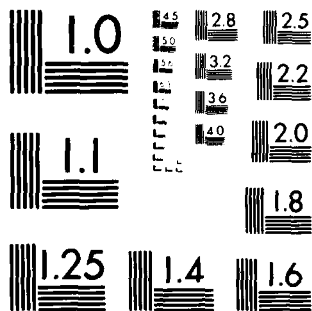
AD-A111 567 FORD AEROSPACE AND COMMUNICATIONS CORP PALO ALTO CA W--ETC F/G 17/2
KSOS SECURE UNIX IMPLEMENTATION PLAN (KERNELIZED SECURE OPERATI--ETC(U)
DEC 80 MDA903-77-C-0333
UNCLASSIFIED WDL-TR7799-REV-2 NL

1 of 1
AD
A111567

END
DATE
FILMED
3-82
DTIC



MICROCOPY RESOLUTION TEST CHART

AD A111567

WDL-TR7799
Revision 2
December 1980

SECURE MINICOMPUTER OPERATING SYSTEM (KSOS)

SECURE UNIX IMPLEMENTATION PLAN

Department of Defense Kernelized Secure Operating System

Contract MDA 903-77-C-0333
CDRL 0002AC

DTIC
ELECTE
MAR 0 3 1982
E

Prepared for:

Defense Supply Service - Washington
Room 1D245, The Pentagon
Washington, DC 20310

DTIC FILE COPY

Ford Aerospace &
Communications Corporation
Western Development
Laboratories Division

3939 Fabian Way
Palo Alto, California 94303

Approved for public release; distribution unlimited.

82 03 03 032

NOTICE

The Department of Defense Kernelized Secure Operating System (KSOS) is being produced under contract for the U.S. Government. KSOS is intended to be compatible with the Western Electric Company's UNIX™ Operating System (a proprietary product). KSOS is not part of the UNIX license software and use of KSOS is independent of any UNIX license agreement. Use of KSOS does not authorize use of UNIX in the absence of an appropriate licensing ageement.

This document, furnished in accordance with Contract MDA 903-77-C-0333, shall not be disclosed outside the Government and shall not be duplicated, used, or disclosed in whole or in part for any purpose other than to evaluate the contractor's performance of Phase I of the contract; upon completion of Phase I of the contract, the Government shall have the right to duplicate, use, or disclose the data to the extent provided in the contract.

The contents of this document shall be handled as proprietary information until 5 April 1978. After that time, the Government may distribute the document as it sees fit.

UNIX and PWB/UNIX are trade/service marks of the Bell System.

DEC and PDP are registered trademarks of the Digital Equipment Corporation, Maynard, MA.

COPY

TABLE OF CONTENTS

1. Programming Techniques 1
1.1. Programming Protocol 1
1.1.1. Language Usage Standards 2
1.1.2. Structure 2
1.1.3. Information Hiding 3
1.1.4. Readability 4
1.1.5. Maintenance 4
1.2. Engineering Models 4
1.2.1. Evaluation Models for High-Risk Components 5
1.2.2. Programming Standards 5
1.3. Product Level Specifications 6
1.3.1. High-Level System Documentation 6
1.3.2. Hierarchical Design Language (HDL) 6
1.4. Specification-Design-Implementation Correspondence 7
1.5. Formal Code Inspection 7
1.5.1. Quality Assurance and Education 9
1.5.2. Security 10
1.5.3. Other Applications 10
1.6. Implementation Tools 10
1.6.1. Interactive Program Creation Editor 10
1.6.2. Source Level Configuration Control 10
1.6.3. Documentation 11
1.6.4. Design/Implementation Continuity 11
1.6.5. Source Back-up Plan 11
1.6.6. Development Support 11

2. Implementation Activity Integrity 13
2.1. Physical Plant 13
2.2. Design Security 13
2.3. Code Security 14

3. Verification Impact 16
3.1. Source-Level Configuration Control 16
3.2. Modification Request System 16
3.3. Programming Standards Enforcement 16
3.3.1. Formal Code Inspection 16
3.3.2. Production Coding 16

4. Performance Impact 17
4.1. Formal Code Inspections 17
4.1.1. Algorithm Review 17
4.1.2. Programmer Productivity 17
4.2. Performance Monitoring 17
4.2.1. Self-Hosted Measures 18
4.2.2. Satellite Processor 18

5. Testing 19

5.1. Module Tests 19
5.2. Thread Testing 19
5.3. Cumulative Testing 20
5.4. Required Testing 20
5.5. Stress Testing 20
5.5.1. Shell Script 20
5.5.2. Externally Generated Load 20

6. External Configuration Management 21
6.1. Supported Configurations 21
6.2. Automated System Generation 21
6.3. Secure Delivery and Installation 21
6.4. Maintenance and Support 21

7. REFERENCES 23

APPENDICES

A. Hierachical Design Language
B. Programming Standards
C. Program Development Tools
D. Documentation Standards

Accession For		
NTIS GRA&I		X
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution/		
Availability Codes		
Dist	Avail and/or Special	
A		

KSOS IMPLEMENTATION PLAN

Ford Aerospace & Communications Corporation
Western Development Laboratories
Software Technology Department
3939 Fabian Way
Palo Alto, California 94303

ABSTRACT

This report presents a discussion of the implementation plan for the KSOS project. Programming techniques, implementation tools, implementation activity integrity, impact of techniques on verification and performance, testing, and external configuration management are discussed.

The implementation plan developed in the sections that follow is intended to provide the reader with an understanding of the implementation methodology that Ford Aerospace & Communications Corporation (FACC) will use during the detailed design and coding of KSOS. This report is divided into major sections that discuss programming techniques, FACC programming support tools, verification and performance implications of FACC implementation methodology, implementation security, testing plans, and configuration management.

1. Programming Techniques

1.1. Programming Protocol

This section discusses the programming protocol to be used during the development of KSOS. A programming protocol is the creative use of standard language constructs within a management defined and enforced structure. This structure will provide style standards for all code produced by the FACC implementation team. Engineering models will be developed in selected high-risk areas of the project in order to provide quantitative performance information for use in detail design decisions. FACC plans to specify program design in a hierarchical design language (HDL) in lieu of flowcharts. HDL is an FACC product that runs under UNIX*tm. Documentation is provided in Appendix A. Formal code inspection [Fagan, 1976] will be used as the vehicle for quality assurance, style enforcement and programmer education.

1.1.1. Language Usage Standards

The subject of standards is generally unpopular with programmers because of the notion that standards stifle creativity. Well-selected standards do just the opposite, since the programmer's creativity need not be spent making mundane stylistic decisions. Standards also have a positive effect on reliability because they encourage uniform programming style. An illustration of the type of programming standards to be applied in the KSOS project is given in Appendix B, which discusses standards for writing C programs. The necessary changes to embrace the KSOS programming language (KPL) will be made to the programming standards document when the KPL is decided. Drastic alteration of the standards is not expected. With the formal inspection procedure that FACC will use, adherence to programming standards will be ensured. A discussion of this inspection process is presented in Section 1.5.

Programming standards are not viewed as absolute rules. Any proposed deviation from these guidelines can be publicly scrutinized and approved, if justified. A set of standards will be drawn up once the programming language for kernel implementation is determined. These standards and other, similiar materials will be collected together into a KSOS Programmer's Notebook. These notebooks will be distributed to all project members, and will be periodically updated. Wherever practical, automated tools for formatting, editing, archiving, and auditing will be used. (See Section 1.5 of this document for additional information on implementation tools.)

1.1.2. Structure

Each procedure will contain a standard header block that specifies the information shown in Figure 1-1.

```
MODULE NAME:  name.c
FUNCTION:  Synopsis of operation.
ASSUMPTIONS:  Limits, etc.
ALGORITHM:  References & explanation.
PARAMETERS:  Input & output.
RETURNS:  Output value.
GLOBALS:  Variables & their meaning.
MODULES CALLED:  Library & programs.
CALLED BY:  What programs use this one.
HISTORY:  Who, when, what, why.
  AUTHOR:
  DATE:     Jan 28 1978
  VERSION:
```

Figure 1-1. Standard Module Header Contents

This header block forms the basic entry in a Module Development Folder that is maintained for each module throughout the development and testing cycle. FACC plans to maintain online versions of the Module Development Folder as a part of the program source.

The procedure format will be maintained with a standardized lexical reformatter/indenter. All data declarations will be commented in the source code, since understanding of the data abstraction is central to understanding the program. The executable statements of each module should not be excessively commented. A reasonable guideline is to read the code (at formal inspections) and insert comments at points where questions arise. Commentary should be notes to the reader, not a description of logic that is fully described by the code itself. Accuracy of the logic description in the standard module header will be ensured by the formal inspection process.

1.1.3. Information Hiding

One of the key style principles will be to delay program binding as long as possible, consistent with efficiency considerations. SRI International's Hierarchical Design Methodology (HDM) [Neumann, 1976] being used for KSOS is based on the notion of successive levels of information hiding and delayed binding. In particular, access functions will be used to manipulate global status information rather than providing direct access to global data structures. The principle of decision, or information hiding [Parnas 72], is central to the design of a provably secure system. This principle also serves to improve reliability and reduce the system's sensitivity to change. A structure designed according to information-hiding principles can often result in a module division that is quite different from conventional decompositions [Parnas 77].

Each design decision about data representation or event sequencing is "hidden" within a module. As a result, assumptions about data structures are confined to a single module and a change in these assumptions affects only one module. Such a change is said to be transparent. As a result of information hiding, system maintainability is expected to be significantly better than that normally available with globally-known system tables.

The cost of rigorous information-hiding may be paid in reduced performance, since information accesses may be converted into procedure invocations rather than direct structure references. This problem may be avoided if the programming language provides inline code-generation facilities; invocation of an inline-procedure must be indistinguishable syntactically from actual procedure calls. FACC accepts the potential reduction in "absolute efficiency" that results from decision-hiding as a necessary price to pay for resilient software that has low sensitivity to structural modification. Violations of the "information-hiding" principle will be explicitly noted in design and code inspections. It is anticipated that the only exceptions to "information-hiding" will be made in program areas found to be on critical timing paths.

Quantative assessment of the performance cost of information hiding is not possible at the present time. When overly expensive hiding is discovered, redesign of data access mechanisms to distribute information structure knowledge to time-critical users will be undertaken when necessary. However, conventional wisdom about program behavior indicated that execution inefficiencies tend to be clustered (10% code -> 90% time) and that significant inefficiencies can be isolated at relatively low expense with proper instrumentation of the program.

1.1.4. Readability

Programming style that emphasizes readabilty and simplicity will be required throughout the KSOS project. A sample of these standards for C programs is presented in Appendix B. Formatting standards will be enforced by means of indenters such as that described in Appendix C.

1.1.5. Maintenance

Life-cycle cost studies indicate that average computer programs have a maintenance cost that is far greater than the development cost. FACC will emphasize maintainability during the code development process in order to minimize the long-term costs associated with KSOS support. Transparency assists maintenance because changes are localized to a single module. Since the kernel must be reverified after any modification, localization is important to change-containment and reduction of reverification costs. Details of FACC's maintenance procedures are discussed in a separate document, the Support and Maintenance Plan.

1.2. Engineering Models

The KSOS project presents significant development challenges on several fronts. Since the implementation effort must address the possibly conflicting objectives of secure operation and high performance, FACC believes that engineering models of significant portions of the security kernel and the UNIX*tm emulator will need to be built before detailed design specifications can be drawn up. FACC intends to build enough of the kernel to evaluate timing limitations of processes and file mechanisms proposed in the kernel design. Instrumentation of the engineering models will allow quantitative performance measurement and an early basis for performance estimation of the production system.

