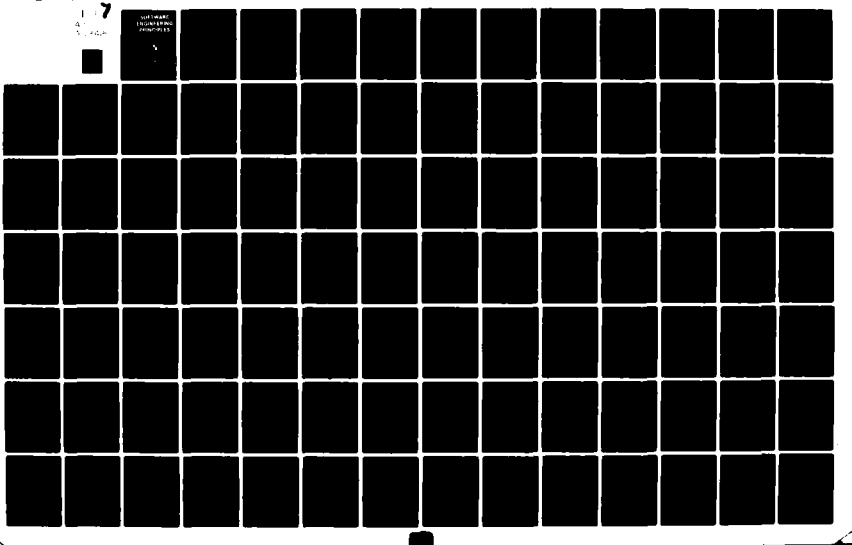
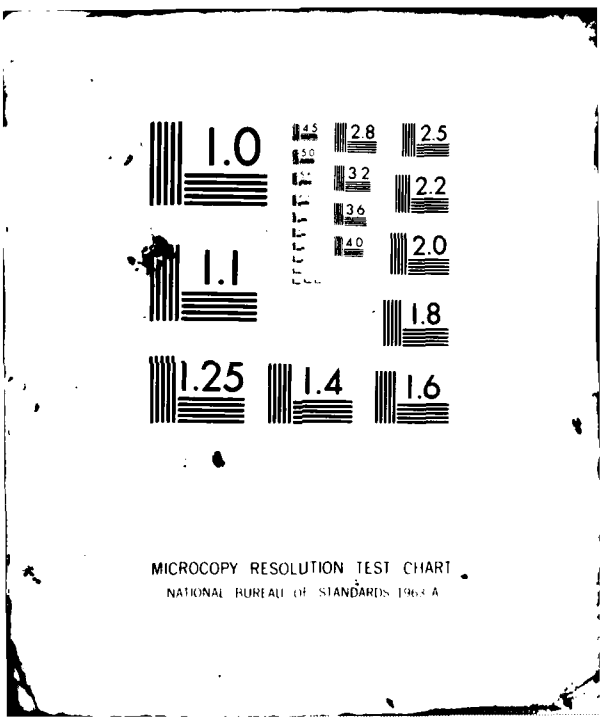
AD-A113 415 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
SOFTWARE ENGINEERING PRINCIPLES 3-14 AUGUST 1981.(U)
AUG 81 L J CHMURA, P C CLEMENTS

UNCLASSIFIED NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

SOFTWARE ENGINEERING PRINCIPLES

3 — 14 August 1981

Information Technology Division
Naval Research Laboratory
Washington, D.C. 20375

Department of Computer Science
and
Office of Continuing Education
Naval Postgraduate School
Monterey, CA 93940

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A113415	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* SOFTWARE ENGINEERING PRINCIPLES	5. TYPE OF REPORT & PERIOD COVERED 1981 Course Notebook
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) L. J. Chmura, P. C. Clements, C. L. Heitmeyer, K. L. Britton, D. L. Parnas, J. E. Shore, D. M. Weiss	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science and Systems Branch Naval Research Laboratory Washington, DC 20375	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS O&MN, NPS NRL Problem 75-M001-X-1
11. CONTROLLING OFFICE NAME AND ADDRESS Computer Science and Systems Branch Naval Research Laboratory Washington, DC 20375	12. REPORT DATE August 1981
	13. NUMBER OF PAGES 614
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

Prepared in cooperation with Office of Continuing Education
Naval Postgraduate School
Monterey, CA 93940

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Computer
Software
Computer Programming
Training

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This is the notebook from the updated edition of the well-received course originated by the Naval Research Laboratory and taught annually for the past six years. It is a two-week technical course for personnel managing a software project or designing software. The purpose of the course is to improve the participant's ability to evaluate software requirements, specifications, design, correctness, and maintainability. Its purpose is not to transform the participant into an expert software designer. → (continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

The course concentrates on technical problems of software design. It introduces generally accepted design practices, as well as software design research that may result in practical design applications in the near future. Design for ease of maintenance is emphasized. All course material is unclassified. Topics covered include information-hiding modules (modules that isolate the effects of changes), abstract interfaces (a technique for designing the interfaces of information-hiding modules), responses to undesired events, cooperating sequential processes (in real-time systems, software tasks in which scheduling and resource allocation decisions are not embedded), disciplined documentation techniques, techniques for formal specification, and designing systems with useful subsets.

Contents

*Distributed during course

CONTENTS

*Distributed during course

*Distributed during course

CONTENTS

Preface

Since 1973, the Computer Science and Systems Branch at the Naval Research Laboratory (NRL) has studied many of the managerial and technical problems connected with Navy software acquisition, development, and maintenance. One observation has been that persons responsible for software could benefit greatly from training in state-of-the-art software engineering technology. Another has been that a better job of software design is essential if software is to meet requirements and be maintained inexpensively. These two observations have led to the course "Software Engineering Principles," which addresses some of the important technical problems concerning software design. First taught in 1976 by NRL, the course is now presented annually by NRL in cooperation with the Naval Postgraduate School (NPS).

In "Software Engineering Principles," we introduce several important design principles that encourage production of correct, understandable, and easily changed software products. We also examine some software engineering research that may result in valuable design principles in the near future. The course will not transform the student into an expert designer, but should improve his ability to evaluate software proposals, progress, and products. He should also better appreciate design approaches, design problems, and ongoing research.

Coverage of each course topic typically involves lectures, examples, exercises, sample solutions, and exercise discussions. The schedule is rigorous; therefore, we recommend that the student summarize the major points raised in each lecture and exercise discussion. We encourage him to ask questions if he is having trouble isolating the major points of a topic or if he is confused by details. The student should also challenge statements or sample exercise solutions that seem in error or that do not agree with his experience. In the past, student questions have led to many lively discussions, cleared up some hidden confusions, and sometimes resulted in changes to course materials.

The following persons have prepared this year's materials.

Kathryn Heninger Britton
Louis Chmura
Paul Clements
Constance Heitmeyer
David Parnas
John Shore
Janet Stroup
David Weiss

PREFACE

Much of the material derives from earlier versions of the course. Kathryn Heninger Britton, Louis Chmura, Paul Clements, David Parnas, John Shore, and David Weiss, together with the following persons, developed those versions.

Honey Elovitz
John Guttag
Richard Hamlet
Cynthia Irvine
Rodney Johnson
Rudolph Krutar
Michael McClellan
Pam Mayo
Lee Nackman
Barbara Trombka
Helen Trop

Dean W. M. Woods and Ms. Ruby Kapsalis of NPS's Department of Continuing Education have handled many of the administrative details associated with the course and were responsible for making the local arrangements for its presentation in Monterey. Professor Gordon Bradley, Colonel Roger Schell, Lt. Commander Ronald Modes, and Mr. William Faulkner of the Computer Science Department were responsible for providing computer services and support for the course programming assignment.

Preparation of the notebook has been in the general charge of Janet Stroup. Her work was facilitated by the use of previous versions of the materials prepared by Ms. Stroup, Georgine Spisak, and Sarah McCray.

"Software Engineering Principles" originated as part of the NRL's Software Engineering Project, which the Naval Electronic Systems Command originally funded under Program Element 62721N, Task XF21-241-021.

Schedule

Day 1, Monday, 3 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders*
0800-0830	Registration			JS
0830-0930	Course Overview	Lecture	GEN.1	DP
0930-0945	Break			
0945-1015	Personal Experiences	Exercise	GEN.2	LC
1015-1045	Results of Exercise	Discussion		LC
1045-1100	Break			
1100-1130	The A-7 Project	Lecture	GEN.3	DP
1130-1200	The MMS Project	Lecture	GEN.4	CH
1200-1315	Lunch			
1315-1415	Program Families: What and Why	Lecture	PF.1	CH
1415-1430	Break			
1430-1530	The UGH Message Processing (MP) System	Reading and discussion	MP.1	CH, FR
1530-1545	Break			
1545-1615	MP as a Family of Programs	Exercise	PF.2	CH
1615-1645	Results of Exercise	Discussion	PF.3	CH
	Pseudo-Code Language Description	Homework	GEN.5	

*CH: C. Heitmeyer DP: D. Parnas DW: D. Weiss EN: E. Newhire
FR: F. Rat JS: J. Stroup KB: K. Heninger Britton LC: L. Chmura
OD: O. U. DeZeeman PC: P. Clements

Day 2, Tuesday, 4 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0830	A Minimal Member of the MP Family	Exercise	PF.4	CH
0830-0900	Results of Exercise	Discussion	PF.5	CH
0900-0930	Pseudo-Code Language Description	Discussion	GEN.5	LC
0930-0945	Break			
0945-1045	Desired Responses to Undesired Events	Lecture	UE.1	LC
1045-1100	Break			
1100-1130	MP and UEs	Exercise	UE.2	LC
1130-1200	Results of Exercise	Discussion	UE.3	LC
1200-1315	Lunch			
1315-1445	Family Development by Stepwise Refinement	Lecture	PF.6	CH
1445-1500	Break			
1500-1530	Applying the Program Family Principle	Lecture	PF.7	CH
1530-1630	Decomposition into Modules	Lecture	MOD.1	DP
	MP Typical Modular Structure	Homework	MP.2 MP.3	

SCHEDULE

Day 3, Wednesday, 5 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0830	MP Typical Modular Structure	Discussion	MP.2 MP.3	DP
0830-0900	Change and the Original MP Modular Structure	Exercise	MOD.2	DP
0900-0930	Results of Exercise	Discussion	MOD.3	DP
0930-0945	Break			
0945-1045	Modular Structure of Complex Systems	Lecture	MOD.4	DP
1045-1100	Break			
1100-1130	MP Secrets	Exercise	MOD.5	LC
1130-1200	Results of Exercise	Discussion	MOD.6	LC
1200-1315	Lunch			
1315-1345	MP Improved Modular Structure	Reading	MP.4	DP
1345-1415	Change and the Improved MP Modular Structure	Exercise	MOD.7	DP
1415-1430	Break			
1430-1500	Results of Exercise	Discussion	MOD.8	DP
1500-1530	The Military Address System (MADDS)	Reading and discussion	MADDS.1 MADDS.2	LC
1530-1545	Break			
1545-1615	MADDS Modular Structure	Exercise	MADDS.3	LC
1615-1645	Results of Exercise	Discussion	MADDS.4	LC
	Using the Computer System	Homework	MADDS.5	

Day 4, Thursday, 6 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0900	Intermodule Interfaces and UEs	Lecture	UE.4	DP
0900-0915	Break			
0915-0945	MP Intermodule Interfaces and UEs	Exercise	UE.5	LC
0945-1015	Results of Exercise	Discussion	UE.6	LC
1015-1030	Break			
1030-1130	What are Specifications?	Lecture	SPEC.1	DP
1130-1200	An Informal Functional Specification for the MP Message Holder Module	Exercise	SPEC.2 MP.5	LC
1200-1345	Lunch (Guest Speaker)			
1345-1415	Results of Exercise	Discussion	SPEC.2 MP.5	LC
1415-1430	Break			
1430-1600	Formal Functional Specifications	Lecture	SPEC.3	DP
1600-1615	Break			
1615-1730	Using the Computer System	Terminal	MADDS.5	PC

Day 5, Friday, 7 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0900	Abstract Interface Modules and Their Value (Part 1)	Lecture	ABS.1	DP
0900-0915	Break			
0915-1015	Abstract Interface Modules and Their Value (Part 2)	Lecture	ABS.1	DP
1015-1030	Break			
1030-1130	Using the MP Abstract Interface	Exercise	ABS.2 MP.6	DW
1130-1200	Results of Exercise	Discussion	ABS.3	DW
1200-1315	Lunch			
1315-1415	Informal Functional Specifications for MADDS Modules	Reading and discussion	MADDS.6	LC
1415-1430	Break			
1430-1530	Coding Specifications	Lecture	SPEC.4	DW
1530-1545	Break			
1545-1700	The Military Address System	Programming	MADDS.1- MADDS.7	PC
	Host-At-Sea (HAS) System	Homework	HAS.1	

Day 6, Monday, 10 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0900	Host-At-Sea (HAS) System: Requirements Summary	Reading and discussion	HAS.1 PF.8	DP
0900-0915	Break			
0915-1015	HAS Data Acquisition and Transmission Software: Program Design Specification	Reading	HAS.2	DP
1015-1045	Evaluation of the Proposed HAS Modular Structure	Discussion	HAS.2	DP
1045-1100	Break			
1100-1200	Identifying HAS Modules	Exercise	MOD.9 HAS.1 HAS.2	DW
1200-1315	Lunch			
1315-1415	Results of Exercise	Discussion	HAS.3	DW
1415-1430	Break			
1430-1530	Hierarchy Survey	Lecture	HIE.1	DP
1530-1545	Break			
1545-1700	The Military Address System	Programming	MADDS.1- MADDS.7	PC

SCHEDULE

Day 7, Tuesday, 11 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0930	Designing a Uses Hierarchy	Lecture	HIE.2	KB
0930-0945	Break			
0945-1015	Uses Hierarchy for an Address System	Exercise	HIE.3	KB
1015-1045	Results of Exercise	Discussion	HIE.4	KB
1045-1100	Break			
1100-1200	The Uses Hierarchy and UEs	Lecture	UE.7	DW
1200-1315	Lunch			
1315-1415	Language Selection	Lecture	LANG.1	DW
1415-1430	Break			
1430-1700	The Military Address System	Programming	MADDS.1-MADDS.7	PC
	A Structured View of HAS	Homework	HAS.4 pp. 13-43 thru 13-47	

Day 8, Wednesday, 12 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0930	Process Structure of Software Systems	Lecture	PROC.1	KB
0930-0945	Break			
0945-1015	MP Process Structure	Exercise	PROC.2	DW
1015-1045	Results of Exercise	Discussion	PROC.3	DW
1045-1100	Break			
1100-1200	A Structured View of HAS	Reading and discussion	HAS.4, pp. 13-43 thru 13-47	DW
1200-1345	Lunch (Guest Speaker)			
1345-1515	Process Synchronization	Lecture	PROC.4	KB
1515-1530	Break			
1530-1600	Academic Poppycock	Reading and discussion	HAS.5	DW
1600-1730	The Military Address System	Programming	MADDS.1-MADDS.7	PC
	A Structured View of HAS	Homework	HAS.4 HAS.5	

SCHEDULE

Day 9, Thursday, 13 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0845	A Structured View of HAS	Debate	HAS.4 HAS.5	EN, OD
0845-0930	A Structured View of HAS	Discussion	HAS.4 HAS.5	KB, PC
0930-0945	Break			
0945-1045	Separation of Concerns	Reading	HAS.6	KB
1045-1100	Break			
1100-1200	Ada	Lecture	LANG.2	DW
1200-1315	Lunch			
1315-1415	Implementing Processes in HAS	Reading	HAS.7	DW
1415-1430	Break			
1430-1700	The Military Address System	Programming	MADDS.1-MADDS.7	PC
	Implementing Processes in HAS	Homework	HAS.7	

Day 10, Friday, 14 August 1981

Time	Topic	Kind of Session	Relevant Material	Session Leaders
0800-0830	Implementing Processes in HAS	Reading	HAS.7	DW
0830-0930	Implementing Processes in HAS	Discussion	HAS.7	DW
0930-0945	Break			
0945-1115	Documentation Guidelines	Lecture	DOC.1	KB
1115-1130	Break			
1130-1230	Course Review	Lecture	GEN.6	KB
1230-1300	Course Evaluation	Evaluation	EVAL.2	JS

GEN.1 Course Overview

LECTURE

I. What is Software Engineering?

A. You already know programming?

B. You already have languages?

C. What then are the special characteristics of software?

1. Multiperson involvement

2. Multiversion production and maintenance

3. Handling of undesired events (UEs)

4. Usual common additional properties

a. Machine "near" — machine dependent

b. Large size

c. Efficiency, reliability important

d. Robustness important

II. Characteristics of well-structured software

A. Can be verified one part at a time

B. Can be changed one part at a time

C. Can be read one part at a time and each part only once -- characteristics of both program and documentation, not just documentation

D. Subsets work -- ability to tailor to actual needs

E. Meaningful error messages

F. Effective utilization of resources

G. Extensibility from outside

III. Characteristics of badly structured software

A. In one eye and out the other (smart people)

B. Must remember many arbitrary facts to understand code changes

C. Modification requires changes in unpredictable places

D. System integration a real effort

IV. Various times at which decisions are made, e.g.,

A. Early design time

B. Program writing time

C. Compile time

D. Load time

E. Run time

V. Decision postponement

VI. Three software structures

A. Module structure

B. Program-uses structure

C. Process structure

VII. The meaning of abstract and the use of abstractions

VIII. Goals of this course: After you finish, you should

A. Be better able to recognize bad software design

B. Be able to recognize good software design

C. Be able to recognize contractor BS, snow, run-around, incompetence, etc.

D. Be able to evaluate contractor performance

E. Have a sense of the state of the art in software engineering

IX. Non-goals of this course

A. You will not be a system designer

B. You will not be a super programmer

C. You will not be a philosopher about software methodology

X. Discussion of contents and schedule

A. Contents (pp. i-iv)

1. Nine topics, three examples

2. Document IDs (e.g., GEN.1)

3. Materials to be handed out during course (e.g., MADDS.7)

4. Pseudo-code language description (GEN.5)

5. Course evaluations

a. Comment sheets (EVAL.1)

b. End of course (EVAL.2)

6. Glossary (GLOS.1)

7. Bibliography (BIB.1)

a. Articles available

b. References for each section

B. Schedule (pp. vii-xvi)

1. Full days

2. Sessions

a. Lectures

b. Case studies

c. Exercises (collected)

d. Discussions

3. Relevant materials

4. Session leaders (e.g., DP)

5. Programming assignment (time allocated)

6. Guest speakers

GEN.2 Personal Experiences

EXERCISE

Name: _____

This exercise gives you a chance to relate the topics mentioned in the overview lecture to your own experiences in software development.

Briefly describe your experiences, if any, for the situations listed below. Attached at back of this exercise, you will find some sample experiences of former students. We will ask a few students to relate their "war stories" duing the upcoming discussion period.

