A SOFTWARE DEVELOPMENT SUPPORT SYSTEM FOR A MICROCOMPUTER ENVIRONMENT

By

JOHN CHARLES WARREN

Bachelor of Music Education
University of Oklahoma
Norman, Oklahoma
1975

Bachelor of Science
University of Oklahoma
Norman, Oklahoma
1976

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fullfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1983

A SOFTWARE DEVELOPMENT SUPPORT SYSTEM FOR A MICROCOMPUTER ENVIRONMENT

Thesis Approved:

Sharilyn A. Thoreson
Thesis Adviser

Michael J. Folk

D. E. Hedrick

Norman N. Durham
Dean of the Graduate College

PREFACE

This study examines some existing software development support systems with the intent of adapting some of the techniques found to a microcomputer software development environment. The motivation for doing this study arose from experiences gained as a part of a microcomputer software development project.

During the course of this project, a number of very annoying events plagued us. Parallel updates and modifications to out-of-date copies of source code necessitated the redoing of work previously thought to be finished. In addition, these events caused much confusion. Another difficulty which was encountered was that on occasion, changes were made which were not propagated into the executable system. As the project grew more complex, the process of linking the system also grew to be a burdensome chore. Procedures were established to deal with these problems as they arose; however, these procedures depended on fallible humans, and mistakes still occurred far too often.

This paper looks at what has been done in regard to these problems and presents an adaptation of some of the previous work to the type of environment in which these experiences took place. The result is intended as a

specification for a system to deal with a number of the above-mentioned difficulties automatically. I hope that this work will serve as a guideline for anyone interested in producing such a microcomputer software development support system.

I would like to express my gratitude to my major adviser, Dr. Sharilyn A. Thoreson, for her valuable guidance and advice throughout my thesis research. I also acknowledge my other committee members, Dr. George E. Hedrick and Dr. Michael J. Folk, for their many helpful suggestions. My appreciation also goes to my supervisor on the above-mentioned project, Mr. Arlen Long, for numerous ideas, encouragement, and a healthy dose of good humor.

I would like to thank the Department of Computing and Information Sciences for financial support during my graduate studies. I also gratefully acknowledge financial support received from the AMOCO Foundation. In addition, I am extremely indebted to my wife's parents and grandparents, Mr. and Mrs. Bill South, Mr. and Mrs. Joe Robinson and Mrs. Ruby South, for their generous support during this time.

The friendship of fellow students has meant a great deal to me, especially that of Mr. Jack Lucas. The faithful support given by my parents, Mr. and Mrs. James D. Warren, has been a constant source of encouragement to me. Finally, I would like to express my appreciation to my wife for her love, patience, understanding, and most of all, just for being there.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Motivation

The need for programming has existed since the inception of the electronic computer. In the years since the introduction of the first computer, there has been significant progress both in making the programming task easier and in increasing the productivity of programmers. The initial laborious production of machine language in absolute form soon gave way to the use of assemblers, macro assemblers and linkers to produce executable machine code. Operating systems assumed the burdens of input and output operations (along with many other tasks). High level languages have allowed yet another step away from the intricate details of machine-level coding. Punched cards and batch programming systems are being replaced by online text editing and interactive programming systems. Tools such as full screen editors, debugging compilers, and interactive debugging systems all aid in furthering the timely development of a functionally correct program.

However, despite this progress, the production of a software product is often more costly and time-consuming

than it might be. Early awareness of the difficulties of producing a large-scale software system came from both governmment and commercial operations. One notable learning experience was the production of IBM's OS/360 operating system. Brooks [5] reports both the problems encountered and some of the lessons learned.

Many have contrasted the remarkable decline in hardware costs with the lack of such a decline in software costs. Regarding hardware and software embedded in weapons systems, Stuebing [31] says:

> There is an optimism concerning hardware system costs because technological breakthroughs are continually reducing the component costs. However, the recurring software problems of late deliveries, poor quality, and especially increasing life-cycle costs have created a somewhat pessimistic attitude regarding software (p. 408).

The term "software crisis" has been used in connection with the current state of affairs in the software industry. Gillett and Pollack [14] have this to say about the situation:

> The present level of concern about the software crisis does not mean that there was a better time, a 'golden age' for programming, programmers, and programs. We have always had problems with software, but most of these eventually were removed or reduced to minor irritants. . . . Now the situation has changed dramatically. The tremendous increase in the amount of 'computing power' that can be bought for a given amount of money has not been paralleled by anything even remotely similar in the way of software productivity. In many instances the software trend is going in the opposite direction. Consequently, the fraction of a system's cost attributable to software has been rising steadily. For example, the U.S. Department of Defense figured that its software costs in the early 1960s

> were about 20 percent of its total computer system costs. In the 1980s the fraction is expected to exceed 90 percent. Findings in a variety of industries and businesses indicate that this is not peculiar to the Defense Department (p. 3).

Because of these factors, much attention is currently directed at means by which more useful software can be provided with the available resources. Of course a part of the means to this end is to enhance the productivity of programmers. The need for a means of producing reliable software at a reasonable cost cuts across all dividing lines in the computer industry. This topic is relevant for mainframe systems, minicomputers and micros; for real time systems as well as data processing systems; and for systems programming as well as applications programming.

Because of the broad scope of this issue, no single solution will work in every situation. Work which has been done until now has for the most part been directed at development activities which take place on the larger systems. However, the power and increasing usage of relatively low-cost microcomputers is bringing about a corresponding need for suitable software for such machines. This paper is oriented towards support for the development of software for microcomputer systems.

Problems in Software Production

When the computer solution to a particular problem is small enough to be written by an individual, there is little difficulty in understanding the total program product being

produced. There are no surprises for the programmer when certain aspects of the system are forced to evolve--he knows the program well enough that ramifications of changes are apparent and can be considered before modifications are made. Even in the initial phases of development (the requirements and design phases) the fact that this is a small, one-man project comes into play. It may be that the program is being written for the programmer's personal use, in which case there is no need for the user-designer interaction necessary for requirements specification. There is no need for communication between designers and implementors concerning modular functions, interfaces, and operation of the system. The programmer himself determines these things.

However, a medium to large size project is another matter. The project may involve a few people or it may involve a few hundred, but while the potential problems differ in magnitude, the nature of those problems remains essentially the same. Of course, some of the problems may be technical problems concerned with the actual process of defining the operation of the system and implementing it. These types of problems are as relevant for the one-man project as they are for the team project. But other problems arise from the dynamics of the group situation. These problems may be either communications difficulties or management difficulties.

Communication must take place in a number of different ways. The requirements must be clearly communicated by the user to the system designer. If there is a breakdown here, the programming team may find that they have done a superb job in solving the wrong problem. The designer (or design team) must in turn communicate with the programmers who will actually implement the system. To the degree that their work interrelates, the programmers must communicate among themselves during the actual implementation process. A potential problem if this intercommunication does not occur is that one programmer may make an intentional modification to code which has unintended side effects. The side effects may arise because of conflicting use of common data or because of some other miscommunication between programmers. When errors in the evolving system are discovered, the testers must communicate with the programmers. Probably most important is the communication of the development team with those who will use and maintain the system by means of documentation.

Management tasks are many and varied. Just a few of the areas in which problems may arise are mentioned here. One area is that of coordinating programmers' activities, i.e., ensuring that all necessary tasks are covered with a minumum of overlap. A second is the management of the code itself. One particular need in this regard is to avoid the possibility of "parallel" updates to source code. A parallel update occurs when two programmers modify distinct

copies of the same module at the same time, resulting in two parallel versions. The two versions cannot both be incorporated into the developing system. Another part of the management of code is version control and configuration management. A third area in which potential problems may arise is the management of error reports. When a deficiency is discovered in the developing system, there must be some means of guaranteeing that the deficiency is corrected. A fourth area is management of the integration task. When modifications are made to source code, action must be taken to incorporate these changes into the executable system. If this does not take place, the source and the executable code becomes inconsistent. Such inconsistency can cause a great deal of confusion.

An interesting aspect of computers is that they themselves function under the control of automated tools to make their own use easier for humans. This has been described as a "bootstrapping" operation [18]. Assemblers, compilers, text editors, operating systems, and interactive debugging systems are all examples of such automated tools. Similarly, many of the problems mentioned above can be solved or at least alleviated by appropriate automated tools. Recently the focus has been shifted away from the use of individual software tools to the use of integrated programming support environments. An integrated environment takes a more comprehensive approach to the task of software development and provides facilities to support the entire

software life-cycle. In the survey that follows, both collections of tools and true integrated programming environments are examined.

Survey of the Literature

A number of different approaches to improved programming environments are found in the literature. The presentation of the systems is based on the type of help that is provided.

Support for Specification and Design

The Information System Design and Optimization System (ISDOS) described by Teichroew and Hershey [32] consists of two parts. The first is the Problem Statement Language (PSL). The second is the Problem Statement Analyzer (PSA). PSL/PSA is an automated system which supports the requirements specification phase of software development. It is specifically targeted for information processing systems. A proposed system is described with the problem statement language. The problem statement analyzer records this description in a database and on demand can perform analysis and produce various reports. The information recorded contains not only data about the system being proposed, but also project management descriptions as well. Reports include a record of changes that have been made to the specifications, properties of particular "objects" of the system, system hierarchy, as well as others. PSL/PSA

currently operates on a variety of larger computers, such as the IBM 370, UNIVAC 1100, and CDC 6000/7000 systems. PSL/PSA has been incorporated into more comprehensive systems such as the Design Analysis System [37].

Another system which has been developed specifically to aid in the requirements specification task is called the Requirements Engineering and Validation System or REVS [2]. It, too, includes a specification language called the Requirements Statement Language (RSL). The two other major components of the system are a centralized database (which stores the requirements in relational form) and a set of automated tools for processing information in the database. One tool is a graphics package which allows display and editing of flow path information. This editor provides an alternative to the use of RSL for specification. There are several different static consistency checkers which analyze structure, data flow, and check for proper hierarchy in the specification. In addition, a simulator generator provides the means for dynamic checking of requirements. A final tool is a generalized reporting system which provides for both user-defined special reports and ad-hoc inquiries of the database. REVS is implemented on a Texas Instruments ASC computer.

A somewhat different approach is reported by Lemaitre, Lemoine, and Zanon [23]. The aim of the SPRAC system is "to extend the assistance given by software work bench, such as UNIX-PWB, to the initial phases of program design (p. 333)."

The system is restricted to the design of tranlators, compilers, and interpreters. The underlying idea of SPRAC is to give the designer "active assistance" in the design process. This is possible because of a well-developed knowledge base in the field of compiler design.

Support for Program Implementation

Several systems exist which provide support for the implementation process beyond that normally available via stand-alone editors and compilers. The Programmer's Apprentice (PA) is one such system [36]. A unique type of program representation called a plan provides the basis for the programmer's apprentice. A plan is a graphical representation of a program which shows both control flow and data flow by explicit arcs. There are five parts to the system: an analyzer, a coder, a drawer, a library of plans, and a plan editor. The analyzer operates on an existing FORTRAN, COBOL or LISP program and produces the corresponding plan. The coder generates a LISP program from a plan. The drawer produces a graphic representation of a plan. The library of plans contains common algorithmic fragments which can be combined and modified to produce a new program. Finally, the plan editor allow a program to be modified by modifying its plan. The greatest advantage of the system is reportedly the ability to quickly build up a program from the algorithmic fragments in the library, edit these, and produce the desired new program.

Another approach is taken by the IDEAL system [28]. IDEAL stands for Interactive Development Environment for an Application's Life-cycle. The design and development facility uses both a high-level design language (Procedure Definition Language) and special purpose fill-in-the-blank forms for data definition, report specification and screen format definition. The support environment includes special purpose editors for the procedure definition language as well as for the various forms used by the system. An integrated database sublanguage provides convenient manipulation capabilities for data objects. A centralized data dictionary is also maintained. This system is intended for the production of data processing business applications.

The Interlisp programming environment is described by Teitelman and Masinter [34]. This system integrates a version of LISP with several very interesting tools. A file package underlies the other portions of the system, keeping track of the location of data within files and noting references and changes to that data. Masterscope performs analysis and cross-referencing of programs and allows interactive querying of the derived information. DWIM (Do What I Mean) attempts to ensure that the system operates reasonably (based on context) when unrecognized input is given it. One facet of DWIM is the spelling corrector. The Programmer's Assistant records a history of the user's interaction with the system and allows the user to REDO, FIX, and UNDO previously performed commands. The user is

able with these facilities to edit, test and debug a developing system interactively.

Another integrated edit/test/debug environment is the Cornell Program Synthesizer [33]. The Synthesizer contains a "smart" editor for a version of PL/I called PL/CS. The editor enforces a top-down approach to program development with its syntax-based approach to editing. Templates corresponding to language constructs are manipulated and placeholders within the templates are filled in as the program takes shape. Semantic checking for problems such as missing declarations is performed during the editing process. Debugging is supported by the use of a visual display as the program is executing, by allowing the user to step through the program execution, and by providing for "reverse execution" (a history of execution makes this possible).

Support for Management

MONSTR (for MONitors Software Trouble Reports) is an error reporting system described by Knobe [20]. Emphasis is placed on the communication flow of error reports within the organization. This flow is controlled by a protocol. The access to the information contained in the reports is also strictly controlled based on the same protocol. The system keeps track of the status of the trouble reports as well. When put in place at the National Software Works, the system reportedly greatly enhanced the ability of managers to

monitor what the programmers were doing. In addition, many programmers indicated that their work environment improved as a result (most had previously been inundated with trouble reports which were redundant, inconsequential, or simply not their responsibility).

A more comprehensive approach to management of the production of software is found in the SAGA system [6]. SAGA uses a management grammar to specify acceptable sequences of programmer action. A similar grammar is used to define a development sequence. An automated system ensures that programmers stay within the quidelines specified by the grammars. The authors suggest this system as a vehicle for experimentation with different management policies.

Integrated Management and Development Support

A number of systems attempt to integrate the total development process, and join the facilities for an improved programming environment with those for management of the project.