In his book of essays on software engineering, [Brooks 75] argues that the initial system developed in a project should not be delivered, but used as a test-bed for the system to be delivered. In a similar vein, [Royce 70] advises commitment of about one third of the project's time budget to "doing it twice", with the first implementation serving as a simulation of the deliverable (second) version. Although FACC plans to devote far less resources than Royce suggests, the basic principal will be followed in KSOS. FACC feels that such a strategy offers the Government far less risk than proceeding ahead with no internal engineering model efforts.

The two KSOS prototypes have been studied (and will be the object of much future study) to provide guidance in the FACC design effort. FACC believes that both prototypes fall short of the requirements for a production version of KSOS. It is clear that simple extensions to either kernel will not satisfy all requirements for a production system (including performance). Thus, FACC/SRI have embarked on a course that seeks to preserve the best features of each system in a unified design.

1.2.1. Evaluation Models for High-Risk Components

An engineering model is a skeletal program that implements crucial algorithms for the purpose of timing, storage, and logic validation. Before embarking on a detailed design effort, engineering models of the most critical components of the kernel and the UNIX*tm emulator will be developed. The results of these modelling efforts will be used to guide the generation of detailed design specifications. Engineering models, while developed as disposable software, will affect the structure of the detailed design and the the style for the full-scale implementation that follows. The resulting programs will serve as models for subsequent full-scale implementation efforts.

Because of the importance of feedback to the design process, the engineering model development will be undertaken very early in Phase II of the KSOS project. The model development team will proceed with a top-down design and development effort based on the formal specifications that result from Phase I. These engineering models will provide the quantative performance basis for a final design that assures adequate speed of operation of the production version of KSOS. This throw-away code is expected to provide quantative data on algorithm performance. In particular, FACC expects to implement engineering models of the Security Kernel and performance-sensitive portions of the UNIX*tm emulator. Although a great deal about design approaches has been learned from the MITRE and UCLA security projects, FACC does not believe that detailed design of an assured-performance system based on an extended MITRE/UCLA design can be undertaken honestly without direct modelling efforts.

The throw-away code produced in the engineering models is expected to reduce the cost of the project by exposing infeasible implementation strategies early in the detailed design phase. Feedback to the designer of data from a "live" system can be extremely valuable in disspelling "folklore."

1.2.2. Programming Standards

The engineering models are also expected to serve as a test for the administrative aspects of the KSOS effort: configuration control, design and code inspections, and documentation methods that FACC will use during detailed design and subsequent coding. FACC believes that we can make better use of our resources if we track them carefully. Furthermore, the use of inspections at the engineering model stage allows us to get feedback on the inspection process itself, as well as train project personnel in its use. The engineering models are expected to provide educational benefits to the whole development team. Overview documents that are understandable, informative and current will be used to document each model. These overview documents will be distributed as updates to the Programmer's Notebook. Every detailed designer must have an understanding of the design based on a world-view of project objectives and design trade-offs. Without such a consistent frame of reference, detailed design can easily compromise performance, security, or maintainability.

1.3. Product Level Specifications

1.3.1. High-Level System Documentation

Subject to Government approval, FACC plans to use a hierarchical design language (HDL) in lieu of flowcharts. System documentation requirements vary with the level of detail. While graphical representations of program structure have been required by tradition, there has been considerable discussion of the utility of flowcharts in recent years.

> Flowcharts show the decision structure of a program, which is only one aspect of its structure. They show decision structure rather elegantly when the flowchart is one page, but the overview breaks down badly when one has multiple pages, sewed together with numbered exits and connectors. The one-page flowchart becomes essentially a diagram of program structure and of phases or steps. [1]

Shneiderman's recent studies conclude that

> Our experiments have not demonstrated the utility of detailed flowcharts in program composition, comprehension, debugging, or modification. [2]

FACC plans to use a hierarchical design language (HDL) to represent all levels of program design. Where a very high level structure overview appears useful, structure charts may be produced.

1.3.2. Hierarchical Design Language (HDL)

The Hierarchical Design Language (HDL) is a tool that aids design and documentation of systems of programs. HDL is modelled after the Caine, Farber and Gordon, Inc. (CFG) Program Design Language system. An HDL design is written in "structured English", with control represented in terms of C-language control constructs. HDL is a program developed by FACC for use on the UNIX*tm system. Appendix A contains documentation for HDL.

Input to the HDL processor consists of files that are created and maintained with the program creation editor (discussed below) that runs under UNIX*tm. Output of HDL is a working design document consisting of a table of contents, a lexically indented listing of the procedure designs, and a cross-reference of procedure calls and global variable usage. Procedure calling trees can be produced from HDL source through use of "nn", another FACC developed tool. Appendix C documents a set of representative program development tools.

HDL output is a replacement for flowcharts. It is machine readable, easier to process, easier to maintain, and easier to read. Because HDL is textual, it channels programmer energies away from mechanical drawing and into productive, technical design. Like a flowchart, HDL can be written at the level of detail appropriate to the current design level. A designer can start with a few pages of HDL that give general system structure. As the hierarchical design expands, the design specification can be expanded precisely and consistently.

---

1. Brooks, 1975, p 168
2. Shneiderman, 1977, p 381

The development of HDL is part of FACC's continuing IR&D in software technology development. FACC has chosen to use HDL rather than PDL because HDL provides:

* Enhanced control structure notation.
* Efficient operation under UNIX*tm.

HDL or associated design-aid tools will provide greater utility and flexibility than CFG's PDL processor.

1.4. Specification-Design-Implementation Correspondence

Synchronization of the formal specification, the HDL design, and the KSOS programming language (KPL) implementation are of significant concern to FACC. The classical specification to design to implementation (i.e., SPECIAL -> HDL -> KPL) evolution requires two levels of interface and translation. FACC plans to derive HDL specifications from the SPECIAL representation of the formal specification for the component. Realization of the KPL implementation will also be made from the formal specification, using the HDL representation as commentary and guidance. The SPECIAL representation will be used as the controlling element at all times.

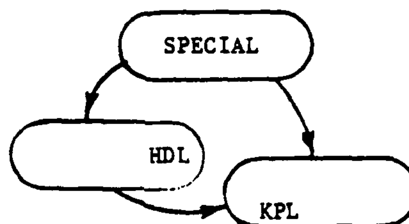


Figure 1-2. Specification-Design-Implementation Correspondence

1.5. Formal Code Inspection

Successful management requires planning, measurement and control. All these elements are especially significant in the software development process, where there is a tradition of highly individualized program development protocols. Under the traditional regimen, the manager is usually limited to planning, with control of the project an ellusive, unrealizable goal because effective measurement is unattainable. Measurement implies known standards coupled with careful inspection. In short, consistent and vigorous discipline is required in setting and enforcing standards. Since establishment of blindly-enforced rules can stifle programming work, a clearly understood mechanism for justifying "rule-breaking" should be specified.

Numerous organizational techniques have been applied to the program development process in order to improve software quality. Among these techniques, variants of the Chief Programmer Team [Brooks 75, Baker 71] have been widely discussed and utilized. Major elements of each of these efforts have been limitation of team size and deep involvement of the team leader in critical details of system design and implementation. Programmer attitude towards their products have also been discussed extensively [Weinberg 71, Aron 75, Yeh 77]. Use of structured walk-throughs and teams that naturally practice "egoless" programming has yielded good results in some development efforts. FACC proposes to take the effort a step further, applying techniques of formal code

inspection [Fagan 76] to improve the quality control in the code development process.

The inspection process is divided into the five general types of activities shown in Figure 1-3. The functions of each activity will be explained in detail in the Programming and Specification Standards that will be prepared as a part of Phase II of the KSOS project.

OPERATION	DESCRIPTION	OBJECTIVES	COMMENTS
PLANNING	Establishing the schedules, designating the participants, and requesting the inspection material	To assure that schedules and participants are established and materials available	Occurs before inspection material is available
PREPARATION	Participants use the distributed material to prepare for the inspection	Allows participants to come to the inspection ready to find errors	Occurs after the material is distributed and before the inspection
INSPECTION	Formal process of inspecting the distributed material	To find errors and ommissions	A meeting led by the Moderator; held after all preparation is complete
REWORK	The process of fixing those problems identified at the inspection	Provide time for identified problems to be corrected	Moderator determined necessity of reinspection based on number and magnitude of problems
FOLLOW-UP	Maintain accountability for problem resolution	To insure that all problems are satisfactorily resolved	Inspection exit criteria require resolution of all problems; on exit, update inspection Data Base

Figure 1-3. Inspection Activities and Objectives

1.5.1. Quality Assurance and Education

KSOS stands or falls on the accuracy of the translation of verified specifications into code. FACC proposes to use an implementation language that provides a basis for subsequent automated verification of the code-to-specification correspondence. Code inspection provides tangible mechanisms for programmer education in matters of style and language usage, as well as a management tool for identification of problem areas and enforcement of programming standards.

Review (inspection) takes place in a formal atmosphere, with a specific review objective of finding errors. Note that redesign is not carried out at review meetings. The review process should include people in the following roles:

a. Moderator: A competent programmer who need not be an expert in the program being inspected. The moderator manages the inspection team and offers leadership. He is the inspection coach and resource manager.

b. Designer: The programmer responsible for the design.

c. Implementor: The programmer responsible for translating the design to code.

d. Tester: The programmer responsible for writing and executing test cases for the design and implementation.

When the roles are not clear-cut, it is often advantagous to bring in others to fill each role. It has been found that four people constitute a good sized inspection team. However, if the code involves a large number of interfaces, programmers of code related to these interfaces may profitably be involved in the inspection process. The inspection process will not be used for programmer evaluation (no matter how tempting it may be to management) since the programmer evaluation mode destroys the candor of an inspection and turns it into an inquisition or a bland rubber-stamp peer approval exercise.

Time for inspections and resulting rework of code will be scheduled and managed with the same attention given to all other project activities. Two inspections are scheduled for each module. The first, a Design Completion Review, is scheduled when detail design has reached a level where each design statement is estimated to correspond to three to ten implementation statements. The second inspection, the Code Completion Review, takes place after the first diagnostic-free compilation of the module is obtained. These are checkpoints in the development process through which every module must pass.

FACC cannot over-emphasize the importance of programmer education. Since KSOS is an advanced development project, with code to be written for a yet-to-be-implemented compiler, FACC/SRI must provide an effective mechanism for programmer education and quality assurance. We believe that a program of formal design and code inspection provides the needed environment.

1.5.2. Security

The use of the formal inspection program will enhance the security of the development process by exposing each module to critical review by a team of knowledgeable inspectors. If simplicity and visibility of code is assumed, introduction of specious or subvertible code would require collusion or deception of the entire inspection team, a highly unlikely occurrence.

1.5.3. Other Applications

Documentation and testing are as important to the success of the KSOS project as coding. We therefore plan to exercise the same quality control on all these activities. Documentation will be subject to formal reviews at the outline and draft stages; test planning will likewise be subject to design and implementation reviews.

1.6. Implementation Tools

KSOS will be implemented in the environment provided by the Programmer's Workbench (PWB) [Ivie 77, Dolotta 76]. This UNIX*tm-based program development, management, maintenance, and testing tool provides the facilities needed to carry out the complex development and maintenance activities required by the KSOS project. In addition to the facilities available through PWB, FACC has implemented a set of development tools as a part of its on-going IRAD software engineering efforts. A discussion of FACC development tools is provided in Appendix C.

