaid
#
# SYNOPSIS:  mkaid [filename]
#
# INPUTS:  optional user defined name for resulting
#          makefile
#
#          Three input files will be checked for and the
#          information contained in them will be used to
#          generate the resulting makefile.  These files
#          are to be named: mk.top, mk.nonleaf, and
#          mk.leaf.
#
# OUTPUTS:  a formated makefile that can be used as input
#          to the make command.
#
#######################

MAKEFILE=$1

TOP=
LOGIN=$HOME
FOUNDFLG=FALSE
CURDIR=`pwd`
SAVDIR=`pwd`

#
# Determine if the user has specified a file name
# for the resulting makefile.  If not, default to
# the name "makefile".
#

if [ -z "$MAKEFILE" ]
then
        MAKEFILE=makefile
fi

#
# Determine the scope of the directory structure by
# finding the mk.top file.  This is accomplished by
# finding the directory between the current directory
# and the login directory that contains a file named
# "mk.top".  Also set the TOP macro to the relative
```

```
# location of the directory in which the mk.top file
# was found.
#

while [ $FOUNDFLG = FALSE ]
do
        #
        # If the current directory is not the same as
        # the login directory
        #

        if [ $CURDIR != $LOGIN ]
        then

                #
                # If the mk.top file is in the current
                # directory, set the TOP macro.
                #

                if [ -f "mk.top" ]
                then
                        if [ -z "$TOP" ]
                        then
                                TOP=`pwd`
                        fi

                        FOUNDFLG=TRUE

                        break

                #
                # If the mk.top file is not in the current
                # directory, change back one directory and
                # add to the relative path in the TOP
                # macro.
                #

                else
                        cd ..
                        CURDIR=`pwd`

                        if [ -z "$TOP" ]
                        then
                                TOP="../$TOP"
                        else
```

```
                        TOP=$TOP/..
                    fi
                fi

        #
        # The current directory is the login directory.
        # If it contains a "mk.top" file, set the TOP
        # macro.
        #

        else

                if [ -f "mk.top" ]
                then
                        if [ -z "$TOP" ]
                        then
                                TOP=`pwd`
                        fi

                        FOUNDFLG=TRUE
                        break
                else
                        break
                fi
        fi
done

#
# If a mk.top file has not been located anywhere between
# the current directory and the login directory, issure
# an error message and stop processing.  Otherwise, set
# the MK_TOP macro.
#

if [ $FOUNDFLG = FALSE ]
then
  echo "ERROR:  a mk.top input file cannot be located"
  exit 1
else
        MK_TOP=${TOP}/mk.top
fi

#
# Get back to the user's working directory
#
```

```
cd $SAVDIR

#
# Determine the type of the current directory (leaf or
# non-leaf).  A leaf directory must contain a mk.leaf
# file and a non-leaf directory must contain a mk.nonleaf
# file.
#

#
# If both the mk.leaf and a mk.nonleaf files exist in the
# current directory, issue an error message and stop
# processing.
#

if [ -f "mk.leaf" -a -f "mk.nonleaf" ]
then
        echo "ERROR:  too many input files in $CURDIR"
        exit 1
fi

#
# If the current directory contains a mk.leaf file, it is
# a leaf directory.  Therefore, call genleaf to generate
# the leaf level makefile.  (NOTE:  genleaf calls findsrc
# and findsrc then calls findinc.)
#

if [ -f "mk.leaf" ]
then

        #
        # Initialize the temporary file that contains the
        # list of files to be included to create a product.
        # This file will be reinitialized and started over
        # for each product in the product list.  It's
        # contents will be appended to the temporary
        # file of make instructions.
        #

        > inclist

        #
        # Gather all the pieces and put them together in a
        # standard format in the resulting makefile.  The
```