A system known as CADES (Computer Aided Development and Evaluation Systems) is one of the earliest attempts at a comprehensive programming environment [26]. CADES was initially developed between 1970 and 1972 by International Computers, Ltd. The approach is based on a methodology called "structural modeling". Emphasis is placed on strict control of the connectivity of a system. A design language

called SDL (System Descriptive Language) is used in a top-down approach to define the software being developed. Consistency checks are made on successive refinements, and when a suitably detailed level of design is reached, high-level language code is automatically generated. The system is organized around a central product database. A more recent version of the system includes tools for configuration management. This latter version runs on ICL's 2900 computers.

The Software Factory is an integrated system developed by the System Development Corporation [4]. There are four facets of the system. The Factory Access and Control Executive (FACE) provides the user with a consistent control interface. The Integrated Management, Project Analysis, and Control Technique (IMPACT) provides project planning and monitoring aids. The third facet is the project development database which is divided into a software development database and a project control database. The latter contains both system and program descriptions as well as management data. The final facet is a set of tools to aid design and development of a system. These tools are: AUTODOC, a so-called "automatic" documentation facility; Program Analysis and Test Host (PATH), which instruments code for program flow analysis; Test Case Generator (TCG); and TOPS, a top-down system developer which provides a design verification facility. The Software Factory has been implemented on an IBM 370 computer system.

The programmer's workbench (PWB) grew from the desire to use the UNIX operating system for program development at Bell Laboratories [3, 12, 21, 29]. Rather than being developed around a specific language or particular methodology, the programmer's workbench is a collection of useful tools for programmers. Besides UNIX itself, facilities include a Remote Job Entry (RJE) system, the Source Code Control System (SCCS), the Modification Request Control System (MRCS), document preparation facilities, and test drivers for program testing and validation. The RJE facility handles job submission to remote computer systems. The Source Code Control System manages source code and maintains a history of changes to that source. Any version of a source file, from the initial version to the most recent, can be recovered by SCCS. The Modification Request Control System provides an online error reporting and tracking mechanism. Another tool available on the UNIX system which is of interest is called MAKE [13]. Although not described as a part of the programmer's workbench in the referenced articles, it interacts with PWB tools and forms a significant addition to the programming environment. MAKE allows the user to specify the sequence of commands needed to rebuild a system after particular changes. MAKE can then determine the correct sequence of actions needed to produce the updated executable form of a system. It will also allow the user to specify which versions of the source modules to use. This in conjunction with SCCS provide powerful configuration management capabilities.

Several systems have been reported in the literature which are either based on the programmer's workbench or are extensions to it. One of these is SMS--the Software Manufacturing System [8]. The primary extension used in SMS is that each source file has a label consisting of name and version number embedded in it. Labels are propagated by the compilers and linkers so that derived files contain the combined labels of all files used in the derivation process. This allows better configuration control and consistency checking. A second system which includes tools that are based on the workbench is called the Communications Software Development Package [24]. This is a JOVIAL support environment which includes tools for text manipulation, documentation production, file management, software implementation, version control, and file revision statistics gathering. A third reported derivation adds a management database along with extensions to SCCS, and provides a standard interface to other tools. This is the Change Control System (CCS) reported by Bauer and Birchall [1]. The three primary extensions to SCCS are (1) to allow multiple views of the system change level; (2) to provide for the management of object code; and (3) to add a finite state change model to control the progress of a change through the system.

A system geared to the development of weapon system software is described by Stuebing [31]. The Facility for Automated Software Production (FASP) is built around a

project database which contains source, object, test cases, interface information, modification histories, and management information. The support software includes a source program librarian, system generators (these generate load tapes for the target computers), software emulators, an automated test analyzer, and software management tools. This last category include tools to produce various management reports such as summaries of cost data, use of host computer resources, and details of FASP operations.

The Gandalf system is described by Haberman [16]. The Gandalf system is a "generic" one which can be used to produce specialized Gandalf environments. A Gandalf environment is tailored to a specific development language. It has tools for (1) incremental program construction; (2) system version control; and (3) project management. The incremental program construction subsystem consists of a language-oriented editor (LOE) and an "incremental" relinker/reloader. The editor, rather than producing a textual representation of a program, produces a syntax tree. This means that there is no need for parsing and the syntax tree can be processed to generate code. The incremental relinker/reloader allows a module to be recompiled and relinked when a breakpoint occurs during program execution. The version control subsystem maintains version and revision number information, records the construction process and automates the task of generating system versions. The project management subsystem tracks development status and controls changes to modules.

A summary of the various types of tools forming the systems reported on is found in Table I. The tools have been grouped into six different categories: specification and design aids, graphical aids, language-specific aids, testing aids, management aids, and other tools.

While all of these tools are very useful, a completely integrated environment containing all or even a large portion of these tools is beyond the scope of this paper. The focus must be narrowed substantially. We choose to examine the area of software management tools, particularly those applicable to implementation and use in a microcomputer environment. There are at least two reasons for this choice. Some relatively sophisticated traditional program development tools already exist for the more common microcomputers (tools such as high-level language compilers, text editors, and interactive debugging systems). Rather than these tools being replaced, they can be augmented by the addition of management tools. Furthermore, in the author's experience, some of the more frustrating problems in a development project arise because of the lack of such management tools.

Capabilities of Existing Tools

The facilities of the existing software management tools are examined in more detail, focusing particularly on those which are applicable to a microcomputer development environment. From these are drawn the ones which are to be included in a development support system.

TABLE I

SUMMARY OF AUTOMATED TOOLS

Tools	Examples
Specification/design aids	
Requirements language	PSL, RSL
Static requirements checking	PSA, REVS
Dynamic requirements checking	REVS
Knowledge-based design aid	SPRAC
Graphical aids	
Graphical representation	REVS, PA
Coding in graphical form	PA
Language-specific aids	
Smart editors	Cornell PS, Gandalf
Program generation	PA, CADES
Testing aids	
Interactive debugging	Interlisp, Cornell PS
Test drivers/generators	Software Factory, PWB
Program flow instrumentation	Software Factory, FASP
Management tools	
Source code management	SCCS, Gandalf
Error report tracking	MRCS, MONSTR
Automated system generators	MAKE, FASP, Gandalf
Management policy automation	SAGA
Other	
Documentation aids	Software Factory, PWB

Source Code Management

The Source Code Control System has three main functions. Source code is stored and managed by SCCS, and all changes to the source code are monitored by the system. This provides the means to eliminate parallel updates. In addition, all versions of each source file are implicitly stored and can be reconstructed by the application of the proper sequence of deltas or changes. Similar functions are found in the source management capabilities of Gandalf.

Storage space is substantially more limited on a microcomputer system than it is on a larger system. While the capability to maintain all versions of a module is undoubtably nice, most of those multiple versions are not useful at all. Perhaps only two or three actually form part of any production configuration. Consequently, this is one capability which is not included in the development support system. However, management of source code and monitoring of changes is provided.

Error Report Tracking

The Modification Request Control System provides the ability to maintain error report information as an online data processing task. In addition, it provides a means of tracking progress in correcting the problems. A third aspect of the system gives the ability to classify the error

reports as to severity and type of action to be taken. The system also allows for the simple, straightforward generation of various summary reports. An additional function is found in the MONSTR system. MONSTR provides precise control of the communication flow of error reports.

The size of a software development group developing a product for a microcomputer will probably be relatively small. While communication is essential no matter what the size of the group, automated control seems superfluous in a group of only a few people. This is why we choose to include no facility such as that provided by MONSTR. The other types of error report tracking capabilities are included.

Automated System Generators

The MAKE system provides several functions. Information previously recorded is used to allow the system to issue the commands needed to rebuild the executable product when requested to do so. A macro substitution facility is provided to allow additional flexibility. In conjunction with SCCS, MAKE allows the generation of multiple versions of the system being developed. This combination of SCCS and MAKE provide configuration management facilities. Similar features are found in the Gandalf version control subsystem.

There are two primary benefits expected from the automatic system generation facility. One is simply to

relieve the programmers of the burdensome task of relinking the system following modifications. The other is to ensure consistency between source and executable code. These benefits both follow from the ability to automatically issue commands needed to rebuild the product, which we include in our system.

The provision for producing multiple executable versions cannot be included because of the lack of such provision in source management. The macro substitution facility is not included, either.

Management Policy Automation

The SAGA system uses a formal management specification along with an automated enforcement mechanism to ensure that only "valid sequences" of activities can occur. The formal specification of management policies and procedures is an excellent idea. However, in the opinion of the author, the rigid enforcement of a set of management procedures indicates a dictatorial and somewhat untrusting approach towards team members. Such a scheme does not form a part of this system.

The balance of this paper presents a detailed description of a system which contains the capabilities indicated above. This is the Software Management System. Chapter two contains an overview of the system, along with a statement of the underlying assumptions made. Chapters three, four, and five each contain details concerning one of

the three major components of the system. A final chapter summarizes the discussion and presents suggestions for future work.

CHAPTER II

SYSTEM OVERVIEW

Approach to the Development Task

As mentioned previously, the Software Management System presented in this paper is intended to support the development of software for microcomputer systems. There are (at least) two possible approaches to the development task and thus two approaches to providing software management tools for the development team.

The first approach is to perform the bulk of the development work on a larger timesharing computer system and download the end result to the target machine. Assuming that an appropriate choice is made for the larger system, existing software management tools could be used. If, for instance, the development team works on a minicomputer with the UNIX operating system and the Programmer's Workbench tools, they would then be able to use the Source Code Control System to manage the source code, and to use the Modification Request Control System to monitor change requests. However, were the team to download the source code to the microcomputer and compile and link on the smaller machine, an important ability would be lost. They

could no longer insure consistency between source files and executable image by automatically propagating changes. To overcome this shortcoming, the team would need to produce or otherwise obtain a special purpose cross-compiler and linker which would execute on the larger system but produce code for the microcomputer. Then this executable image could be moved to the smaller machine for testing. MAKE would provide the facilities to automate the system rebuilding and integration tasks.

The second approach is to do the development work at independent microcomputer workstations. These workstations would either be identical to or compatible with the target microcomputer system. In this case, software management tools must be produced which will execute on these machines.

This second approach has several advantages. Access to the larger system could be provided either by purchasing the system or by buying time on a commercial timesharing system. In using the workstation approach there is no need for the relatively large capital investment required to purchase such a system, and no extra costs associated with its operation. The alternative of purchase of time on a remote computer implies that proprietary information (i.e. the source code of the program product) would have to be stored at the remote site with no direct control over it. This increased security risk is eliminated by using local microcomputer systems. The needs for specialized cross-compilers and linkers and for communications links between

the development system and the target system are also eliminated by this choice.

In addition, Gutz et al. [15] mention several other advantages of a local workstation as compared to a timesharing system. Among these advantages are 1) improved reliability, 2) improved performance, and 3) private storage. Reliability is improved because the multiple microcomputers provide redundant capability so that productive work can proceed even if one machine is inoperable. With a centralized approach to development, failure of the primary system would essentally bring work to a halt. Performance is improved, or at least made more predictable, because response time is not dependent on the amount of other work being performed simultaneously. The private storage enhances security and makes the use of experimental versions of programs more feasible.

The main disadvantage of the workstation approach arises because the microcomputers are single-user systems. This means that shared data cannot be accessed by more than one team member at a time, and that time may be lost because of waiting. This does not seem to be a serious drawback as long as the team remains relatively small. Because the advantages seem to outweigh the disadvantages, the multiple workstation environment is the basis of the Software Management System.

Additional Environment Assumptions

The author's experiences that have motivated this study took place in the context of a small programming team, developing a fairly complex application system for the microcomputer market. Much of what is said is based on that team environment and on the characteristics of the small, single-user systems which were used in that project. While the system is intended for a similar environment, projects with differing characteristics could also benefit from its use. A single-programmer project could profit from the automation of tasks which is provided. A project developed in a multi-user environment could benefit from most of the facilities if they were appropriately modified. Nevertheless, the specifics of the primary target environment should be kept in mind.

There are several assumptions about the way the team interacts which need to be mentioned. As was indicated above, the team members work at individual microcomputer workstations. At each workstation are copies of editors, compilers, linkers, and other software needed for program development. In order to provide a common location for source files and allow better coordination of the team's activities, there is a central machine which contains "official" copies of the source. For a team member to modify a source file, he must obtain a working copy from the central machine ("check out"), edit the copy at one of the

workstations, and then transfer the updated source back to the central machine ("check in"). Figure 1 illustrates this physical operating environment.

Several reasons for the strategy of modifying copies of the official source can be given. Because the microcomputers are single-user systems, shared access to data in the central file system is not possible. Editing copies of the official source file (rather than the central copies themselves) is needed to make the workstation concept viable. In this way, the team members can all be working productively. In addition, this allows changes to be made without overwriting the current copy so that it can be used as a backup.

Probably the nicest way to handle transferring files between workstations would be to have the machines connected in a network. However, for the purposes of the Software Management System, any means of moving the files is sufficient. One straightforward means of doing so is to use flexible diskettes (floppy disks) as both work storage and as a means of file transfer. The programmers merely insert a diskette into a disk drive and obtain the working copy, remove the diskette and take it to another machine. The use of floppy disks as a removable and transportable storage medium is common practice on microcomputers.