1.6.1. Interactive Program Creation Editor

An editor that is language-sensitive, understanding syntax and limited semantics, can enforce programming and documentation standards as well as provide language-dependent formatting and text generation capabilities. The FACC program creation editor (PCE) checks for balanced parenthesis, generates standard module headers, automatically indents program text lexicographically, and contains a mechanism for rapid language construct insertion. In addition, the PCE contains a number of editing commands (a superset of those provided by the UNIX*tm editor, "ed").

1.6.2. Source Level Configuration Control

The development schedule for KSOS requires that several programming teams will be concurrently writing and testing portions of the system. In order to provide an auditable system at every phase of implementation, configuration control will be enforced through use of an automated library control and maintenance system. FACC will implement source-level configuration control through the Source Code Control System (SCCS) [Rochkind 75] that is part of the Programmer's Workbench (PWB/UNIX*tm).

Use of SCCS will provide a complete audit trail for systems development, repeatable incremental systems, and the basis for subsequent maintenance of the developed system. SCCS will be used for the system integration library as well as individual development libraries. Each version will receive a unique version number. This number will be used for inspections, modification requests, and documentation. The SCCS will be used to control all forms of machine-readable text created or maintained with source editors.

1.6.3. Documentation

Machine readable document source will be generated for all non-pictorial documents required by the KSOS project. All such prose-form documentation will be produced under configuration control. Sample documentation standards are discussed in Appendix D. Configuration control will be achieved by use of project-standard documentation macros for use with troff/nroff and formal documentation inspections.

1.6.4. Design/Implementation Continuity

The FACC/SRI design and specification team assembled in Phase I of the KSOS project will continue to exist as an implementation team during Phase II, with most personnel assuming positions as detailed designers and lead programmers for the implementation phase.

Figure 1-4 shows the proposed top-level structure for the KSOS Phase II implementation team. All members are currently participating in the Phase I design specification effort. Personnel named in Figure 1-4 will have responsibility as team leaders in their respective functional areas. All personnel in the KSOS project will be available to work as members of the various teams. The Implementation Manager will report directly to the KSOS Program Manager, Dr. M. S. Pliner. Dr. E. J. McCauley will continue to provide technical support to the KSOS project throughout Phase II; he will spend approximately half of his time on the KSOS project during Phase II, continuing in his role as principal engineer and technical adviser to the project.

1.6.5. Source Back-up Plan

All implementation source and documentation files will be maintained in machine readable form on the GFE PDP-11/70. In order to ensure that data loss due to a hardware, software, or human error will not adversely affect the project, a back-up archival tape system will be maintained. This system will be entirely separate from any secure copy system. Should this back-up be used to restore part or all of the source files, a verification check will be made of the differences between the restored version and the last secure copy. Backup copies of development files will be made on a daily basis with a weekly tape being saved for one month and the last weekly tape of the month being saved for the duration of the project. This mechanism will provide both an immediate back-up and an archive extending over the life of the project. The weekly and monthly save tapes will be stored at a different secure location to reduce the chances of loss due to a physical plant disaster. In addition to the above measures, the PWD SCCS will provide a running file of DELTAs for all configuration-controlled software.

1.6.6. Development Support

A DEC PDP-11/45 running PWB/UNIX*tm will be used to support the KSOS effort at no cost to the Government. This machine will be equipped with 124k of memory, a large disk, tape drive, printer, and at least 8 terminals. It will be connected by a dedicated line to the SRI DECsystem 20 and will be able to access the specification and verification tools available on the SRI machine. The PDP-11/45 will be located in a controlled environment near the

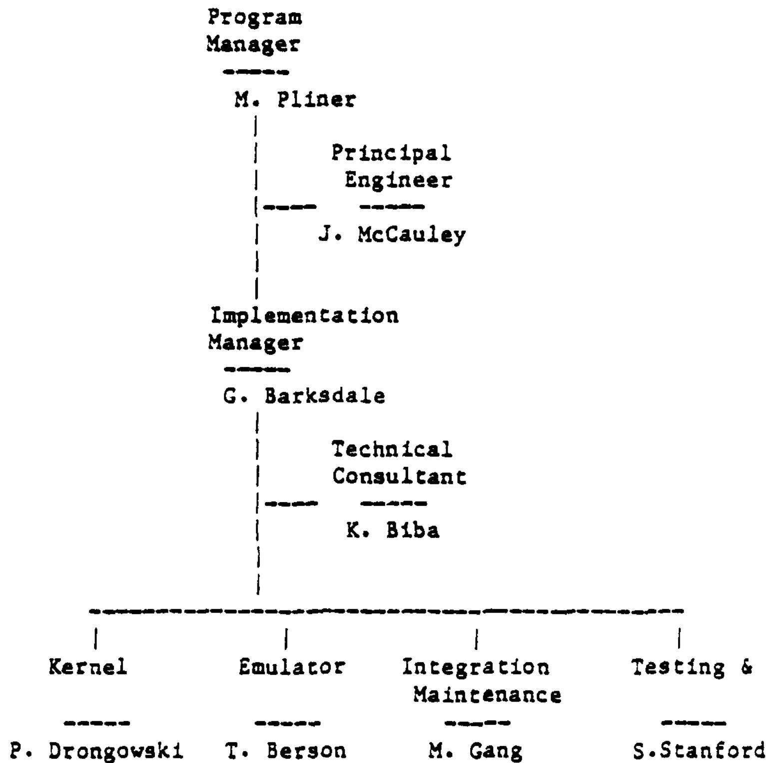


Figure 1-4. KSOS Implementation Organization

GFE PDP-11/70 and will be available as an alternate development machine in the event of extended GFE machine hardware failure. When secure operation of either PDP-11 is required, all communications lines will be disconnected from the computer system.

2. Implementation Activity Integrity

FACC provides a physical plant appropriate for protecting and maintaining the KSOS development effort. This section also discussed techniques for secure configuration control of the formal specifications for KSOS and the resulting source code.

2.1. Physical Plant

FACC is located on a 52 acre site in Palo Alto, California. Floor space totals more than one million square feet, of which more than 85,000 square feet are vaulted areas suitable for intelligence systems. There are raised floor equipment areas that can be run from UNCLASSIFIED through TOP SECRET.

FACC has TOP SECRET facility clearance. The cognizant government security office is: DCASR, (D&FM) San Francisco, 866 Malcolm Road, Burlingame, CA 94010.

FACC has a proprietary Guard Force. Plant guards and five video entrance booths monitor all entrances to FACC on a 24-hour basis, 7 days a week. The control console for the video booths is located in Plant Protection. Identification is made by photographic identity badges. Preemployment, as well as postemployment, investigations on all FACC personnel are conducted by a FACC employed professional investigator. Visitor Control and Master Document Control are integrated units within Government Security Operations. Document Control custodians are assigned custodial duties for Department of Defense classified material in FACC Master Document Control and the FACC Program Office.

FACC has a secure communication channel through an AUTODIN Terminal. When necessary, as directed by customer requirements, the Armed Forces Courier Service is available. Delivery and pickup are on Monday, Wednesday, and Friday. The courier station is located in the Presidio of San Francisco. The U.S. Postal Service can also be utilized for registry and delivery of classified material. All such material, with the exception of special intelligence information, is delivered directly to Master Document Control. It is opened there, and the material is brought into the accountability system.

2.2. Design Security

FACC will fully comply with both the letter and the spirit of RFP Paragraph J-4 and the applicable DoD security policies. The master copy of the design and code will be maintained at the SECRET level. Copies can be treated as UNCLASSIFIED. FACC proposes to accomplish this in the following way. The master copy of the design (i.e., the formal specifications in SPECIAL) will be maintained on a secure system. Under ARPA-TTO sponsorship, a secure TENEX is available at Moffett-ARC, located approximately three miles from FACC. This system regularly provides a SECRET-level time-sharing environment. This system will be used for maintaining the master copy of the design. Only cleared personnel have access to the Moffett-ARC system.

As stipulated earlier, all designing and coding of KSOS will be done by people with at least SECRET clearances. A copy of the design will be moved to the SRI TOPS-20 system as needed. There, the automated tools for design support and analysis will be applied. Should these tools discover design flaws,

the design will be changed to eliminate them. The master copy of the design will be kept at the Moffett-ARC. All updates to the design will be made in a controlled manner by cleared FACC personnel. By retaining every old master copy, a complete secure audit trail of the evolving design is assured.

A Memorandum of Understanding between FACC and the Moffett ARC will be executed to provide for services and access procedures for FACC personnel to the Moffett-ARC. This access will provide secure storage for master copies of the KSOS Specification sources that were developed in an unclassified environment. The procedure is designed to allow controlled, infrequent access to tapes stored at the ARC by FACC personnel.

1. By standard clearance transfer procedures, FACC will forward names to ARC to establish an access list. All personnel will be cleared for SECRET access. ARC will issue photographic identification badges to FACC personnel to facilitate base entry and computer terminal access to the ARC PDP-10.

2. At least 24 hours prior to FACC use ARC facilities, a netmail "Use Request" sent to ARCSYS@I-4. The "Use Request" will list the personnel involved, time of use, serial numbers of tapes to be used, and mode of tape use (read, write, deposit, remove). This request will be acknowledged and approved/denied by netmail to the originator of the "Use Request".

3. Tapes at the ARC will be stored in a controlled-access storage area. Each access will be authorized by the netmail "Use Request" previously discussed. An access by that lists authorized FACC users and contains each approved use request will serve to control access. FACC will format the "Use Request" so that signature start time and finish time are indicated. The form will then be self-prompting.

4. In addition to tape storage, the ARC will be used to compare two source tapes and list differences between the tapes. The difference list will be used by FACC to verify that all differences between the two tapes are authorized. The difference list will be treated as a SECRET-level document and will be sent from the ARC to FACC as a controlled document.

5. FACC's use of the ARC will require ARC local terminal access, machine time for installation and operation of the file-compare program, tape and disk access, and basic system familiarization. Since the ARC has only a single 9-track 800 bpi tape drive, ARC will provide temporary disk storage to FACC to hold working copies of the tapes to be compared. Controlled access storage for no more than twenty-five 10.5 inch magnetic tape reels is also required.

2.3. Code Security

Changes to source code under configuration control will require explicit authorization by the FACC KSOS Configuration Control Board. Each approved source change will correspond to an SCCS DELTA and will be documented in a configuration control notebook. The approved DELTAs and their supporting justification will form an audit trail for each controlled item.

The basic physical security for the GFE PDP-11/70 during implementation will be in accordance with long standing, government-approved FACC policies. The implementation of KSOS will be done using both the GFE PDP-11/70 and FACC's capital PDP-11/45 UNIX*tm system. Both will be located in secure

facilities where access by uncleared personnel will be restricted. Primary access to these systems will be through hard-wired terminals.

If and when external (dial-up or ARPANET) communications lines are enabled, the master copy of the KSOS software will not be present. This will be done by using a mountable file system, and removing the disk when the dial up communication lines are enabled. This master pack will be handled as if it were a SECRET level object. Working copies of the software can continue to remain on the system during periods when communication lines are enabled, since working copies are treated as UNCLASSIFIED. Conversion of working copies to master copy status requires the same review process that is used for approving updates made to the specifications, viz, Configuration Control Board approval. FACC expects to utilize the UNIX*tm file comparison utilities for these checks. Of course, only KSOS project personnel will be given passwords on the GFE system and only project related work will be carried out there. These actions effectively prevent access to the KSOS implementation by any but cleared personnel with a valid need to know.

The effect of these measures is that the master copies of the design and implementation are completely protected at the SECRET level. There, an uncleared agent cannot systematically alter the design or the tools to introduce security flaws.

In summary, the proposed FACC/SRI KSOS design and implementation process will satisfy all Government security and control requirements.