1. Describe a situation in which a small change in the requirements resulted in many changes all over a system.

2. Describe a situation in which a subset of an existing system was needed, but it was not possible simply to remove the unneeded parts -- rewriting was necessary.

3. Describe a situation in which the information that you needed in order to understand one module in a system was finally found somewhere else in the system documentation.

4. Describe a situation in which a decision that was made when a system was specified could and should have been postponed until assembly time or run time.

SOME PAST ANSWERS TO GEN.2

Question 1. Describe a situation in which a small change in the requirements resulted in many changes all over a system.

1. "In 1976, it was requested that the Order of Merit System (at USMA) be dropped in favor of an alphabetical graduation scheme. This seemingly simple administrative decision required not only a complete overhaul of the term end processing system, but also impacted many seemingly unrelated areas — such as the Army promotion list for the new graduates which was found to be based on the order of merit, also the branch drawing system — based on order of merit."

2. "Requirement to utilize an optional H/W control selection to compensate for system level malfunction needed only single bit manipulation in H/W to effect. Result was 9 S/W module changes, some redesign in S/W because — some S/W hard coded element not expecting it to ever change, some S/W read data base item and then overwrote it in S/W, some S/W got data base item from wrong place — this was found 3 years after deployment and was costly to correct — large documentation cost incurred."

3. "This example perhaps describes the converse of the question, i.e., a small change was not done because the overall impact was too costly. The simple change was to 'hard copy' upon operator command the results of menu changes on a CRT-like device. The program was designed such that the change required a pulling apart of the display code and restructuring it to accomplish the desired result."

Question 2. Describe a situation in which a subset of an existing system was needed, but it was not possible simply to remove the unneeded parts — rewriting was necessary.

1. "USMA wanted to implement the Air Force 'CIAPS' system of providing procurement support to HQ's local procurement division. The 'AF' CIAPS system consisted of approximately 167 programs of which 'we' needed 16. Resulting problem was that CIAPS was developed for their base level Burroughs 3500 — we were configured on H6000 — and planning a conversion to Univac 1110. Rewrite was decided upon — using the CIAPS programs as a basis."

2. "It was desired to update a sequential file using tape, as well as cards. However, because the formats of the input cards and tapes differed so greatly (several tape records = 1 card record) it was simpler to write a whole new program rather than try to incorporate the new requirement into the existing program."

3. "In developing a program for an agency which must communicate on TADIL-B, it was not possible to use an existing module which does TADIL-B processing for another agency. This has happened repeatedly. For every type agency which uses TADIL-B, there is a unique TADIL-B processor."

Question 3. Describe a situation in which the information that you needed to understand one module in a system was finally found somewhere else in the system documentation.

1. "Such a situation occurred in an EW system where parameters required by a controlling program which interacted with a number of distributed original processors were not specified in the documentation of the main control program. All of the distributed programs had to be examined to understand all the parameters being used and what the specific functions of the controlling program were."

2. "System documentation included a statement that read 'S+A=P pg 33Wrth' which meant page 33 in Wirth's book, Structure Plus Algorithms Equals Programs."

3. "Involved Navy personnel strength projections performed with a simulation model. Model was written to perform certain processes sequentially, and the processes shared common routines, i.e., rounding routines, random number generator, etc. In documentation, the descriptions of the common routines were spread out throughout system's description."

Question 4. Describe a situation in which a decision that was made when a system was specified could and should have been postponed until assembly time or run time.

1. "A data reduction system for handling A-7 flight data had to be able to handle flight recorder outputs from several different OFPs (Operational Flight Programs). The same variables were often in different positions on the different tapes. The programmer introduced an ungodly number of flags and complicated branching logic in order to handle all the 'special cases.'"

2. "Memory size of TRIDENT Command and Control system has grown throughout system development with impact on the executive and in many cases on the subsystem modules. It would appear that memory allocation could have been generalized in the development process and then after total requirements were known, memory allocation could have been determined."

3. "Because hardware must be procured at the beginning of development, no one can tell if it is sufficient for the job. Only after compilation (just how efficient is that compiler anyway?) should the exact amount of hardware be gotten. At the very least, some experimental coding must be done."

GEN.3 The A-7 Project

LECTURE

I. Problems with tactical software for DoD aircraft as seen by Navy's A-7 maintainers

A. Mostly unstructured assembly language code

B. Little documentation

1. Design and analysis documents were not purchased from the contractor or have not been maintained

2. The reasons for doing things have been lost

C. Additions and deletions are risky

1. Almost impossible to assess impact or magnitude of a change

2. Ripple effect

3. Personnel making a change must understand entire program

4. Total program must be retested

D. Difficult to validate

1. No independent statement of the requirements except the code

2. Reliability always unknown

E. Training of personnel is difficult, requiring about 1 year before a significant contribution can be made

II. Claimed benefit of software engineering principles is well-structured software

— Can be verified one part at a time

— Can be changed one part at a time

— Can be understood one part at a time

III. Questions

A. Why doesn't the DoD use software engineering principles?

1. No convincing test

2. No models to emulate

B. Would the DoD benefit from well-structured programs? How can we find out? — By building and using one.

C. Is it feasible to follow software engineering principles while building embedded systems with tight memory and time constraints? How can we find out? — By building an operational system and following the principles.

D. How much memory and execution time does good structure cost? How can we find out? — By comparing two equivalent programs, one with good structure, one without.

E. What if we fail?

1. Learn why

2. Stop preaching non-truth

IV. The A-7 Project

A. Redeveloping the operational flight program for the Navy's A-7 aircraft using the following principles

	PROBLEMS					
PRINCIPLES	Low Level Code	Poor Doc.	Change Risky	Validation Hard	Training Hard	Requirements Change
Requirements Definition Techniques		X		X	X	X
Information Hiding Modules	X	X	X		X	X
Abstract Interfaces	X	X	X		X	X
Cooperating Sequential Processes	X			X	X	X
Process Synchronization Primitives	X	X	X			
Uses Hierarchy			X		X	X
Resource Monitor Modules	X		X			X
Formal Specifications			X			
Disciplined Programming	X		X	X	X	
Program Verification			X	X		

B. Stages

1. Define requirements, August 1978

2. Redesign program, November 1980

3. Rebuild program, July 1982

4. Undergo Naval Weapons Center (NWC) acceptance tests, November 1982

5. Compare new program to old, January 1983

C. Status

1. *Software requirements specification for the A-7E aircraft*

2. Publication of specification methodology

3. High-level design documentation for new A-7E program

4. Interface specifications for device modules: Model of abstract interface methodology

5. Specifications for virtual machine

D. Influence on this course

V. References

Heninger, K. L. 1980. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application." Trans. on Software Engineering, vol. SE-6, no. 1, pp. 2-13.

Heninger, K. L.; Kallander, J.; Parnas, D. L.; and Shore, J. E. 1978. Software Requirements for the A-7E Aircraft, Naval Research Laboratory Memorandum Report no. 3876.

GEN.4 The MMS Project

LECTURE

I. History of message system development in DoD

A. Military crises in late 1960s led to several Congressional investigations into military communications

B. Problems of military message systems uncovered by Congress

1. Delayed message delivery

2. Human errors

3. Lack of standardization

4. Duplication of effort

C. Congressional directives regarding future message system development in DoD

1. Centralized approach

2. Greater standardization

D. Areas of expected cost savings

1. Development

2. Maintenance

3. Documentation

4. Training

II. The Military Message Systems (MMS) Project

A. One message system will not suffice for DoD

1. Some organizations require special functions

2. Varying organizational procedures and preferences

3. Different computer hardware

4. Different terminals

5. Different incoming message volumes

6. Different message storage requirements

B. In future years, DoD will need several message systems with many common features but many differences as well

C. Project goal: Develop a family of military message systems using current software engineering principles

1. Provide useful product to DoD

2. Demonstrate the application of software engineering principles to a complex problem area in DoD

a. Family methodology

b. New techniques for specifying requirements

c. Abstract data types

d. Information-hiding modules

e. Abstract interfaces

D. Stages

1. Requirements specification for family

2. Design specification for family

E. Status

1. Investigation of existing family members (HERMES, SIGMA, NMIC-SS)

2. Requirements analysis

a. Functions

b. Primitive operations

c. Impact of multilevel security

III. Questions

A. Is it feasible to follow software engineering principles when building a set of systems with a wide range of characteristics?

B. Do the claimed benefits apply to large, complex systems or are they confined to small, simple programs?

IV. Reference

Heitmeyer, C. L.; and Wilson, S. H. 1980. "Military Message Systems: Current Status and Future Directions." IEEE Trans. on Communications, vol. COM-28, no.9, pp. 1645-1654

GEN.5 Pseudo-Code Language Description

READING

I. Introduction

Algorithms sometimes appear in this notebook to illustrate concepts being introduced. The algorithms are presented as abstract programs written in a programming language similar to ALGOL and FORTRAN. This document briefly sketches the pseudo-code language constructs and their meanings.

II. Character Set and Basic Constructs

The character set used to construct pseudo-code constructs comprises the 26 letters, both upper and lower case, the 10 decimal digits, the space character, and the following special characters.

" ( ) * + , - . / : ; = _ [ ]

There are six elementary language constructs: identifiers (such as Alpha, sensor_reader), numbers (such as 1, 32.4), strings (such as "I am a string."), delimiters (such as ;), keywords (such as begin, for), and operators (such as or, +).

Identifiers must start with an uppercase or lowercase letter but then may contain any character and may be of any length. Numbers may be integers, which contain no decimal point, or reals, which do. Strings are characters enclosed in quotation marks. The primary delimiter is the semicolon, which terminates declarations and statements. Tables 1 through 3 following this description list all keywords and operators. Keywords are always underlined.

Variables, or data objects, are provided identifier names, data types, and other attributes in declarations and are manipulated in statements. Expressions consist of variables and numbers strung together by appropriate operators.

III. Statements

Statements manipulate variables and control the order of execution of other statements. All statements terminate with a semicolon. Statements can appear anywhere on a line.

A. Assignment statement

Example: x := e;

Explanation: := is the assignment operator. The variable x is assigned the value represented by expression e.

B. Comment statement

Example: comment This is a comment statement;

Explanation: A comment starts with the keyword comment. All text between the keyword and the next semicolon is assumed to be the comment text. A comment may be longer than one line.

C. Statement label

Example:
```
labelname: x:= y; comment any statement may have a label
                                associated with it;
```

Explanation: A statement label is an identifier followed by a colon appearing before the statement being labeled. Note: Because this language does not have a "go to", labels serve only as convenient markers.

D. Compound statement

Example:
```
begin
comment All the statements enclosed between begin and end
        make up a compound statement;
x:= y;
y:= z * 3;
label: temp:= y + 5 * x;
first:= 1; second:= 2; comment three statements on one
                                line;
end;
```

Explanation: A compound statement is a way of grouping several statements together; each statement is then executed sequentially. This is accomplished by enclosing the statements between begin and end. Any legal statement may appear within the begin-end pair.

General form:

```
begin
  stm 1;
  stm 2;
   .
   .
   .
  stm n;
end;
```

where each stm i is a legal statement.

E. If statement

Example:
```
if x gt y
then x:= y;
else y:= x;
end-if;
```

Explanation: The if statement is used to test for a specified condition. In the above example, this test is for x greater than y. If the condition is true, the then part of the statement is executed. If the condition is false, the else part of the statement is executed. The else part is optional; if this part does not appear, execution continues with the statement following the if statement.

General form:

```
if logical expression
then statement sequence
else statement sequence
end-if;
```

where logical expression is any expression resulting in a true or false evaluation.

F. While statement

Example:
```
while x le y do
   begin
      z:= A(I);
      x:= x + 2;
      i:= x;
   end;
end-while;
```

Explanation: The while statement is a looping construct. As long as the expression following the while is true, the compound statement is executed.

General form:

while logical expression *do* statement sequence end-while;

where logical expression is as previously defined.

G. Case statement

```
Example: case getmsgtype(message) of

             //ship//
                 begin comment This compound statement is executed
                               when the value of getmsgtype(message)
                               is ship;
                 shiprec:= shiprec + 1;
                 report_request:= true;
                 end;

             //air//
                 begin comment This compound statement is executed
                               when the value of getmsgtype(message)
                               is air;
                 airrec:= airrec + 1;
                 report_request:= true;
                 end;

             //history//
                 begin comment This compound statement is executed
                               when the value of getmsgtype(message)
                               is history;
                 historyrec:= historyrec + 1;
                 report_request:= true;
                 end;

         end-case;
```

Explanation: In the above example, identifiers enclosed in double slashes (//) are used to name different situations and their values. The statement following the matched //name// field is then executed.

General form:

```
case arithmetic expression of
    //expression 1//
          statement
    //expression 2//
          statement
            .
            .
            .
    //expression n//
          statement
end-case;
```

where the value of arithmetic expression matches one of the expressions enclosed in slashes. When there is no match, execution continues after the end-case.

H. For statement

Example:
```
for I:= J step K until N do
    A[I]:= I + 1;
    B[I]:= A[I + 1];
end-for;
```

Explanation: The for statement is a looping construct with parameterized step size. The loop variable I is initialized to J and incremented once for each traversal of the loop by the step value K until I is greater than the upper bound variable N. Each execution of the loop executes the two assignment statements. Note: This statement is equivalent to

```
I:= J;
while I le N do
    A[I]:= I+1;
    B[I]:= A[I+1];
    I:= I + K;
end-while;
```

General form:

```
for var:= expression step expression until expression do
statement sequence end-for;
```

where var is a variable and expression is an arithmetic expression.

IV. Data Types and Declarations

All variables must be declared. A declaration provides a variable with an identifier name and identifies the variable's type and scope (see section V). A variable may also be declared to be private to the routine where it is used, global, or a parameter to the routine.

A. integer, real, and array

Example:
```
integer x;
real y;
real reel1, real2;
integer array z[1:10];
comment The following declaration defines a two-
        dimensional array of real values. The index for
        the first dimension may take on values from 1 to
        15, for the second dimension from 0 to 30;
real array two_dimensions[1:15, 0:30];
integer int1, int2, int3;
```

Explanation: integer indicates that the variable or variables following may assume only integer values.
real indicates that the variable or variables following may assume only real values.
integer array declares an array variable to be an array of integers. The array's dimensions are enclosed in brackets; lower bounds and upper bounds must both be indicated for all dimensions. All bounds must be integers. Arrays may be of any legal type: integer, real, boolean, character, string, semaphore (see below). Multidimensional arrays are declared by separating the dimensions by commas.

B. boolean

Example:
```
boolean x;
boolean x, y, z;
```

Explanation: boolean variables are variables that may have only one of two possible values — true or false. Boolean is synonomous with FORTRAN LOGICAL.

C. string and character

Example:
```
string str;
character char;
```

Explanation: Statements such as str:= "THIS IS A STRING ASSIGNMENT"; are permitted provided str is declared a string. Character variables are string variables of length one

D. buffer

Example:
```
integer buffer buf;
real buffer buf2;
string buffer bufstr;
integer buffer array buf[1:5];
real buffer array buf[5:12];
```

Explanation: Buffer variables hold information of a specified data type. For example, integer buffer buf; means that buf holds integers. The only legal operations on buffers are accept and deposit. The operator accept removes one unit of information from the buffer specified; deposit places one unit of information into the buffer specified. The unit of information is determined by the buffer's type. After an accept operation, the buffer has one fewer item; a deposit operation causes the value to be copied into the buffer. A buffer is a first-in first-out storage device. The number of spaces actually available in a buffer are specified at system generation time (details of the specification are not relevant here). Buffer space management is handled by the accept and deposit operations and is transparent to the buffer user. An accept (deposit) operation may result in delays of the programs using it when a buffer is empty (full).

Examples of the use of accept and deposit follow. Assume that buffer buf has previously had a value deposited.

```
begin
   integer buffer buf;
   integer item;
   accept (item, buf);
end;
```

The above example causes one integer from buffer buf to be removed from buf and placed into item; buf contains one less integer.

```
begin
   character buffer array buf[1:5];
   integer int;
   character char;
   int:= 2;
   char:= "a";
   deposit (char, buf[int]);
end;
```

This example causes the character in char to be placed in the second buffer of the buffer array buf. The contents of char are unchanged.

E. program and procedure

Example:
```
procedure proc(x,y,z);
program proc2(synch);
reentrant procedure reentproc(a);
integer procedure intproc(x,y,z);
```

Explanation: A procedure is a means of grouping together often-used code (sometimes called a subroutine); each time the procedure is called, the code is executed. If the procedure is a function (i.e., it returns a value), its declaration specifies the type of the returned value.

A program is used to specify the sequence of events within a process, i.e., the program defines the process. Several processes may be controlled by the same reentrant program (see below).

Reentrant procedures (reentrant programs) are procedures (programs) written in such a way that several processes may use the procedure (program) simultaneously. To accomplish this, the code must be separate from all data that is changed during execution. The code is shared but each process has its own copy of changeable variables.

Parameters to procedures and programs are enclosed in parentheses and separated by commas. Parameters must be declared within the procedure or program body to be of type parameter.

All declarations within a procedure or program must precede the compound statement that makes up the executable portion of the procedure or program.

```
Example:  procedure proc(x,y,z);
          parameter integer x;
          parameter real y;
          parameter string z;
          begin
            .
            .
            .
          end;
```

Procedures are called as in ALGOL by using the procedure name and a list of parameters enclosed in parentheses.

Note that there is no restriction on the data types of parameters so procedure names may be passed as parameters to other procedures. Consequently, the following declarations may occur:

```
procedure proc(parproc, x,y);
parameter procedure parproc;
```

F. semaphore

```
Example:  semaphore seml;
          semaphore array ten-sem[1:10];
          semaphore array five-sem[1:5];
```

Explanation: Semaphore is a variable type designed for the synchronization of processes that are proceeding in parallel at unknown speeds. Just as semaphores in railway systems are used to inform one train of the activities of another, semaphores in computer systems are used to inform one process of the activities of another.

There are only two legal operations on semaphores possible — P and V; they are invoked in the same way as procedures are called:

```
P(seml);
V(seml);
```

P and V are operators defined by Dijkstra for the data type semaphore. A V operation on a semaphore allows exactly one more P operation on the same semaphore to complete. This may cause a waiting process to continue.

A semaphore array is a set of semaphores with the same name; they are distinguished by an integer subscript, e.g., ten_sem[1], just as array elements. Semaphore arrays may only have one dimension unlike integer, real, or other kinds of arrays.

V. Scope of Variables

The scopes of all variables must be declared. There are three possible scope declarations.

1. parameter
2. global
3. private

Examples:
```
procedure proc(x);
parameter integer x;
global integer y;
private integer z;
```

Explanation: parameter indicates that the variable is a parameter to the procedure; global indicates that the variable is global to the procedure; private indicates that the variable is private (local) to the procedure.