```
# standard format is:  standard macros, mk.top
# file, mk.leaf file, and generated instructions.
#

genleaf $MK_TOP

#
# If genleaf exited with an error, then exit
# this module with an error.
#

RET=$?
if [ $RET -eq 1 ]
then
        exit 1
fi

#
# Print the resulting makefile in the following
# format:  standard macros for leaf level,
# the mk.top file, the mk.leaf file, and the
# generated make instructions.
#

echo "##########" > $MAKEFILE
echo "#" >> $MAKEFILE
echo "# Standard Macros" >> $MAKEFILE
echo "#" >> $MAKEFILE
echo "##########" >> $MAKEFILE
echo >> $MAKEFILE
echo "TOP = $TOP" >> $MAKEFILE
echo "MK_TOP = $MK_TOP" >> $MAKEFILE
echo "CC = cc" >> $MAKEFILE
echo "CFLAGS = -O" >> $MAKEFILE
echo "DEFS = " >> $MAKEFILE
echo "CC_CMD = $(CC) -c $(CFLAGS)" >> $MAKEFILE
echo "CCLD_CMD = $(CC) $(LDFLAGS)" >> $MAKEFILE
echo "LD = ld" >> $MAKEFILE
echo "LDFLAGS = -o" >> $MAKEFILE
echo "LD_CMD = $(LD) $(LDFLAGS)" >> $MAKEFILE
echo "TARGET = all" >> $MAKEFILE
echo >> $MAKEFILE

#
# If the inclist temporary file has something in
```

```
        # it, sort the contents, edit the file to remove
        # the '\' on the last line, and then add its
        # contents to the INC_LIST macro in the resulting
        # makefile.
        #

        if [ -s inclist ]
        then
                echo >> $MAKEFILE
                echo "INC_LIST=\" >> $MAKEFILE
                sort -u -o inclist inclist

ed - inclist > /dev/null <<!
$
s/.$//
w
q
!

                cat inclist >> $MAKEFILE
                rm inclist
                echo >> $MAKEFILE
        fi

        echo "all:" >> $MAKEFILE
        echo "install:     all" >> $MAKEFILE
        echo >> $MAKEFILE
        echo >> $MAKEFILE
        echo "##########" >> $MAKEFILE
        echo "#" >> $MAKEFILE
        echo "# mk.top" >> $MAKEFILE
        echo "#" >> $MAKEFILE
        echo "##########" >> $MAKEFILE
        echo >> $MAKEFILE

        cat $MK_TOP >> $MAKEFILE

        echo >> $MAKEFILE
        echo >> $MAKEFILE
        echo "##########" >> $MAKEFILE
        echo "#" >> $MAKEFILE
        echo "# mk.leaf" >> $MAKEFILE
        echo "#" >> $MAKEFILE
        ec  > "##########" >> $MAKEFILE
        echo >> $MAKEFILE
```

```
        cat mk.leaf >> $MAKEFILE

        echo >> $MAKEFILE
        echo >> $MAKEFILE
        echo "###########" >> $MAKEFILE
        echo "#" >> $MAKEFILE
        echo "# Instructions for making executables" >> \
                $MAKEFILE
        echo "#" >> $MAKEFILE
        echo "###########" >> $MAKEFILE
        echo >> $MAKEFILE

        cat tmpmake >> $MAKEFILE
        rm tmpmake

#
# If the current directory contains a mk.nonlead file,
# it is a non-leaf directory.  Therefore, call gennonleaf
# to generate the non-leaf level makefile.
#

elif [ -f "mk.nonleaf" ]
then

        #
        # Gather all the pieces and put them together in
        # a standard format in the resulting makefile.
        # The standard format is:  standard macros,
        # mk.top file, mk.nonleaf file, and generated
        # instructions.
        #

        gennonleaf

        #
        # If gennonleaf exited with an error, then exit
        # this module with an error.
        #

        RET=S?
        if [ $RET -eq 1 ]
        then
                exit 1
        fi
```