The central machine requires some special consideration. This machine must be able to store all of the source files for the team as well as information

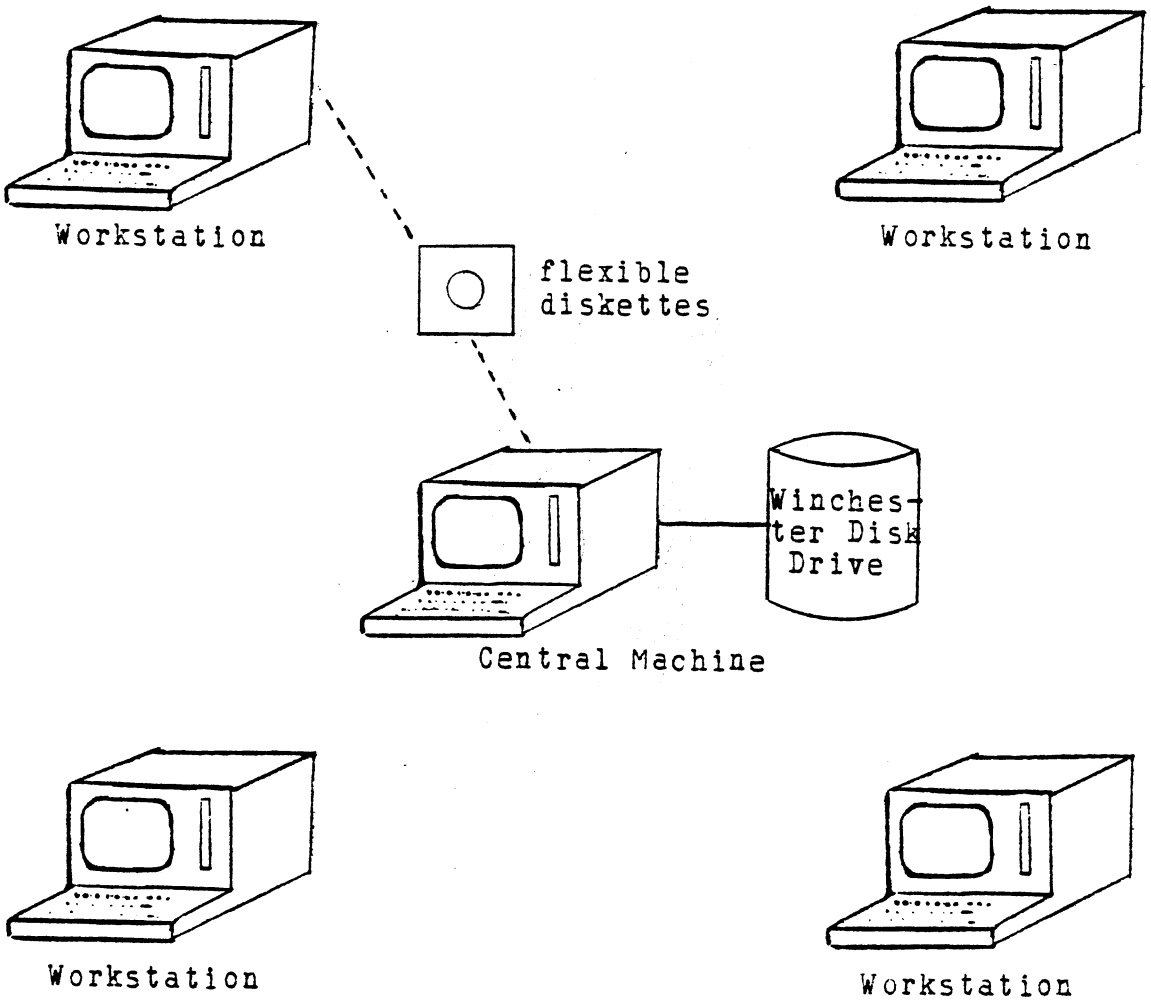


Figure 1. Operating Environment

necessary to the operation of the Software Management System. This could quite easily be more data than could fit on one or even several diskettes, and so floppy disks are not an adequate storage medium. Thus secondary storage is provided by a winchester hard disk drive. An additional benefit derived from using a hard disk is the faster access time which is possible. Printed output will be needed for reports of several kinds. For this reason a printer must be connected to the central machine. It should be mentioned that the central machine (and especially the hard disk) represents a weak link in the workstation approach; but with the multiple CPUs, another system could substitute for the central one in case it went down.

The operating system used is a significant factor in any computing environment. For the eight bit microcomputers, one of the more common operating systems is the CP/M operating system.[1] Because of its popularity, this operating system is a likely choice for the software being developed. This is also an appropriate choice for the Software Management System. A CP/M operating system is assumed in what follows. There are two main advantages to this choice. First, there are a large number of small machines which support CP/M. Secondly, there is a great deal of existing software which will run under CP/M. Of particular interest are the basic software development tools such as compilers, linkers, editors, and debugging aids.

---

[1]CP/M is a registered trademark of Digital Research, Incorporated.

There are also some limitations of CP/M which particularly affect the Software Management System. One limitation is the comparatively unsophisticated file directory structure. For one thing, there is no hierarchical directory structure. In addition, file name and location are the only information contained in the directory; all other information such as organization, creation date, version number, and access date must be maintained by the application program. Another limitation is the lack of a virtual memory management scheme which means that any memory management necessary is the responsibility of the applications programmer. A final limitation is the fact that CP/M is a single-user system, which, as mentioned, eliminates the possibility of shared access to data.

Another assumption concerns the type of language used by the development team. A language which supports modular program development through separate compilation and linking is fundamental to much of what is done. Because of the lack of virtual memory capability, the language is also assumed to support the use of overlays as a memory management technique.

As is evident from the survey of existing systems in chapter one, most of these systems include a project information database. This is in keeping with the requirements for ADA programming support environments as specified in the Stoneman report [30]. This specifies that

a database is to be a central feature in the kernel of the environment, and that all communication between tools is to take place via the database. Several advantages are gained by the use of a central database management system for storing, organizing, and retrieving project data. The central database gives the system a logical view of the data independent of its actual organization, and information about the state of the system is recorded at each step of the way on a relatively stable medium. This should facilitate recovery should some unexpected event occur such as sudden power failure. Another motivation for this is that by communicating only through the data base, the various subsystems gain a certain independence from each other. This means that modifications and enhancements to the system are easier, because one need not worry about restructuring the communication paths between the various portions of the system.

Functional Description

The Software Management System consists of three basic subsystems: the source file access control subsystem, the recompilation and relinking subsystem, and the error reporting and tracking subsystem. These will be discussed in that order, followed by discussion of some additional aspects of the system which don't fit into a single subsystem.

Source File Access Control Subsystem

The source file access control subsystem has the overall task of maintaining the integrity of source files. This is done by placing restrictions on the ability to gain access to those files. This does not mean that its purpose is to provide security against malicious tampering. The intent is to provide a disciplined approach to change as an aid to the programmers.

As with access to any information, access to the source files can be classified either as read-only or as update. There need be no restrictions on team members concerning reading the source files. The potential problems arise in the area of update access to those files. The system needs to insure first of all that any team member wishing to modify a particular file does so to an up-to-date copy of the file. In addition, the system must prevent any other modifications from being made to that file until the first is complete. This is done by having users "check out" any file which they intend to change by requesting a copy with update rights. If the file is not already checked out, the system grants the request and makes the copy. If the file is already checked out, the request is denied. The user at this point may wish to know who has the file out, and so the system provides this information. The user who has completed a modification returns the file to the system through a "check in" procedure. The system now needs to

guarantee that the file is indeed a modified version of the up-to-date copy which was checked out.

As a part of ensuring the integrity of these source files, it is important to determine that they are in some sense correct. As a minimum requirement, the system should ensure that program source files are syntactically correct by seeing that they compile without errors. If the user has several files to check in at the same time, it would not be desirable to force him to wait as each is compiled. So modules are placed in a provisional status until compilation either succeeds or fails. If the module compiles correctly, the check-in process is completed, but if it does not, the check-in attempt is rejected. If compilation fails, the user is notified of the fact so that the module can be corrected.

Another aspect to be covered is the need for a way to cancel a check-out. There are a number of events which can destroy data in this kind of microcomputer environment. Static electricity or an unexpected power failure can ruin the data on a diskette; a hardware or software failure may cause the destruction of a file; and the medium itself is prone to error, so that data can be lost.[2] After such an occurrence, one needs a way to recover by cancelling the previous check-out.

---

[2]Because of the possibility of data loss on the hard-disk as well, it should also be backed up frequently. While such backup is important to the system, it is peripheral to the main task, and so is not discussed further.

The information gathered when a file is checked out is useful beyond the time when that file has been successfully checked in. A history of check-out information is maintained by the Software Management System. A significant motivation for keeping this information available is to provide it to the programming team in accessible and useful form. The history of modifications provided by this subsystem is particularly useful when unexpected side effects occur as a result of some change. Various summary reports can be produced indicating changes to the system since a particular date, or changes to a specific module since a particular date. These reports can aid the team in locating the source of the unexpected trouble.

Figure 2 summarizes the functions performed by the source file access control subsystem.

Recompilation and Relinking Subsystem

When changes are made to a source file, these changes must be integrated into the executable system. The task of the recompilation and relinking subsystem is to automate this and to ensure that it is done consistently. The operation of this subsystem is based on the fact that source files are used to derive other files, which may in turn be used to derive others. Commands must be issued to invoke the programs which perform these derivations. Thus there exist derivation relationships among files. There are also corresponding commands which invoke the appropriate

routines. The relationship among source, object, and executable image files is the primary example of this. A program object file is derived from the source code file by issuing a command to invoke the compiler (or assembler). Similarly, an executable file is derived from the component object files by issuing a command to invoke the linker.

Other examples of this type of relationship can be given. For instance, consider a program which uses a table-driven parser to interpret commands issued by the user. If the file containing the grammar for the command language is changed, a table-building routine must be invoked to produce a new parsing table. The relationship here is between a grammar file and the parse table file. Another example is the relationship between several object files and the derived library of object files. We need to be able to deal with all of these types of relationships. In addition, there may be other relationships which are unique to a specific project. The relationship information is stored in the data base.

The actions taken by the recompilation and relinking subsystem are straightforward. The system first determines which files have been modified. It then draws upon the derivation information in the database to determine what commands must be issued to make the derived files once again consistent with the source. These newly derived files may in turn be the source for other derivations, and so an entire sequence of actions may be initiated. When all

derivation operations have been performed, the production of the new executable system should be complete.

Most compilers and linkers have a variety of options which can be used. It would not be desirable to limit users to a single set of options, so provision is made to allow the users to specify options with which to invoke the compiler or linker. For instance, the user may wish to suppress or enable the production of a printed listing.

Figure 3 contains a summary of the functions of the recompilation and relinking subsystem.

Error Reporting and Tracking

In any software development process it is necessary to test the results to determine if certain criteria are met--does the system meet the requirements? Do the requirements really match the needs of the intended users? Is the performance acceptable? There will almost invariably be areas in which the observed system behavior falls short of that desired. The system must then be modified to bring it into line with the test criteria. The programmer on a one-man project could probably keep track of necessary modifications by memory aided with brief notes, but as the number of people working on a project grows, the need for a formal means of reporting observed shortcomings and tracking progress on them becomes apparent.

Changes to a software system may be requested because of errors in its operation; they may also be requested

1. Maintains module check-out status.
2. Obtains name of person checking out a file and a description of the reason for the check-out.
3. Grants or denies request for check-out based upon check-out status.
4. If a request is denied, indicates who has the file.
5. Verifies that proper file is checked in.
6. Performs the copy functions for check-out, check-in, and read-only access.
7. Allows cancellation of a check-out.
8. Maintains history of check-out occurrences and provides for summary reports of check-out information.

Figure 2. Source File Access Control Functions

1. Guarantees correct source-object relationship by performing the compilation (or other derivation operation) of modified source.
2. Either rejects any source file which cannot be compiled correctly and notifies the programmer, or completes the check-in of files which compile successfully.
3. Determines the minimal subset of commands necessary to rebuild the executable system.
4. Performs the system rebuilding task.
5. Allows the specification of compile and link options.

Figure 3. Recompilation and Relinking Functions

simply to alter the method of handling a task. Enhancements to the system form a third category of change requests. The term "error report" includes requests for enhancements and modifications as well as reports of erroneous behaviour.

A straightforward method of tracking errors is to provide forms to be filled out with a description of the error encountered or change requested. These forms can then be referred to the appropriate team member for action. When modifications or corrections are completed, the programmer can indicate the nature of the correction and the form sent to the tester with an indication that this is ready for testing. The success or failure of testing can be noted and the report either closed and filed away or returned to the programmer for further action. However, this type of treatment means that the team members must manage additional paperwork. The error reporting and tracking subsystem reduces the amount of paperwork by performing the same job with electronic forms, treating this much like any other data processing task.

The basic items of information which need to be recorded are the name of the module or subsystem affected and a description of the error or of the change requested. In addition, an estimation of the severity of the error is recorded to aid the team members in assigning priorities to the tasks which must be done. Another necessary item is the name of the team member to whom to refer this report. This will allow this individual to be notified upon his next use

of the system. Additional items kept are the date the report was made and the name of the reporting party. This last item is saved so that the receiving team member can know whom to contact for additional information.

The error reports can be placed into four categories: those upon which no action has yet been taken (pending), those which are currently being dealt with (in progress), those which have been corrected and are ready for testing (completed), and those which have successfully passed the testing stage (closed). An error report is "opened" when it is entered on the system. These newly opened reports initially have the status pending. When a programmer checks out a file or files and indicates the purpose is to correct this error, the report moves to the in progress status. Upon check-in of those files, the status can be changed to completed, and after testing is finished successfully, the request is closed.

In some instances it is useful to move a report directly from the pending status to the closed status. For instance, this is the case when an error report is made which the team chooses to deal with by altering or relaxing the requirements. Or perhaps a change request may be made and, after examining the task, the project leader decides that the benefit is not worth the cost involved, so the request is denied. A third reason might be that a situation is reported as erroneous when in fact it is not.

Another special case occurs when a tester determines that a change did not correct the error which was supposedly fixed. In this case, the report is moved from the completed status back into the pending status.

A summary of the functions of the error reporting and tracking subsystem is found in Figure 4.

Other System Functions

There are a few other aspects of the Software Management System which have not yet been described. First of all, there is the matter of the overall operation of the system. As was mentioned earlier, this system resides on a central machine which is essentially dedicated to the task of maintaining the code library and of performing the software management functions. The system will probably be accessed frequently throughout the workday for information, for error reporting, for check-out and check-in, and will in addition perform recompilation and relinking. Because all of these operations are implemented by the Software Management System, the system controls the central machine as long as the machine is operating.

One of the items of information used at several times in the operation of the system is the date. In addition, there will sometimes be the need for the time as well. Unlike larger systems which generally operate around the clock, microcomputer systems are turned on and off as needed. On most microcomputers there is no way to keep

track of time and date when the machine is off. In order to initialize the internal clock and set the date, the first item of business when the system is started is to obtain time and date.