3. Verification Impact

3.1. Source-Level Configuration Control

As discussed above, FACC plans to use a source control system similar to that provided by the UNIX*tm-based Programmer's Workbench [Ivie 77]. This source control system will allow reconstruction of any level of source code (release and version), thus providing a complete version-to-version audit trail. Updating of the master library will be restricted to the Programming Support Librarian, who is on the Implementation Manager's staff. Updates to the master development library are entered only after testing and approval by the cognizant verification team. At any point in time, any version or release of the system can be reconstructed on demand, allowing complete auditing of all differences between system versions.

3.2. Modification Request System

By subjecting all modification requests to the same formal design and code inspections required for initial development, and by using the source control system, FACC will ensure the same high level of integrity throughout the development cycle. Subsequent maintenance functions will build upon the control and request systems used during development.

3.3. Programming Standards Enforcement

Consistent and correct implementation of the proven design is of utmost importance to the success of the KSOS project. The KSOS programming language (KPL) and the implementation style will emphasize the development of code in a form that enhances ultimate provability. Although complete formal code proofs will not be undertaken by FACC, an emphasis on simple (and traceable) realizations of the specification will improve the probability of successful automation of code proofs in the future. As previously discussed, correspondence of the implemented system to the formal specifications will be mandated and enforced by formal inspection techniques.

3.3.1. Formal Code Inspection

The formal code inspection process will serve to enforce programming standards. One objective of these standards is to provide the Government with programs that are in a form suitable for automated code verification. To this end, FACC will use programming constructs that will simplify the code proof process. As an incidental benefit, these measures also make testing somewhat easier. Mutual inclusion of test-team and implementation-team members in complementary formal inspections helps to ensure good communication between the teams. Studies [Larson, 1975, p. 7] indicate significant savings in personnel time (85 percent!) over traditional testing methods.

3.3.2. Production Coding

FACC intends to subject all designs and coding to the formal inspection process. Inspection gives the highest practical level of visibility to each module. Because inspection subjects the module to the multiple view-points of the inspection team, FACC feels that the code produced under the formal inspection process will meet uniformly high quality standards.

4. Performance Impact

The design and coding methodology used by FACC balances the potentially conflicting objectives of verifiability and performance. Each formal inspection will address these issues, judging the design or code in terms of the engineering trade-offs that must be made. The initial implementation of any module will usually sacrifice performance for simplicity and verifiability. If performance bottlenecks are found to exist, then internal redesign of the module may be necessary. SRI International's Hierarchical Design Methodology (HDM) permits this type of change by completely identifying the interfaces. As long as the interface is not changed, the module may be freely altered, due to the transparency provided by the methodology.

4.1. Formal Code Inspections

The FACC commitment to formal inspections is expected to be a wise investment, because the thorough review process will expose programming errors early, before testing is underway. FACC anticipates that the reduction in programming and testing costs will more than offset the inspection costs.

4.1.1. Algorithm Review

One of the functions of a design code inspection is algorithm review. The inspection team is required to methodically compare the proposed design with its controlling specification. Incomplete or inappropriate algorithms should be detected early in the development process, thus allowing early correction. The cost benefits of early error detection and correction have been found to be enormous (ten to one hundred times cheaper to fix an error in design than after delivery. [Boehm 76]) This is one reason that FACC is implementing the formal inspection process at design time and at code-completion time (before integration and testing begin).

4.1.2. Programmer Productivity

Various studies have indicated that programmers can be significantly more productive when programming teams can be effectively formed [Weinberg 71]. The KSOS project will form a small number of teams organized to capitalize on special capabilities of the FACC/SRI KSOS project staff. To the extent possible, staffing of teams will be based on individual capabilities rather than seniority.

The review time required by the formal inspection program appears to be an excellent investment. Studies by [Fagan 76] indicate that inspection improves programming productivity by more than twenty percent in comparison to coding subjected to structured walk-throughs only. In addition, the inspection sample had thirty-eight percent fewer errors during the period between unit test completion and system test.

4.2. Performance Monitoring

Measurement of dynamic behavior of programs is a necessary part of both performance validation and software testing. FACC will utilize both self-hosted and satellite-processor based measurement techniques.

4.2.1. Self-Hosted Measures

A timer-based program counter sampler will be used to measure performance of programs at the user level. This facility will be similar to the "profile" command in UNIX*tm. It is expected that timer-based sampling facilities will also be usable in measuring emulator performance.

4 2.2. Satellite Processor

In order to avoid synchronization artifact between the system clock and the program execution sampler, an external, independent sampling control is necessary. Whenever sampling is not independent of system time, the sampler is biased and will not "see" certain events. We call this phenomenon "synchronization artifact." FACC is studying the feasibility of using a satellite processor (the FACC Software Development Facility machine, a PDP-11/45) to provide remote stimulation and monitoring through an inter-processor link made from a pair of DR11-K's. The DR11-K is a 16-bit parallel interface similiar to the DR11-C. The DR11-K acknowledges data reads automatically, thus eliminating the need for software ACK. Should FACC determine that the necessary resources can be committed, details of the testing setup will be discussed in the formal test plan.

5. Testing

Testing and verification are integral parts of the development process. Unless testing is carefully planned, large amounts of time can be spent without exercising the system adequately. For this reason, a logically separate testing and maintenance team will be established during Phase II of the KSOS project. One function of the formal reviews of design and code is to establish test criteria for acceptance of the module for subsequent use in the system.

Test cases are selected in relation to the formal specifications such that there is at least one test case for the TRUE and FALSE cases of every specified exception condition, for both x=TRUE and x=FALSE conditions of every specified IF (x) THEN-ELSE, for every specified (x)=>Q, and for every possible type of x in every specified TYPECASE (x) OF ... Note that testing is essentially negative in nature, since the purpose of a test is to find errors. A test "failure" shows that no errors were found, but can not guarantee the absence of errors. A discussion of proposed verification and testing methodology is presented in separate documents, the KSOS Verification Plan and the Maintenance and Support Plan.

5.1. Module Tests

Testing of individual modules is the responsibility of the implementor and the testing and maintenance team. As modules are moved from development to integration testing, the implementor's unit test results are made a part of the Module Development Folder and are turned over to the Test group. Modules are recompiled and exercised by the Test group before approval for inclusion in the master source library. Wherever possible, unit testing will be carried out within an existing (skeletal) system structure that has evolved from a top-down hierarchical design and implementation. To the extent possible, continuous integration of the system will be carried out.

A minimum criterion for unit testing is to execute all branches at least once in each direction except for those branches representing defensive programming. All implementation errors will be corrected by source-level modification in an auditable manner. No object-level code modification will be allowed.

5.2. Thread Testing

Partial integration testing (called thread testing by FACC) is an attempt to provide early integration of parts of major systems functions. Thread testing is a variant on top-down integration using module stubs. Thread testing and continuous integration are complementary integration techniques, since the notion of a "thread" is defined in terms of a group of related modules that simulate the complete product environment. It is this environment that provides the vehicle for continuous integration testing. Ultimately, the environment into which a module is being integrated will be the entire system. At that point, continuous integration will evolve into system integration. The key points to remember are that integration activity begins very early in the coding process, and that integration is a fundamental part of module acceptance into system libraries.

Testing will reveal errors in the implementation of the KSOS design. The error source must be traced so that corresponding specifications can be corrected. For example, design errors require formal specification changes, while only HDL changes may be required in other cases. Simple programming logic errors may require no changes to the specifications.

5.3. Cumulative Testing

Test cases for each module will be accumulated for application during integration testing and for release testing of an integrated system. The accumulation of tests can be run at any time to verify consistent behavior across releases. Release-dependent tests can be handled through use of the SCCS from PWB/UNIX*tm.

5.4. Required Testing

FACC will perform all applicable Category I and Category II tests and prepare the appropriate test reports. In addition, a comparison of KSOS performance with Western Electric UNIX*tm (Distribution 6.0) will be made using the MITRE benchmark scripts, along with any other tests that FACC or the Government deem appropriate.

5.5. Stress Testing

A common failure mode of systems occurs when an apparently working system is exercised in a way that forces resource exhaustion. Stress testing attempts to subject the system to extreme "pressures" to find areas of unacceptable performance degradation or system failure. Such testing is particularly important in KSOS because resource exhaustion can be used as a leakage channel. KSOS will utilize explicit resource quotas, probably in the form of quotas on a per-process or per-user basis.

While stress testing attempts to exercise the system to the breaking point, volume testing attempts to expose the system to massive amounts of data over an anticipated operational peak period. In the KSOS project, volume testing will be used to measure system performance for comparison with Western Electric Distribution 6.0 UNIX*tm and to estimate throughput of the system.

5.5.1. Shell Script

Benchmarks and basic performance testing can be carried out in a self-hosted mode by means of a UNIX*tm shell command file. This can provide (moderately) biased timing estimates useful for basic system performance measurement. The MITRE benchmark scripts will be used to provide a comparison basis between 6.0 UNIX*tm and KSOS.

5.5.2. Externally Generated Load

Use of the FACC Software Development UNIX*tm system as an external load generator will provide a terminal simulator for stress and volume testing of the KSOS system. Response time and throughput statistics can be gathered by the external computer system without introducing timing artifact into the KSOS system.

6. External Configuration Management

6.1. Supported Configurations

KSOS configuration support will be limited to systems that contain only the equipment present in the GFE PDP-11/70 provided for Phase II development system. Possible exceptions to this restriction will be in cases where multiple peripheral units are supported by a single controller (e.g., multiple TU16 tape transports, multiple RP05 disk drives, etc.).

6.2. Automated System Generation

Manual generation of systems of variable configuration is too error-prone to provide the confidence-level necessary for production release of KSOS. One simple approach to the configuration problem requires that identical (software) systems be delivered to all sites, with system initialization code responsible for disabling code for all missing control units.

At the present time, FACC favors a system-build dialogue that causes the interrupt vectors and device tables to be built automatically. The loaded system can then contain only the necessary device drivers, giving more storage to the user programs. The major technical factor involved in this approach is concerned with proof that a secure system that has less peripheral storage than the originally certified system is still certifiably secure. FACC will use a system structure that ensures the certifiability of subset systems.

6.3. Secure Delivery and Installation

Since KSOS is expected to have wide-spread usage, secure logistics must be implemented. With an automated system generation facility, it appears reasonable to send a self-starting generation system (at the SECRET level) by U.S. mail. Under separate cover, at appropriate security levels, a generation script, documentation, a password, and system-tape check-sums would be forwarded. The initially booted system contains a tape validity check program, an authentication program, and a KSOS system build restorer. The actual installation process must be carried out at the "system-high" classification level, the highest classification level that the system will ever process. Further discussion of the installation process is given in the Support and Maintenance Plan.

6.4. Maintenance and Support

The necessity for reverifying KSOS whenever any changes are made to the system requires that stringent configuration control be exercised. KSOS will not be a static system, since new devices will need to be interfaced to it, enhancements to its operation will certainly be suggested, and implementation errors may be discovered. Thus FACC will develop a formal system for introducing certified changes to KSOS.

The companion Maintenance and Support Plan addresses:

a. A system for implementation of enhancements to KSOS

b. A method for control and distribution of new versions and updates to

KSOS.

c. A system security standard for secure operation and maintenance of KSOS systems.

7. REFERENCES

[Aron 75] J. D. Aron, The Program Development Process: The Individual Programmer. Addison-Wesley (Menlo Park) 1975.

[Baker 71] F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Sys. J. 11, 1 (1972).

[Boehm 76] B. W. Boehm, "Software Engineering," IEEE Trans. Computers, C-25, 12 (Dec 1976) 1226 ff.