```
#
# Print the resulting makefile in the following
# format:  standard macros for non-leaf level,
# the mk.top file, the mk.nonleaf file, and the
# generated make instructions.
#

echo "##########" > $MAKEFILE
echo "#" >> $MAKEFILE
echo "# Standard Macros" >> $MAKEFILE
echo "#" >> $MAKEFILE
echo "##########" >> $MAKEFILE
echo >> $MAKEFILE
echo "TOP = $TOP" >> $MAKEFILE
echo "MK_TOP = $MK_TOP" >> $MAKEFILE
echo "MAKE_CMD = make \$(MKFLAGS) \$(MKAIDENV) \
        \$(AIDENV)" >> $MAKEFILE
echo "MKAIDENV = \$(MKFLAGS) \$(TARGET) \
        \$(AIDENV)" >> $MAKEFILE
echo "TARGET = all" >> $MAKEFILE
echo >> $MAKEFILE
echo "all:" >> $MAKEFILE
echo >> $MAKEFILE
echo >> $MAKEFILE

echo "##########" >> $MAKEFILE
echo "#" >> $MAKEFILE
echo "# mk.top" >> $MAKEFILE
echo "#" >> $MAKEFILE
echo "##########" >> $MAKEFILE
echo >> $MAKEFILE

cat $MK_TOP >> $MAKEFILE

echo >> $MAKEFILE
echo >> $MAKEFILE
echo "##########" >> $MAKEFILE
echo "#" >> $MAKEFILE
echo "# mk.nonleaf" >> $MAKEFILE
echo "#" >> $MAKEFILE
echo "##########" >> $MAKEFILE
echo >> $MAKEFILE

cat mk.nonleaf >> $MAKEFILE
```

```
        echo >> $MAKEFILE
        echo >> $MAKEFILE
        echo "##########" >> $MAKEFILE
        echo "#" >> $MAKEFILE
        echo "# Instructions for making sub-directories" \
                >> $MAKEFILE
        echo "#" >> $MAKEFILE
        echo "##########" >> $MAKEFILE
        echo >> $MAKEFILE

        cat tmpmake >> $MAKEFILE
        rm tmpmake

else

        #
        # If neither a mk.leaf or mk.nonleaf files exists
        # in the current directory, issue an error message
        # and stop processing.
        #

        echo "ERROR: no input files in $CURDIR"
        exit 1
fi

exit 0
```

```
#####################
#
# NAME: gennonleaf
#
# INPUTS: none
#
# OUTPUTS: a temporary file of generated make
#          instructions called tmpmake.
#
#####################

#
# Get the PRODUCTS line out of the mk.nonleaf file.
# Strip off the word "PRODUCTS" and the equal sign,
# leaving a blank separated list of directories.
#

PRODS=`grep PRODUCTS mk.nonleaf | sed -e "s/PRODUCTS//" \
        -e "s/=//"`

#
# Initialize the temporary file of generated make
# instructions by setting the all target.
#

echo "all:          $PRODS" > tmpmake
echo >> tmpmake

#
# Check each directory listed on the PRODUCTS line to
# see if it really is a directory under the current
# directory. If is isn't, issure an error message and
# stop processing. If it is, create a dummy target for
# it, make the dummay target depend on nothing (so the
# directory will be made everytime), and then create a
# one line instruction that will make the directory.
# The one line instruction will cause the make command
# to save the name of the current directory, echo a
# message that specifies which directory is being made,
# changes to the directory, and executes the make
# command (this will accomplish a recursive execution
# of make).
#

for prod in $PRODS
```

```
do
        if [ ! -d $prod ]
        then
                echo "$prod is not a directory"
                exit 1
        else
                echo "$prod:      d$prod" >> tmpmake
                echo "d$prod:" >> tmpmake
                echo "   @d=\`pwd\`; \\" >> tmpmake
                echo "   echo \"Making \$\$d/$prod\"; \
                        \\" >> tmpmake
                echo "   cd $prod; exec \$(MAKE_CMD) \
                        \$(TARGET)" >> tmpmake
                echo >> tmpmake
        fi
done

exit 0
```

```
#####################
#
# Name:  genleaf
#
# INPUTS:  MK_TOP - path to the mk.top file
#
# OUTPUTS:  a temporary file that contains make
#           instructions.
#
#####################