Another item which is needed often is the name of the individual using the system. The user would be annoyed if he were asked to enter his name every time this information is needed, and so the system obtains identification from the user when he begins using the system. Now in order to know when to ask for a new user name, the team members are asked to indicate when they are finished using the system. This also allows the system to proceed with recompilation of files which are in the provisional check-in status. This is somewhat comparable to logon and logoff of users on a multi-user system.

The notification mechanism is another aspect of the system which needs further description. This includes notification of error reports, notification of rejected check-in attempts, notification of cancelled check-outs (if not performed by the team member who had the file checked out), and notification for testers that errors have been corrected and are now ready for further testing. The individual subsystems place information in the database concerning these types of events. When a team member logs on, this mechanism is invoked to check for notices for that user. If any are found, they are displayed before proceeding.

When a new project is to be started, information about it must be placed in the database. One possible means of performing this initialization task is to independently use the access provided by the underlying database management system. However, setting up the database in this way would require not only knowledge of the database system, but also of the details of how the Software Management System expects the data to be organized. To facilitate project initialization, the system includes an interface to the database management system for entering new information. This facility is also useful when modifications to the information in the database must be made.

There are several system parameters which control its operation. These include such things as the location of the various types of files, the default compiler options and the default link options. The system provides convenient facilities to examine and modify these parameters as needed.

A summary of these other aspects of the Software Management System is found in Figure 5.

1. Obtains the name of the module or subsystem affected, information about the nature of the error encountered or change being requested, and an indication of the severity of the problem.

2. Records the reporting person's name and the date.

3. Obtains the name of the team member to whom to refer this report.

4. Tracks the status of the change; is it pending, is work in progress, is work completed, is it closed?

5. Records the date closed.

6. Provides a list of untested changes and corrections to the team member or members responsible for performing the tests.

7. Maintains a history of error reports and provides for various summary reports.

Figure 4. Error Reporting and Tracking Functions

1. Obtains date and time upon initial startup.

2. Obtains user name with a "logon" process and requests "logoff" upon completion.

3. Notifies team members of new error reports, corrected reports, a rejected check-in attempts.

4. Produces summary reports by team member and by project.

5. Provides interface with database system for initialization and modification.

6. Allows examination and alteration of system parameters.

Figure 5. Other System Functions

CHAPTER III

SOURCE FILE ACCESS CONTROL SUBSYSTEM

The term source file is used to mean any file which is directly modifiable by the user. Some examples of possible types of source files are program source files, included files (i.e., files which are included during the compilation of program source files), data files read during execution of the system being produced, and text files (e.g., documentation files). As was mentioned earlier, this subsystem has the task of maintaining the integrity of the source. This is done in part by restricting access to the files. This means that users are required to check out and check in files. Source file check-out and check-in thus are the primary functions of this subsystem from the user's point of view; others are check-out cancellation, production of reports, and read access to files.

Use of Check-out Keys

Requiring check-out of a source file in order to modify it allows the system to prohibit parallel updates and to ensure that the user has an up-to-date copy to modify. When a file is checked in, the system must verify that it is indeed a modified version of the copy which was checked out.

Obviously, an attempt to check in a file which has not first been checked out must be prohibited. Assuming the file has been checked out, the system must have some means of matching the copy being checked in with the one which was checked out. First, consider the use of a simple test of team member's name. If the system verifies merely that the same individual is checking the file in as checked it out, the crucial question of whether or not the copy of the file is the correct one remains unanswered. For suppose that when this team member checked out this particular file he also had an old copy of the same file on a separate diskette. The simple mistake of switching diskettes could cause an incorrect version to be checked in. A better solution is to issue a key value for each file upon check-out. This value is then stored in the central database. When the file is checked in, the key of the copy to be checked in is compared with the value stored in the database. If the values do not match, the check-in attempt is rejected.

One attribute which a key value requires is that of uniqueness. Two possible candidates for use as key values come to mind. The first is a simple numeric value. Each successive check-out would take the next value in sequence. This is certainly straightforward to implement. One problem with this approach is the possibility of overflow; that is, of exceeding the maximum number size. However, by choosing a representation with a sufficiently large maximum, this can

be prevented. A second candidate is a time/date stamp. This certainly provides uniqueness (as long as the time and date are properly initialized); and there is no problem with overflow. An advantage of this type of key is that the system also uses the date a file is checked out for the check-out history, and so the key can serve a dual purpose. Aside from this, either type of key seems perfectly suitable, so the choice is somewhat arbitrary. This system uses the time/date stamp as the key.

There must also be a means of associating the time/date key value with the file as it is checked out. One possibility is to embed the key in the text of the source file. For most source files, a text editor will be used to make modifications. The user will thus have access to the embedded key just as he does to the rest of the file. It is reasonable to assume that the user will not intentionally attempt to thwart the check-out mechanism by modifying this value. A more likely occurrence is an unintentional action which deletes, overwrites, or otherwise modifies the key. As the file cannot be checked in without the proper value, the user would need to cancel the check-out and begin the modification sequence again if this happens. Furthermore, this method of associating the key with the file places the user in contact with a detail of the check-out/check-in process which would be better hidden from view. One means of handling these objections would be to configure the text editor to pull the key value from the file as it is read in,

save it, and replace it when the modified file is written back to the diskette. In this way the user couldn't inadvertently destroy the key and the existence of the key would be hidden. But modifying a text editor may present some difficulties. If one can gain access to the source code for an editor, then it may be fairly straightforward to make appropriate changes. However, if the source code cannot be obtained, the only choices are to write an editor from scratch or to attempt to "patch" the object for the editor. The potential cost of performing this configuration makes it unattractive as an option.

Another item to consider is how the use of an embedded key is affected by constraints imposed on us by other tools. Specifically, the key value could be placed within a comment for program source files so that it would be ignored during compilation. As the method of delimiting comments varies from language to language, the system would need to be flexible about precisely how the embedding is done. Again, an alternative to embedding in comments would be to produce or obtain compilers which could recognize the key value. If all compilers and the linker were appropriately modified, the key values could be propagated to the derived object files. This would allow the object files to be checked in as the source is checked in, with the system verifying that the object module as well as the source module has a correct key.

Such a method of propagating embedded file identification was reported by Cristofor, Wendt, and Wonsiewicz [8] in their description of a Software Manufacturing System. In this report, the values embedded were version and release number, but the use of these was supported by integrated editors, compilers, and linkers. Again, the principle disadvantage to this approach is the potential difficulty of configuring the tools to match the Software Management System. This also makes the system dependent on the specific tools which have been so configured. Greater flexibility can be achieved by choosing to use existing tools as they are and to the degree possible, design the system to work independently of specific tools.

Rather than embed the check-out key in the source file, the system uses a separate "key" file to store the key values. The user of course has access to the key file through the normal file handling operations. The file appears in the directory listing and can be edited, renamed, or erased. For this reason, the user must be aware of the use of these files. However, the chance of inadvertently destroying the key while editing is decreased with this method of handling the keys. The chance of problems can be further reduced if the file is protected against writing or erasure. This can be done under CP/M by indicating that the file is "read only".

Another choice which arises in considering the use of key files concerns the number of such files. When several files are checked out at the same time, it would be possible to have one key file with a set of file names and keys or to have a separate key file associated with each source file. The first would be more economical in terms of disk space and directory space. This may be especially important because under CP/M the number of directory entries is limited and each key file, besides requiring a new directory entry, occupies a fairly large amount of space relative to the amount of space required for a key. On the other hand, the diskettes are used primarily as temporary workspace and so space constraints are not all that critical. One can simply use a second diskette if space is too limited on the first.

This second approach to key files gives a one-to-one correspondence between source files and key files. If for some reason a user desires to move a source file to another diskette, it is a simple matter to carry along the key by moving the associated file. Were the first approach to be taken, the entire set of keys would have to be copied, which does not seem very tidy. In addition, when checking out a new file or checking in a modified file, the system has to deal with updating the key file rather than creating or erasing the corresponding file. The choice again seems somewhat arbitrary, but because of the savings in space, the use of a single key file is the method employed.

Provisional Check-in

When a program source file is to be checked in, we would like to verify its syntactic correctness by seeing whether or not it compiles successfully. As long as it compiles correctly, the check-in can be completed, making this copy the new "official" copy of the file. If compilation fails, the check-in is aborted and the previous version remains the official copy. So the system must keep the previous version in addition to the new version at least until the compilation succeeds or fails. CP/M does not have facilities to maintain multiple versions of the source files in a single directory. A solution to this problem is to make use of the "type" portion of the file name. The three character suffix of the file name is the file type, and although certain type-naming conventions are generally followed under CP/M, there is no limitation to certain predefined types. So, for example, when the PL/I-80[1] program source file "PROG.PLI" is checked in after modifications, the old version of the file is renamed "PROG.OLD" and the new version entered in the directory under the original name.

Fundamental to the idea of having a provisional check-in is the ability to determine success or failure of compilation. The recompilation and relinking subsystem actually invokes the compiler and determines the success or failure of the process. This information must be

---

[1]PL/I-80 is a trademark of Digital Research, Inc.

communicated to the source file access subsystem. A compilation status flag is used to indicate one of three states: either the compilation has not yet been performed, the compilation was successful, or the compilation failed. This information will allow the check-in to be completed or rejected as appropriate.

Location of Source Files

To this point the simplifying assumption has been made that all files used by the system are on a single disk drive unit. This seems to follow from the requirement that the central machine use a hard disk for storage. However, a single hard disk unit may be divided into two or more logical drives. The smaller space available on each logical drive may require that files be divided between the drives. If we allow the user to specify where particular types of files or even where individual files are to be stored, there is a great increase in flexibility. The additional flexibility adds complexity, however. If a file is to be compiled which includes other files, those files need to be available. This necessitates extra checking and perhaps some movement of files between drives prior to compilations.

New Files

When a project is first started and from time to time during the development process, new files will be added to the system. These files have never been checked out, so

they cannot be treated in the manner described earlier. The first step in adding a new file to a project must be to place information about it in the database. The file can then be checked in as usual except that there is no key to be matched and there are no old files to rename. This then can be compiled by the system as usual.

Interaction with Error Report Subsystem

A large proportion of modifications to a system under development are made in response to a particular report of an error or problem. Others are made to satisfy requests for alterations or enhancements to the system. These error reports and change requests are to be recorded by another subsystem, and we wish to record the relationship between modifications and error reports. The user is asked to specify the connection by providing the error report numbers which identify the corresponding error reports. There may be a single error which is to be corrected; and it may be that there are multiple errors which the programmer intends to correct at the same time. It may also be that the modification has no corresponding report.[2] This might occur because the programmer wants to "clean up" some code, which although correct, is not well-structured and would be

[2]This relationship between error reports and check-out occurrences is in general a many-to-many relationship. It may be that there are several files which must be modified to correct a single reported error. On the other hand it may be that a programmer can correct several errors within a source file at one time. In this case there would be only one check-out occurrence and several error reports which are to be corrected by this check-out.

difficult to modify in the future. Or perhaps there is an error which has not been recorded.

Change History and Summary Reports

Changes to a complex system are sometimes accompanied by unintended side effects. These may appear in code which previously worked correctly, and may be in a portion of the system not clearly related to the code which was modified. These kinds of errors can be very difficult to locate. One piece of information needed to solve such a puzzle is a listing of recent changes. This can indicate possible places to begin looking for the problem. Maintaining a history of check-out/check-in information provides us with the data needed. When a file is checked in, we can record the date, and store this information along with the information obtained when the file was checked out. Of course, depending on the size of the project, it may not be feasible to keep all of the history records in the database. Periodically records of changes made prior to some cutoff date can be moved to a backup medium and purged from the database.

When one of the team members wishes to examine the history of changes made, he requests a "recent changes" report and gives a date from which to start it. The system compares the date entered here with the date recorded as files were checked in and retrieves the information. This report can be further refined by limiting the scope to a particular subsystem, module, or programmer.

Another report which should prove valuable is a summary of files checked out. A particular programmer may want a reminder about what work he has begun, and so this report can be restricted in scope to the files checked out by a single programmer. The team leader may desire to see information about files checked out for the entire project, and so the report can be generated for the project as a whole. These reports, along with those produced by the error reporting subsystem, provide better "visibility" for the work being done on the project and provide valuable help for its management.

Data Summary

A listing of the information which must be maintained in the database in order to support the functions of this portion of the system is found in Figure 6.

The files each have a corresponding modification flag which is used to indicate to the recompilation and relinking subsystem whether they have been changed. The check-out record actually indicates a relationship between a source file, a team member, and perhaps one or more error reports. The reason for check-out is a textual description of what must be changed; however, instead of such a description, the connection with the error report which prompted the change should be sufficient. All of the information but the check-in date is obtained during the check-out process.

General Information

File names
Modification flags
Compilation status flags

Check-out record

Check-out key
File name
Name of programmer
Reason for check-out
Error report number(s)
Date checked out
Date checked in

Cancellation record

Check-out key
Name of cancelling programmer
Reason for cancellation
Date cancelled

Figure 6. Source File Access Control Data

The cancellation record contains information needed if a check-out does need to be overridden. The one who cancels may be the same person who had the file checked out; or it may be that one programmer cancels the check-out of another. In the latter case, the one whose check-out was cancelled should be informed and provided the name of the one responsible and the reason. Hopefully, such occurrences will be rare.

CHAPTER IV

RECOMPILATION AND RELINKING SUBSYSTEM

The task of the recompilation and relinking subsystem is to automate the rebuilding of a system after modifications are made at the source level. The term "compilation" refers to the process of taking a high-level language program and producing the corresponding machine language program, usually in relocatable object form. The use of this term is not intended to imply that only high-level language source code can be used; for some functions it may be advantageous to use assembly language. In addition, as was pointed out earlier, there may well be source files which are not program files and derived files other than program object files. The terms compilation and linking are used because these derivations are by far the most common. However, while relinking is used to refer specifically to the process of resolving external references and producing an executable image from a set of relocatable object files, recompilation is used rather loosely to refer to any other type of derivation.

In order to know what portions of the total system must be rebuilt to regain a consistent state, the system must be able to determine which files have been modified since the

last recompile/relink processing was done. Some operating system directory structures are helpful in this regard as they provide information about the creation date of a file. MAKE, for instance, compares the time/date stamp of derived files with that of those upon which they are dependent to determine if the dependent files must be derived again [13]. CP/M does not provide us with this feature. This means that the system must maintain a modification flag to indicate if a file has been modified since the last recompile and relink sequence.