Table 1. STATEMENT-RELEVANT KEYWORDS

KEYWORD	USE
begin	indicates start of compound statement
end	indicates end of compound statement
case	indicates start of case statement
//item//	marks compound statement in a case statement that should be executed if the arithmetic expression matches the item
of	part of case statement
end-case	indicates end of case statement
comment	starts comment statement
for	starts for statement
step	indicates the step value in for statement
until	indicates the upper bound in for statement
end-for	ends for statement
if	starts if statement
then	starts the part of if statement to be executed if the expression is true
else	starts the part of if statement to be executed if the expression is false
end-if	ends if statement
infinite	used for the largest number that can be represented
null	used to represent a null value, nothing (different from zero)
true	logical value (type boolean)
false	logical value (type boolean)
while	starts while statement
do	part of while statement
end-while	ends while statement

Table 2. DECLARATION-RELEVANT KEYWORDS

KEYWORD	USE
array	declares a variable to be an array
boolean	declares a variable to be a logical variable that may take on the values true or false
buffer	declares a variable to be a buffer
buffer array	declares a variable to be an array of buffers
character	declares a variable to be a character
global	declares a variable to be global
integer	declares a variable to be an integer
parameter	declares a variable to be a parameter to a procedure
private	declares a variable to be local to procedure or program
procedure	procedure heading declaration
program	program heading declaration
real	declares a variable to be a real
reentrant	declares a procedure to be reentrant
semaphore	declares a variable to be a semaphore
semaphore array	declares a variable to be a vector of semaphores
string	declares a variable to be a string

Table 3. OPERATOR SYMBOLS

OPERATOR SYMBOL	MEANING
or	logical operator or
and	logical operator and
not	logical operator not
not equal	logical operator not equal
=	logical operator equal
lt	logical operator less than
gt	logical operator greater than
le	logical operator less than or equal to
ge	logical operator greater than or equal to
:=	assignment operator
*	multiplication operator
+	addition operator
-	subtraction operator
/	division operator
**	exponentiation operator
;	statement terminator
:	statement label terminator
( )	used in expressions or to enclose parameter lists or to group subexpressions
,	used to separate identifiers in declarations or parameter lists
[ ]	used to enclose array indexes and semaphore vector indexes
"	starts and stops a string constant
P	semaphore operator to request a "pass"
V	semaphore operator to permit a "pass"
accept	buffer operator to remove one item from a buffer
deposit	buffer operator to put one item into a buffer

GEN.6 Course Review

LECTURE

I. Characteristics of well-structured software

A. Can be written one independent part at a time

1. Writing later parts doesn't require rewriting earlier parts

2. Writing based on fixed, written specifications and assumptions -- reduces need to communicate or negotiate

B. Can be verified one part at a time

C. Can be changed one part at a time

D. Both program and documentation can be read one part at a time

E. Subsets give ability to tailor software according to needs and resources

F. Effective utilization of resources

II. Groundwork for a flexible design — describe more than functional requirements

A. Possible changes — systems can't be flexible in every way

B. Desired response to undesired events — part of desired behavior but it helps to think about it separately

C. Useful subsets

III. Dividing the system into modules

A. Review of modules

B. Hiding secrets based on expected changes

C. Finding good work assignments

D. Iterating for submodules

E. Other benefits of modularity

F. Examples from HAS

IV. Identifying and specifying the access programs for modules

A. Review of interfaces

B. Abstract interface modules

C. General module interfaces

D. Examples from HAS

V. Identifying internal programs

A. Part of the work assignment

B. Internal programs may be functions or process definitions

C. Examples from HAS

VI. Designing the uses hierarchy

A. Based on expected subsets

B. Based on implementation and testing considerations

C. Based on degradation considerations

D. Example from HAS — see diagrams (pp. 1-46 to 1-48)

VII. Expressing module interfaces and program designs

A. Informal specifications

B. Formal specifications of module interfaces

C. Abstract programs for functions and process definitions

D. Examples

VIII. Processes

A. Could be executed in parallel

B. Capture major sequencing decisions

C. Allow changes in configuration

IX. Review in terms of basic priciples

A. Information hiding

B. Separation of concerns

C. Being explicit about design decisions

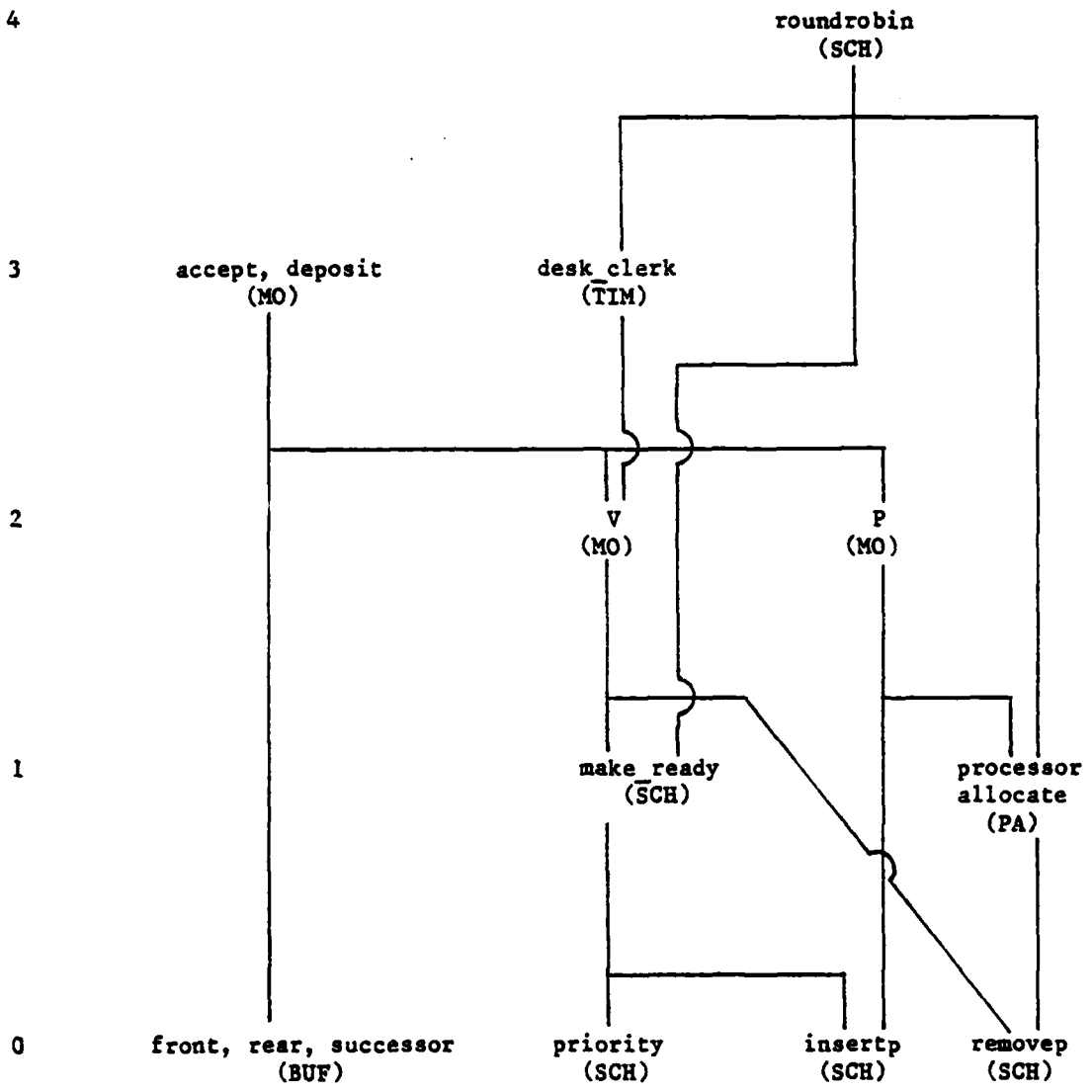
D. Deferring design decisions

E. Program families

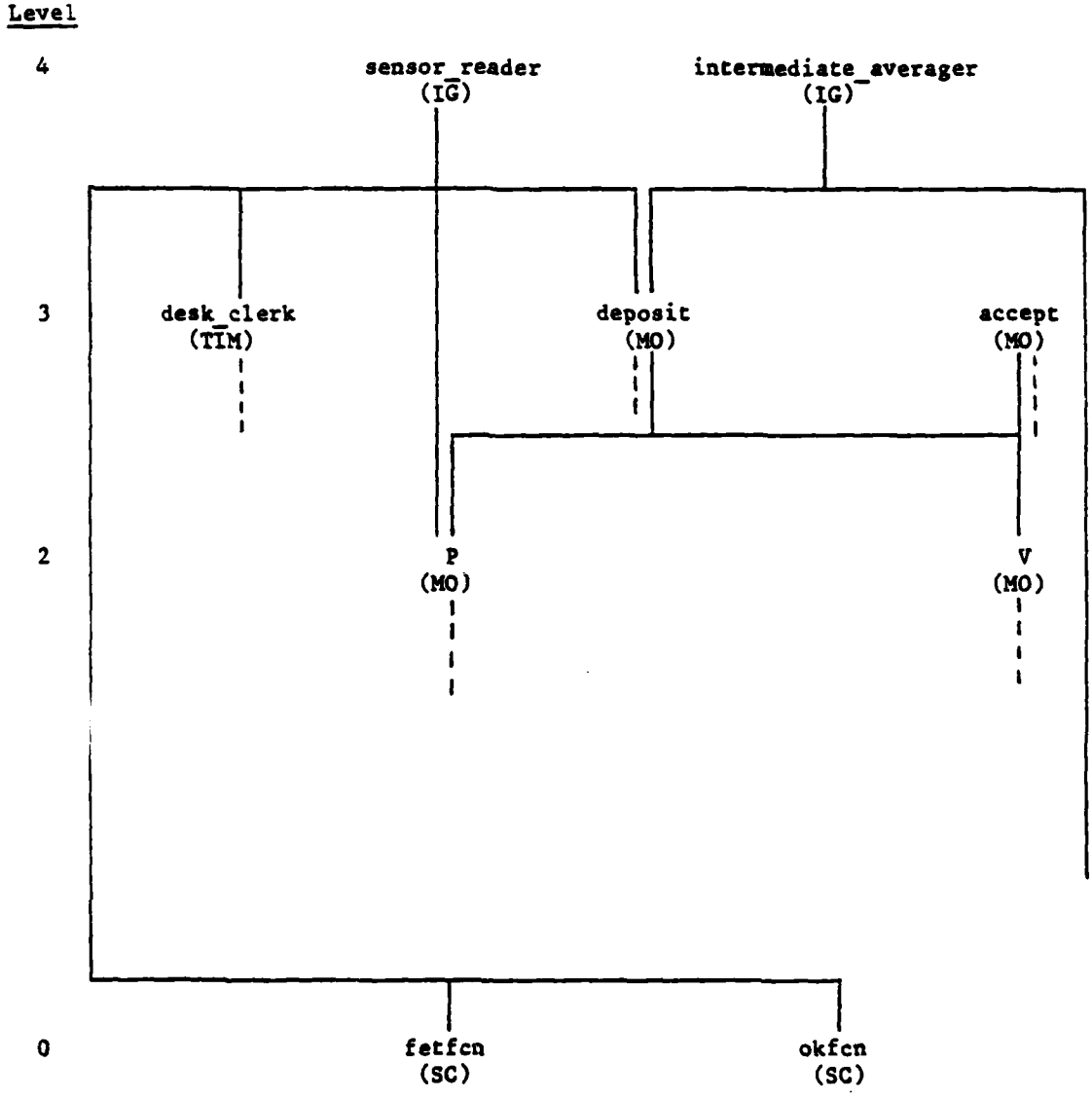
F. Discipline in design

Discipline in documentation

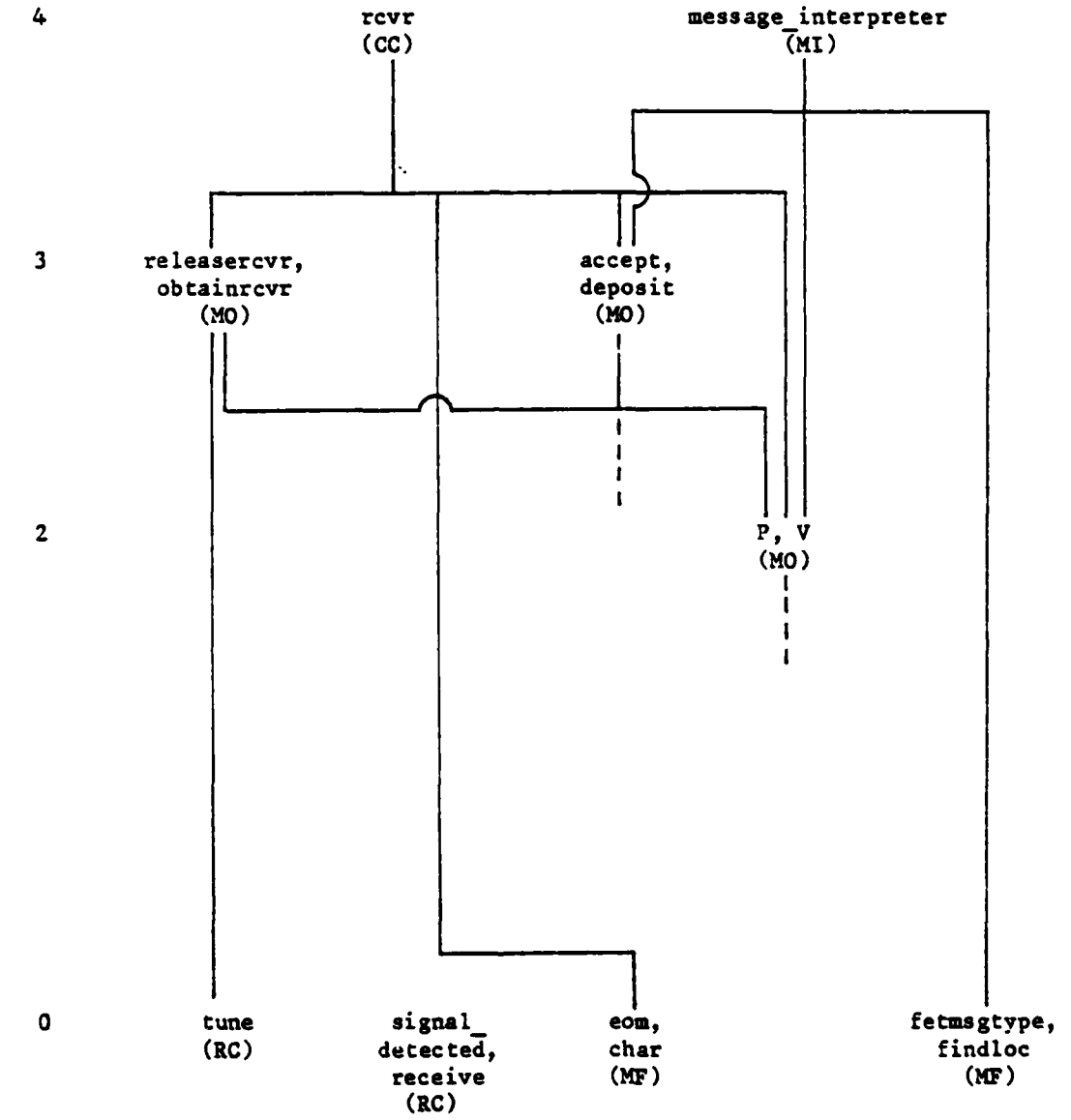
Discipline in programming


Level
4
3
2
1
0
roundrobin
(SCH)
accept, deposit
(MO)
desk_clerk
(TIM)
V
(MO)
P
(MO)
make_ready
(SCH)
processor
allocate
(PA)
front, rear, successor
(BUF)
priority
(SCH)
insertp
(SCH)
removep
(SCH)

A


Level
4
sensor_reader
(IG)
intermediate_averager
(IG)
3
desk_clerk
(TIM)
deposit
(MO)
accept
(MO)
2
P
(MO)
V
(MO)
0
fetfcn
(SC)
okfcn
(SC)

B


Level
4
rcvr
(CC)
message_interpreter
(MI)
3
releasercvr,
obtainrcvr
(MO)
accept,
deposit
(MO)
2
P, V
(MO)
0
tune
(RC)
signal_
detected,
receive
(RC)
eom,
char
(MF)
fetmsgtype,
findloc
(MF)

C

PF.1 Program Families: What and Why

LECTURE

I. Definition: A set of programs will be called a program family if they have so much in common that it pays to study their common characteristics before investigating the special properties of individual programs.

II. Hardware analogy: System/370.

III. Typical program family: The various versions and releases of a manufacturer's operating system.

IV. Why do large organizations so often have sets of programs with similar functions?

A. Different parts of an organization develop programs with similar purposes without knowledge of each other.

B. A program developed on the basis of one set of constraints turns out to perform poorly under other conditions and is too hard to adjust.

C. The general program is late in development; emergency project produces special case for immediate use. Incompatibilities result in long-term existence of system intended only as a temporary measure.

D. A program developed for a large installation turns out to be impractical or even unusable for a small one.

E. A program developed for a small installation turns out to be unable to make effective use of the resources of a large installation.

F. One user wants services which were not anticipated by the developers of an earlier system.

G. Different computers.

H. "I can do it better."

V. What are the disadvantages to an organization of having a set of similar, independently developed systems?

A. Incompatibilities lead to duplication of interfacing programs.

B. Maintenance personnel get confused by false similarities and misleading or superficial differences.

C. Organization-wide changes must be incorporated in each system.

D. Increased costs for storage, documentation, etc.

E. Effort to improve must be distributed and cannot be shared.

VI. What are the disadvantages to an organization of having programs which were not designed to change?

A. Some changes are made poorly.

B. Some changes cannot be made at all.

C. Maintenance and improvement costs are higher.

D. Readiness is impaired because of long completion times.

VII. Solution: Simultaneous development of a set of programs (Program Family Development).

A. Maximize what they have in common.

B. Minimize the differences.

C. Localize the differences.

D. Reduce development costs by sharing among versions.

E. Reduce maintenance costs by sharing among versions.

F. Reduce documentation costs, training costs, etc.

VIII. How can the members of a program family vary?

A. Programs can be functionally identical but make different resource trade-offs.

B. Programs can be identical except for size parameters.

C. Programs can be subsets of the same super program.

D. Programs can be built on a common "base" (kernel) but provide different user interfaces to meet varying needs.

E. Programs can have a common "facade" but a different base.

IX. Sequential completion versus abstract design.

A. What are design decisions?

B. Why is the order of design decisions significant?

C. Decision trees.

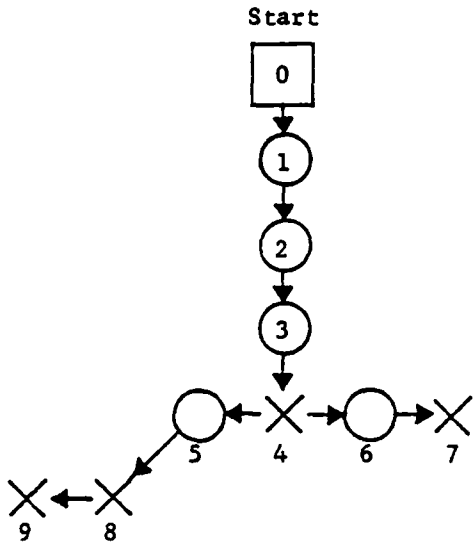


Figure 1 - Representation of development by sequential completion. Note: Nodes 5 and 6 represent incomplete programs obtained by removing code from program 4 in preparation for producing programs 1, 8, and 9.