MK_TOP=$1


#
# Get the PRODUCTS line out of the mk.leaf file.
# Strip off the word "PRODUCTS" and the equal sign,
# leaving a blank separated list of products that are
# to be made.
#

PRODS=`grep PRODUCTS mk.leaf | sed -e "s/PRODUCTS//" \
        -e "s/=//"`

#
# Initialize the temporary file of make instructions
# by setting the all target.
#

echo "all:          $PRODS" > tmpmake
echo >> tmpmake

#
# Check if each of the products in the products list
# has been specified as a target in the mk.leaf file.
# Also strip off the product name and the ":", leaving
# a space separated list of dependencies.  If it hasn't
# issue an error message and stop processing.
#

for prod in $PRODS
do
        DEPS=`grep $prod: mk.leaf | sed -e "s/$prod://"`

        if [ -z "$DEPS" ]
```

```
then
        echo "ERROR: product not specified as a \
              targetg
        rm deplist
        exit 1
fi

#
# Check each dependency to see if it is a ".o"
# file.  If it isn't, skip it.
#

for prod in $DEPS
do

  if [ `echo $prod | grep "\.o"` ]
  then

        #
        # If it is, set a variable to the same
        # file name with the ".o" replaced with
        # a ".c".
        #

        DOT_C=`echo $prod | sed -e "s/\.o/\.c/"`

        #
        # Call findsrc with the ".c" file name.
        # Findsrc will determine the location
        # of the ".c" file, get all the included
        # files from the ".c" file, and then
        # call findinc to determine if there are
        # any files included in the included
        # files.
        #

        findsrc $DOT_C $MK_TOP

        #
        # If findsrc exited with an error, then
        # exit this module with an error.
        #

        RET=$?
        if [ $RET -eq 1 ]
```

```
                then
                        exit 1
                fi

                #
                # Uniquely sort the temporary dependency
                # list file that was created by findinc
                # and initialize the temporary file with
                # the results.  Then append the $(CC_CMD)
                # macro line to the temporary file.
                #

                sort -u deplist >> tmpmake
                rm deplist
                echo "   \$(CC_CMD) $DOT_C" >> tmpmake
                echo >> tmpmake
        fi
    done
done

exit 0
```

```
#####################
#
# Name:  findsrc
#
# INPUTS:  There are two inputs:
#
#         $1 is the name of the ".c" file
#
#         $2 is the location of the mk.top file
#
# OUTPUTS:  deplist - a temporary file which is
#                     initalized to the ".o":".c"
#                     target:dependency pair
#
#####################

DOT_C=$1
MK_TOP=$2

FOUND=FALSE
LEVEL=1

#
# Check if the ".c" files is in the current directory.
#

if [ -f $DOT_C ]
then

        #
        # Current directory contains the ".c" file.
        # Determine the name of the ".o" file and
        # initialize the temporary dependency list
        # file with the ".o":".c" target:dependency
        # pair.
        #

        FOUND=TRUE

        DOT_O=`echo $DOT_C | sed -e "s/\.c/\.o/"`
        echo  "$DOT_O:$DOT_C" > deplist
else
        #
        # Current directory does not contain ".c" file.
        # Get the SEARCH line from the mk.leaf or mk.top
```

```
        # file.  Remove the word SEARCH and the equal sign,
        # leaving a space separated list of directories.
        # Then check each directory to see if the ".c"
        # file is in the directory.  If it is, add the
        # ".o":".c" target:dependency pair to the
        # temporary dependency list file.
        #