[Brooks 75] F. P. Brooks, The Mythical Man-Month. Addison-Wesley (Menlo Park) 1975.

[Caine 76] PDL - Program Design Language Reference Guide (Processor Version 3). Caine, Farber & Gordon, Inc., Santa Monica, CA. 1976.

[Dolotta 76] T. A. Dolotta and J. R. Mashey, "An Introduction to Programmer's Workbench," Proc. 2nd Int'l. Conf. of Software Engineering. (13-15 Oct. 1976) San Francisco, 164 ff.

[Fagan 76] M. E. Fagan, "Design and code inspections to reduce errors in program development," IBM Sys. J. 15, 3 (1976). 182 ff.

[Ivie 77] E. L. Ivie, "The Programmer's Workbench -- A Machine for Software Development," CACM 20, 10 (Oct 1977) 746 ff.

[Myers 76] G. J. Myers, Software Reliability. Wiley-Interscience (New York) 1976.

[Neumann 76] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K.N. Levitt, and L. Robinson, A Provably Secure Operating System: The System, Its Applications, and Proofs, SRI Final Report, Project 4332, 11 February 1977.

[Parnas 72] D. L. Parnas, "A Technique for the Specification fo Software Modules with Examples," CACM 15, 12 (Dec 1972) 1053 ff.

[Parnas 77] D. L. Parnas, "The Influence of Software Structure on Reliability," in Current Trends in Programming Methodology. Prentice-Hall (Englewood Cliffs) 1977. 111 ff.

[Rochkind 75] M. J. Rochkind, "The Source Code Control System," IEEE Trans. Software Engineering, SE-1, 4 (Dec 1975) 364 ff.

[Royce 70] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," TRW Systems Engineering and Integration Division, Redondo Beach, CA. Tech. Report TRW-SS-70-01. (Aug 1970).

[Shneiderman 77] B. Shneiderman, et al, "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," CACM 20, 6 (June 1977) 373 ff.

[Weinberg 71] G. M. Weinberg, The Psychology of Computer Programming. Van Norstrand Reinhold (New York) 1971.

[Yeh 77] R. T. Yeh, ed., Current Trends in Programming Methodology: Software Specification and Design. Prentice-Hall (Englewood Cliffs) 1977.

Hierarchical Design Language

This Appendix documents the FACC HDL processor and gives an example of its use.

NAME
hdl - hierarchial design language

SYNOPSIS
hdl [-ficl2lpug] [-d dname] [-m mname] [-r rfile] hdl files

DESCRIPTION
Hdl is a tool that allows one to describe a project, system or module in English, using the higher level language C's control flow. Hdl provides a verbal flow chart (the hdl source), a trace of the static call structure, an index into the hdl source to where procedures, globals and undefined procedures are used, and an index by page number to all the procedures and hdl segments.

Hdl provides the following defaults. The hdl source is reproduced in the same format as it was input. Output format 2, for short procedure and global names, is used to produce all hdl index tables. All output is formatted for the printer. The last and only critical default is that unless otherwise informed through the use of the m flag, hdl will assume the presence of the top level procedure main.

-f Keep the procedure and global names on files instead of in core. This allows one to process very large hdls.

Input Specifiers

-d dname Place the design title dname in the page header, which is printed at the top of each page. The design title is limited to a maximum length of 40 characters.

-m mname Use mname as the top level procedure of the hdl.

-r rfile Read the file rfile to obtain the names of the hdl source files.

Output Specifiers

-i Indent the hdl source to reflect the hdl control flow.

-c Format the output for the crt. Note that the page size has been set to be compatible with the re (rand) editor. This editor allows one to examine a file by page addressing.

-l Format the index tables to accommodate long procedure and global names.

-2 Format the index tables to accommodate short procedure and global names. If output is to be formatted for the crt procedure and global names should be shorter than 20 characters, otherwise names should be shorter than 32 characters. Note that format 2 will accommodate longer names. It may however, page the output incorrectly.

Output Suppression Specifiers

-l Suppress printing of the hdl source.

-p Suppress printing of the procedure index table.

-g Suppress printing of the global index table.

-u Suppress printing of the undefined procedure index table.

FILES

/tmp/hdl???	temporaries
/usr/sys/source/util/hdl	source files
/usr/bin/hdl	executable file
/usr/sys/source/util/hdl/hdl_doc	documentation file

SEE ALSO

indent (I)
d (I) Automatic indentation during input (ac mode).
re (I)

DIAGNOSTICS

Syntax errors are reported. They normally involve unbalanced double quotes, parentheses or curly brackets.

BUGS

Hdl does not support pure C syntax. See hdl_doc for syntactical anomalies.

HDL: HIERARCHICAL DESIGN LANGUAGE

HDL is a design tool that allows one to describe a project, system or module in English, using the structured constructs of the C programming language. HDL provides a verbal type of flowchart with the added pluses of cross-referencing and easy modification.

HDL produces the following outputs.

An indented listing of the source. All lines within a procedure will be numbered to the left of the source. If an elsewhere defined procedure is used, the page number where that procedure is defined will be placed to the left of the procedure line numbers.

A trace of the static procedure call structure. Note that unless otherwise informed through the use of the m flag, HDL will assume the presence of the top level procedure "main".

An index into the HDL source to where procedures,globals, and undefined procedures were used.

An index by page number to all the procedures and HDL segments.

Output from HDL may be formatted for the printer or the crt. The crt format has been so constructed to be convenient to be viewed on line using the re (rand) editor. Since the rand editor can view a file by page addressing, one can send the last segment of the HDL output, the HDL index, to the printer and use the HDL index to find the desired segment.

Those familiar with C syntax may skip the following syntax discussion. However please read the section entitled special syntax notes.

HDL Syntax and Control Statements

Basic syntax vocabulary:

identifer: can be any combination of letters, digits and underscore up to a length of 60 characters. The first character must be a letter.

string: is any text enclosed in double quotes. Special care should be taken to make sure that all double quotes are balanced or an HDL error will occur.

expression: when evaluated becomes a value. The expression may be text or mathamatical.

conditional: when evaluated becomes a value TRUE or FALSE. Note as a matter of convention if the conditional is numeric, a non-zero value will be

```
        taken as TRUE and a zero value as FALSE.

sentence:any sequence of characters with neither  parentheses  nor  unbalanced
        strings present.


HDL:
     <procedure definition> <statement>  |
     <global definition>  |
     <HDL>

global definition:
     <identifier>

procedure definition:
     <identifier> ( <sentence> )   <complex statement>
        Note that the procedure definition cannot be terminated  with  a  sem-
        icolon.  Procedure definitions cannot be nested, i.e., defined within
        other procedures.

statement:
     <sentence> ;   |
     <complex statement>  |
     <procedure reference>  |
     <for statement>  |
     <while statement>  |
     <do statement>  |
     <if statement>  |
     <switch statement>  |
     <labeled statement>  |
     <goto>  |
     <break>  |
     <continue>  |
     <return statement>


complex statement:
     { <statement list > }

statement list:
     <statement>      |
     <statement list>


procedure reference:
     <identifer> ( < sentence> );

for statement:
     for ( <initialization> ; <conditional> ; <incrementation> )
         <statement>   |

     for ( <sentence> )
         <statement>

         forml: The initialization is performed once. The conditional  is  then
         tested.  If  true  the following is performed until the conditional is
```

false. The statement is executed, the incrementation is performed and the conditional is tested again.

form2: The statement is executed until the task the sentence describes is completed.

while statement:
    while (<conditional>) <statement>

The statement is executed repeatedly as long as the conditional is true.

do statement:
    do <statement> while (<conditional>);

The statement is executed then the conditional is tested. If true, the statement will be executed until the the conditional is false.

if statement:
    if (<conditional>) <statement> |

    if (<conditional>) <statement 1> else <statement 2>

Form 1: If the conditional is true, the statement is executed.

Form 2: If the conditional is true, statement 1 is executed, then control is passed to the first statement following the if statement. If the conditional is false, then statement 2 is executed and control is passed to the first statement following the if statement.

switch statement: Is a clear way to express a nested group of if else statements. Its syntax is described in the following three productions.

switch statement:
    switch (<expression>) <case statement list>

case statement list:
    <case statement> |
    <case statement> <case statement list>

case statement:
    <statement> |
    case <expression> : <statement> |
    default : <statement> |
    break;

The following is an example of a switch statement.
switch (<expression 0>)
{
    case <expression 1> : <statement 1.1>

```
                    <statement 1.2>
                    break;

      case <expression 2> :
      case <expression 3> : <statement 3.1>

      case <expression 4> : <statement 4.1>
                    break;

      case default:
                    <statement d.1>
                    <statement d.2>
}
```

The control flow for the switch statement is as follows. Expression zero is evaluated. Then the expressions following the word case are evaluated in order. When the first expression equal to expression 0 is found, control is passed to the statements following the colon. The control flow will pass from one statement to the next, ignoring case expression labels, until a break or the end of the switch is encountered. A break statement passes control to the first statement after the switch statement.

Note that if none of the case expressions equal expression 0, control will pass to the statements following the default case label if it is present, otherwise control will pass to the statement following the switch statement.

Special statements:

break: break;

This statement terminates the smallest enclosing while, do, for or switch statement. Control is then passed to the statement following the terminated statement.

continue: continue;

This statement passes control to the loop-continuation portion of the smallest enclosing while, do, or for statement. For example:

```
while (...)       do                  for(...)
{                 {                   {
  ...               ...                 ...
  contin:;          contin:;            contin:;
}                 }                   }
```

A continue statement in any of these loops is equivalent to the statement:

```
goto contin;
```

labeled statement: <identifier> : <statement>

Any statement may be preceded by a label. However labels make for

poorly structured and difficult to understand HDL's. They, therefore should be avoided at (all - 1) costs.

goto: goto <identifier>;

This statement passes control to the statement preceded by the label indentifier.

return statement:

```
return;
return ( <expression> );
```

The return statement terminates the present procedure, control is then passed to the calling procedure. Note that, if necessary, a value can be returned to the calling procedure.

Special syntax notes:

Parentheses are used for procedure definitions and references. They, therefore, cannot be used for any other purpose within an HDL. The user has two options for grouping a thought, some delimiter such as square brackets could be used instead of parentheses, or the parenthesized sentence could be enclosed in double quotes.

HDL scans, that is, does not parse, the HDL source. For this reason *the left parenthesis of a procedure* definition or reference must occur on the same line as the procedure name. A left parenthesis on any given line must always be preceded by an identifer.