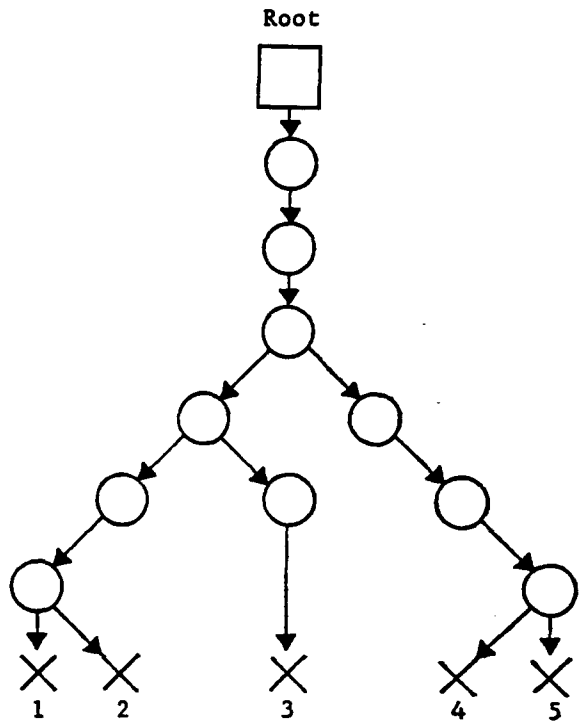


Figure 2 - Representation of program development using "abstract decisions."

Symbols:  is the set of initial possibilities;

 is the incomplete program;

 is the working program.

C. What possible orders have been proposed?

1. Outside in (top down).

2. Bottom up.

3. Most solid first.

D. Implications of thinking about the family -- how to tell good design decisions from bad ones.

X. Reference

Parnas, D. L. 1976. "On the Design and Development of Program Families." IEEE Trans. on Software Engineering, vol. SE-2, no. 1, pp.1-9.

PF.2 MP as a Family of Programs

EXERCISE

Name:_____

Based on your understanding of MP.1, briefly answer each of the following questions.

1. Why might different versions have different memory requirements?

2. a. List some reasons why we cannot assume that the same type of terminal will be used in all MP installations.

   b. List several properties that we can assume will be true of all terminals used.

3. Why might we have to use different programming languages for different versions?

4. Why might the information included in the logs be different in different versions?

5. Why might the operator interface on some versions of MP differ from the operator interface on others?

PF.3 MP as a Family of Programs

EXERCISE SOLUTION

The following answers are just some of the possibilities.

1. Why might different versions have different memory requirements?

   - Different numbers of users to be served.
   - Different response characteristics (dependent on minimum acceptable response time, AUTONOYS message load).
   - Different rates of incoming and outgoing message volumes.
   - Retention time for messages.
   - Functional capabilities chosen.
   - Internal message conventions.

2.a. List some reasons why we cannot assume that the same type of terminal will be used in all MP installations.

   - Mobile terminals must be special ruggedized versions.
   - Some users may want inexpensive terminals.
   - Some users may want quiet terminals.
   - Some users may want sophisticated terminals.
   - May be <u>required</u> to use terminals already on ships.
   - Different print speeds.
   - Some users may require secure terminals.

b. List several common properties, i.e., universal properties, that we can assume will be true of all terminals that will be used.

- Accept characters one at a time and display them.
- Send characters one at a time.
- Will have letters A-Z and numbers 0-9 and no others can be assumed; cannot assume a character encoding.

3. Why might we have to use different programming languages for different versions?

- The CPU's specified may not have any one language in common; writing a compiler is considered too expensive.

4. Why might the information included in the logs be different in different versions?

- Different user requirements.
- Different number of entries in the log.
- Different methods of operation for the organization accessing the logs.

5. Why might the operator interface on some versions of MP differ from the operator interface on others?

- Different operator terminals (i.e., CRT or hard copy).
- Different standard manual operator procedures that are useful to preserve.
- Different levels of operator skills (sophisticated, naive).
- Different captains on board may require tailored interface.

PF.4 A Minimal Member of the MP Family

EXERCISE

Name: _____

Document MP.1 described a "family" of full service message processing systems designed to be useful in demanding situations. The task of producing family members is inherently difficult and the programs are inherently large.

Examine the list of design decisions below and describe the smallest member of the family, Small MP (SMP), that could operate with reduced computer facilities and still provide some useful services. Describe the situations in which SMP could be used. Assume the AUTONOYS message conventions will be used.

The minimal, useful member of a family is not necessarily more powerful than previous manual procedures. Think of it in terms of:

a. a trivial software system that can gradually be extended to a more powerful version by adding programs, and

b. a backup capability in case a large part of the computer goes down, making it impossible to run the full version.

By looking for such subsets, you can avoid an all-or-nothing approach during both development and operation.

1. The MP software can produce messages in the following formats: (The MP alternative is marked by "**".)

   a. simple formats incompatible with existing ones,
   **b. AUTONOYS,
   c. AUTONOYS plus error-correcting codes,
   d. any format, because there is general message-definition facility.

2. Because some AUTONOYS channels are noisy, an MP does the following:

   a. nothing, it accepts anything it sees (hears?),
   **b. checks for errors and notifies operator,
   c. checks for errors and makes "likely" corrections,
   d. uses formal error-correcting codes for all single, double, ..., errors,
   e. checks for errors and initiates auto retransmit when they are found.

3. An MP "screens" incoming messages as follows:

   a. it does not screen incoming messages but merely accepts all,
   b. accepts only messages that match a built-in set of addresses,
   **c. accepts only messages that match a watch list (the watch list can be updated),
   d. accepts messages based on interactions with a centralized system that locates persons.

4. An MP routes messages as follows:

   a. the operator supplies routing information for each message,
   b. the operator-supplied routing information is checked against internal constants,
   **c. the software automatically supplies routing indicators using an indicator list that the operator can update,
   d. the software automatically supplies routing indicators using an indicator list that is updated by AUTONOYS.

5. Possible processor configurations for supporting an MP are:

   a. no processors are used; there are only teletypes and hard-wired recognizers,
   **b. a single UGH-20,
   **c. a single UGH-VAN,
   **d. a single UGH-2PIE,
   e. combinations of b, c, and d.

6. An MP handles message traffic as follows:

   a. serially, one message at a time,
   b. in parallel, with many messages stored in core,
   **c. in parallel, using both core and mass storage to prevent loss of any message.

7. An MP retains messages as follows:

   a. not at all, no messages are retained,
   b. core copies are retained until space is needed,
   **c. all messages are retained for a fixed period of time,
   d. all messages, old ones are archived.

8. An operator can get information about messages as follows:

   a. by requesting a dump of core and mass memory,
   b. by supplying a special code or message ID,
   c. by supplying content information in a rigid format,
   **d. by using a flexible query language.

9. Software support for the operator interface consists of the following:

   a. minimal, the operator simply reads information printed at the console and types in complete messages,
   **b. there is a prompting package for message input by the operator, and there is the RMD option,
   c. there is a general text processing package usable for all aspects of system operation.

10. There is the following capability for on-line testing of an MP:

   a. none, any testing of an MP must be done off-line,
   **b. there is test generation and transmission controllable by the operator,
   c. there is auto test and evaluation,
   d. there is auto test and fault correction.

PF.5 A Minimal Member of the MP Family

EXERCISE SOLUTION

SMP, the new, small member of the MP family, prints any messages it receives on a terminal, buffering messages if necessary.

SMP does not

1. provide redundancy checks or error correction,
2. screen messages for relevant ones,
3. check or add any routing information,
4. assist in preparation of messages,
5. retain messages after printing them,
6. provide any information retrieval or automatic logging, or
7. perform any self checking.

SMP is useful for producing hard copies of messages that were received at a relay point connected by high quality transmission facilities to the sender. The hard copy can be manually scanned, distributed, and logged.

SMP is useful in developing and testing MP. It provides minimal service in the event of a casualty to part of the computer.

PF.6 Family Development by Stepwise Refinement

LECTURE

I. Review of the decision tree representation of the family development process

II. Dijkstra's Prime Number Program Development*

A. Decisions: one thousand primes, compute before print

```
begin  variable table p;
       fill table p with first thousand prime numbers;
       print table p;
end;
```

B. Debate about the order of decisions — should one design "table" or an algorithm "fill with first thousand prime numbers"

III. Wulf's KWIC Index Program*

A. Stage 1 PRINTKWIC

Design decisions:

---

* The original notation of this algorithm has been changed slightly in order to be consistent with the abstract programming language presented in GEN.5.

B. Stage 2

```
PRINTKWIC:
     begin
        generate and save all interesting circular shifts;
        alphabetize the saved lines;
        print alphabetized lines;
     end;
```

Design decisions:

C. Stage 3

```
PRINTKWIC:
  begin
     comment generate and save all interesting circular shifts;
     for each line in the input do
       begin
       generate and save all interesting shifts of this line;
       end;
     end-for;
     alphabetize the saved lines;
     print alphabetized lines;
  end;
```

Design decisions:

IV. Development of the memory allocator family of programs by stepwise refinement

A. Stage 1

```
begin
type1 BEST_YET, CANDIDATE;
type2 ACTUAL;
boolean NOT_ALL_SPACES_CONSIDERED, BETTER_SPACE_MIGHT_EXIST;
BEST_YET:= null;
while NOT_ALL_SPACES_CONSIDERED and BETTER_SPACE_MIGHT_EXIST do
   begin
      CANDIDATE:= FIND_NEXT_ITEM_FROM_FREE_SPACE_LIST;
      BEST_YET:= BEST_OF(BEST_YET,CANDIDATE);
   end;
end-while;
if (BEST_YET = null) then ERROR_ACTION end-if;
ACTUAL:= FRAGMENT(BEST_YET);
ADJUST(BEST_YET,ACTUAL);
ALLOCATE(ACTUAL);
end;
```

Note: type1 and type2 are ways of declaring variables without actually specifying the type. See section on design decisions not made in Stage 1.

1. Assumptions made to verify that the above is correct

   a. The memory is initially divided into frames of different sizes. A request is always for an amount of memory no greater than one frame. When memory areas are returned to the list of free spaces, adjacent sections of the same frame will be coalesced. The amount of space requested will be known to the program.

   b. BEST_YET is a variable that indicates an item from the list of free spaces. Null, a possible value of BEST_YET, indicates no item.

c. BETTER_SPACE_MIGHT_EXIST is a boolean variable that is true as long as it is possible that a "better" space can still be found. The criteria for "better" have not yet been specified.

d. NOT_ALL_SPACES_CONSIDERED is a boolean variable that is true until the loop has checked each free space once.

e. CANDIDATE is a variable of the same type as BEST_YET.

f. FIND_NEXT_ITEM_FROM_FREE_SPACE_LIST is a function that returns a value that indicates one of the items on the free space list. If there are n items on the list, a sequence of n function calls of the procedure will deliver each of the n items once.

g. Adding new items during program execution will not occur.

h. BEST_OF is a procedure that takes two variables of the same type as BEST_YET (i.e., type1) and returns the better of the two according to some unspecified criterion. If neither is suitable, null is returned.

i. ERROR_ACTION is a procedure that performs the action that should be performed if no suitable space can be found. ERROR_ACTION does not return control to this program except at the beginning.

j. type2 is a class of variables that can describe a storage area.

k. FRAGMENT is a procedure that returns a variable of type2 after the procedure determines which part of the free space should be allocated. The free space is identified by the parameter.

l. ADJUST is a procedure that adjusts the list of free spaces to reflect the allocation. The parameter BEST_YET indicates which item is to be removed from the list. ACTUAL describes the amount of space to be allocated in case the unused fragment of the original space is to be left on the list.

m. ALLOCATE is a procedure that gives the storage area to the requesting program.

2. Stage 1 design decisions

a. No items will be added to or removed from the free space during execution of the program until the final selection has been made.

b. Once execution of the program begins, no other execution of it will begin until the executing program is completed (critical section).

c. The only other program that might change the data structures involved is one that adds items to the free space list when space is returned.

d. The program that adds free spaces to the list compacts two or more contiguous free spaces that are part of a frame into one space represented by a single item on the list.

e. A candidate is not removed from the list while it is being considered.

f. Before the search begins, there is no check to determine if allocation is possible (e.g., check for empty free space list, check for size of largest available space).

3. Design decisions not made in Stage 1

   a. The representation of the free space list.

   b. The type of the variables BEST_YET, CANDIDATE and ACTUAL.

   c. The order in which the free spaces are stored on the list.

   d. The order in which the items on the free space list are searched.

   e. The criteria used in BEST_OF.

   f. The decision to allocate all of the space found or allocate only that part needed, (i.e., the action taken in FRAGMENT).

   g. The ERROR_ACTION that will be taken.

B. Stage 2

```
begin
integer BEST_YET, CANDIDATE, N;
boolean BETTER_SPACE_MIGHT_EXIST;
type2 ACTUAL;
BEST_YET:= 0;
CANDIDATE:= 0;
while (CANDIDATE lt N) and BETTER_SPACE_MIGHT_EXIST do
   begin
      CANDIDATE:= CANDIDATE + 1;
      BEST_YET:= BEST_OF(BEST_YET,CANDIDATE);
   end;
end-while;
if BEST_YET = 0 then ERROR_ACTION end-if;
ACTUAL:= FRAGMENT(BEST_YET);
ADJUST(BEST_YET,ACTUAL);
ALLOCATE(ACTUAL);
end;
```

1. Stage 2 design decisions

   a. The list of free spaces is represented by a table with N entries. Each entry represents a valid free space. The first space searched is identified by entry 1 and the last space searched is identified by entry N.

   b. The variables BEST_YET and CANDIDATE are integers (that will be used as array indices) so that the test for NOT_ALL SPACES_CONSIDERED can be an integer test on the value of CANDIDATE.

c. BETTER_SPACE_MIGHT_EXIST is a boolean variable that is true as long as it is possible that a "better" space can still be found. If it is set to false, that will be done by BEST_OF.

2. Design decisions not made in Stage 2

a. The characteristics that describe an item in the free space list (e.g., starting address and length, or starting address and end address).

b. The order in which the entries are stored on the list.

c. The relation of the variables CANDIDATE and BEST_YET to the items in the free space list.

d. The policy or selection criteria in BEST_OF.

e. The decision to allocate all of the space found, or allocate only that part needed and leave the rest on the free space list.

C. Stage 3

```
begin   integer BEST_YET, CANDIDATE,N, T, REQUIRED_LENGTH, OLD_T, I;
        integer array LAST[1:N], START[1:N];
BEST_YET:= 0;
CANDIDATE:= 0;
OLD_T:= infinite; comment infinite stands for the largest integer
                          that can be represented;
while (CANDIDATE lt N) do
  begin
    CANDIDATE:= CANDIDATE + 1;
    T:= LAST[CANDIDATE] - START[CANDIDATE] + 1;
    if (T ge REQUIRED_LENGTH) and (T lt OLD_T) then
      begin
        BEST_YET:= CANDIDATE;
        OLD_T:= T;
      end;
    end-if;
  end;
end-while;
if BEST_YET = 0 then ERROR_ACTION end-if;
ACTUAL:= (START[BEST_YET],OLD_T); comment the single variable ACTUAL
                                        is represented as two
                                        integers that are always
                                        used together;
N:= N - 1;
comment close up the gap caused by removing that element;
for I:= BEST_YET step 1 until N do
    begin
      LAST[I]:= LAST[I+1];
      START[I]:= START[I+1];
    end;
end-for;
ALLOCATE(ACTUAL);
end;
```

ADDITIONAL ASSUMPTION

The length of the requested space, REQUIRED_LENGTH, is input to the program. It must be known by BEST_OF.

1. Stage 3 design decisions

   a. Each item in the free-space table has the starting address (START[item]) and the ending address (LAST[item]) of the free space identified by the item in the arrays START and LAST.

   b. The entire free space that is selected will be allocated, not just the part of the space that is needed.

   c. The integer values of CANDIDATE and BEST_YET are indices into the table containing free space information.

   d. A policy of "best fit" is used to select the smallest free space with a length greater than or equal to REQUIRED_LENGTH. The boolean variable BETTER_SPACE_MIGHT_EXIST is true until "all spaces considered" is false, so that it need no longer be included as one of the loop termination conditions. If the policy "first fit" were used, BETTER_SPACE_MIGHT_EXIST would become false as soon as the first suitable space were found.

2. Design decisions not made in Stage 3

a. The order in which the entries are stored in the list.

b. The ERROR_ACTION that will be taken.

c. Implementation of ALLOCATE.

V. Another member of the memory allocator family

A. *Design decisions and assumptions*

1. All the assumptions made for Stage 1.

2. There is a list of free spaces represented by two arrays. Each can be accessed by an index into the array identifying the free space. Each free space is represented by its starting address (START[item]) and its length (LENGTH[item]).

3. Both free space arrays have at least N entries and all entries between 1 and N represent valid free spaces. The first space searched is identified by the first entry, and the last space searched is identified by the Nth entry.

4. During execution of the memory allocator program, no items will be added to or removed from the free space list by other programs.

5. The only other program that might change the free space list is one that adds items to the list. This program compacts two or more contiguous free spaces into one space represented by a single entry on the list so that the list will never contain two contiguous areas.

6. Once execution of the program begins, no other execution of it will begin until the executing program is completed.

7. Before the search begins, there is no check to determine if allocation is possible (e.g., check for empty free space list). After the search is performed, if no suitable free space was found, a subroutine ERRORACTION is to be called.

8. The length of the requested space, an integer called REQUIRED_LENGTH, is input to the program.

9. A policy of "best fit" is used to select the smallest free space whose length is greater than or equal to REQUIRED_LENGTH.

10. While a candidate is being considered, it is not removed from the free space list.

11. The space that is allocated is equal in length to REQUIRED_LENGTH and is taken from the beginning of the free space selected by the "best fit" algorithm.

12. The free space list is adjusted to reflect the allocation of the necessary part of the selected free space.

13. A procedure named ALLOCATE is supplied with information about the space to be allocated and gives the space to the requesting program.