Compilation and Linking

The two processes--the recompilation process and the relinking process--are handled in significantly different ways. A link requires many or all of the object files of the system as input. In addition, relinking an entire system may be quite time consuming, especially on a microcomputer. Because of these factors, there is a real advantage in "batching" the link processing in order to avoid the overhead of performing the same work repeatedly. To illustrate this, suppose the system under production contains thirty object files which are linked to obtain the executable image file. Suppose then that two of these files, FILEA and FILEB, have been modified. If batching is not used, immediately after FILEA is checked in and compiled the link command is issued (processing all thirty files). Subsequently, FILEB is checked in and compiled. The same

linking command is then issued again, processing twenty-nine files which are precisely identical to those for the link just completed. If, on the other hand, the system were to defer linking and batch the processing, a single link command would suffice.

In contrast to this, a compilation requires only one primary input file, with secondary input possible from any included files. This means that each modified source file usually corresponds to a distinct compile command. This is not strictly the case because of included files, but because of the way in which included files are likely to be used it is generally true. Included files often contain information such as global data declarations and syntactically replaced constants which are common to a number of files. As such, included files are much less likely to require corrections and alterations than the program source files themselves. It is possible that files included during compilation of another file be changed one at a time and that the primary file be repeatedly compiled. However, such an occurrence would be quite unusual. So, in general, there is no advantage to be gained in batching compilations. Thus they can be performed incrementally, as each modified file is returned to the system.

An additional distincton between the compile and link portions of the rebuilding process is the way commands are determined. Because of the essentially one-to-one relationship between source files and recompile commands,

these commands can be stored directly in the database and retrieved as needed. The link process presents a more interesting situation, particularly when the use of overlays is involved. A complex sequence of commands, each dependent on previous ones, may be required to relink a system in which overlays are used. Depending on the circumstances, some subset of this entire sequence may be adequate to restore a system to a correct state. In order to allow the system to determine the minimum command sequence needed for each circumstance, the link commands are not stored directly; information about the system (i.e. about the overlay structure) is used to construct the necessary command sequence. This will become clearer as the use of overlays is described in more detail later.

Recompilation

Before proceeding into the discussion of the recompilation process, perhaps it would be useful to elaborate on a choice which has not yet been explained. The question is, why is it necessary for the system to perform recompilation? Why not have the users compile source files at their workstations and check in the object code as well as the modified source? The users will want to recompile the modules anyway to make sure that they did not inadvertently introduce any syntax errors as they made their changes, so there would be no extra burden on them. This would also relieve the central machine of a time-consuming

task. The answer, of course, is that the system has no way of verifying that source and object really match each other without actually compiling the source. Maintaining consistency between source and object in this way is an essential part of the task of maintaining the integrity of the system being produced.

Compilations are performed as files are checked in. Several source files may be returned by the user at one time, and so there may be several files waiting to be compiled at any given time. The user will have to indicate when he is finished so that the system can proceed with compilations. Now even though the one team member has indicated that he is through using the system, other team members may need to have access to it. A potential problem arises because the system is going to be compiling while a team member is forced to wait. As each compile may take several minutes (depending on the size of the module being compiled), this could prove to be quite frustrating. One of the underlying goals of the Software Management System is to be helpful to the programming team. Making a user wait for an entire series of compilations would be a hindrance rather than a help. For this reason the compilation sequence can be "interrupted" by a user who wishes to access the system.

On a single-user system, only one process has control at a time. Thus when the compiler is operating, it has complete control of the CPU. The recompilation and relinking subsystem cannot control what takes place while

the compiler has the CPU. However, most microcomputers buffer input from the keyboard. This means that one possible way to allow interruption of the compilation sequence is to check for keyboard input between the individual compilations. The user would still have to wait, but only for the completion of a single compile, rather than the entire sequence.[1]

Drive Usage

Before a compilation command is issued the recompilation and relinking subsystem needs to determine that all necessary files are available for processing. The drive which contains the primary input file can be specified in the command itself, but the included files may cause difficulty. The included files will need to be copied to the drives where they will be expected by the compiler. The include statement within the program source file may contain a drive specification which indicates on which drive the included file is to be found. If no drive specification is given, the compiler assumes the included file is on the current default disk drive (called the "logged" drive).

It might be convenient to store all included files on a particular drive, and to require that all the include statements reference this particular drive. This avoids the

---

[1]Actually, one microcomputer compiler which the author has used has the annoying habit of aborting if any key is pressed. This being the case, any time the sequence is aborted, the system is forced to restart the compilation. Besides being somewhat inefficient, this also complicates checking for correct compilation.

necessity of moving the included files prior to compilation, but there is a signficant drawback. The workstations used may not have the same drives as the central system. Because the user will want to compile modified source prior to checking it in, the choices for the drive to be used for included files would be limited to one which would be found on the workstations as well as on the central machine. An alternative is to require that the programmers always omit the drive specification in the include statement. This means that the currently logged drive must contain any included files. In this way the individual can be responsible for obtaining copies of any included files needed to compile at the workstation, and the system can easily make sure that needed files are available on the logged drive before issuing the compile command. If the system is configured so that all included files are on the same drive, this should be the logged drive during compilation. In this way no copying or moving of files would be required.

Once the compilation has been completed, the system needs to determine its success or failure. Unlike compilers which are set up to run in a batched environment, there is no "return code" as such from those which operate under CP/M. They usually display messages on the console screen for the programmer which note any errors present. These messages can usually be directed to a disk file. So a scan of this file can be used to determine whether or not the

compilation was completed successfully. The form of these messages will of course be dependent on the compiler used, and so the system must know what to look for. A simpler way to achieve the same end is to determine whether or not the compiler has produced an object module. This of course assumes that no code will be generated if there are errors in the compilation process. In order to be able to find out if an object file is produced, the system must know that no copy of the object file was present before the compilation attempt. This can be guaranteed by erasing the old object file before proceeding, but then if the compilation fails, we would need to recompile the old program source file to restore the system to a consistent state. By renaming the old object file, we can determine if a new object file is produced and restore the old one easily if it is not.

The Linking Process

Options

The various options available for the link process cover a wide variety of features. Options are used to indicate specific modes of operation for the linker, to indicate characteristics of the code generated, and to redirect information produced by the linker. Some options may be necessary for the success of the link, such as one instructing the linker to use the disk for workspace when available primary memory is not sufficient. Others may only

be used occasionally to obtain details of the link process not normally produced. Other options may normally be set in one way, but may sometimes need to be changed. Thus there are some options which will be used always, some which will be "defaults" that may be changed, and some which are used for a single link. Options in the last two categories can be entered by the user when the link process is initiated. These are then used in the link commands which are subsequently generated.

Use of Overlay Techniques

The operating systems of many large scale general purpose computers now have virtual memory capabilities, freeing the applications programmer from any concern with memory management. However, there are also a large number of small systems without such capabilities. One memory management technique used on such systems is that of overlaying portions of memory with different code when needed during the course of program execution [22, 25]. This technique is considered obsolete on most larger systems, but is quite necessary as long as there are systems with real storage management only.

A program which uses overlays works generally as follows. There is some portion of memory which contains code that must be resident throughout the execution of the program. This portion, sometimes referred to as the "root", contains common data structures and common routines as well

as the code necessary to drive the rest of the system. The memory not occupied by the root or by the operating system is used for "overlays", code segments which are read from secondary storage as needed.

Figure 7 illustrates the organization of memory for a simple overlay scheme. The root contains the driver and all common data. The first phase of processing is performed by overlay 1. When this phase is complete, overlay 2 is loaded at the same address (overwriting overlay 1), and the second phase of processing is performed.

A more complicated overlay scheme may use several overlay areas, each of which is used for several program segments. This allows one overlay to invoke any other which does not use the same region of memory. In Figure 8, overlay 1 can invoke overlays 3, 4, or 5 but not overlay 2.

A further increase in memory space utilization can be achieved by allowing the relaxation of divisions between separate "areas". For instance, suppose that overlay 1 of Figure 8 uses only overlays 3 and 4, while overlay 2 uses only overlay 5. This means that overlay 2 may be allowed to extend into the memory space used by overlays 3 and 4, provided that overlay 5 is loaded above overlay 2. This type of arrangement is shown in Figure 9.

The motivation for supporting the use of overlays is quite simple--the program being developed may be larger than will fit into memory otherwise. Again, if virtual memory were available, the use of overlays would be unnecessary.

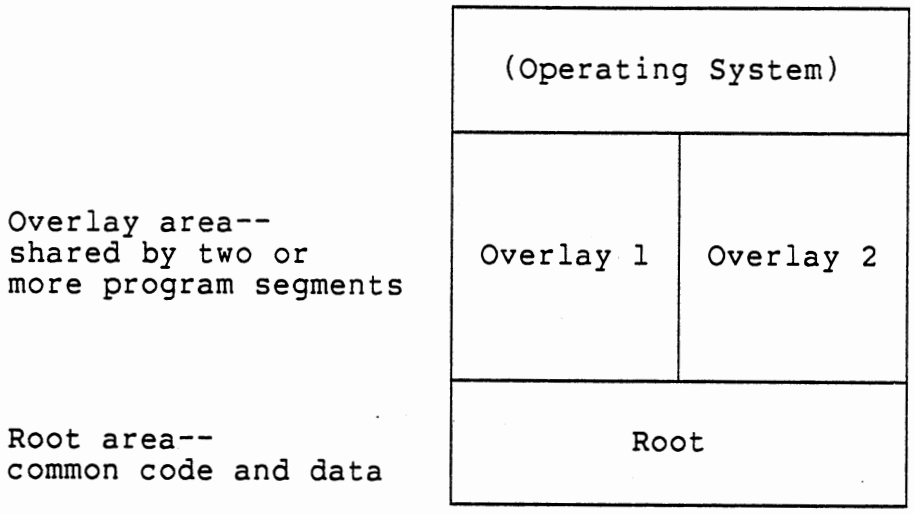


Figure 7. A Simple Overlay Scheme

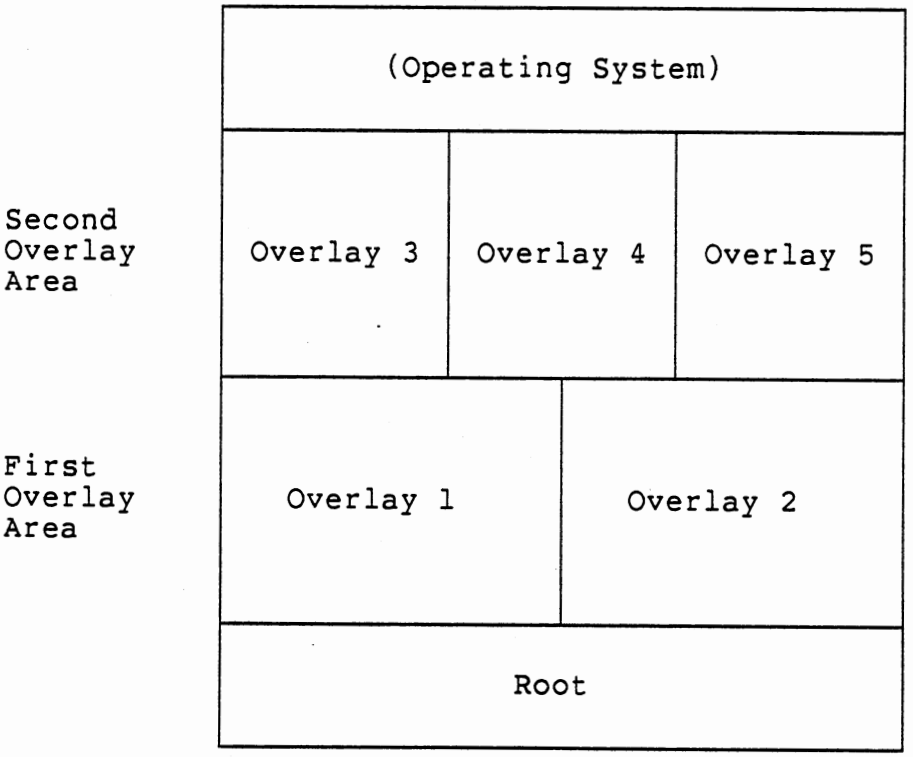


Figure 8. Use of Multiple Overlay Areas

Without virtual memory, overlay techniques allow tasks to be performed which would not otherwise be feasible. Two linkers which support overlays for CP/M programs are LINK-80 by Digital Research [9] and PLINK-II by Phoenix Software Associates [18].

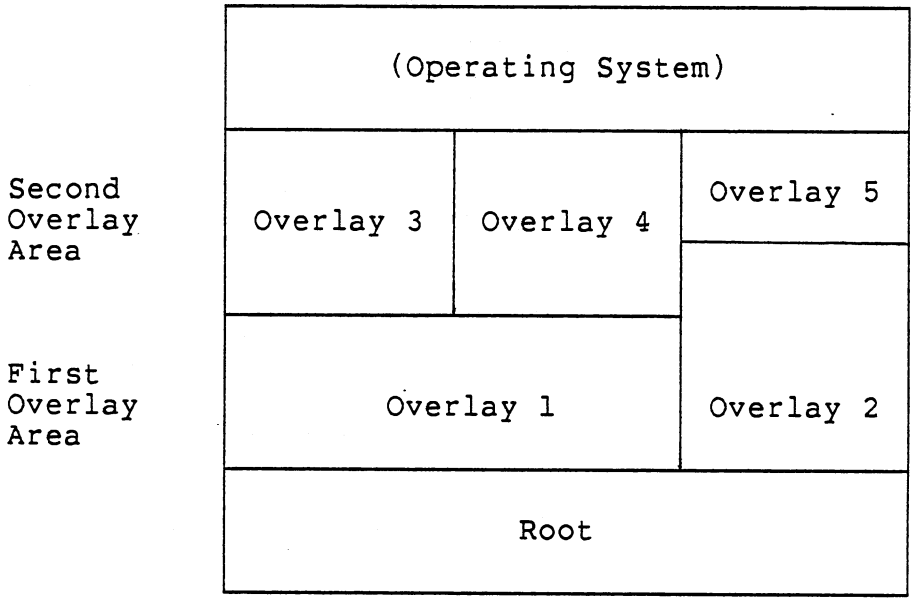


Figure 9. A More Complex Overlay Scheme

Stages in the Link Process

It can be seen from the above illustrations that the location of an overlay in memory is dependent upon the use of that overlay by other overlays. Overlays which are used

by another may not occupy the same memory space as the overlay which uses them. As long as there are no further use relationships between these files, they may share the same overlay area.