HDL PROCEDURES

PROCEDURE: main

```
         1  main ()
         2
         3      /* DESCRIPTION: This procedure will construct the static call
         4         structure of the procedures that it is given as input */
         5
         6      {
         7          Invoke cref to process the input procedures and produce a
         8          temporary output file;
         9
   3    10          Construct_the_list_of_procedures ();
        11
        12          Make_procedure_branch_array ();
        13
   5    14          Construct_the_static_call_tree ();
   |    15
   |    16          Print_the_call_tree ();
   |    17      }
   |     |
   |     |
   |    LINE NUMBER IN PRESENT PROCEDURE
   |
 REFERENCE TO A PROCEDURE DEFINED ON PAGE 5
```


DEFINED GLOBALS

GLOBALS

```
     int buffer[256];
     int node;
      |
      |
     IDENTIFIER TYPING
```

HDL PROCEDURES

PROCEDURE: Construct_the_list_of_procedures

```
 1  Construct_the_list_of_procedures ()
 2
 3      {
 4          Open the cref temporary output file;
 5
 6          get_line (character buffer);
 7
 8          First_string = get_first_string_from_line ();
 9          Second_string = get_second_string_from_line ();
10
11          while (not end of file)
12          {
13              if (Second_string is a _ then first string is a global)
14              {
15                  Allocate a node;
16                  Fill a node with characters pointed to by First_string;
17                  Link this node to first node of the procedure branch
18                  array;
19              }
20              else
21              {
22                  if (First_string_is_in_the_list_of_procedures () )
23                  {
24                      if (First_string is a new procedure)
25                      {
26                          Fill the present procedure with the First_string;
27
28                          Allocate a new branch and node;
29                          Link the end of the branch array branch->next_branc
30                          to the new branch;
31                          Link the branch->pointer_to_node to the node just
32                          allocated;
33                          Make node->pointer_to_node_name point to the name
34                          of the procedure in the procedure array;
35                      }
36                      else
37                      {
38                          Allocate another node;
39                          Make the node->pointer_to_node_name point to the
40                          name in the procedure array;
41
42                          if (if the present procedure is the first son
43                              in this branch )
44                              Link the present node to the last node's
45                              node->pointer_to_son;
46                          else
47                              Link the present node to the last node's
48                              node->pointer_to_brother;
49                      }
50                  }
51              }
52              get_line (character buffer);
53
54              First_string = get_first_string_from_line ();
```

HDL PROCEDURES

PROCEDURE: Construct_the_list_of_procedures

```
55              Second_string = get_second_string_from_line ();
56          }
57
58          return (pointer to the first element of the list);
59      }
```

HDL PROCEDURES

PROCEDURE: Construct_the_static_call_tree

```
        1  Construct_the_static_call_tree (pointer to branch array, pointer to node )
        2
        3      {
        4          if (Check_if_node_is_recursive (pointer to the node) )
        5          {
        6              Add_a_blank_star_to_the_end_of_the_procedure_name (pointer
        7                  to the node);
        8
        9              Set the node's node->pointer_to_son to NIL; That is don't
       10              allow the node to adopt his sons;
       11
       12              if (the node->pointer_to_brother is not NIL)
       13              {
5      14                  Construct_the_static_call_tree (branch array,
       15                      node->pointer_to_brother );
       16              }
       17          }
       18          else
       19          {
       20              Check_branch_array_and_return_a_pointer_to_the_node_s_son (
       21                  pointer to the branch array, pointer to node);
       22              if (the pointer returned is equal to NIL)
       23              {
       24                  Set pointer to node->pointer_to_son to NIL;
       25              }
       26              else
       27              {
       28                  Set present node->pointer_to_son to point at this
       29                  node's sons;
       30
       31                  if (the node->pointer_to_son is not equal to NIL)
       32                  {
       33                      push_procedure_on_active_procedure_stack (node);
       34
5      35                      Construct_the_static_call_tree ( pointer to the
       36                          branch array, node->pointer_to_son);
       37
       38                      pop_active_procedure_stack ();
       39                  }
       40              }
       41              if (node->pointer_to_son is not equal to NIL)
       42              {
5      43                  Construct_the_static_call_tree (pointer to the branch
       44                      array, node->pointer_to_brother);
       45              }
       46          }
       47      }
```

STATIC PROCEDURE CALLING STRUCTURE

```
main

        Construct_the_list_of_procedures

        Construct_the_static_call_tree

                Construct_the_static_call_tree

                Construct_the_static_call_tree

                Construct_the_static_call_tree
```

```
                INDEX To PROCEDURES

PROCEDURES

  3  Construct_the_list_of_procedures
              called by:
                         1  main
                               at line(s):    10

  5  Construct_the_static_call_tree
  |           called by:
  |                      5  Construct_the_static_call_tree
  |                            at line(s):    14      35      43
  |                      1  main
  |                      |     at line(s):    14
  |                      |                     |
  |                      |                 LINE NUMBER IN main WHERE
  |                      |                 consruct_the_static_call_tree
  |                      |                 IS CALLED.
  |                      |
  |                  PAGE NUMBER WHERE main
  |                  IS DEFINED.
  |
PAGE NUMBER WHERE
Construct_the_static_call_tree
IS DEFINED.
```



```
                INDEX To GLOBALS

GLOBALS

       buffer
              referenced by:
                         3  Construct_the_list_of_procedures
                               at line(s):    52      6

       node
              referenced by:
                         3  Construct_the_list_of_procedures
                               at line(s):    15      16      17      17      28      31
                                              33      38      39      44      44      45
                                              47      47      48
                         5  Construct_the_static_call_tree
                               at line(s):    10      12      15      21      24      28
                                              29      31      33      36       4      41
                                              44       7       9       9
```

INDEX To UNDEFINED PROCEDURES

UNDEFINED PROCEDURES

```
        Add_a_blank_star_to_the_end_of_the_procedure_name
                called by:
                                5  Construct_the_static_call_tree
                                      at line(s):     6

        Check_branch_array_and_return_a_pointer_to_the_node_s_son
                called by:
                                5  Construct_the_static_call_tree
                                      at line(s):    20

        Check_if_node_is_recursive
                called by:
                                5  Construct_the_static_call_tree
                                      at line(s):     4

        First_string_is_in_the_list_of_procedures
                called by:
                                3  Construct_the_list_of_procedures
                                      at line(s):    22

        Make_procedure_branch_array
                called by:
                                1  main
                                      at line(s):    12

        Print_the_call_tree
                called by:
                                1  main
                                      at line(s):    16

        get_first_string_from_line
                called by:
                                3  Construct_the_list_of_procedures
                                      at line(s):    54       8

        get_line
                called by:
                                3  Construct_the_list_of_procedures
                                      at line(s):    52       6

        get_second_string_from_line
                called by:
                                3  Construct_the_list_of_procedures
                                      at line(s):    55       9

        pop_active_procedure_stack
                called by:
                                5  Construct_the_static_call_tree
                                      at line(s):    38
```

INDEX To UNDEFINED PROCEDURES

UNDEFINED PROCEDURES

```
push_procedure_on_active_procedure_stack
      called by:
                5  Construct_the_static_call_tree
                      at line(s):    33
```


INDEX to HDL

PAGE

GLOBAL VARIABLES 2

PROCEDURES

main 1
Construct_the_list_of_procedures 3
Construct_the_static_call_tree 5

HDL REFERENCE TABLES

Static Procedure Calling Structure 6
Index to Procedures 7
Index to Globals 8
Index to Undefined Procedures 9

Programming Standards

This Appendix discusses programming standards for use with C-language programs. It is a prototype of the standards to be used with the KSOS programming language.

UNIX Coding Guidelines

Ford Aerospace & Communications Corporation
Western Development Laboratories
Software Technology Department
3939 Fabian Way
Palo Alto, California 94303

ABSTRACT

This paper presents guidelines and standards for C language programmers in the writing of C programs for UNIX. The rationale is to provide a minimum level of coding quality so that programs are easy to read, understand and support.

1. General Rules

Comments should be used freely. Remember that a comment should explain what is happening. Comments which state the obvious are a waste of time. Blank space and blank lines make code easier to read. A blank line should separate sections of code and declarations from statements. Programs should be made into small modules of one or two pages except in cases where the code is long but sequential. Wherever possible, functions/procedures should be less than one page long.

1.1. Identification Information

Each program or function must have a preamble in the form of a comment containing the following information:

```
 * /*
 * Name: name of module, program or function
 * Function: purpose of module. General functional description of what the
   module does.
 * Algorithm: detailed description of how the program or module accom-
   plishes its task. May refer to another document if the algorithm is
   very technical.
 * Parameters: list of the names, meanings, and values of all parameters
   used to invoke the function. If a main program, a list of the arguments
   and options and their meanings.
 * Returns: if a function, the type and meaning of any return values. If a
   main program, the meaning of any exit codes.
 * Globals: a list of all global variables used.
 * Calls: a list of all functions invoked by this program including a list
   of all system calls made.
 * Called by: a list of all (known) functions that invoke this function.
```

* Files and Programs: a list of all files used by this program (other than those passed in via the argument list). A list of all programs called by this module via the exec system call.
* Notes and memorabilia: information not intuitively obvious to the general programmer. Program bugs, deficiencies and/or special features. Warnings on installation problems that may result. System dependencies (for example, assumes Version 7 of the C compiler).
* Installation instructions: details on how to compile the program. This should be a set of UNIX commands. Also, names of any other library routines, etc.
* History: date of coding and author version number. This will be followed by an entry for each change.

1.2. Configuration Information

Each separately compiled module should have an embedded string with configuration information. Such a facility is available using the key word features of the Source Code Control System of Programmer's Workbench. This string contains a set of four special characters that are very unlikely to appear in any normal situation. They are followed by the configuration information. Sample strings are:

```
char ftp_id[] "~|^` UNIX Server FTP Version 1.3";
char id[] "~|^` main.c  1.3";
```

All library modules loaded with the program should also have such a configuration line whenever the module is not a standard module. Programmer's Workbench has a program named "what" that extracts this information from a file or files.

1.3. Release History

For each version or release of the software module a line must occur in the preamble giving the following information:

* version/release
* author/modifier
* date of change
* history or reason for change.

1.4. Use the C Pre-Processor

The C pre-processor should be used to assist the reader. Use the #define feature whenever there are manifest constants, strings, and data structures.

1.4.1. Data Structure Definitions

All standard data structures should be defined by use of an include statement. General system data structures should all be defined by the same file. A common library of definitions, data structures and file names should be maintained. Data structure and field names should be the same as those specified in the UNIX Programmer's Reference Manual. Don't use the field name fd when the manual calls it fildes.

1.4.2. Conditional Compilations

Use the conditional compilation feature to include debug statements. Use it to allow custom or installation dependent features. For example, if your installation favors the use of the program "dir" to obtain directory listings instead of the old reliable "ls", use a conditional compile:

```
#ifdef CAC
    execl ("/bin/dir","dir",arg,0);
else
    execl ("/bin/ls","ls","-1",arg,0);
#endif
```

1.4.3. Embedded Strings

The software should not contain strings which have embedded file names; login names other than root or bin; program names; etc. If any of these strings are required, they should be declared as defines or includes at the head of the module immediately following the preamble and configuration/identification string. For example, use the following:

```
#define PASSWD "/etc/passwd"
fd = open(PASSWD,0);
```

and not

```
fd = open ("/etc/passwd",0);
```

1.4.4. Manifest Constants

Manifest constants should be specified as parameters, e.g.,

```
while( a != EOF ) ...
```

and not

```
while( a != 0) ...
```

and

```
while (TRUE)
```

instead of

```
while(1)
```

System constants (e.g., 512, 16) should be specified by the compiler construct sizeof or parameterized as BLKSIZ.

```
count = read(dirfile,&dirent,sizeof dirent);
```

NOT

```
count = read(dirfile,&dirent,16);
```

1.5. Error Messages

Messages printed by the program should be specified as constant strings declared at the head of the program or module, or as include files.

Error messages should be of the form
<program> : <file name> : <diagnostic> (<messages number>) For example,
delta:sam.c:write error(203)

Error numbers should be assigned centrally. For each error number a short (1 to 5 line) explanation should be provided giving information on what the message means, corrective action to be taken by the user, etc. This can be integrated into the users manual or installed into a help file.

1.6. Enclosing Braces

Each bracketing pair (the pair of braces "{" and "}") should be identified if they are separated by more than a few lines. Examples:

```
} /* main */
```

and,

```
} /* switch(*cp) */
```

and,

```
} /* for(i=1;... */
```

1.7. Null Statements

Null statements should be specified as deliberate by placing the ";" on a separate line. Thus,

```
while((c=getchar()) != ',')       ;     /* consume input line to next  comma
*/
```

specifies the intent clearly.