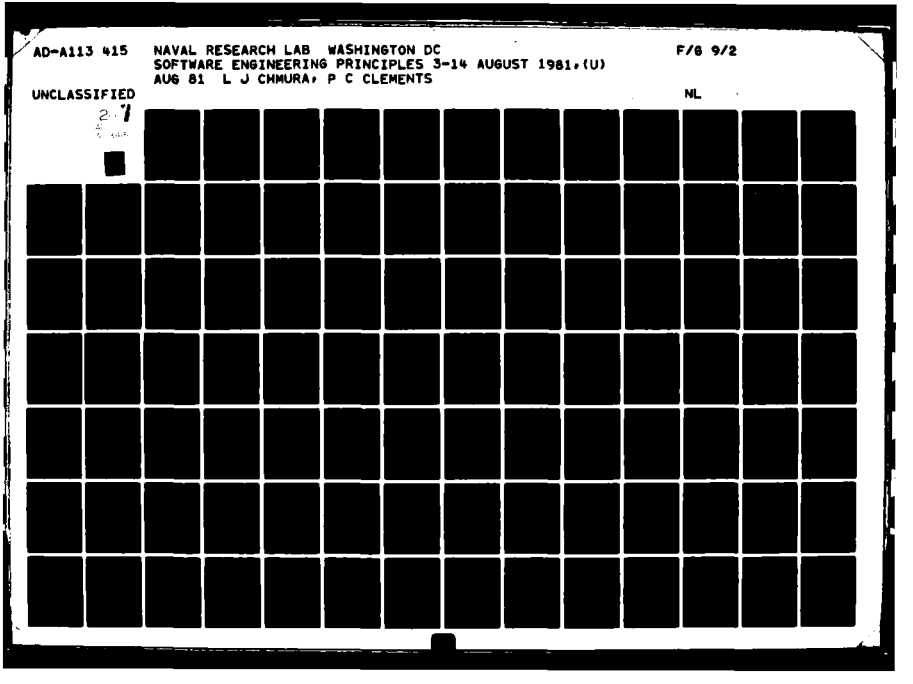
14. The variables BEST_YET and CANDIDATE are integers, so that the test for NOT_ALL_SPACES_CONSIDERED can be an integer test on the value of CANDIDATE. The value null indicates no item on the list.

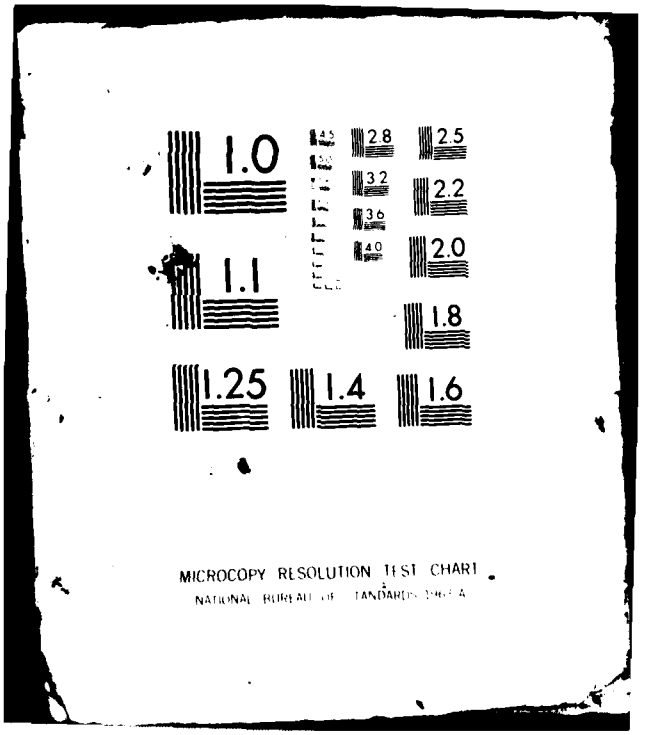
B. Comparison of the two family members

1. Same assumptions and design decisions as Stage 2.

AD-A113 415 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
SOFTWARE ENGINEERING PRINCIPLES 3-14 AUGUST 1981.(U)
AUG 81 L J CHMURA, P C CLEMENTS

UNCLASSIFIED NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

2. Differences in Stage 3 design decisions.

C. Alternative ways to develop the program

1. Start from scratch.

2. Start with Stage 3. Scan line by line and make required changes.

3. Go back to Stage 2. Develop new program from there.

D. Abstract program for another member of the memory allocator family

```
begin  integer BEST_YET, CANDIDATE, N, T, REQUIRED_LENGTH, I;
       integer array LENGTH[1:N], START[1:N];
BEST_YET:= 0;
CANDIDATE:= 0;
T:= infinite;
while (CANDIDATE lt N) and LENGTH [CANDIDATE] ne REQUIRED_LENGTH do
    begin
        CANDIDATE:= CANDIDATE + 1;
        if (LENGTH[CANDIDATE] ge REQUIRED_LENGTH) and
           (LENGTH[CANDIDATE] lt T)
        then
            begin
                BEST_YET:= CANDIDATE;
                T:= LENGTH[BEST_YET];
            end;
        end-if;
    end;
end-while;
if BEST_YET = 0 then ERROR_ACTION end-if;
ACTUAL:= (START[BEST_YET],REQUIRED_LENGTH);
if (REQUIRED_LENGTH lt T) then
    begin
        START[BEST_YET]:= START[BEST_YET] + REQUIRED_LENGTH;
        LENGTH[BEST_YET]:= T - REQUIRED_LENGTH;
    end;
else
    begin
        N:= N-1;
        for I:= BEST_YET step 1 until N do
            begin
                START[I]:= START[I+1];
                LENGTH[I]:= LENGTH[I+1];
            end;
        end-for;
    end;
end-if;
ALLOCATE(ACTUAL);
end;
```

PF.7 Applying the Program Family Principle

LECTURE

I. Steps of the family methodology

A. Identify the characteristics shared by family members

1. Example of feature common to all: Capability of ..ers to display a received message.

2. We may need to identify a larger set of characteristics than that required by any single member.

B. Identify and encapsulate the differences among family members

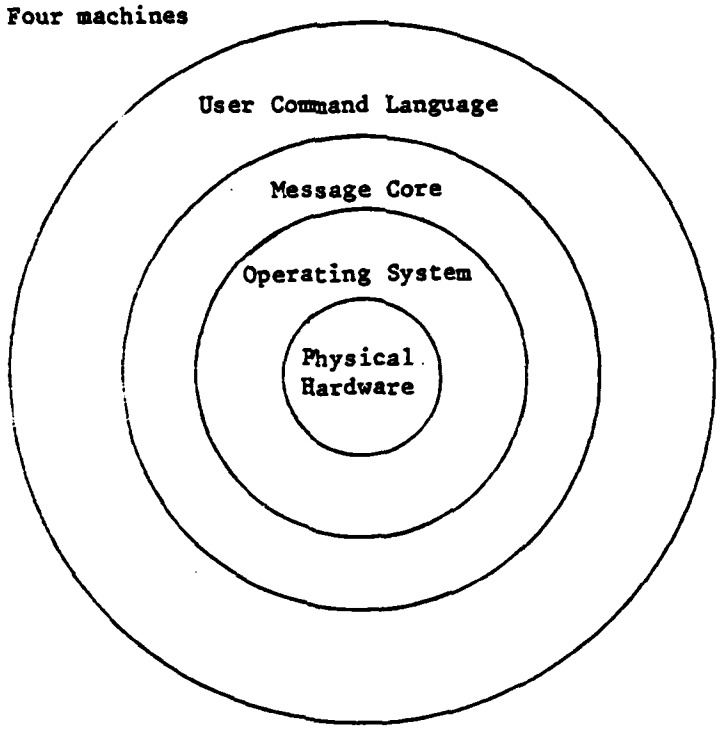
1. A software structure suitable for all family members is formulated.

2. In that structure, the software is partitioned into modul encapsulate the various distinguishing characteristics.

II. Application of the family methodology to military message systems

A. To identify commonalities and differences among members, a model, based on a series of nested machines, was developed to represent each member of the family.

1. Four machines



2. The data objects and operations of each extended machine are constructed from the data objects and operations of another machine.

3. Examples of each machine's data objects and operations.

Machine	Objects	Operations
Hardware	registers; memory words	load, store; move
Operating System	processes; segments	create_process, destroy_process; move_segment, copy_segment
Message Core	messages; message_files	create_message,setfield_message, getfield_message; create_msgfile,addmsg_msgfile, rmvmsg_msgfile, destroy_msgfile
User Command Lang.	messages; message_files	COMPOSE MESSAGE, DELETE MESSAGE, PRINT MESSAGE, SEND MESSAGE; CREATE FILE SECRET, PRINT FILE, DESTROY FILE

4. Difference between message core machine and user command language machine.

   a. Message core operations are typically less powerful than user command language statements.

   b. To the extent feasible, all message core data types are specified independently of one another; i.e., an operation on a data object of a given type affects only that object and no others.

5. Examples of user command language statements and the different data objects each affects.

a. User Command: CREATE MESSAGE AUTODIN SECRET CHINA
Message System: SIGMA
Operations: Create an AUTODIN message at Secret and insert it in a message_file named CHINA
Data Objects Affected: message, message_file

b. User Command: CREATE FILE USSR CONFIDENTIAL
Message System: SIGMA
Operations: Create a message_file named USSR at Confidential and insert an entry for it in the user's directory of message_files
Data Objects Affected: message_file, message_file_directory

c. User Command: PRINT SEQUENCE.S TEMPLATE.T
Message System: HERMES
Operations: Print every message in SEQUENCE.S using the display format defined by TEMPLATE.T
Data Types Affected: message, message_file, sequence, template

B. Step 1: Identify the characteristics shared by family members

1. Family members differ.

a. User command languages

- different organizational procedures

- different habits and preferences

b. Physical hardware

- special requirements aboard ships

- different processing speeds and/or memory requirements

c. Operating systems

- different hardware has different operating system

- same hardware supports more than one operating system

2. The significant shared characteristics of military message systems are functional capabilities.

a. Create, coordinate, send, distribute, display, and delete messages

b. Create, destroy, add messages to, and delete messages from message files

3. Shopping list of message core data types and operations.

		SIGMA		HERMES	
message		X		X	
	create_message		X		X
	delete_message		X		X
	send_message		X		X
	setfield_message		X		X
	getfield_message		X		X
			.		.
			.		.
			.		.
message_file		X		X	
	create_messagefile		X		X
	destroy_messagefile		X		X
	addentry_messagefile		X		X
	rmventry_messagefile		X		X
			.		.
			.		.
			.		.
filter		X		X	
	create_filter		X		X
	destroy_filter		X		X
	modify_filter		—		X
			.		.
			.		.
			.		.
template		—		X	
	create_template		—		X
	destroy_template		—		X
	update_template		—		X
			.		.
			.		.
			.		.
user_profile		—		—	
	create_profile		—		—
	addterm_profile		—		—
			.		.
			.		.
			.		.

C. Step 2: Identify and encapsulate the differences

1. Every user command language statement can be expressed as a sequence of message core operations.

2. Similarly, every message core operation can be translated into a sequence of operating system calls and/or machine instructions.

3. The message core insulates the operating system/hardware from changes in the user command language.

   a. When the user command language changes, the sequence of message core operations that implements each user command language statement will require change.

   b. The lower level code that supports each message core operation will not require revision.

4. The message core also insulates the user command language from changes in the operating system/hardware characteristics.

   a. Each message core operation must be reimplemented using the new operating system calls and/or new machine instructions.

b. The translation of each user command language statement into a sequence of message core operations will remain unchanged.

III. Lessons learned

A. We developed sets of shared features. Each family member is associated with some subset of each set.

1. Shopping list of message core data types and operations

2. Shopping list of semantics of user command language statements: Examples

a. CREATE TEMPLATE T

b. EDIT FILTER F

c. PRINT MESSAGE id [print-template]

d. CREATE MESSAGE [type] [security-level] [message-file] [compose-template]

B. We limited the range of family members.

1. Examples

a. Number of message core machines

b. Semantics of user command language

2. These are examples of early design decisions. In making these decisions, we had to confirm that we didn't rule out any features that would be needed later on.

C. We studied existing family members in detail. It is helpful if some family members already exist, since it is easier to determine the requirements of existing systems than systems that will be built at some future date.

PF.8 Design Decisions in HAS Requirements

EXERCISE

Name: _____

In the HAS requirements summary (HAS.1), many decisions are already made that implicitly rule out family members that might be useful later. Because these are early decisions, they are likely to permeate the design and be very difficult to change later.

For example, the Navy may eventually require large, moored, repairable buoys to collect weather data in key locations. If HAS software designers consider the decisions "drifting buoy" and "disposable" absolutely fixed, the software may be designed so that it is not reusable in the new buoys, even though the functions are similar.

Study HAS.1, looking for four or five software-related design decisions that have already been made. List each decision, along with several reasonable, but rejected, alternatives. Indicate the alternative that was chosen for HAS. You may want to format your answers as multiple choice questions, as shown in the example below.

EXAMPLE

1. The HAS software transmits the following information: (the alternative chosen in HAS.1 is marked with "**".)

   a. No information,
   b. A summary at fixed intervals,
   c. A summary when requested,
   d. A small set of predefined reports on request,
** e. A summary at fixed intervals and a small set of predefined reports on request,
   f. Answers to specific queries.

PF.9 Design Decisions in HAS Requirements

EXERCISE SOLUTION

Listed below are some of the software-related design decisions implicit in HAS.1, along with some reasonable alternatives.

1. The software will operate on the following hardware:

   **a. BEEN computer
   b. NOVA computer
   c. a microprocessor
   d. PDP-11 computer
   e. any of the above

NOTE: Choice of computer can profoundly affect software construction; for example, consider availability of support software.

2. The software designer should assume the sensor quality will be:

   **a. poor, because HAS will use the cheapest sensors available
   b. variable, depending on accuracy required at a particular location
   c. consistently good

NOTE: Assumptions about sensor quality affect decisions about frequency of reasonableness checks and the complexity of filtering algorithms.

3. The data retained in the system will be:

   a. none — no data retained
   b. most recent data only
   **c. most recent data with a limited history
   d. extensive history data

4. The various deployed buoys may or may not be connected as follows:

   a. master/slave mode where several buoys in predefined area collect data but only one transmits
   **b. no connection
   c. sophisticated network with cooperation and comparison

5. The communications system will consist of:

   a. wires with low data rate
   b. transmitter only, for regular broadcasts
   c. transmitter and receiver, for operating in broadcast burst mode on request
   **d. transmitter and receiver, for broadcasting regularly and responding to requests
   e. multiple transmitters for broadcasting reports simultaneously on different channels

6. Geographic location will be determined:

   a. never — geographic location information will not be available
   b. by initialization on deployment — for moored buoys
   **c. by Omega fix
   d. by NAVSAT fix

7. The message format is:

   **a. RAINFORM
   b. determined by the HAS system designer
   c. flexible — must change to meet varying user requirements

UE.1 Desired Responses to Undesired Events

LECTURE

I. Introduction

A. Murphy's Law: If anything can go wrong, it will

B. In the technical description of a system or subsystem, there is the useful dichotomy of wanted and unwanted inputs and outputs (Hall 1962)

C. Built-In Test (BIT) (Coppola 1979)

D. Definitions

1. Undesired Situation: a condition that is unfortunately true. Examples of undesired situations are: (1) data was not received when expected, (2) data was received when not expected, (3) data does not meet specifications, and (4) a device or function has malfunctioned.

2. Undesired Event (UE): the moment in time that an undesired situation arises. A UE (indeed any event) may or may not be detected (i.e., noticed) by a system, or it may be detected sooner or later. Sometimes the functioning of a system is interrupted by a UE. Sometimes a system detects a UE by examining a condition.

E. Our contention: The behavior of a system in the face of UEs is a specification, not implementation, issue.

II. The existence of alternatives when something goes wrong

A. Example: A garbled address is found on an input tape

1. Alternatives

a. Skip it

b. Print it with known errors — no change

c. Print it with erroneous parts missing

d. Print it with erroneous parts replaced by "?"

e. Print it with erroneous parts marked

f. Use minimal correction methods to correct errors

g. Search for "closest" address in files

2. For each alternative, there is an appropriate situation

3. In practice, decision often not made in specifications — although it is visible behavior

B. Example: A memory bank in a multiprogramming system fails — no data lost — insufficient memory available

1. Alternatives

a. Kill the job(s) that were currently using that memory bank allowing the others to continue normally

b. Use swapping to allow all of the jobs to continue more slowly

c. Pick the newest (most recently started) job and kill it, continuing with this procedure until those remaining function normally

2. All alternatives technically feasible if expected and planned for

3. Should and can be part of system specifications

C. Example: Message system operator types date of 1781 on message

1. Alternatives

a. System rejects the message because year is "out of range"

b. System files the message as the most recent message in its logs

c. System queries the operator, accepts message if he insists

d. System files the message as the oldest message in its logs

2. All alternatives easily achieved technically

3. Each useful in some situation

III. The existence of UE alternatives when designing the system

A. What, me worry?

B. Maintain redundant information necessary to detect UEs

C. Maintain redundant information needed to recompute state

1. Spend the time in recovery actions if something occurs

2. Maintain information necessary to restore state "immediately"

IV. Cost factors associated with UEs

A. Cost of preparation for recovery whether or not a UE occurs

B. Cost of no recovery and no recognition if a UE occurs

C. Cost of recognition but no recovery if a UE occurs

D. Cost of actual recovery if a UE occurs

E. Frequency of UEs must be a deciding factor

V. Incidents and crashes (*Kaiser and Krakowiak 1974*)

A. Incident: expected UE; system action depends on occurrence; presumably system recovers without significant long-range cost

B. Crash: unexpected UE; system fails to perform intended effect; presumably a high penalty is paid

C. Computer system example

1. Incident: most recent version of file lost; reconstructed from old copy and activity log

2. Crash: file lost

D. Incidents and crashes, reexamined

1. Incidents with higher costs, crashes of little significance

2. Incident: UE resulting in something unfortunate
   Crash: UE resulting in something catastrophic

3. Successful UE handling =

4. Two of Gilb's Laws (Bloch 1977)

   a. Undetectable errors are infinite in variety, in contrast to detectable errors, which by definition are limited.

   b. Investment in *reliability will increase until it exceeds* the probable cost of errors, or until someone insists on getting some useful work done.

VII. *Classes of UEs — guide to error anticipation*

A. Resource failure

1. Detectable by examining input only

2. Detectable by comparison with internal data

3. Detected externally

4. Reported to user by means of incorrect output

5. Two considerations

   a. With information loss vs. without information loss

   b. Temporary vs. long term

B. Incorrect input data

   1. Detectable by examining input only

   2. Detectable by comparison with internal data

   3. Detected externally (after input)

   4. Reported to user by means of incorrect output

C. Incorrect internal data

   1. Detected by internal inconsistency

   2. Detected by comparison with input data

3. Reported to system in terms of incorrect output data

4. Uncertain data (e.g., after reporting of hardware parity error)

VIII. Strategies (conclusions)

A. Above list provides an approach to listing of classes of UEs, appropriate responses should become part of specification

B. External interfaces suggested by above list must be present

C. Tests on internal and external data, as well as resources, that are to be performed by the software must be specified. Both exits must be shown.