Any given code segment has a lower bound and an upper bound in memory. The lower bound is the address at which the first instruction is placed when loaded (that is, when it is read in from disk). This is where execution begins when the segment is invoked. This is referred to as the load address of the segment. The upper bound is known as the module top. The linker generally requires the load address as input and provides module tops as output as each overlay is linked. The module tops of all segments which use a particular overlay are used to derive the load address for that overlay. Thus information necessary for the linking of one overlay area is not available until the completion of the link for the previous overlay area. So for any overlay scheme using multiple areas, the link process is divided into stages. The module top information obtained from one stage becomes input to the next. The number of stages is of course dependent on the number of overlay areas.

Full Links and Partial Links

For an overlay scheme such as we have been describing, the linker does not produce a single executable file but multiple files. There is a file for the root and one for each overlay in the system.

As was mentioned, common data structures and common code are found in the root of the overlay structure. Individual overlays may access any of these common data structures or routines. Thus the overlays must have correct addresses for these. For this reason the entire system is dependent upon the root addresses and the complete link process must be performed if these addresses change. Such a complete link is referred to as a "full" link.

On the other hand, when changes are made to individual overlay routines, these changes do not necessarily affect other portions of the system. Overlays used by the routine may be affected if a change in its module top occurs; otherwise, the only output file needing relinking is the one which was modified. This type of link is known as a "partial" link.

A full link is, with the exception of the load addresses used, a fairly static set of commands. There is no "minimum command sequence." However, a partial link may be quite short compared with the full link process. This is where the ability to determine a minimal set of commands pays off. If the system being produced uses ten overlay files, and only one has changed, only that one must be relinked (assuming that its module top does not affect the load addresses of other overlays).

Construction of the Link Commands

The overlay structure is stored in the database by means of the uses relationship. Although similar to the calling relationship between modules, the two are not identical. For one thing, any of the overlays may call routines found in the root, but this information is not needed during the construction of the link commands. In addition, an overlay may consist of several external procedures linked together. The calling relationships within the overlay aren't relevant to the link process. Only the use by one executable segment of other overlays is recorded by this relationship. The term executable segment is used to encompass the root as well as the overlays.

The determination of the link commands necessary is governed by three "rules". The fundamental rule is that an overlay used by other executable segments must be loaded in memory above those segments which use it. The second rule has already been mentioned--if the root segment changes, a full link must be performed regardless of whether or not other segments have been modified. The third rule is dependent upon the particular overlay manager being used and is difficult to automatically enforce. There is sometimes a maximum number of "active" overlays permitted by the overlay manager. An active overlay is one which is currently resident in memory. In the LINK-80 system with which the author has worked, this maximum was five.

This limitation has some implications. First, the breakdown of the system into overlays must take the limit into account. The programmers must see to it that no more than five overlays need to be active at one time. Second, the following problem must be handled. For illustrative purposes, let us assume a maximum of three active overlays. Say that at a particular point in the system execution, exactly three overlays are active. Say further that there is an unused portion of memory between OVL2 and OVL3 (this can arise because OVL3 is used by other segments which have a higher module top than does OVL2). This situation is illustrated in Figure 10(a).

Now suppose a new overlay is loaded that is logically supposed to replace OVL3, and that it is used only by OVL2. This allows the load address to be the top of OVL2. The new overlay is small enough that its top is still at or below the load address of OVL3. This is illustrated in Figure 10(b). The overlay manager considers an overlay "active" until it has been displaced by another overlay, so there are apparently four active overlay areas. This event may cause abnormal termination of program execution.

Automatic detection and correction of this problem is not incorporated into the command construction process. As a means to solve this problem, the system allows the user to specify an explicit load address for an overlay. If this is present and is higher than that derived from the module tops of the segments which use the overlay, it is used as the

load address. This allows the load addresses to be set so that the intended overlay is displaced.

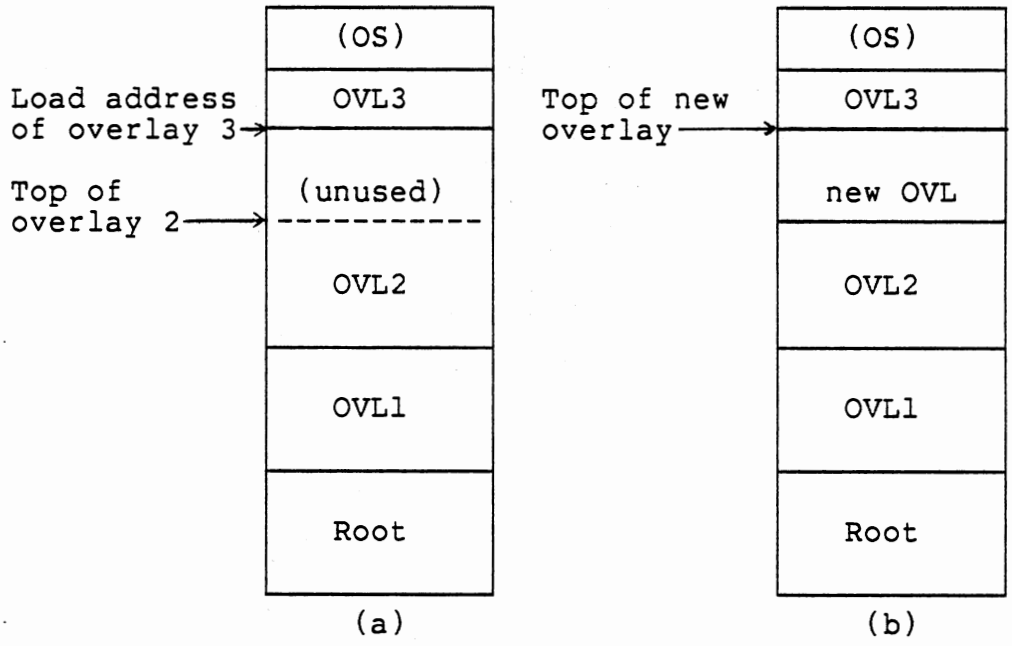


Figure 10. Potential Overlay Problem

A preliminary step in the relinking process is thus to determine if the root will change. If so, this fact will dictate that a full link be done. Otherwise a partial link is sufficient. The root of the program under CP/M is a file of type "COM". If any of the object files which are used to derive this file have changed, the root will change. If a full link is needed, stage one of the process is to link the

root and all overlays used only by the root. The derives relationship is used to determine names of object files to be linked into the root. The uses relationship is used to determine which overlays are used by the root. Then any overlays which are also used by other overlays are removed from the list of overlays to be linked in stage one. Then the command is generated in the proper form and the linker invoked with this command. Upon completion of stage one, the module tops are updated.

Commands for the subsequent stages are determined as follows. The system lists all unlinked overlays which have been used by segments linked thus far. It then eliminates those overlays used by as yet unlinked segments. The load address of each overlay is then determined by taking the maximum of the module tops of all those segments which use that overlay. Again, if a higher load address is explicitly specified, it overrides that determined by the preceeding method. Now the command can be generated and subsequently executed. Upon completion of each state, the module tops are appropriately updated. This repeats until all overlays have been linked. Upon conclusion of this process, the highest of any of the module tops becomes the system module top.

A partial link proceeds somewhat similarly. The root modules must be linked with each command, so that all overlays have access to global data and routines. In addition, all overlays which will change (corresponding to

modified object files) are listed. Based upon the uses relationships, all overlays which are directly or indirectly used by others being linked are eliminated from the list. This means that overlays from the lowest area are linked first. Module tops are determined based upon information from prior links. Now the command is produced and executed.

Upon completion of each stage, module tops are updated and at the same time a check is made for changes which will affect the load addresses of any unmodified overlays. If any load addresses are changed, the affected files must be added to the list of files to be relinked. The process is repeated until all modified files have been linked. One additional consideration is necessary for a language such as PLI-80. This language uses available memory for a run-time stack and for dynamically allocated storage. Because of this, the root module is "backpatched" with the system module top by the linker. Consequently, the last link operation performed must contain the module with the highest top of any in the system. This is true anyway for full links; the system must ensure that it is done for partial links.

Data Summary

In conclusion, a summary of the data required to support the functions of the recompilation and relinking subsystem is presented. Figure 11 lists this information. The modification flags are used to determine which files

have changed. The module top information is collected and used during the link process. The load addresses are those explicitly specified by the user as described earlier. In most cases, no explicit load address will be present. The derives relationships each have a corresponding command. The exceptions are relationships associated with the link process. For these, an indication that relinking must be performed is stored in place of a command. The uses relationship was described earlier. Default compiler and linker options are also stored. A final type of information is the compilation status flag, which is used to communicate with the source file access subsystem.

General Information

File names
Modification flags
Compilation status flags

Derivation Information

Derives relationship
Derivation commands

Overlay Structure

Uses relationship
Explicit load addresses
Module tops
System module top

Other Data

Default compile and link options

Figure 11. Recompilation and Relinking Data

CHAPTER V

ERROR REPORTING AND TRACKING SUBSYSTEM

Identifying Error Report Responsibility

The job of recording and tracking error reports and change requests is handled by the error reporting and tracking subsystem. The primary purpose of this subsystem is to see that when an error is reported or a change is requested, one of the team members follows through to make the necessary modifications or corrections. So when an error report is entered on the system, one of the team members must be notified so that he can take action upon it. A key question which must first be answered is how should the job of identifying the proper person to whom to refer each report be done.

One approach which could be taken is to require that the individual entering the report supply the name of the team member who should be responsible for dealing with the error. This is certainly straightforward as far as the Software Management System is concerned. The team member indicated can be notified when he next uses the system. However, an assumption is made with this approach which may not always be valid. That is the assumption that the

testers will be able to identify the correct individual. This may be the case if the tester is a team member who has been with the project for some time; this will probably not be the case if the tester is an outside party brought in for the purpose of finding errors. It may be possible to provide testers with a list of module names or functional areas and the names of team members responsible for each. This, along with the name of a person to whom to give all reports which don't fit into one of the specified categories, will allow the testers to provide the information the system needs. Another possible approach is to give all error reports to a single individual (such as the team leader) who can then divide the work appropriately. This is a flexible approach but means added work for this one person. However, it also provides better "visibility" of progress to the project leader. This can aid in the ability to see if the project is on schedule and to determine if productivity goals are being met.

Given the proper information as input, it is possible to automate the identification of the team member responsible for each error report. The team may be organized so that each module is "owned" by a particular team member. That is, the owner of a module has primary responsibility for that portion of code. If this "ownership" information is available in the system database, the one making an error report can identify a module and the system can associate the report with the responsible person.

Of course, the tester may not be familiar with the underlying modular organization of the project, and even if he is, it may not be clear which module to identify as the culprit. In fact an error may be caused as the result of some complex interaction between a number of modules, so that there is no single module responsible.

Another difficulty with this approach is that the tester's view of a system may be quite different than the actual modular organization. The user sees various functions which may or may not correspond to particular divisions in the code. A possibility that is used by some groups is to assign functional responsibilities to team members. The system then obtains information which indicates the location of the problem in terms of function when the error report is entered. This seems to be a more natural approach for those entering the reports. Even this does not solve all difficulties, however. Even in a well-structured system, there is room for a great deal of confusion as to how to classify a problem. For instance, suppose the project being developed has three primary functional areas. In addition, there are several "utility" functions, such as one to obtain user input, one to display information, and perhaps a set of editing functions. If an error is noted which occurs as the user is providing input for the first functional area, does that error originate in the user input utility or in functional area one? Perhaps only further investigation by one of the team members can really answer the question.

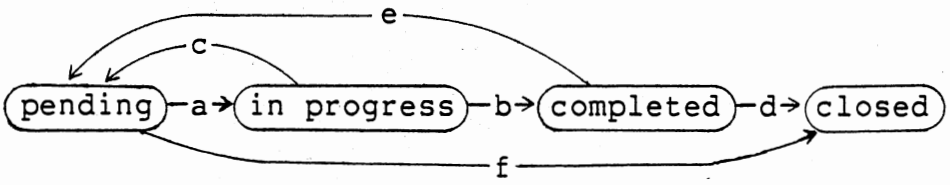
This latter approach to the problem--that of maintaining information on functional areas of responsibility and having the tester specify the location of the problem in terms of function--seems to give the best balance of practicality and useability. The tester indicates the area to which the error report applies, and the system determines the individual to whom to give the report. This is the method used by the error reporting subsystem.

Whatever the approach to this problem, there will most probably be times that a user decides either that a particular error report does not "belong" to him (it is not in his area of responsibility) or that for some reason the problem should be handled by another person. This could be due to the need to balance the work load among team members or due to the abilities of the individuals involved. Because of this, there needs to be some way of "forwarding" an error report to a new person. In this way, even if the initial decision about assigning a report to a person is incorrect, there is a means of getting the report to the right person.

Status Transitions

Tracking of error reports within the system is accomplished by placing them in four different statuses. The pending status indicates that no action has yet been taken on the report. In progress means that the error is

currently being dealt with. Once the programmer is satisfied that the problem has been fixed and is ready for testing, the error report is moves to the completed status. Closed indicates that the testing was indeed successful and that no further action is required on this particular problem. A report is initially placed in the pending status, and from there usually moves successively to in progress, then to completed, and finally, to closed. Three other special transitions may also occur. These are a transition from in progress back to pending, a transition from completed back to pending, and a transition from pending to closed. Figure 12 illustrates the transitions possible between the various statuses.



a. Check-out for correction
b. Report ready for testing
c. Report referred to another team member
d. Successful completion of testing
e. Test failure
f. Decision to take no action

Figure 12. Error Report Status Transitions

The transition from pending to in progress can be handled in one of two ways; either the status can be changed as soon as the first file is checked out in order to satisfy a particular error report, or upon explicit indication by the user. In the first case, the user must specifically state when checking out a file which error reports are to be corrected. This approach has the advantage that it is automatic and it can be implemented fairly easily. The latter approach would allow the system to reflect the state of the project more accurately, because there is usually examination, testing, analysis and redesign work done well in advance of any actual change. So work on an error report actually begins prior to the modification of files. However, saying that a correction is not in progress until a file is actually being modified seems a satisfactory choice and does allow an automatic transition. This is how the error reporting subsystem handles it.