1.8. Position Critical Code

Code or declarations which are position dependent, or cannot have intervening code inserted must be designated. Avoidance of these constructs is better yet!

1.9. Function References

References to functions returning values other than integers and references to external variables used as constants in initializatin of data structures must be declared (via extern) prior to use. All side effects of function references must be documented.

2. Beauty

All programs should be processed by the standard indenter (indent) before release. This will ensure compliance with installation standards.

2.1. Line Length

Lines longer than 72 characters are in bad taste. CRT oriented editors have problems with them. Also, it is hard to print the program on 8 1/2 by 11 paper for binding.

2.2. Major Comments

Comment blocks of any length should be distinguished as follows:

```
/*
 *  Place a line of stars as shown.  Indent or tab
 *  for the body of the comment.  Be sure to end the
 *  line of stars with the appropriate ending (*/)
 */
```

The major reasons are

* to make major comments obvious and readable,
* to prevent the loss of an end comment by it having too far from the

start,

* and to prevent a person from adding a line to the comment in such a way as to end the comment unintentionally.

3. Expressions and Operators

All operators involving the equal sign (=) should be separated by blanks to prevent ambiguous code. I.e., this

```
  a = *b;
  a = -b;
  a =- b;
  a == b;
```

and not

```
  a=*b;
  a=-b;
  a==b;
```

Surrounding arithmetic and logical operators with blanks improves code readability considerably.

4. Summary

The most ingenious coding is of no value if people other than the writer of the code cannot read and understand it. The above guidelines suggest how to write C programs so other people will find it easier to read and maintain them. There is value in following guidelines in that the programs will be easier to understand and modify.

5. Acknowledgements

Anyone who has read much of the UNIX Operating System software can see examples of many of the above suggestions. The Network UNIX software also uses some of the ideas. The concept of special strings being embedded in a program module for identification purposes and the concept of error messages with numbers for more help are ideas found in the Programmer's Workbench software. This paper is an adaptation of work by Dennis L. Mumaugh of the Department of Defense.

6. References

1. Kerninghan, B. and Plaugher, P.J. The Elements of Programming Style. McGraw-Hill, 1974.

2. Kernighan, B. and Plaugher, P.J. Software Tools. Addison-Wesley, 1976. Pages 27-28 are of special interest, but the whole book has a wealth of examples.

Program Development Tools

Programming development tools discussed in this Appendix will be used to support the KSOS programming process. When the KSOS programming language (KPL) is determined, comparable facilities will be developed to support KPL development. Programs discussed in this Appendix are:

- d - the program creation editor
- hdl - the hierarchical design language
- indent - the C-indenter
- nn - the procedure calling tree lister

NAME
d - Enhanced UNIX ed editor.

SYNOPSIS
d [-] [name]

DESCRIPTION
The following is a list of differences between 'enhanced' editor (from UCLA, UCB, CAC and NPS) and the standard UNIX editor.

1) Interline editing: The interline edit mode is invoked with the command o (for 'open'). This opens a line for interline editing. At that point. The following commands can be used:

(Space) Moves the cursor 1 ahead in the line, and prints the character.

Moves the cursor 1 back in the line, and prints the character passed over. NOTE: Characters which have been backspaced over and characters which have been deleted are shown between found signs ('#') to indicate that they do not form part of the final line.

D or ^A Deletes the next character in the line.

^H Deletes the next character in the line, and backspaces to correct position on print line.

e Deletes the previous character in the line.

c Substitutes the character typed after it for the next character in the line (may be aborted by typing a ^D)

s(char) Does the equivalent of (space) until the next (space) would print the specified (char). It always skips at least one space. If the specified (char) is not found, the cursor is left at the end of the line.

k(char) Like s, but does the equivalent of a d command. If the specified character is not found, the cursor is not moved.

i(string)^D Inserts the typed (string) into the line.

r(string)^D Is identical to di(string)^D.

l Prints the rest of the line, then resumes editing at the beginning.

p Prints the rest of the line, then resumes editing at the current cursor location.

u Restores the line to its original (pre-edited) state.

<Newline> Prints the rest of the line, then returns to the top-level editor.

Del or ^C Aborts the interline edit, restores the line to its original

form, then returns to the top-level editor.
The commands (space), #, d, e, c, s, and k may be preceeded by a signed repeat count with the obvius result. (-<command> = -1<command>). Also, note that the insert and change commands only go forward (sigh).

2) If a hangup occurs, the command "w ed-hup" is executed before the editor exits.

3) Prompting is default. Prompting and file counts may be inhibited with the option '-' when invoking ed. In addition a "#" will alternate the state of the line number prompting.

4) Line numbers at the beginning of each line is now the default. These can be turned on and off using the 'n' command.

5) 'g' and 'e' commands are intercepted the first time if the file has been modified since the last 'e' command. Uppercase Q and E will execute without the warning message.

6) Addressing outside the bounds of the file will in general be automatically corrected to within the bounds.

7) The bug which caused extra temp files to file up in /tmp has been fixed.

7) The ed command '!' will, if followed by a null line, fork a shell which remains in control of the console until an end-of-file (^D) is received.

9) The 'l' command shows all non-graphics, including the 177.

10) tmp files now take the decimal process id as part of their name. (Previously it was the octal process id.)

11) ^a and ^h do appropriate things in the o command.

12) The 's' command may now be given without search and replacement strings. The replacement strings as well as the search strings is remembered from the previous s command. The g and p suffixes may be used with this abbreviated form. Lower case letters may no longer be used as search/replacement string delimiters.

13) The single character command '/' and '?' may be used where the commands '//' and '??' made sense previously.

14) The 'z' command has been added. Its syntax is:

(.)z(.1+)(n)

The (.) represents an optional line address. n represents a page size, the number of lines on a screen. If not specified, its previous value is used. The default value is 20. The flags (.1+1-) specify the mode in which z is to operate.

. means print the current line in the middle of the screen;
- means print the current line at the bottom of the screen;
+ means print the current line at the top of the screen. (The entire screen is fill d with the necessary context lines to do that.) + is the default and need not be specified. Additionally, if z+ is used with no

specific address, the line pointer is incremented before Z is executed. Thus the Z command may be used to step through a file page by page.

15) A second temporary file, called /tmp/Exxxxx where xxxxx is the process id like the first temporary has been added. This contains indices into /tmp/exxxxx for each line in the ed file. The E tmp is automatically updated after an r or e command, and after FLUSHLIM lines have been modified by the user. FLUSHLIM is currently set at 25. The buffer can also be updated (like sync) by the 'x' command. The program "drestore" takes two files as arguments, the E and e files respectively, and puts on its standard output the file which ed had created. This feature is essentially for restoring from crashes, but could become the standard working file format. Currently, ed deletes its two temporaries whenever normally exited (though they can be saved by renaming them while in ed).

16) The 'u' command has been added. This command will undo the effects of an s or o edit. The last 8 s and o commands are saved on a lifo stack.

17) 'x<char>' will bring in the appropiate header from code modules. "escape cr" is used to go to the next subheading. Presently the follwing are supported: x no comment delimiters are provided xc Invokes "c" header
    xf Invokes the Fortran header.
    xa Invokes the assemble header.

18) The b command concerns itself with function keys that can be defined and used in the d editor.

b <function key definition file>. - This will read the file and load the function keys.

b - This will list out the values that have been set for the function keys.

The set of possible function keys is the set of lower case letters. The editor will also set up some default function keys that relate to the date.

The function keys will operate in ac append mode or in the x command (see x command). They are invoked by: <control Y> <function key>

The function key definition file consists of separate lines to define each function key. The lines are of the form: r<text to be printed out when the r function key is invoked>

For example, a function key definition such as the following pProject: Pascal compiler

would enter into the edit file the string "Project: Pascal compiler" when <control Y>p was entered.

The date default keys are displayed and updated by the b command. There values are:
Y - Year
M - Month of the year
W - Week of the month

D - Day of the month
H - Hour of the day
N - miNute of the day
S - Second of the hour
T - Time in unix standard format

Suggested usage - Using the function keys and a form file that is read in with the x command one is able to generate flexible forms that can reflect todays date, the current project, the users name etc.

19) 'ic<num>' and 'ac<num>' invoke modes that know about 'c' syntax. Keywords are automatically generated using control characters. The <num> defines the beginning indentation level, during an edit session this level is saved from one c append or insert to the next. The following is a summary of the various control character 'escape' often used to get to the next part of a construct.

control char	c construct	escape char	carriage return
^a	char		
^b	break;		
^c	(exits append mode)		
^d	#define		
^e	case	:	:
^f	for (	;	; ;) {
g	register		
^k	/*	*/	
^l	#include "	"	"
^n	int		
^o	main (argc,argv) char **argv;		
^p	printf(	);	);
^r	return (	);	);
^s	struct		
^t	switch (	){	){
^u	continue;		
^w	while	){	){
^x	extern		

FILES

/tmp/Exxxxx index file
/tmp/exxxxx temporary edit file
/usr/bin/drestore crash restore file

NAME
hdl - hierarchial design language

SYNOPSIS
hdl [-ficl2lpug] [-d dname] [-m mname] [-r rfile] hdl files

DESCRIPTION
Hdl is a tool that allows one to describe a project, system or module in English, using the higher level language C's control flow. Hdl provides a verbal flow chart (the hdl source), a trace of the static call structure, an index into the hdl source to where procedures, globals and undefined procedures are used, and an index by page number to all the procedures and hdl segments.

Hdl provides the following defaults. The hdl source is reproduced in the same format as it was input. Output format 2, for short procedure and global names, is used to produce all hdl index tables. All output is formatted for the printer. The last and only critical default is that unless otherwise informed through the use of the m flag, hdl will assume the presence of the top level procedure main.

-f Keep the procedure and global names on files instead of in core. This allows one to process very large hdls.

Input Specifiers

-d dname Place the design title dname in the page header, which is printed at the top of each page. The design title is limited to a maximum length of 40 characters.

-m mname Use mname as the top level procedure of the hdl.

-r rfile Read the file rfile to obtain the names of the hdl source files.

Output Specifiers

-i Indent the hdl source to reflect the hdl control flow.

-c Format the output for the crt. Note that the page size has been set to be compatible with the re (rand) editor. This editor allows one to examine a file by page addressing.

-l Format the index tables to accommodate long procedure and global names.

-2 Format the index tables to accommodate short procedure and global names. If output is to be formatted for the crt procedure and global names should be shorter than 20 characters, otherwise names should be shorter than 32 characters. Note that format 2 will accommodate longer names. It may however, page the output incorrectly.

Output Suppression Specifiers

-l Suppress printing of the hdl source.

-p Suppress printing of the procedure index table.

-g Suppress printing of the global index table.

-u Suppress printing of the undefined procedure index table.

FILES

/tmp/hdl???	temporaries
/usr/sys/source/util/hdl	source files
/usr/bin/hdl	executable file
/usr/sys/source/util/hdl/hdl_doc	documentation file

SEE ALSO

indent (I)
d (I) Automatic indentation during input (ac mode).
re (I)

DIAGNOSTICS

Syntax errors are reported. They normally involve unbalanced double quotes, parentheses or curly brackets.

BUGS

Hdl does not support pure C syntax. See hdl_doc for syntactical anomalies.

NAME

indent - produce an indented c program source

SYNOPSIS

indent [ifile [ofile]][args]

DESCRIPTION

The arguments that can be specified are:

ifile Input file specification. If this is omitted, input is from standard input.

ofile Output file specification. If this is omitted, output is to standard output. Ofile cannot be specified without ifile. -lnnn This gives the maximum length of an output line. The default is -l120.