IX. References

Bloch, A. 1977. Murphy's Law and Other Reasons Why Things Go Suoɹɯ! Los Angeles: Price/Stein/Sloan.

Coppola, A. 1979. A Design Guide for Built-In-Test (BIT). Report RADC-TR-78-224. Available as DTIC Document ADA069384.

Hall, A. D. 1962. A Methodology for Systems Engineering. Princeton: D. Van Nostrand.

Heninger, K. L.; Kallander, J.; Parnas, D. L.; and Shore, J. E. 1978. Software Requirements for the A-7E Aircraft. Naval Research Laboratory Memorandum Report no. 3876.

Kaiser, C.; and Krakowiak, S. 1974. "An Analysis of Some Run-Time Errors in an Operating System." IRIA Rapport de Recherche, no. 49.

Parnas, D. L. 1975. "The Influence of Software Structure on Reliability." Proceed. of the 1975 International Conf. on Reliable Software, pp. 358-362.

Parnas, D. L.; and Wurges, H. 1976. "Response to Undesired Events in Software Systems." Proceed. of Second International Conf. on Software Engineering, pp. 437-446.

Randell, B.; Lee, P. A.; and Treleaven, P. C. 1978. "Reliability Issues in Computer System Design." Computing Surveys, vol. 10, no. 2, pp. 123-165.

UE.2 MP and UEs

EXERCISE

Name: _____

Introduction

The description of the MP system (MP.1) generally fails to mention the behavior desired of the system in the event that something goes wrong. One exception is the frequently occurring undesired event, noisy message data. There are, however, many other UEs that should be mentioned in a system specification. Examples:

1. What services should be provided if the disk fails? Full service is obviously impossible, but partial service can be expected. What services have priority?

2. What services should be provided to assist in the event that messages that were thought to have been transmitted were not transmitted because of a failure beyond the scope of MP. MP's data structures now contain incorrect data.

3. How should the system react if an obviously incorrect date is inserted in a message being composed (e.g., 1781)?

Assignment

Think about the functions provided by MP and try to supplement the above list. The UE classification scheme in the lecture outline should provide some help in organizing your efforts.

UE 3 MP and UEs

EXERCISE SOLUTION

The following list of possible UEs is not complete. It merely illustrates the kinds of things that must be considered when defining the desired behavior of the MP system.

1. Resource failures

   a. What services can be provided if the operator's console fails?

   b. What should the system do if the UGHTRANS equipment fails to respond? Should the system detect this or wait for external notification?

   c. What services should be provided if a bank of memory goes down?

2. Incorrect input data

   a. What should the system do if the operator reports an incorrect destination during the time that the message is being transmitted?

   b. What should the system do if it does not receive a response within the required response time, for a message that requires a response (e.g., an emergency command precedence message)?

   c. What should the system do if the operator tells it that the date on all of the last 20 messages (Feb. 30) is incorrect?

3. Incorrect internal data

   a. What should the system do if it discovers that there are parity errors in the Watch List?

   b. What should the system do if it discovers that its internal directory used to find log data contains an impossible disk address?

   c. What should the system do if two messages being composed have grown so large that deadlock prevents either from being finished before the other finishes?

UE.4 Intermodule Interfaces and UEs

LECTURE

I. Introduction

A. UE handling can result in an order of magnitude decrease in the frequency of crashes

B. UE handling tends to introduce interprogram dependence

C. Preparation for UEs can avoid this

II. Probability considerations and UE handling

A. Need for redundant information — information that would not be needed if no UE occurs

B. Detection vs. correction

C. Number of simultaneous errors

1. Example: Warning lights in automobiles

2. Example: Archive files on <u>different</u> devices

D. No error-free system — probability of error can be arbitrarily reduced

E. Extra complication — situations in which multiple errors are highly probable

III. The effect of structure on UE handling

A. Proper response to an error requires efforts from various modules

1. Example: Unreadable block on a tape file.
Detected by tape handler.
Attempt to correct by tape handler.
File system knows which file, location of other data.
Data system knows how to reconstruct data.

2. Example: Part of memory becomes error prone (no parity).
   User program detects inconsistent data.
   Memory allocator must not assign this area.
   Deadlock prevention must know of reduced resources.
   Background memory system must attempt to restore data.

B. Conventional response: Write a program that uses all relevant tables for common cases. Introduce "connections between modules."

C. Conclusion — interface must include UE communication possibilities

Examples:

1. Tape handler reports nature of error to file system in terms of block number and tape — not: file system reads bits

2. File system reports to data system in terms of file/line

3. Data system knows file/line — storage of redundant data

4. Memory user has "complaint box"

5. Banker can be informed of catastrophe, bad loans

6. Banker needs interface to "job killers"

7. Banker needs alternate entry to recompute deadlock danger

IV. Abstractions that interface with error recovery

A. Useful information can be presented abstractly — not likely to change

B. Hiding too much can prevent recovery

1. Example: "parallel changes" to a file

2. Example: indistinguishable error classes in hardware

V. Software traps as an error reporting mechanism

A. Reduce code complexity by separating normal behavior, detection, and response

B. Decrease likelihood of undetected errors

C. Ease the removal of detection code when not needed

D. Four possibilities for actions after UE detection

1. continue

2. retry

3. clean up

4. give up — only if no higher level remains

VI. Classes of UEs to report

A. Incorrect call

B. Incorrect results

C. Report earlier incorrect call

D. Resource failure

E. Unlikely actions

VII. "Impossible" vs. possible states after a UE

VIII. Example of abstract reporting of defects — tree specification

IX. Summary — to make the modular concept work, all communication must be over the predefined interface — UEs included

X. Reference

Parnas, D. L.; and Wuerges, H. 1976. "Response to Undesired Events in Software Systems." Proceed. of Second International Conf. on Software Engineering, pp. 437-446.

UE.5 MP Intermodule Interfaces and UEs

EXERCISE

Name: _____

In the improved MP system (document MP.4), recovery from mishaps will often require the cooperation of several modules. A program of one module may discover and handle the problem initially, but programs of other modules must be informed to take appropriate action. Example: If a message is so badly garbled that a program of the external interface module (EI) assigns a low probability of its being correct, this should be noted in the logs by some program of IR/LOG. The interface between programs of the modules must provide for communication that will allow this. Examine the description of the improved MP. For each UE listed below, explain which modules are affected and describe the information that must be communicated between their programs.

1. Persistent errors in output buffer.

2. Persistent high rate of errors in part of memory.

3. Operator's console is no longer functioning; repair will take some time.

4. For several messages under composition, an operator has indicated desire to use an UGHTRANS channel that does not exist.

5. All available disk space is allocated; freeing of space is not immediately likely.

UE.6 MP Intermodule Interfaces and UEs

EXERCISE SOLUTION

1. Parity errors in output buffer.

The CM module requests the EI to rewrite the same information into other core locations.

2. High rate of errors in a part of memory.

The MH module requests the AL not to allocate the error-ridden area unless no alternative exists. It requests DS to report the problem to the operator.

3. Operator's console is no longer functioning.

TC detects that an operator's console is no longer functioning. The assignments for that console must be reassigned. (Resources allocated by AL to users of that console can be temporarily reassigned.) Partially processed messages must be either finished or removed from the system.

4. An operator tries to use an UGHTRANS channel that does not exist.

EC must inform MH to correct messages containing that channel number. EI informs the operator who has been using that channel.

5. All available disk storage space is allocated.

PS must inform the AL and the system operator. Space should likely be freed by putting old logs or messages in archival storage off-line. Message archiving must be done by use of MH and log archiving by IR.

UE.7 The Uses Hierarchy and UEs

LECTURE

I. Introduction

A. UEs correspond to the violation of a specification

B. Uses hierarchy as the key to understanding UEs

II. The problem of designing for UEs

A. Characteristics of UEs

1. Relatively infrequent

2. Often caused by higher levels in uses hierarchy, detected by lower levels

3. Recovery best performed at the higher levels

B. UE detection and handling should not be afterthoughts in the design

C. How can we incorporate UE communication in such a way that appropriate (application dependent) recovery techniques can be added?

D. Problem — UE handling requires efforts of several modules and levels

E. Result — potential program interdependence that violates design principles

III. Criteria of successful design for UE's

A. UE communication doesn't violate information hiding

B. UE handling doesn't interfere with subsettability

C. Design permits change in UE handling without change in system structure

D. New levels in the uses hierarchy can be added (with UE handling) without changing the lower levels

IV. Review of the uses hierarchy

A. Given program A with specification S_a and program B, we say that A uses B if A cannot satisfy S_a unless B is present and satisfies some non-trivial specification S_b. The assumed specification S_b may differ for different users of B.

B. If A doesn't care about any S_b, a call on B is not a use of B

V. The problem of UE communications in a uses hierarchy

A. "Uses" provides a means for communicating downwards in the hierarchy

B. Since recovery possibilities are usually at higher levels, lower levels need to communicate UE detection upwards

C. Lower levels can make no assumptions about upper levels (i.e., they can call programs at higher levels, but they can't use programs at higher levels)

VI. Using "traps" to communicate upwards in the uses hierarchy

A. The analogy with hardware

B. Consider each uses hierarchy level as a virtual machine with traps for the UEs detectable by the virtual machine

C. Hardware trap = branch to fixed trap location;
virtual machine trap = call (not use) of routine with reserved name

D. Actual trap routines (UE handlers) provided by users of virtual machine; can change dynamically

E. Virtual machine traps provide upwards UE communications without violating uses hierarchy, make no assumptions about who will receive information and what will be done with it

VII. Problems in designing the virtual machine traps

A. Can be advantageous to report on a class of UEs with a single trap

B. Further information about the UE can be passed by parameters

C. Information reported upwards must be in terms of appropriate abstraction — must respect information hiding

1. No references to internal data structures or programs

2. No references to partially computed results

3. Information only in terms of specification of the virtual machine

4. Different versions of the same module must provide same UE reports

D. Trade-offs

1. Amount of information in UE handler name and parameters vs. ease of analysis at user level

2. Number and detail of virtual machine traps vs. diagnostic programs at the user level

E. UE information may also be reported "sideways" — reports to the programmer

VIII. Classes of UEs to consider

A. Parameter values

B. Capacity limits

C. Undefined information requests

D. Operations in certain order (e.g., open before read)

E. Detecting actions likely to be unintentional

IX. Sufficiency

X. Trap priority — call only one at a time

XI. Detecting errors — redundancy vs. efficiency — early development

XII. Summary in terms of criteria for success (see section I)

A. Respect information hiding by communicating UEs in terms of suitable abstractions

B. Maintain subsettability by using traps that don't violate uses hierarchy

C. Software traps allow changes in UE handling without changes in system structure

D. UE detection based on specifications permits addition of higher levels without change

MOD.1 Decomposition into Modules

LECTURE

I. Limitations of stepwise refinement

A. Level of detail (assumptions must be stated) — intuitive understanding assumed

B. Sequencing decisions are implied

C. Postponement principle: Postponement of sequencing decisions

D. Size of programs expands — more people, work involved

II. What else can we decide besides the order of events?

A. The design of data structures

B. Interfaces

C. Work assignments — modules

D. Parameters that characterize the program family

III. History of modular decomposition

A. Unit of measure = 3.27 square meters

B. Parts to be put together

IV. Modules of hardware: How you put them together is obvious; there are well-known physical constraints. Hardware is a physical object.

V. Modules of software — when are parts put together?

A. Write time

B. Assembly time

C. Memory load time

VI. The three (or more) meanings must not be confused

A. Constraints different

1. Write time — intellectual coherence for programmer

2. Assembly time — name conflicts

3. Memory load time — fitting into core things needed at same time

B. Myth of overmodularization: Modules should be as small as possible

C. Inefficiency results from forcing coincidence

VII. In this course, modules are always design-time or change-time entities

A. Units of change

B. Redesign = throw away

C. So small that changing does not help

VIII. The KWIC INDEX example

A. Conventional structure

1. Input Module

2. Circular Shift Module

3. Alphabetizing Module

4. Output Module

5. Master Control Module

B. Decisions likely to change

1. Input format

2. How stored in memory

3. Output table sorted completely before output

C. Alternative structure

1. Line Holder Module — special purpose memory to hold lines of KWIC index

```
GET_CHAR(lineno, wordno, charno)
SET_CHAR(lineno, wordno, charno, char)
CHARS(lineno, wordno)
LINES
WORDS(lineno)
DELETE_LINE(lineno)
DELETE_WORD(lineno, wordno)
```

2. Input Module — reads from cards; calls line holder programs to store in memory

   INPUT

3. Circular Shift Module — uses line holder programs to get data from memory; may make table, may not

   CS_SETUP
   CS_CHAR(lineno, wordno, charno)

4. Alphabetizer Module

   ALPH
   ITH(lineno)

5. Output Module — calls ITH and circular shift programs

   OUTPUT

6. Master Control Module — calls INPUT, CS_SETUP, ALPH, and OUTPUT

D. Claim

1. Not getting any really different program

2. Different way of cutting up — so that change is confined in one person's work (recall module definition)

3. System organized into set of modules so that it is clearly seen what needs to be changed

4. Not necessarily better *algorithms or data structures*

5. Simplifies interfaces

IX. Terminology

A. Information-hiding modules

— identify the design decisions that are likely to change

— have a module for each changeable design decision

-- each changeable decision is a "secret" of a module

B. The secret of a module

Exactly the one design decision that might change -- only the implementor knows

1. Line holder -- how lines are represented in memory

2. Input -- input format

3. Circular shift -- how circular shifts are represented

4. Alphabetizer -- time in which alphabetization is done and sorting method used

5. Output -- output format

C. Structure redefined -- terms of modular structure

1. Connections between modules are assumptions that they make about each other (interface)

2. Mistake — flowchart boxes become modules

D. Frequency of switching from module to module

1. Steps-in-processing approach — low frequency of switching

2. Information-hiding — has many separate, callable routines

3. Macros — after expanded may be same mess but to change the software, one looks at the information-hiding representation

X. Hiding information about the design at write-time and not information at run-time — reduce the connectivity between the modules at write-time and not at run-time

Run-time information versus design-time information

XI. References

Parnas, D. L. 1971. "Information Distribution Aspects of Design Methodology." Proceed. of IFIP Congress 71, pp. 339-344.

Parnas, D. L. 1972. "A Technique for Software Module Specification with Examples." Comm. ACM, vol. 15, no. 5, pp. 330-336.

Parnas, D. L. 1972. "On the Criteria To Be Used in Decomposing Systems into Modules." Comm. ACM, vol. 15, no. 12, pp. 1053-1058.

Linden, T. A. 1976. "The Use of Abstract Data Types to Simplify Program Modifications." Proceed. of Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Special Issue, vol. 11, pp. 12-23.

Parnas, D. L.; Shore, J. E.; and Weiss, D. M. 1976. "Abstract Data Types Defined as Classes of Variables." Proceed. of Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Special Issue, vol. 11, pp. 149-154. Also Naval Research Laboratory Report no. 7998

Parnas, D. L. 1977. "The Use of Precise Specifications in the Development of Software." Proceed. of the IFIP 1977, pp. 861-867.

MOD.2 Change and the Original MP Modular Structure

EXERCISE

Name:_____

1. (Hardware change) Suppose that a bulk core device were added to the optional hardware in the UGH 2PIE system. The device has roughly the capacity of the small disk used (but not that of the extended mass storage or large disk option); it is very fast (about 1/5 the speed of the primary memory, where the disk was 1/10000 for access); and code may be executed from bulk core. If MP is to make the best use of such a device, what modules must be changed? Explain why the change is not confined to one or two modules.

2. (Message format change) Suppose that a message format change is announced in which Format Line 5 may be entirely omitted from a message of the lowest security class. What modules will be affected by this change? Explain why the change is not limited to one or two modules.

MOD.3 Change and the Original MP Modular Structure

EXERCISE SOLUTION

1. The bulk core can replace small disks, but it cannot replace large disks. This means that logs, if kept, will still be on disk, but that all other data can now be in the bulk core. Unfortunately, this could change every module in the system, some substantially, others only slightly (but the slight changes might still be hard to make).

In practice, DK would probably be left in place, modified to use the core. This shouldn't really be considered a triumph for modularization, because it will multiply the overhead by perhaps a factor of ten. Any module that uses DK will now sacrifice a lot of the hardware performance available by not directly using the bulk core.

Changes to EX and DC and DK cannot be avoided, and they are substantial. EX and DC must now choose whether to use main memory or leave the data in bulk core, and they have no algorithms to do anything like this. The complex interaction between DK and EX is no longer required (indeed, since most executions might be done in bulk core, only the scheduling and interrupt service of EX will be required). DC now has little reason to allocate main memory at all and thus may acquire a third class of "disk" allocation for which it is unprepared.

Of course, if IR and LM are included in the system with bulk core, they will profit greatly by using it, and will require large changes to do so. The changed modules will be inappropriate to use on a system with only disk (the UGH-VAN and UGH-20 do not have the bulk core possibility).

Two mistakes in MP are shown here. The first is that disk addresses are used throughout the system, on the assumption that DK would always be present to handle reads/writes. The other was that the core/disk distinction was an early one, which was reflected in the module structure and hence hard to reverse.

2. Certainly CO, SC, and MA will be affected by this change. (SC, even though it doesn't use FL5, because it must skip over it to find the addressee list.) But there will be smaller changes in OP, IR, LM, and even DC and perhaps TO to deal with the changed messages.

The mistake is an obvious one: too many modules duplicate the process of pulling a message apart, using slightly different algorithms, but making a common set of assumptions about message format. Finding all the places where these assumptions enter into code is liable to be very difficult in the completed system.

MOD.4 Modular Structure of Complex Systems

LECTURE

I. Review: Information hiding as a criterion

A. What is a secret?

B. What are some typical secrets?

(See Figure 1, p. 4-19, and Figure 2, p. 4-20)

II. What is different about large complex systems?

A. How do we deal with unstructured lists of modules?

B. How can we tell when we have them all?

C. How does everyone remember the names?

D. How do we avoid duplications?

III. Why should we group modules into classes?

A. Put some structure in the list

B. Help to check for completeness

C. Leads to more helpful naming conventions

D. Makes duplications less likely

IV. What are some possible classification criteria for modules?

A. By level in hierarchy

B. By similarity of interface

C. By type of function served or service provided

D. By nature of the secret

E. Similar programming problems

V. What are the classes of modules in the A-7?

A. The Extended Computer class of modules

Secrets: Implementation of common data types, I/O, etc.

Characteristics of TC-2 computer such as registers, memory structure, etc.