The transition from in progress to completed could similarly be made when all files which were checked out to satisfy this particular report have been checked in. However, files which have not yet been checked out may also need modification, and so the system cannot really know that the change is complete. This transition must be indicated by the user. It is possible that the user has checked out a file or files intending to correct a certain error report but for one reason or another has decided that another team member could better handle the job. He may either cancel

the check-out or check in the modified file with other corrections he has made. In either case he can then specify that the report be moved back into the pending status.

Once an error report has reached the completed status, it must be tested. When the tester is satisfied that a particular error has been corrected, he can indicate that the corresponding report is now closed. But what if the tests are not successful? It is possible that the observed shortcoming is still present, and that the modification had no effect on it. It is also possible that in correcting one problem, another problem has been created. In the first case, the tester can indicate that the original report is to be moved back to the pending status. In the second case a more reasonable action is to close the first error report and open a new error report on the new problem. So two conditions indicate a transition from completed to closed: successful results from testing and unsuccessful test results which point to a new problem. Only in the case in which the original error has not been corrected should the transition from completed back to pending be made. In this situation, the programmer responsible is informed of this change in status when he next uses the system.

A final special case arises when for some reason the decision is made to ignore a particular error report. In this case we wish to bypass the intermediate statuses and move the report directly from pending to closed. This may be because the error report itself is incorrect; it may also

be that the solution to the discrepancy between requirements and operation is to change the requirements. Another possible reason for moving a report directly from pending to closed is that a request for a change has been made but the group (or team leader) decides that the change should not be made now.

History of Error Reports

Once error reports have been moved to the closed status, no further action need be taken to deal with them. They no longer play an active role in the development process. For this reason it may seem at first that they should be purged from the system database. However, there are good reasons for keeping this information available. Information from one project indicating the types of errors found and their frequency could prove helpful in learning what types of problems to guard against in a second project. Information about the length of time between opening and closing of error reports may suggest need for changes in team organization or in the management of the project. Other uses can no doubt be suggested for this data as well. So closed error reports are kept for these historical purposes.

While such information should be saved for later analysis, its accumulation is somewhat peripheral to the main task of the Software Management System. In addition, the volume of data represented by the error reports and

change requests for a lengthy project may be very substantial. Consequently, some means of archiving the data is needed. That is, the system must provide a way to move "old" data to a backup medium of storage and clear the records from the database. Any available backup medium could be used for this purpose. One possible choice is to use floppy diskettes, as the system is assumed to have floppy disk drives; a better choice if available is digital magnetic tape because of its reliability compared to the diskettes. Another possible archival medium is that of printed listings. This choice has the disadvantage that it is no longer in machine readable form; but if the reports are not likely to be needed in this form, this is not a serious drawback.

Archiving may remove all closed reports to the backup medium, or it may remove only those reports which have been closed for a certain length of time. The date closed is compared with a cutoff date to determine which reports should be archived. The choice of the cutoff date is left to the user.

Summary Reports

Easy access to the error reporting subsystem information is provided by means of the summary reports. A programmer may wish to know what future tasks need to be done, and for this he may wish a listing of all pending error reports which "belong" to him. Or perhaps he wishes a

reminder of what corrections are currently being made, and so desires a list of his error reports which are in progress. Reports may be requested for the completed and closed statuses as well (the report for closed error reports is of course dependent on whether or not these have been archived yet). So one parameter for generation of summary reports is the status indicator for the error reports desired.

A second parameter used in narrowing the scope of these reports is whether the report is for a single team member or for the whole project. Reports for the whole project can be used to give the team leader a feel for the progress being made, possible problem areas, and other information useful for managing the team. These reports also help give "visibility" to the work being done. The third parameter useful for specializing the reports is a functional area parameter. This can be used to gather all reports pertaining to a particular portion of the system so that the necessary corrections can be made at the same time.

Additional Aspects of the System

Notification Mechanism

The system communicates to the users via the notification mechanism, so it is a quite important portion of the system. There are four types of notification items: rejected check-in attempts, cancelled check-outs, new error

reports, and completed error reports. A rejected check-in needs to be dealt with by the individual who last modified the file, and so this person is the one informed of the rejection. Each file modification is associated with a particular check-out record, and it is this record which must be uniquely identified. The check-out key performs this identification function. The check-out record then indicates the individual to be informed of the problem. Similarly, a cancelled check-out is associated with a unique check-out record which then points to the individual to be notified. However, there is no need for notification if the individual who cancelled the check-out is the same as the one who had the file out.

The error reports are each uniquely identified by an error report number. The error report contains an indication of the functional area to which it applies. The database holds information which links these areas with the individuals responsible. This information is used to determine whom to notify.

The final type of item, notification of completed error reports, may be handled in at least two ways. If one individual is responsible for testing, he can of course be notified automatically. It may be though that all of the team members share some responsibility for testing. If each one has a particular subset of the project to examine, we could ask for the name of the team member to notify as each error report is completed, and notify that individual of

just the reports which affect his particular subset. On the other hand, it might be that the responsibility is not divided and that the team members choose what to test; in this case we could allow the users to request a listing of all error reports ready for testing. This last possibility seems an undesirable way to organize the team, as there would undoubtably be portions of the project which would not be tested as thoroughly as one would like. We will assume that there is a specific team member to notify for each error report.

The system may have to notify a user of several error reports at one time. In order to make the information manageable, the system displays only the unique report identification number, the name of the module or subsystem in question, and a summary of the error. The user can later retrieve a complete error report, probably producing a printed copy. The notifications of corrected errors also reference error reports, and so the same type of summary display is used for them as well. The notification of check-in problems has only one item of information for each message, and that is the name of the file rejected. The only reason this can occur is that the compilation attempt failed so the check-in was not completed.

Another item to consider concerning the notification mechanism is how to control its display. That is, how many times should the person see these items? A simple choice would be to say that he is notified only once, and that

thereafter he must explicitly request the information. Perhaps a better method is to have a "reminder" for him until he actually looks at the complete report or takes action for a rejected check-in. No action is necessary for cancelled check-outs, and the user can be reminded of the cancellation if he does try to check in the file without rechecking it.

There is a certain amount of indirection necessary in determining whom to notify, especially for the error reports. Because a search for notices will need to be made each time someone logs onto the system, it is important that excessive search time be avoided. To lessen search time, each notification item could contain the name of the team member to be notified. The question of whether to store team member name is an implementation detail which will not be addressed further in this thesis.

System Utilities

The system provides convenient access to project information contained in the database. This access is set up with the structure of the information used by the Software Management System in mind. The fundamental project information can be initialized and modified. This includes such items as: project name, file names, the uses relationships, the derivation relationships, the derivation commands, and explicit load addresses. Of course, the information used only internally by the system is not accessible through this utility.

The other "utility" function is that which provides access to system parameters. These are such things as: location of files (by type); default compiler and linker options, and rename types (e.g. "OLD" source files, and "ORL" for old relocatable object files).

Data Summary

Figure 13 contains a summary of data used by the error reporting and tracking subsystem. The error report number is assigned by the system to new error reports as they are entered. An integral number is used in this case because the users will need to reference these in order to identify error reports as reasons for checking out files. For this reason, ascending numbers seem preferable to something like a time/date identifier.

The name of the team member to whom to forward the report can be filled in initially by the system, based on functional responsibilities. If the report needs to be forwarded, the user can then change this field. When this occurs, a notification item is entered as for a new report. The status field is used to track progress in correcting the error. When a team member indicates that a particular error report is completed, he enters the description of the correction made. The date closed is recorded by the system when the tester indicates that a report is to be closed.

Figure 14 lists other data used by the system. The notification items are triples consisting of type, person,

and either error report number or check-out key depending on notification type. General project information needed includes project name, team member names, and functional responsibility information. File location information consists of file type/drive name pairs. The remaining information should be self-explanatory.

Error Report Record

Error report number
Summary of error
Description of error
Functional area
Severity level
Date of report
Reporter
Team member to whom to forward report
Status
Description of correction
Date closed

Figure 13. Error Reporting Subsystem Data

Notification Items

Error report notices
Check-out notices

General Project Data

Project name
Team member names
Functional area responsibilities

System Parameters

File locations
Old source rename type
Old object rename type
Root file type
Default compile options
Default link options

Figure 14. Other Data Used by the System

CHAPTER VI

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE WORK

Summary and Conclusions

Programming tools and programming environments aid in the process of software development. A number of systems have addressed the problems of development of software for medium and large computer systems; similar solutions are needed for microcomputer software development. Software development using microcomputer workstations is a viable approach to the task of developing program products for microcomputer systems. There is a need for software management aids suitable for such an environment. A Software Management System has been presented which incorporates aids for the management of source files, for the management of the integration task, and for recording and tracking of error reports.

The source file access control subsystem has the job of maintaining the integrity of the source files. Parallel updates to code are prevented by requiring check-out of files prior to modification. A check-out key is used to

ensure that the file being checked in is an updated version of the copy which was checked out. Syntactic correctness of program source files is assured by placing modified files in a provisional status until compilation has been successfully completed. In addition, a history of check-outs is maintained. Useful reports can be produced from the information gathered by this subsystem.

The recompilation and relinking subsystem uses the derivation relationships to determine what action is to be taken when a source file changes. By performing the derivation of object code by compiling the source, it ensures consistency between these. Success or failure of this is then determined and communicated with the source file access control subsystem, so that files can be removed from the provisional check-in status. These types of derivations are performed incrementally, as the files are checked in. In contrast, the relinking of the system is "batched" to avoid redundant operations. The automation of this task frees the programmers of a burdensome task and also guarantees that it is done in a consistent, accurate manner. An additional advantage of the automation is that it provides a convenient way to manage the linking of a system with a complex overlay structure. The overlay structure is recorded in the database by means of the uses relationship.

The error reporting and tracking facility provides a structured communications medium between product developers

and product testers. It supports the idea of functional areas of responsibility and uses this information to aid in directing error reports to the right person. In addition, the ability to track progress in dealing with the reports provides the team leadership with information very useful for the management of the project. The system also maintains a history of errors reported and of the time taken to correct them. The reporting capabilities of this subsystem and those of the source file access subsystem also provide useful help to programmers as well as management.

Suggestions for Future Work

The implementation of the Software Management System is a starting point for future work. Several questions must be addressed as it is implemented. One interesting question is that of how to issue commands from within the program and then regain control after the completion of an operation (such a facility is needed for the recompilation and relinking subsystem).

Another question is what sort of shape should the user interface take? A fundamental issue concerns the method by which the user indicates what operations the system is to perform. The two principle alternatives are to make the system menu driven or to make it command driven. The menu approach makes all possible options visible to the user, but he may need to traverse a hierarchy of menus to get to the desired option. In a command driven system, the user must

be able to remember the available options and the syntax for the commands. On the other hand, the user can directly enter his commands and can shift from one type of action to another quite readily. The menu-driven approach seems more suitable for novice users, and perhaps also for experienced users when they desire to invoke a seldom-used feature of the system. The command-driven approach seems most suitable for those who are experienced in the use of the system. By adding the ability to move directly between options to the menu approach, we can have both the ease of use for the experienced user and the information needed for the novice. This approach sounds promising. Perhaps additional insight can be gained from the field of human factors engineering.

Once this decision is made, additional details must be worked out. For instance, if a menu-driven approach is taken, what options are available from each menu and what sort of menu hierarchy is to be used? Alternatively, for the command driven approach, what are the commands? What is the syntax to be used?

Additional details of the man-machine interaction must be determined as well. At various points the system will request several items from the user at one time. For instance, when entering an error report, the user is asked for a summary, a full description of the error, the severity of the error, the functional area, and so forth. It would be extremely frustrating to enter one field and proceed to the next, then realize that a mistake was made in the first

if no means were available for backing up and correcting the error. So in any situation in which multiple fields are entered on the screen, the user should be allowed to edit any of them until the information is correct. Another type of situation to avoid is that of requiring the user to repeatedly select a certain option in order to perform that task on different entities. So, for instance, when the user elects to check out files, it would be nice to allow him to check out as many as desired before proceeding to another option.

Another implementation detail is the selection of a database system to be embedded. Existing systems could be evaluated for suitability for this application. Some microcomputer database products are intended to be self-contained. They have an internal language and lack ability to interface with existing languages. Others are intended to be embedded within other program products. The systems could then be evaluated for efficiency and performance as well.

There are several ideas for enhancements to the system which come to mind. According to Pearson [26], provision for multiple version handling and configuration management is one of the more important facilities for any large-scale software development task. Providing such facilities would involve fairly extensive changes to both the source file access control and the recompilation and relinking subsystem.

Another enhancement which should prove very useful to a software development team is to expand the database to include information on calling structure of the program, the variable usage within the routines, and cross-reference information. Such a facility along with a corresponding question answering ability would aid not only the development group but would also provide useful documentation for the maintenance phase.

A third possible enhancement could take a number of forms. This extension would be to provide support for the early phases of software development. A requirements language or some type of design support tool would be a step in this direction.

This paper has presented a description of a system to support software development activities for a microcomputer environment. It is hoped that this work will stimulate other investigation in this area, and that the developers of microcomputer software will be provided with a more productive environment in which to operate.

SELECTED BIBLIOGRAPHY

[1] Bauer, H. A. and Birchall, R. H. "Managing Large Scale Software Development with an Automated Change Control System." Proceedings of the Second International Computer Software and Applications Conference (Nov. 13-16, 1978), 13-18.

[2] Bell, T. E., Bixler, D. C., and Dyer, M. E. "An Extendable Approach to Computer-Aided Software Requirements Engineering." IEEE Transactions on Software Engineering, Vol. SE-3, No. 1 (Jan. 1977), 49-60.