-cnnn This gives the column in which comments will start. The default is -c57.

-cdnnn This gives the column in which comments on declaractions will strt. The default is for these comments to start in the same column as other comments.

-innn This gives the number of spaces for one indentation level. The default is -i4.

-dj,-ndg -dj will cause declarations to be left justified. -ndj will cause them to be indented the same as code. The default is -i4.

-v,-nv -v turns on "verbose" mode, -nv turns it off. When in verbose mode, indent will report when it splits one line of input into two or more lines of output, and it will give some size statistics at completion. The default is -nv.

-bc,-nbc If -bc is specified, then a newline will be forced after each comma in a declaraction. -nbc will turn off this option. The default is-bc.

-dnnn This option controls the placement of comments which are not to the right of code. The default of -d2 means that such comments will be placed two indentation levels to the left of code. -do would line up such comments with the code. See the section on comment indentation below.

-br,-bl Specifying -bl will cause complex statements to be lned up like this:

```
if (...)
{
  code
}
```

Specifying -br will make them look like this:

```
if (...){
   code
}
```

The default is -bl.

Indent is intended primarily as a c program indenter. It will not take an arbitrary c program and produce a source file which conforms to the dsg documentation standards. It will, however, isolate the programmer from much of the work required to get his/her source into such a format. Specifically, indent will:

indent code lines
align comments
insert spaces around operators, where necessary
break up declaration lists, as in "int a,b,c;".

It will not break up long statements to make them fit within the maximum line length, but it will flag lines that are too long. Lines will be broken so that each statement starts a new line, and braces will appear alone on a line. (see the -br optin for the exception to this.) also, a feeble attempt is mde to line up identifiers in declarations.

multi-line expressions

Indent will not break up complicated expressions that extend over multiple lines, but it will usually correctly indent such expressions which have already been broken up. Such an expression might end up looking like this:

```
x =
      (
         (arbitrary parenthesized expression)
         +
         (
             (parenthesized expression)
             *
             (parenthesized expression)
         )
      );
```

COMMENTS

Indent recognizes four kinds of comments. They are straight text, "box" comments, unix-style comments, and comments that should be passed through unchanged. The action taken with these various types is as follows:

"box" comments: The dsg documentation standards specify that boxes will be placed around section headers. Indent assumes that any comments with a dash immediately after the start of comment (i.e. "/*-") is such a box. Each line of such a comment will be left unchanged, except that the first non-blank character of each successive line will be lined up with the beginning slash of the first line. Box comments will be idented (see below). Unix-style comments: This is the type of section header which is used extensively in the unix system source. If the start of comment

('/*') appears on a line by itself, indent assumes that it is a Unix-style comment. These will be treated similarly to box comments, except the first non-blank character on each line will be lined up with the '*' of the '/*'. Unchanged comments: any comment which starts in column 1 will be left completely unchanged. This is intended primarily for documentation header pages. The check for unchanged comments is made before the check for unix-style comments. Straight text: all other comments are treated as straight text. Indent will fit as many words (separated by blanks, tabs, or possible. Straight text comments will be indented.

Comment indentation

Box, unix-style and straight text comments may be indented. If a comment is on line with code, or followins a non-blank line, it will be started in the "comment column", which is set by the arameter. Otherwise, the comment will be started at nnn indentation levels less than where code is currently being placed. Where nnn is specified by the -dnnn command line parameter. (Indented comments will never be placed in column 1.)

If the code on a line extends past the comment column, the comment will be moved to the next line.

Diagnostics

Diagnostic error messages, mostly to tell that a code line has been broken or is too long for the output line, will be printed on the controlling tty.

Bugs

None known at this writing, but they surely exist. If (when) any are found, plese notify David Willcox.

NAME
nn - print n-squared tree of procedure calling structure

SYNOPSIS
nn source file ...

DESCRIPTION
Nn invokes cref to produce the temporary file produced by cref. It then, using this temporary file, produces an n-squared type tree of the calling structure. The default mode assumes that the top of the tree is the procedure main.

-m Use the first argument following the flags as the top of the tree.

-c Include all calls to the C library in the procedure tree.

-g Print out globals.

-s Sort all output.

-d Delete all duplicates and sort all output. All duplicate globals are removed and also all duplicate procedures on any level are removed.

-h Print active procedure list at the top of each page. This is useful if a very large tree is to be printed.

FILES

/tmp/nn??	temporary files
/usr/sys/nsource/util/nn	source files
/usr/bin/nn	executable file

SEE ALSO
cref(I)

DIAGNOSTICS
If more than one top of tree is found the message:
" multiple top of tree found "
will print out. Nn will use the first top of tree it finds as the top of the output tree. One should check the files that were given to nn. See BUGS.

BUGS
If files with multiple versions of the same procedures are passed to nn, nn will produce a tree that is a mixture of these versions.

Cref incorrectly scans procedure type declarations as procedure definitions. Since nn post-processes cref output, this error is passed on.

Documentation Standards

This Appendix describes documentation standards for use with C programs. It is a prototype of the standards to be used in the KSOS Project.

Guidelines for Documenting UNIX Software

Ford Aerospace & Communications Corporation
Western Development Laboratories
Software Technology Department
3939 Fabian Way
Palo Alto, California 94303

ABSTRACT

This paper presents guidelines and standards for the documentation of UNIX software for release to external organizations. The rationale is to provide the minimum level of documentation necessary to make programs easy to use, understand, and support.

1. General Rules

Documentation for a UNIX software distribution package should consist of the following items:

* Installation guide.
* Page(s) for the UNIX Programmer's Reference Manual.
* User's guide or Manual.
* Technical manual or software notebook pages.

As may be expected, one or more of these items may be missing or combined. The installation guide may be an appendix to the technical manual, or both the software notebook and the installation guide may be included as the first few pages of the actual software.

1.1. General Format and Preparation

All machine readable documentation should be in a format to be input to the nroff program (nroff source). This is no hardship to anyone since all UNIX systems have nroff.

Whenever possible use a "standard" macro package for the files. Good choices are the "ms" [1] or the "mm" [2] packages. The primary reason for this requirement is to ensure that the site using the package has the appropriate macros. Also, both of the above packages can be used with the photo-composition typesetter, which most "home brew" packages cannot.

----

[1] The package tmac.ms is distributed with the standard UNIX system.
[2] The package tmac.mm is provided with the Programmer's Workbench system.

All documents having more than one part (chapter, section) should have a master file set up as follows:

```
.so begin
.so next
.so chap1
...
.so last
.so tail
```

This is to allow a user to be able to understand how to generate the document. The user should be able to say:

```
nroff file or,
troff file
```

Please note that due to differences in line printers and terminals page lengths are not always the same and the files should allow for this parameter to be changed easily.

All macros should be referenced by the following method:

```
.so /usr/lib/tmac.s
```

This leaves no doubt to the user as to what to do to generate the file or where the macros are expected to be.

2. Installation Guide

The installation guide should provide all necessary information to install the software. It may be a separate manual, it may be included in the technical manual or software notebook, or it could be a README file in the directory with the software.

2.1. Cover Letter

A general cover letter should be included describing the contents of the package. It should provide the following information:

* any restrictions, proprietary or legal concerning the package such as second or third part licenses required.
* general overview and description of the package. If the package is distributed on tape a description on how the tape is organized. (This should include a listing of all files see section 2.4.)
* Instructions as to how to extract the software and where to place it in the UNIX directory system. If the tape is a tp tape, information as to what files to extract first, etc.

2.2. Installation Procedures

A detailed step by step procedure to install the software. No step is too small to include, no detail is too small to mention. A Heathkit-type procedure is recommended.

2.2.1. Install Object Programs

If the tape has pre-compiled object programs, there should be instructions on how to install these in their proper places.

2.2.2. Compile Sources

This should specify exact sequences for compiling sources. It should give command strings to use including any special switches (e.g., -n, -i, or -lp). Also list any special libraries to include files needed.

2.2.3. Test Software

This should describe how to test the software for correct operation. It might involve the use of special test programs.

2.3. Shell or Run Files

A copy of all shell or run files should be included in the installation guide so they can be read and digested before use.

2.4. List of Files and Directories

A list of all files and directories on the distribution medium (tape) should be given with a description of the use or meaning of each.

2.5. Appendix

The following items should also be given:

* A list of all programs or files to be placed in the public file spaces of the system
* A list of all embedded or compiled file names (or user names or ids) and the modules in which they appear
* A list of all libraries, routines and modules required which are not part of the Version 6 UNIX
* A list of any "non-standard" features or operating system calls (those not in Version 6.0 of UNIX)
* All include files required and where they should be located
* All documentation macros that are not distributed as part of UNIX.

One may fairly assume that copies of the UNIX User's Group software distribution center tapes are generally available and can be referred to for copies of specific items. Do not assume that all (or any) of that software has been installed, since many installations are still running a standard distribution version of UNIX.

2.6. README Files

If software packages have special files such as README files they should also be included.

2.7. Manual Pages

We recommend adding a macro .RV to the manual macros as follows:
.RV who when where
The *who* specifies the last person to revise the manual page. The *when* gives the date of revision. The *where* gives the sources of the software (such as RAND, USNPGS, CAC, SDC1, etc.) This line of information is printed at the bottom of each manual page. Remember that more than one manual page is required if special formats or files are involved.

3. User's Manual

Often, more information is required than the format of the standard manual page provides. The following subsections suggest some helpful additions to the information contained in the manual.

3.1. Error Messages and Diagnostics

A listing of all possible diagnostics generated by the software informs the user of the kind of error conditions he may expect. Suggestions for corrective action to be taken by the user would also be extremely helpful.

3.2. Usage Distributions

In order to help the user get started, include a sample usage session with the program(s). To guarantee authenticity, the command script and program responses should be collected from an actual session.

4. Technical Manual

This manual is sometimes called a software notebook. It should contain everything an unfamiliar systems programmer needs to know about the software in order to maintain it.

Again, there are standards for such manuals and this paper covers only UNIX related areas.

4.1. Narrative Overview

This could be a short version of the preliminary design specification. It should give a brief description of what the program does and how it does it. Specific UNIX concepts and features such as pipes, lock files, signals, temp files, and inter-process communication primitives are to be used. For example, "The program spawns a child process which does an execl of 'cat /usr/help/tom.hf' to present the help information."

4.2. User Interface

Describe the behavior of the program exactly as the user sees it. Items of special interest might include the fact that input is done in raw mode, or that the program traps the interrupt signal.

4.3. Data Structures

Describe each significant data structure used by the program. For each field the meaning of the field and the range of allowed values should be given.

4.4. Files

There shold be a listing of all files used or generated by the program (with full path names). (Include implicit files such as the /etc/psswd file used by getpw.) Normal system files need not have formats specified, but their use should be noted.

5. Summary

The purpose of documentation is to allow a person who has never seen the software package to install, use, and maintain it with minimal effort. The last thing one should do before releasing the documentation for a software product is to give it to a stranger and ask him to use it or to install the software.

6. Acknowledgments

The Version 7 C compiler distribution and the Programmer's Workbench Export 1.0 package were the source of many ideas for this paper. This paper is an adaptation of a working paper by Dennis L. Mumaugh of the Department of Defense.

7. References

1. Department of Defense. Automated Data System Documentation Standards Manual. Manual 4120.17-M. December 1972.

2. Mumaugh, D. L. UNIX Coding Guidelines (Working paper).

3. Mumaugh, D. L. Guidelines for the Preparation of UNIX Software Packages for Export Release. (Working paper).