B. The Device Interface Modules

Secrets: Device characteristics

C. The Physical Model Modules

Secrets: Models of physical phenomena

D. The Data Banker Modules

Secrets: Source and updating policies for common data

E. The System Status Modules

Secrets: How the program keeps track of the status of the system and detects state changes of interest to the system

F. The Function Driver Modules

Secrets: Algorithms for performing requirements functions

G. The System Generation Modules

Secrets: Implementation of the tools used to assemble the system from the library of components

VI. Concluding dilemma:

How do you deal with the fact that large systems divided into modules that are small enough to be understood have confusingly many modules?

Figure 1*: Common Secrets in Data Processing Systems

Secret	Typical Reasons for Changes
Data base structure (logical)	- New fields in records - Field sizes changed - More records required - Faster access required for particular fields
Algorithms	- Different time-space tradeoffs required - More accurate or efficient algorithms invented
Data storage (physical)	- Size of available storage changed - Type of available storage changed (e.g., from one tape drive model to another, or from tape to disk) - Faster access required
Input	- Input medium changed (e.g., from cards to OCR) - Fields rearranged within records - More extensive error-checking required - Input sequence changed (e.g , from unsorted to sorted)
Output	- Change in output device (e.g., from printer to computer-output microform)
Operating system interface (e.g., JCL)	- Manufacturer issues new release
Software functions as seen by user	- New types of reports required - Client requires changes in report formats

* From Kathryn Heninger and John Shore, "Designing Modular Programs — Methodology," Auerbach Portfolio 14-01-11.

Figure 2*: Common Secrets in Real-Time Systems

Secret	Typical Change
Computer characteristics	- Computer replaced by faster, larger, or cheaper model - Computer replaced by standard model (e.g., military standard)
Peripheral devices	- Sensors replaced by more accurate, more reliable, or faster models - Displays replaced by more flexible or more reliable models
Resource allocation (e.g., scheduling)	- Relative priorities or activities changed - Single computer replaced by set of micros - Capacity of resources changed, e.g., additional memory added
Algorithms	- More accurate or faster algorithms invented - More general algorithm invented that can replace several more specialized algorithms
Software functions	- User preferences changed, including New modes needed Transition between modes changed New responses required to user inputs New displays needed - Computer-driven devices used for different purposes

* From Kathryn Heninger and John Shore, "Designing Modular Programs — Methodology," Auerbach Portfolio 14-01-11.

MOD.5 MP Secrets

EXERCISE

Name:_____

The modular structure of a system should be based on the aspects of the system that are likely to change. Each changeable aspect should become the secret of a module. Thus it is important during requirements definition and during design to list the changeable aspects of a system. Referring to the MP documents MP.1, MP.2, and MP.3, make a list of the changeable aspects of the MP system. Hint: Some areas of possible change are algorithms, data structures, formats, strategies, and hardware characteristics.

Example:

1. The internal representation of a message.

MOD.6 MP Secrets

EXERCISE SOLUTION

Listed below are some aspects of the MP system that are likely to change and therefore are secrets to be hidden in modules.

1. The internal representation of a message.
2. The external message format(s).
3. The protocol(s) of the communication device(s).
4. The WATCH LIST of messages of interest.
5. The method of controlling UGHTRANS devices.
6. The format of terminal commands and displays.
7. The terminal characteristics.
8. The commands needed to compose and edit a message.
9. The organization of log data.
10. The paging and backup-store system.
11. The resource allocation strategy.
12. The interrupt handler.
13. The organization of WCB queues.
14. The format of the WCBs.
15. The search algorithm for the WATCH LIST.
16. The configuration of the system.
17. Message analysis.

MOD.7 Change and the Improved MP Modular Structure

EXERCISE

Name:_____

1. (Hardware change) Suppose that a bulk core device were added to the optional hardware in the UGH 2PIE system. The device has roughly the capacity of the small disk used (but not that of the extended mass storage or large disk option); it is very fast (about 1/5 the speed of the primary memory, where the disk was 1/10000 for access); and code may be executed from bulk core. If MP is to make the best use of such a device, what modules must be changed? Explain why the change is not confined to one or two modules.

2. (Message format change) Suppose that a message format change is announced in which Format Line 5 may be entirely omitted from a message of the lowest security class. What modules will be affected by this change? Explain why the change is not limited to one or two modules.

3. (Message format change) As a result of the Freedom of Information Act, every message must include a declassification date in both FL4 and FL12. Which modules will require a change?

4. (Hardware change) A microfilm printer has been added to produce hard copy. This device contains two rolls of film, one for unclassified messages, the other for classified. Which modules will require a change?

MOD.8 Change and the Improved MP Modular Structure

EXERCISE SOLUTION

1. This change can be confined to the paging module (PS) because it is the only one that "knows" the nature of the storage devices. Some performance improvement might be obtained by examining other parts of the system, which make a choice about when they request pages be removed or brought in. The optimum choice is a function of the speed of access. Nevertheless, it is possible to use the system initially without such improvements.

2. Only the EI module is affected by this change.

3. This change requires a change in the information supplied by the MH module. The new format violates the assumptions that went into the design of the module (namely, that no such information was present or relevant). The danger exists that every function that uses the functions in MH may have to be changed. For example, the programs that display messages to the operator must be altered to include the new information in the display. Similarly, such information must be included in logs, so that the programs that store messages in logs may also be subject to change. The EI module will also have to change to accommodate the new format.

4. None of the modules requires a change. The control of the new device is done by a new program that can obtain all of the information that it needs from MH. If these microfilm files are considered "logs," the program can be considered part of the log module. The system must consider this new storage as "write only" memory.

MOD.9 Identifying HAS Modules

EXERCISE

Name: _____

The HAS modular structure given in HAS.2 is sensitive to specification changes such as adding a CPU and second transmitter, changing the time interval between sensor readings, or eliminating history report transmissions by storing them internally on a floppy disk (to be picked up by passing ships or by trained dolphins). Propose an improved modular structure for HAS based on the system description given in HAS.1. Use the information-hiding principle to organize the system into modules. Guide your design by using the table below, first filling in the secrets column, and then the module that hides each secret. Derive the secrets from consideration of possible changes described or implied in HAS.1. Give a short description of each module, including its functional capabilities. Recall that secrets include items such as algorithms, data structures, formats, and hardware characteristics. An example entry has been included.

PROPOSED HAS MODULES

Secret	Module Name	Module Capabilities
Sensor Characteristics	Sensor Control	Hidden in this module are the sensor characteristics that might change if we replaced one sensor with another that delivers the same information. The programs that take readings from sensors are in this module; they know the HAS-BEEN instruction sequences that perform sensor input and the hardware defined memory location corresponding to each device.

Secret	Module Name	Module Capabilities

SPEC.1 What Are Specifications?

LECTURE

I. What are "specifications"?

A. General definition of specification

Specific information about the object

B. Engineering definition

Specific information about the requirements the object must meet

C. We will use it in the engineering sense

II. Why do we need specifications?

A. Multiperson projects

B. Multiversion projects

C. "Our inability to do much" (E. W. Dijkstra)

Each subtask should have a definition independent of the rest of the job

D. Making early decisions explicit and precise

1. Intramodule assumptions

2. Decision postponement

III. Why must specifications be precise?

A. Early, distributed design decisions are hard to correct

B. Prevent incompatibility between parts

C. Remove the need for excessive information distribution

D. Minimize forbidden assumptions

IV. Why must specifications be abstract?

A. Abstraction — one model, many realizations

B. Must allow many versions

C. State only requirements

Example: fictitious sort

D. Less information to comprehend

E. User only concerned about that which he could eventually discover for himself

V. What do we mean by formal?

A. Not "superficial"

B. Based on forms and rules

1. No chance of misinterpretation

2. Conceivably interpretable mechanically

VI. Why not English (German, French, Dutch, ...?)

A. Interpretation may (often does) require an elaborate legal system

B. Examples of subtle ambiguities

1. Delivers the top of the stack

2. Delivers the address of the new PSW

3. Removes the top element from the stack

4. The date three months from today

VII. Stating the visible effects of functions on each other

A. The basic technique of formal specifications

B. Refusal to mention internal or invisible effects

The way to abstract specifications

C. Leaving some externally visible values undefined

The way to restrict statements to requirements

VIII. Stating the "syntactic" properties of functions

A. Does the function have a value? What is its type?

B. Input parameters: How many? What type?

C. Output parameters: How many? What type?

1. This information specifies the syntactically allowed calling combinations

2. More information can be added by defining more types

IX. Stating the semantic properties of functions

A. The immediately visible effects

B. Relations among functions (F1 + F2 = 3)

C. Equivalent sequences — enough to define all the effects

D. Effect after a sequence

E. References to "history"

X. Describing "don't cares"

XI. Describing forbidden or undesired actions

A. Stating the allowed actions

B. Stating the effects of restriction violations

C. Specifying conventions for reporting internal errors

XII. Example: Stack with limitations

XIII. Example: Stack without limitations

XIV. Example: Tree

XV. Example: Queue

XVI. Example: Sorting queue

XVII. Completeness and consistency

A. Derivation of value or "undefined" for all sequences

B. Only one value derivable

XVIII. Important vs. notational aspects

A. Rules about content important

B. Syntactic invention still needed

C. Don't let syntax control content

XIX. References

Parnas, D. L. 1972. "A Technique for Software Module Specification with Examples." Comm. ACM, vol. 15, no. 5, pp. 330-336.

Guttag, J. V. 1975. The Specification and Application to Programming of Abstract Data Types. University of Toronto Computer Systems Research Group Technical Report CSRG-59.

Parnas, D. L.; and Handzel, G. 1975. More on Specification Techniques for Software Modules. Fachbereich Informatik, Technische Hochschule Darmstadt.

Guttag, J. V. 1977. "Abstract Data Types and the Development of Data Structures." Comm. ACM, vol. 20, no.6, pp. 396-404.

Parnas, D. L. 1977. "The Use of Precise Specifications in the Development of Software." Proceed. of the IFIP 1977, pp. 861-867.

Liskov, B.; and Zilles, S. 1975. "Specification Techniques for Data Abstractions." IEEE Trans. on Software Engineering, vol. SE-1, no. 1, pp. 7-19.

Bartussek, W.; and Parnas, D. 1977. Using Traces to Write Abstract Specifications for Software Modules. University of North Carolina Report no. TR 77-012.

SPEC.2 Using an Informal Functional Specification

EXERCISE

Name: _____

Using the informal functional specification of the message holder module given in document MP.5, answer the following questions. Quotes enclose a character string. Thus, "RUSSIA" represents the six-character string RUSSIA.

1. What is the value of GET_ROUTING_INDICATOR after executing SET_ORIGINATOR_ROUTING_INDICATOR("RUSSIA")?

2. After executing SET_TEXT(25,30,"tricky"), what is the value of GET_TEXT(30,30)?

3. What is the effect of SET_TEXT(25,25,"tricky")?

4. The original text is "HAPPY BIRTDAY"; what command will correct it?

5. The original text is "HAPPY BIRTHDAY"; what is the text after executing BLANKIT(4)?

6. If the message already contains an addressee, what is the effect of SET_ADDEE("SECNAV")?

7. We have executed BIND(47), stored some text in the message, and then executed NEW_MESSAGE(47); what happens as a result?

8. What will happen if a program executes SET_SERIAL("serial")?

9. How long is the string GET_TEXT(30,30)?

10. If the text of a message currently contains 35 characters, what is the effect of SET_TEXT(50,71,"MP SOFTWARE")?

SPEC.3 Formal Functional Specifications

LECTURE

I. Purpose: Stating requirements that an implementation of an information-hiding module must satisfy

A. State everything that is required

B. State nothing that is not required

C. Leave no room for doubt

II. Three views about showing internals

A. Mention no internals

Why:

Why not:

B. Mention hypothetical "ridiculous internals"

Why:

Why not:

C. Mention hypothetical "suggested internals"

Why:

Why not:

III. Syntax in a specification

A. What is a type?

B. "Presentation" presents type information

IV. What is a trace?

A. Execution history of a module from creation

B. A subtrace is part of a trace

C. Notation for describing traces and subtraces

1. F(a, b, c)

2. F(a, b, c).Y(2, 3, 4)

3. ␣

4. V(T)

5. S^n

V. What kinds of assertions can be made about traces?

A. Which traces must be legal?

B. When are two traces equivalent?

1. Equivalent traces must be indistinguishable from outside the module

2. Two traces are not equivalent unless shown to be equivalent

C. Values of traces in terms of previously defined types

VI. Specification of an unbounded stack

A. Syntax:

```
PUSH:     ⟨integer⟩ x ⟨stack⟩ -> ⟨stack⟩
POP:                  ⟨stack⟩ -> ⟨stack⟩
TOP:                  ⟨stack⟩ -> ⟨integer⟩
DEPTH:                ⟨stack⟩ -> ⟨integer⟩
```

B. Semantics:

1. Legality:

a. $\lambda(T) \Rightarrow \lambda(T.PUSH(a))$
b. $\lambda(T.TOP) = \lambda(T.POP)$

2. Equivalences:

   c. T.DEPTH ≡ T
   d. T.PUSH(a).POP ≡ T
   e. λ(T.TOP) => T.TOP ≡ T

3. Values:

   f. λ(T) => V(T.PUSH(a).TOP) = a
   g. λ(T) => V(T.PUSH(a).DEPTH) = 1 + V(T.DEPTH)
   h. V(DEPTH) = 0

The above specification assumes that only one stack exists and omits the stack parameter in the trace assertions.

VII. Specification of an unbounded queue

A. Syntax:

```
ADD:       ⟨integer⟩ x ⟨queue⟩ -> ⟨queue⟩
REMOVE:                ⟨queue⟩ -> ⟨queue⟩
FRONT:                 ⟨queue⟩ -> ⟨integer⟩
```

B. Semantics:

1. Legality:

   a. λ(T) => λ(T.ADD(a))
   b. λ(T) => λ(T.ADD(a).REMOVE)
   c. λ(T.REMOVE) = λ(T.FRONT)

2. Equivalences:

   d. λ(T.FRONT) => T.FRONT ≡ T
   e. λ(T.REMOVE) => T.ADD(a).REMOVE ≡ T.REMOVE.ADD(a)
   f. ADD(a).REMOVE ≡ ⌴

3. Values:

   g. V(ADD(a).FRONT) = a
   h. λ(T.FRONT) => V(T.ADD(a).FRONT) = V(T.FRONT)

The above specification assumes that only one queue exists and omits the queue parameter in the calls on the access programs.

VIII. Specification of a sorting queue

A. Syntax:

INSERT: ⟨ integer ⟩ x ⟨ squeue ⟩ -> ⟨ squeue ⟩
REMOVE: ⟨ squeue ⟩ -> ⟨ squeue ⟩
FRONT: ⟨ squeue ⟩ -> ⟨ integer ⟩

B. Semantics:

1. Legality:

a. $\lambda(T) \Rightarrow \lambda(T.INSERT(a))$
b. $\lambda(T) \Rightarrow \lambda(T.INSERT(a).REMOVE)$
c. $\lambda(T.FRONT) = \lambda(T.REMOVE)$

2. Equivalences:

d. $\lambda(T.FRONT) \Rightarrow T.FRONT \equiv T$
e. $T.INSERT(a).INSERT(b) \equiv T.INSERT(b).INSERT(a)$
f. $INSERT(a).REMOVE \equiv \sqcup$
g. $\lambda(T.FRONT)$ cand $(V(T.FRONT) \leq b) \Rightarrow T.INSERT(b).REMOVE \equiv T$

3. Values:

h. $V(INSERT(a).FRONT) = a$
i. $\lambda(T.FRONT)$ cand $V(T.FRONT) \leq b \Rightarrow V(T.INSERT(b).FRONT) = b$

NOTE: The value of X cand Y is false if X is false, and the value of X cand Y is the value of Y is X is true. Y need not have a defined value if X is false.

IX. Specification of a stack that overflows at the bottom

A. Syntax:

PUSH: ⟨stac⟩ x ⟨integer⟩ -> ⟨stac⟩
POP: ⟨stac⟩ -> ⟨stac⟩
VAL: ⟨stac⟩ -> ⟨integer⟩

B. Semantics:

1. Legality:

For all T, $\lambda(T)$

2. Equivalences:

$0 < N \leq 124 \Rightarrow PUSH^{N}(a_i).POP \quad PUSH^{N-1}(a_i)$
$PUSH(a_0).PUSH^{124}(a_i) \equiv PUSH^{124}(a_i)$
$T.VAL \equiv T$
$N \geq 0 \Rightarrow POP^{N}.PUSH(a) \equiv PUSH(a)$

3. Values:

$V(T.PUSH(a).VAL) = a \bmod 255$

X. When is a specification complete? Do we always want completeness?

XI. When is a specification consistent?

XII. Effect of programming languages

A. Lack of choices in some languages leads to simplification

B. Lack of "functions" leads to minor complication

C. User-defined types can simplify specifications

XIII. Dealing with nonsequential systems

XIV. Three heuristics

A. Minimal subset approach

B. Looking for sequences that make the system "forget"

C. Canonical forms for showing completeness

XV. Open problems

SPEC.4 Coding Specifications

LECTURE

I. Motivation

A. It is sometimes useful to have a level of documentation between module interface designs and module implementation code

1. Can ease maintenance (code is sometimes hard to read)

2. Can provide a useful form for reviewing module implementation design decisions

3. Can enable the design of module implementations valid for more than one language

B. Such documentation can ease problems that sometimes occur with *programmers responsible* for implementing modules or parts of modules

1. May not have "big picture"

2. Will (naturally) optimize locally

3. May not have much experience

4. May misinterpret requirements

II. What are coding specifications?

A coding specification for a given program is a document in which pseudo-code or abstract programs are used to constrain the selection of algorithms and data structures or to specify them completely. Whatever the extent of the constraints imposed, the coding specification should contain all information (or references) required to write complete and correct code for the program.

III. Why not just write programs?

A. The programming language may not be well-suited to communicating algorithms to people

Examples: use of case, while, if ... then ... else.

```
if  flag                          IF (FLAG)  GO TO 10
      then action_1;                 action_2
      else action_2;                 GO TO 20
end-if;                         10 action_1
                                20 CONTINUE
```

B. Since constraints imposed on different code may vary, we need a degree of informality

Example: "sort A"

vs.

"sort A using algorithm efficient for N less than 15"

vs.

"bubble sort A"

vs.

"for I:= 1 step 1 to N do ..."

C. Often desirable to introduce special notation to aid in specification, communication, or maintenance

1. Example: Physical field references

R[5:9] or R["opfield"]

vs.

RSHIFT(LSHIFT(R,4), 28)

2. Example: names instead of values

if I gt SYSNUM then error(TOOBIG); end-if;

vs.

if I gt 796 then error(3); end-if;

IV. Selecting a code specification "language"

A. The actual programming language might be a suitable base

B. It should be straightforward to translate to the programming language — don't use a base like APL or LISP for coding specifications if the system will be written in FORTRAN

C. Balance formality and informality

1. Formality can facilitate automatic indexing, cross-referencing, and checking

2. Informality provides the advantages discussed in section III

V. Using coding specifications

A. Have module designers write them and other designers review them

B. Have author and coder collaborate on necessary changes

C. Have author review resulting code

D. Use in designing and analyzing system tests

E. May be appropriate to keep them current and use as long-term documentation (depends on readability of code)

F. Keep on-line

G. Use utility programs

VI. Summary

CODING SPECIFICATION EXAMPLE

Interface Specification

)CLWTCT(

FUNCTION: Clears the contents of the write access counters (WCOUNT) for ATRTAB entries other than labels, procedures, or constants.