        SRCHPATH=`grep SEARCH mk.leaf | sed -e \
                "s/SEARCH//" -e "s/=//"`

        if [ "$SRCHPATH" ]
        then
                SRCHPATH=`grep SEARCH $MK_TOP | sed -e \
                        "s/SEARCH//" -e "s/=//"`
        fi

        for DIR in $SRCHPATH
        do
                if [ -f $DIR/$DOT_C ]
                then
                        FOUND=TRUE
                        echo "DOT_O:   $DIR/$DOT_C" \
                                > deplist
                        break
                fi
        done
fi

#
# If the ".c" file wasn't found in any of the directories,
# issure an error message, remove the dependency list
# temporary file and stop processing.
#

if [ $FOUND = FALSE ]
then
        echo "ERROR: $DOT_C cannot be located"
        rm deplist
        exit 1
fi

#
# The ".c" file was found.  Determine the files that the
# ".c" includes.
```

```
#

INCS=`grep '^#include" $DOT_C | sed -e "s/\#include//"
        -e "s/\"//g" -e "s/\<//" -e "s/\>//"`

#
# If there are some files in the list, call findinc
# to locate each of the files.
#

if [ -n "$INCS" ]
then
        for INC in $INCS
        do
                findinc $INC $DOT_O $MK_TOP $LEVEL

                #
                # If there was an error in findinc,
                # then exit with modules with an error.
                #

                RET=$?
                if [ $RET -eq 1 ]
                then
                        exit 1
                fi
        done
fi

exit 0
```

```
#####################
#
# Name:  findinc
#
# INPUTS:  There are three inputs:
#
#         $1 is the name of the include file to be found
#
#         $2 is the name of the ".o" file that will
#                 depend on the include file
#
#         $3 is location of the mk.top file
#
# OUTPUTS:  two temporary files are created:
#
#             deplist - contains the target:depency pairs
#                 for each ".o"
#
#             inclist - contains the list of directories
#                 in which include files were actually
#                 located
#
#####################

FILE=$1
DOT_O=$2
MK_TOP=$3
LEVEL=$4

MAXLEVEL=7
FOUND=FALSE

#
# findinc is recursive so it is possible that a circular
# include could put it in an endless loop.  Therefore,
# check the number of times findinc has called itself.
# This also checks that the include files are not nested
# too deep.  If the number of levels becomes too many,
# issue an error message, remove the temporary files,
# and stop processing.
#

if [ $LEVEL -gt $MAXLEVEL ]
then
        echo "ERROR: Possible circular include or files \
```

```
                nested too many levels"
        rm tmpmake
        rm inclist
        rm deplist
        exit 1
fi

#
# Increment the number of times findinc has been called
#

LEVEL=`expr $LEVEL + 1`

#
# If the include line ($FILE) is a full path name,
# use it just the way it is.
#

FIRST=`echo $FILE | cut -c1`
if [ $FIRST = '/' ]
then
        if [ -f $FILE ]
        then
          FOUND=TRUE
          echo "-I${FILE}\\" >> inclist
          echo "$DOT_O:         ${FILE}" >> deplist
          DEPS=`grep "^#include" $DIR/$FILE | sed \
                -e "s/\#include//" -e "s/\"//g" \
                -e "s/\</>//" -e "s/\>//"`
          if [ -n "$DEPS" ]
          then
                for DEP in $DEPS
                do
                  findinc $DEP $DOT_O $MK_TOP $LEVEL

                  #
                  # If there was an errer in findinc,
                  # then exit this module with an
                  # error.
                  #

                  RET=$?
                  if [ $RET -eq 1 ]
                  then
                        exit 1
```

```
                fi
            done
        fi
    fi
else
    #
    # The directories to be searched for an included
    # file is a concatenation of the current
    # directory, the directories listed in SEARCH,
    # the directories listed in INC, and the directories
    # listed in UINC. SEARCH, INC and UINC can be set in
    # either the mk.top or the mk.leaf file.  If they are
    # set in # both files, the mk.leaf has presidence.
    # The /usr/inclue directory will be added to UINC
    # whether it the user has set it or not.
    #