[3] Bianchi, M. H. and Wood, J. L. "A User's Viewpoint on the Programmer's Workbench." Proceedings of the Second International Conference on Software Engineering (Oct. 13-15, 1976), 193-199.

[4] Bratman, H. and Court, T. "The Software Factory." Computer, Vol. 8, No. 5 (May 1975), 28-37.

[5] Brooks, F. P. The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley Publishing Co., Reading, Massachusetts (1975).

[6] Campbell, R. H. and Richards, P. G. "SAGA: A System to Automate the Management of Software Production." AFIPS Conference Proceedings (May 4-7, 1981), 231-234.

[7] Chen, P. P. "The Entity-Relationship Model--Towards a Unified View of Data." ACM Transactions on Database Systems, Vol. 1, No. 1 (March 1976), 9-36.

[8] Cristofor, E., Wendt, T. A., and Wonsiewicz, B. C. "Source Control + Tools = Stable Systems." Proceedings of the Fourth International Computer Software and Applications Conference (Oct. 27-31, 1980), 527-532.

[9] Digital Research. LINK-80 Operator's Guide, Pacific Grove, California (1980).

[10] Digital Research. PL/I-80 Applications Guide, Pacific Grove, California (1980).

[11] Digital Research. PL/I-80 Language Manual, Pacific Grove, California (1980).

[12] Dolotta, T. A. and Mashey, J. R. "An Introduction to the Programmer's Workbench." Proceedings of the Second International Conference on Software Engineering (Oct. 13-15, 1976), 164-168.

[13] Feldman, S. I. "MAKE - A Program for Maintaining Computer Programs." Software - Practice and Experience, Vol. 9, No. 4 (April 1979), 255-265.

[14] Gillett, W. D. and Pollack, S. V. An Introduction to Engineered Software, Holt, Rinehart, and Winston, New York (1982), 3.

[15] Gutz, S., Wasserman, A. I., and Spier, M. J. "Personal Development Systems for the Professional Programmer." Computer, Vol. 14, No. 4 (April 1981), 45-53.

[16] Haberman, A. N. "System Development Environments." in Tools and Notions for Program Construction, Neel, D. (ed.), Cambridge University Press, New York (1982), 247-272.

[17] Herschman, D. PLINK-II User's Manual, Lifeboat Associates, New York (1981), 21-27.

[18] IEEE. "Tools Working Group." Proceedings of the 23rd IEEE Computer Society International Conference (Sept. 15-17, 1981), 353-359.

[19] Kernigan, B. W. and Mashey, J. R. "The UNIX Programming Environment." Software - Practice and Experience, Vol. 9, No. 1 (Jan. 1979), 1-15.

[20] Knobe, K. "Early Experience with MONSTR: a Software Maintenance Management Tool." Proceedings of the 23rd IEEE Computer Society International Conference (Sept. 15-17, 1981), 214-218.

[21] Knudsen, D. B., Barofsky, A., and Satz, L. R. "A Modification Request Control System." Proceedings of the Second International Conference on Software Engineering (Oct. 13-15, 1976), 187-192.

[22] Lanzano, B. C. "Loader Standardization for Overlay Programs." Communications of the ACM, Vol. 12, No. 10 (Oct. 1969), 541-550.

[23] Lemaitre, M., Lemoine, M., and Zanon, G. "SPRAC: A Computer Assisted Software Development System." in Tools and Notions for Program Construction, Neel, D. (ed.), Cambridge University Press, New York (1982), 329-345.

[24] McMahon, E. M. "A JOVIAL Programming Support Environment." AFIPS Conference Proceedings (June 7-10, 1982), 319-325.

[25] Pankhurst, R. J. "Program Overlay Techniques." Communications of the ACM, Vol. 11, No. 2 (Feb. 1968), 119-125.

[26] Pearson, D. J. "The Use and Abuse of a Software Engineering System." AFIPS Conference Proceedings (June 4-7, 1979), 1029-1035.

[27] Reifer, D. J. and Trattner, S. "A Glossary of Software Tools and Techniques." Computer, Vol. 10, No. 7 (July 1977), 52-60.

[28] Rin, R. A. "An Interactive Applications Development System and Support Environment." in Automated Tools for Information System Design, Schneider, H.-J., and Wasserman, A. I. (eds.), North-Holland Publishing Co., Amsterdam (1982), 177-213.

[29] Rochkind, M. J. "The Source Code Control System." IEEE Transactions on Software Engineering, Vol. SE-1, No. 4 (Dec. 1975), 364-370.

[30] STONEMAN. "Requirements for ADA Programming Support Environments", Defense Advanced Research Projects Agency, Arlington, Virginia (1980).

[31] Stuebing, H. G. "A Modern Facility for Software Production and Maintenance." Proceedings of the Fourth International Computer Software and Applications Conference (Oct. 27-31, 1980), 407-418.

[32] Teichroew, D. and Hershey, E. A. "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems." IEEE Transactions on Software Engineering, Vol. SE-3, No. 1 (Jan. 1977), 41-48.

[33] Teitelbaum, T. and Reps, T. "The Cornell Program Synthesizer: a Syntax Directed Programming Environment." Communications of the ACM, Vol. 24, No. 9 (Sept. 1981), 563-573.

[34] Teitelman, W. and Masinter, L. "The Interlisp Programming Environment." Computer, Vol. 14, No. 4 (April 1981), 25-33.

[35] Ullman, J. D. Principles of Database Systems, Computer Science Press, Potomac, Maryland (1980), 3,10-17.

[36] Waters, R. C. "The Programmer's Apprentice: Knowledge Based Program Editing." IEEE Transactions on Software Engineering, Vol. SE-8, No. 1 (Jan. 1982), 1-12.

[37] Willis, R. "DAS - An Automated System to Support Design Analysis." in Tutorial: Automated Tools for Software Engineering, Miller, E. (ed.), IEEE Computer Society Press, New York (1979), 105-111.

APPENDIX

SCHEME FOR THE SOFTWARE MANAGEMENT SYSTEM DATABASE

A brief discussion of the scheme for the system database is found in what follows. The term scheme is used here to mean the conceptual plan for the organization of the information (see Ullman [35]). An entity-relationship diagram (ERD) is used to illustrate that organization.

The entity-relationship diagram is a graphical representation of the organization of a database. It was first used by Chen [7]. An entity is something which can be uniquely identified. A relationship is an association between entities. In an ERD, it is not particular entities which are represented but rather entity sets. Similarly, the diagram illustrates relationship sets which exist between the entity sets. Rectangles are used to indicate entity sets; attributes of the entities are shown as ovals connected to those rectangles. The relationships are represented by diamond-shaped figures with edges connecting them with the associated entity sets.

For the sake of clarity, a somewhat simplified entity-relationship diagram of the system database is shown in Figure 15. The primary simplification is the omission of most of the attributes.

The meaning of the contains relationship is hopefully obvious. The developers relationship is intended to include testers as well as the designers and programmers for the project. It should be mentioned that in the case in which there is only one project using the Software Management System, both of these relationships can be implicit. Each of the team members is responsible for various functional areas.

The files entity set includes both source files and intermediate files derived from the source (such as object files). Now while ideally the actual contents of the files would form a part of the database, the most likely way to implement this would be to store only the filenames and types in the database. The contents would be stored using the normal CP/M file mechanisms. This allows the system the freedom to use existing compilers and other tools without modification. The executable segments are also files but are represented by a different entity set because we wish to store slightly different information regarding them. In addition, the uses relationship applies only to executable segments. The derives relationship is used both as a recursive relationship within the files entity set and to connect the file and executable segment sets.

A number of the relationships illustrated in the diagram are not actually considered as separate items but as one of the fields of an entity set. For both the file_out and user_out relationships, a field is found in the

check-out record which indicates the association with a member of the corresponding entity set. The reason for this is that when these records are archived, this information should be stored as well, and we wish to keep it together in the database. Similarly, the cancels relationship is stored in the cancellation record, and both the reported and the deficiency relationships are indicated by fields found in the error report.

The notification items are not illustrated in the diagram. These items connect either an error report or a check-out with a team member who is to be notified of some event. Nor are the system parameters shown. These parameters form a rather diverse group of individual items. Because of their nature, there may be some question as to whether they should be located in the system database or stored in some other way. Table II lists the entities and the corresponding fields; Table III lists the relationships, the entity sets connected, and fields for those which are to be represented separately from the associated entity sets.

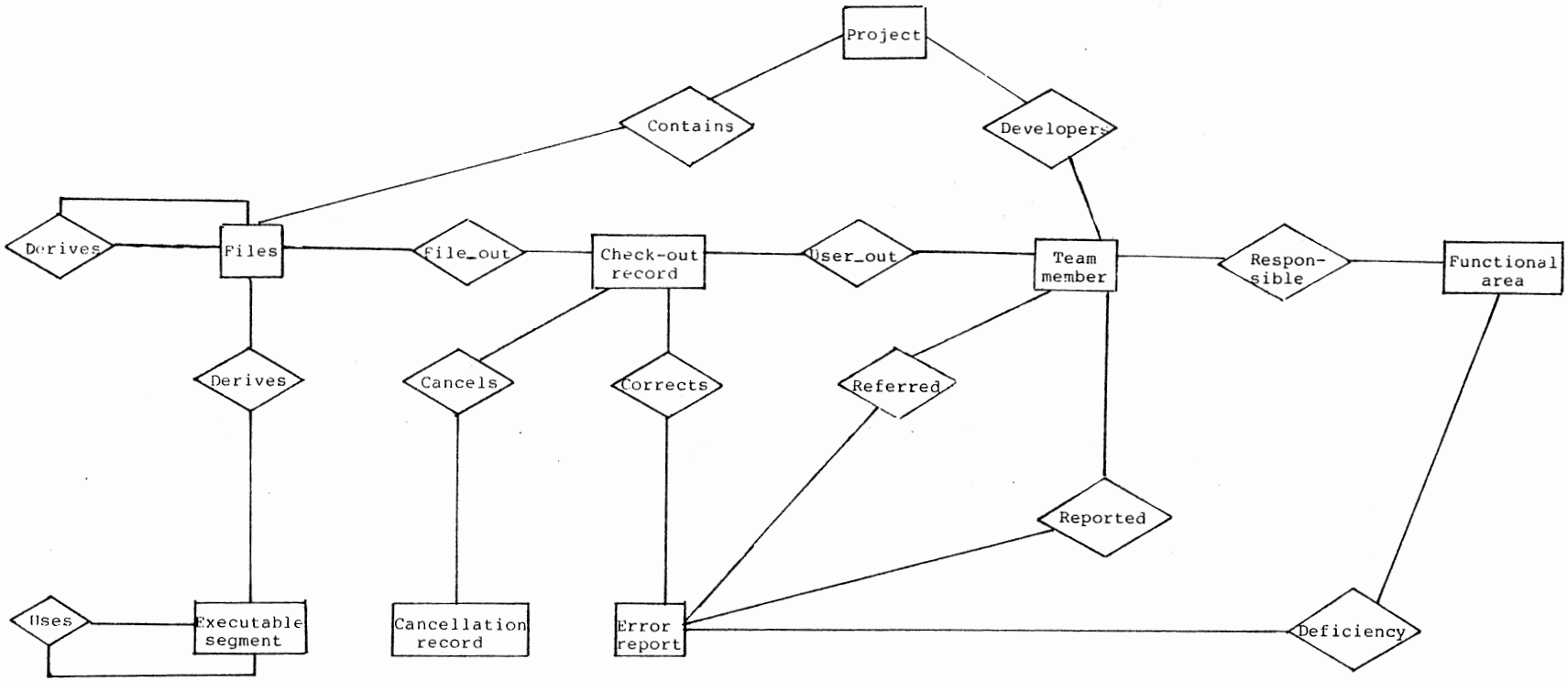


Figure 15. Entity-Relationship Diagram

TABLE II

ENTITY SETS IN THE SYSTEM DATABASE

Entity sets	Fields
Project	(project_name)
Team_member	(member_name)
Functional_area	(area_name)
File	(filename,type,modification_flag, compilation_status)
Executable_segment	(segment_name,type,load_address, module_top)
Check-out record	(check-out_key,filename,type, member_name,reason,date_out,date_in)
Cancellation record	(check-out_key,canceller,reason,date)
Error report	(report_number,summary,description, area_name,severity,reporter,date, correction,status,date_closed)
Error report notice	(type,member_name,report_number)
Check-out notice	(type,member_name,check-out_key)

TABLE III

RELATIONSHIPS IN THE SYSTEM DATABASE

Relationships	Related Entities	Fields
Contains	Project ↔ files	(Project_name, file_name, type)
Developers	Project ↔ team members	(Project_name, member_name)
File_out	Check-out ↔ file	
User_out	Check-out ↔ team	
Corrects	Check-out ↔ error report	(Check-out_key, report_number)
Cancels	Cancellation ↔ check-out	
Reported	Error report ↔ team member	
Referred	Error report ↔ team member	(Report_number, member_name)
Deficiency	Error report ↔ functional area	
Derives	File ↔ file	(Source_name, source_type, derived_name, derived_type, command)
[Derives]	File ↔ executable segment	(Source_name, source_type, derived_name, derived_type, [link])
Uses	Executable segment ↔ executable segment	(File_name, type, used_name, type)

VITA[2]

John Charles Warren

Candidate for the Degree of

Master of Science

Thesis: A SOFTWARE DEVELOPMENT SUPPORT SYSTEM FOR A MICROCOMPUTER ENVIRONMENT

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Dallas, Texas, January 18, 1953, the son of Mr. and Mrs. James D. Warren.

Education: Graduated from Norman High School, Norman, Oklahoma, in May, 1971; received Bachelor of Music Education degree from the University of Oklahoma, Norman, Oklahoma, in December, 1975; received Bachelor of Science degree in Mathematics from the University of Oklahoma in July, 1976; completed requirements for the Master of Science degree at Oklahoma State University, Stillwater, Oklahoma, in July, 1983.

Professional Experience: Graduate teaching assistant, Department of Computing and Information Sciences; Oklahoma State University, Stillwater, Oklahoma, August, 1980 to May, 1981. Applications Programmer, Time Management Software, Stillwater, Oklahoma, May 1981 to May 1983. Member of ACM since 1981.