    #
    # Add the current directory to the ALLPATH list
    #

    ALLPATH=`pwd`

    #
    # Add the SEARCH directories to the ALLPATH list
    #

    SEARCH=`grep SEARCH mk.leaf | sed -e \
            "s/SEARCH//" -e "s/=//"`

    if [ -z "$SEARCH" ]
    then
            SEARCH=`grep SEARCH $MK_TOP | sed -e \
                    "s/SEARCH//" \
                    -e "s/=//"`
    fi

    #
    # Add the INC directories to the ALLPATH list
    #

    INCPATH=`grep INC mk.leaf | sed -e 's/INC//' \
            -e 's/=//'`

    if [ -z "$INCPATH" ]
```

```
then
        INCPATH=`grep INC $MK_TOP | sed -e \
                's/INC//' -e \
                's/=//'`
fi

#
# Add the UINC directories to the ALLPATH list
#

UINCPATH=`grep UINC mk.leaf | sed -e "s/UINC//" \
        -e "s/=//"`

if [ -z "$UINCPATH" ]
then
        UINCPATH=`grep UINC $MK_TOP | sed -e \
                "s/UINC//" -e \
                "s/=//"`
fi

UINCPATH="$UINCPATH /usr/include"

ALLPATH="${ALLPATH} ${SEARCH} ${INCPATH} \
        ${UINCPATH}"

#
# Determine the location of an include file by
# looking for the file in each directory listed
# in the concatenated ALLPATH.
#


for DIR in $ALLPATH
do
        #
        # If an include files is found, add it to
        # the temporary include list file, add the
        # ".o":include file target:dependency pair
        # to the temporary dependency list file,
        # and check to see if the include file
        # includes other files.  If it does,
        # recursively call findinc with the new
        # dependency list.
        #
```

```
                if [ -f $DIR/$FILE ]
                then
                        FOUND=TRUE
                        echo "    -I${DIR}\\" >> inclist
                        echo "$DOT_O: $DIR/$FILE" >> \
                                deplist
                        DEPS='grep '^#include" $DIR/$FILE \
                                | sed -e "s/\#include//" \
                                -e "s/\"//g" -e "s/\<//" \
                                -e "s/\>//"'
                        if [ -n "$DEPS" ]
                        then
                          for DEP in $DEPS
                          do
                            findinc $DEP $DOT_O \
                                $MK_TOP $LEVEL

                            #
                            # If there was an error in
                            # findinc, then exit this
                            # module with an error.
                            #

                            RET=$?
                            if [ $RET -eq 1 ]
                            then
                                exit 1
                            fi
                        done
                        fi
                        break
                fi
        done
fi

#
# If an include file was not found in any of the
# directories in ALLPATH, issue an error message,
# remove the tmporary files and stop processing.
#

if [ $FOUND = FALSE ]
then
        echo "ERROR: Include file $FILE not found in \
                search path"
```

```
        rm deplist
        rm tmpmake
        rm inclist
        exit 1
else
        exit 0
fi
```

DESIGN OF A MINIMAL TOOL SET FOR A SOFTWARE DEVELOPMENT ENVIRONMENT UNDER UNIX

by

Ruth A. Pfaffmann

B. S., Western Illinois University, 1967
B. A., North Central College, 1982

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

ABSTRACT

This report proposes a minimal tools set that will provide a software development environment under the UNIX Operating System. The objective of the software development environment is to provide a tool set that will assist in the deliver of high quality software systems, improve the staff's productivity, and aid in the effective use of resources. Each tool in the proposed tool set is described along with the problems that the tool will solve. Recommendations about the order of acquisition are also made.

Included in the report is the design and implementation of a tool that will aid in the administration of projects. This tool is a makefile writing aid call *mkaid*. The makefile writing aid enforces good programming practices, but still allows the user a great deal of flexibility. One benefit is that developers need not have intimate knowledge of the organization of the system being developed. The *mkaid* system will generate target:dependency pairs correctly, identify circular includes or too many nested includes, and produce resulting makefiles with uniform formats.