COMMENTS AND DESCRIPTION: Counters can be incremented or cleared but not set to a specific value.

CALLING SEQUENCE: CALL CLWTCT(ERRCODE)

PARAMETERS:

ERRCODE	INT;R	+ or - the function identifier of caller ⟨ERRINC⟩ Codes: ⟨OK⟩ ⟨NOBIND⟩ ≡ no ATRTAB entry bound ⟨ILLTYP⟩ ≡ label, procedure, or constant bound

Tables Referenced	Entry Types Referenced	Logical Components Referenced
ATRTAB	ARRAY	WCOUNT
	REG	WCOUNT
	TREG	WCOUNT
	TFLAG	WCOUNT

Interface Specification

)INWTCT(

FUNCTION: Conditionally increments the write access counter (WCOUNT) for ATRTAB entries other than labels, procedures or constants. If the STATUS field indicates that counting is enabled, counting is performed.

COMMENTS AND DESCRIPTION: Counters can be incremented or cleared but not set to a specific value.

CALLING SEQUENCE: CALL INWTCT(ERRCODE)

PARAMETERS:

ERRCODE	INT;R	+ or - the function identifier of caller ⟨ERRINC⟩ Codes: ⟨OK⟩ ⟨NOBIND⟩ ≡ no ATRTAB entry bound ⟨ILLTYP⟩ ≡ label, procedure or constant bound

Tables Referenced	*Entry Types Referenced*	Logical Components Referenced
ATRTAB	ARRAY	WCOUNT
	REG	WCOUNT
	TREG	WCOUNT
	TFLAG	WCOUNT

Coding Specification

Routines to clear access counters

ROUTINE	[PHYSICAL FIELD]
)CLRDCT(	[READ COUNT]
)CLWTCT(	[WRITE COUNT]
)CLMOCT(	[MONITOR COUNT]

Routines to increment access counters

ROUTINE	[PHYSICAL FIELD]
)INRDCT(	[REAL COUNT]
)INWTCT(	[WRITE COUNT]
)INMOCT(	[MONITOR COUNT]

PARAMETER FILES:

ACERRI.REQ	(ERRINC) Codes for calling)ERR(
ACNAM.REQ	Parameter definitions for routine name codes used for calling)ERR(function
ACATIP.REQ	ATRTAB entry type codes
ARFSIZ.REQ	Parameters for array size declarations and table entry size definition

COMMON BLOCK DEFINITION FILES:

ACSTOR.REQ	Primary data structure for support of table access routines

EXTERNAL REFERENCES:

)ERR(	for error reporting
(Byte and Halfword routines)	As required to extract table entries (See references (c), (d), and (e))

```
ALGORITHM:

    /* ⟨ ROUTINE⟩ is the parameter containing the name of the routine being
       coded. */

    !if !DEBUG
       !then /* code the following with 'D' in column 1 */

          if ATRBD {ACSTOR} = -1
             then

                call ERR(ERRCODE, ⟨NOBIND ⟩ + ⟨ERRCSZ⟩ * ⟨ ROUTINE⟩)
                return

             end

          if ATRTP {ACSTOR} is not appropriate for the desired [PHYSICAL FIELD]
             then

                call  ERR(ERRCODE, ⟨ILLTYP ⟩ + ⟨ERRCSZ⟩ * ⟨ROUTINE⟩)
                return

             end

          !end /* !DEBUG */

       !if routine being coded clears a counter
          !then

       [PHYSICAL FIELD]:= 0

          !else /* routine must increment a counter */

       if CTSTAT {ACSTOR}
          then /* counting will be effective */

             [PHYSICAL FIELD]:= [PHYSICAL FIELD] + 1

          end

          !end

       return
```

ABS.1 Abstract Interface Modules and Their Value

LECTURE

I. Introduction -- review

A. The value of being explicit about design decisions and assumptions

1. Example -- the "fundamental assumptions" list for the A-7

2. Program families — choosing the order of decisions

Difficulties:

3. Modules and information hiding

Difficulties:

B. What is an interface?

1. More than just syntax or format

2. An interface between two programs is defined by the set of explicit and implicit assumptions they make about each other

II. What's special about DoD software interfaces?

A. Not just a question of size or real-time demands

B. Definition -- an embedded computer system is considered a module in some larger system

C. Some distinguishing characteristics of embedded computer systems

1. Designer not free to define interface

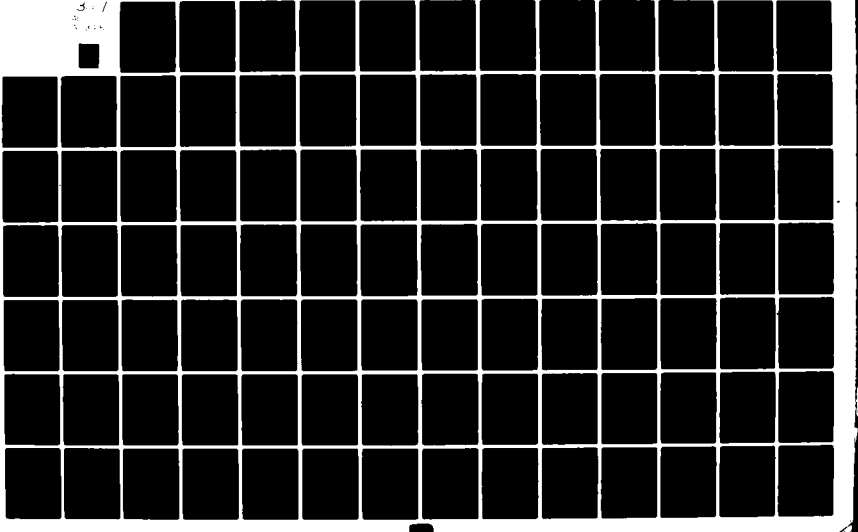
2. Interface constraints may be strict and arbitrary, but we can't ignore them

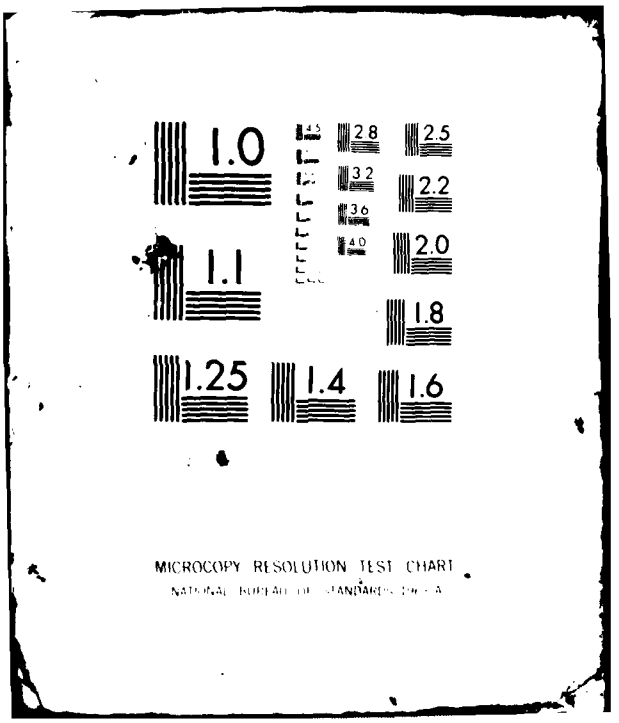
3. Several similar interfaces may be involved

4. Interface will change during development

AD-A113 415 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
SOFTWARE ENGINEERING PRINCIPLES 3-14 AUGUST 1981,(U)
AUG 81 L J CHMURA, P C CLEMENTS

UNCLASSIFIED NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

5. Cost of changing computer system not considered seriously when changes in total system are made

6. Commercial contrasts

D. A contractual dilemma

1. Contract must constrain contractor by providing testable specifications

2. For above reasons, final interface must be considered unknown

3. System for "wrong" interface will be hard to change

4. Lack of competition makes changes afterward unreasonably expensive

E. Preponderance of embedded systems -- a partial explanation for the high cost of DoD software

1. Reason may not be functional complexity, programming tools, programmers' abilities

2. Technical advances can help

III. Examples of embedded systems

A. The address holder system (our programming problem)

Constraints that may change:

B. MP

Constraints:

C. Radar data analysis systems

Constraints:

D. Computers in weapons systems — e.g., TC-2 computer in the A-7

Constraints:

IV. Examples of implicit assumptions in an interface and their effects on application programs

A. Address holder

B. A-7

V. Applying "information hiding" to solving the interface problem when externals will change

A. Review of information hiding

B. Use an "abstract interface" to "hide" the actual interface

VI. Abstract interface modules

A. What do we mean by abstract?

1. Do not mean vague or highly mathematical; abstract means conceived apart from special cases

2. Abstract implies a many-to-one mapping that models some aspects but not all

3. Examples of abstractions

a. Circuit diagrams

b. Address holder assumptions

c. Graphs

d. Algorithms

e. Data types

B. Why are abstractions useful?

1. If all properties of the abstract system correspond to properties of the real system — we can learn about the real system by studying the abstraction

2. Abstraction is simpler (in principle, but abstract thinking may be unfamiliar)

3. Results about abstraction may be "reused"

C. What is an abstract interface?

1. Represents many possible actual interfaces

2. Models some properties of actual interface but not all

3. All things true of the abstract interface are true of actual interfaces

VII. How can abstract interface modules help?

A. Define the abstract real-world interface

B. Procure applications programs based on abstract interface, preventing exploitation of facts that happen to be true of today's actual interface

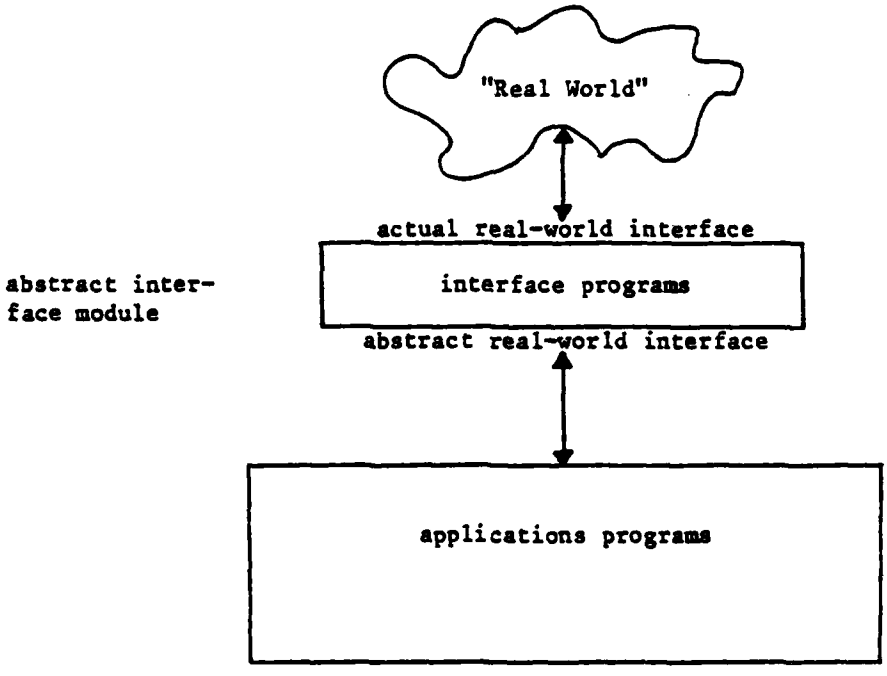


Figure 1. Abstract Interface Module.
The interface programs implement one instance of the many-to-one mapping between the actual real-world interface and the abstract real-world interface.

C. When actual interface is fixed, build interface programs

D. "Real-World" changes that affect actual interface should only affect the interface programs

E. Simple example — a date interface. Possible formats in actual interfaces:

February 10, 1941	(month day-in-month, year),
10 February 1941	(day-in-month month year),
10 February 41	(day-in-month month last-two-digits-of-year),
10.2.1941	(day-in-month.integer-encoded-month.year),
2/10/1941	(integer-encoded-month/day-in-month/year),
41.2.10	(last-two-digits-of-year.integer-encoded-month. day-in-month),
41 February 10	(last-two-digits-of-year month day-in-month),
41,41	(day-in-year,last-two-digits-of-year).
15015	(days since the first day of 1900)

VIII. How to design an abstract interface module

A. Prepare a list of assumptions about properties of all the possible real-world interfaces to be encountered — have this list reviewed

B. Express these assumptions by defining a set of "functions" representing possible system inputs and outputs and by stating relations between these functions

C. Perform consistency checks

1. Verify that any property of the function set is implied by the assumptions

2. It should be possible to write bulk of system in terms of these functions; if not, return to A

D. Contractor is required to write bulk of system in terms of the functions defined in B, and programs must be correct for any implementation of those functions that satisfies the description B

E. Illustration of this procedure for the address holder (programming example)

1. Initial assumptions

The following items of information will be contained in addresses and can be identified by analysis of the input data; this information is the only information that will be relevant for our computer programs:

Last name
First name
Organization
Street address
City, state and zip code (single line with a comma between city and state)

2. Objections

3. Refined assumptions

F. Another example — abstract interfaces for the A-7 Device Interface Module

IX. Refining/extending the interface for a subset of the interfaces

A. Some useful applications programs may not be generally applicable

B. Confinement of the specialized program

C. Specialization (refinement) by adding functions *not* generally implementable

D. Specialization (refinement) by stating additional properties of functions

E. Deviant actual interfaces

F. The family tree again

X. When won't it work?

A. Success depends on our ability to predict change (oracle assumption)

B. Success depends on existence of commonality between actual interfaces (interface programs smaller than applications programs)

XI. Summary

A. An interface is equivalent to a set of assumptions

B. The abstract interface is a precise, formally specified interface

C. The abstract interface is a model of all "expected" actual interfaces

D. Contractor is more tightly constrained than in conventional procedure — his program is not allowed to make assumptions that limit applicability

E. Actual interface is met by writing additional programs — not by modifying programs that were written based on the abstract interface definitions

XII. Abstract interface module as an application of fundamental principles

A. Being explicit about assumptions and design decisions

B. Encapsulation of likely changes

C. Abstract interface modules can solve the embedded computer system problem by hiding the embedding from the computer!

D. Abstract interface modules are just a special case — use same method for other information-hiding modules

XIII. Reference

Parnas, D. L. 1977. Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems. Naval Research Laboratory Report no. 8047.

ABS.2 Using the MP Abstract Interface

EXERCISE

Name:_____

The message on page 6-18 has been assembled according to the message format rules in the original MP design (MP.3). These rules are summarized on pages 6-16 and 6-17 of this exercise.

Your job is to use the abstract interface functions in MP.6 to assemble the same message. To do this, complete the list of function calls started below.

```
CREATE(bdaymsg)

SET_ORIGIN_ROUTE_PART(bdaymsg, DB)

SET_CHANNEL_ID(bdaymsg, 3)

SET_TIME_CREATED(bdaymsg, CLOCK24(12, 00))
```

MESSAGE FORMAT FROM ORIGINAL MP DESIGN

To summarize the original MP message format, we provide a few general rules, a message template, and a list of the fields in the template.

General rules

A message consists of a number of Format Lines numbered beginning with one (these are abbreviated FL1, FL2, etc.). Capital letters represent themselves, and where given, must appear exactly. Where information is to be supplied, a lower-case name will appear, explained in the list below. Where items are optional, they are enclosed in square brackets; where a choice of items is permitted, these are shown one above the other. The spaces shown are nonrepresentative: the characters begin in the first column, and continue to the end of the format line without spacing unless explicit spaces are indicated by the symbol b. Each line ends with a sequence of two-carriage-returns-and-a-line-feed, not shown. When an item is superscripted, it is repeated that many times; superscript ^{n} means an indefinite repeat (but at least once).

Fields in the message template

The following list describes the form and content of the fields in the template, given on the next page.

"origin-route-part":	2-letter part of the originating routing code (the 3rd-last and 2nd-last letters of the code),
"channel":	3-digit number identifying the channel,
"precedence":	1-letter code from a standard list,
"origin-media":	1-letter language media code from a standard list,
"dest-media":	1-letter language media code from a standard list,
"classification":	1-letter security classification letter from a standard list,
"content-action":	4-letter identifier from a standard list,
"sender-orig-route":	7-letter routing indicator of the sender,
"serial":	4-digit number supplied by the sender,
"day-in-year":	3-digit Julian day-of-the-year, for date when message created,
"time":	4-digit GCT at which the message created,
"addressee-route":	7-letter routing indicator for the addressee,
"year":	2-digit year when message created, e.g., 76 for the bicentennial year,
"originator":	either "sender-orig-route" or plain text,
"addressee":	the plain text corresponding to the routing indicator it follows. (In the final addressee item the period replaces the comma, and similarly in FL8, 9.)
"subj-code":	6-character code composed of the letter N and 5 digits
"text":	the message text,
"null":	an empty line (but with the usual ending),
"lf":	line-feed,
"day-in-month":	2-digit day in month.

Message template

FL1: VZCZC origin-route-part channel

FL2: precedence origin-media dest-media classification content-action b sender-orig-route serial b date time b classification[4] addressee-route

FL3: DE b sender-orig-route serial date time b year

FL4: ZNR b classification[5] T [addressee-route]

FL5: precedence b day-in-month time Z b JAN ... DEC b year b

FL6: FM b originator

FL7: TO b [routing / addressee,]^{n}.

FL8: [INFO b [routing / addressee,]^{n} .]

FL9: [XMT b [routing / addressee,]^{n} .]

FL11: BT

FL12: classification //subj-code// text

FL13: BT

FL15: # serial

FL16: null lf[7] NNNN

MESSAGE TO BE REASSEMBLED

	Format Line (not part of message)
VZCZCDB003	FL1
RTTUZYUW RUCLDBA2355 1861200 UUUURUHHLFA	FL2
DE RUCLDBA23551861200 76	FL3
ZNR UUUUU	FL4
R 041200Z JUL 76	FL5
FM COMNAVTELCOM WASHINGTON DC	FL6
TO RUHHLFA/ALCOM.	FL7
BT	FL11
U//N09999//	FL12
HAPPY BIRTHDAY	FL12
BT	FL13
#2355	FL15
(8 blank lines)	FL16
NNNN	FL16

ABS.3 Using the MP Abstract Interface

EXERCISE SOLUTION

CONSTRUCTING MESSAGES WITH THE MP ABSTRACT INTERFACE

```
CREATE(BDAYMSG)

SET_ORIGIN_ROUTE_PART (bdaymsg, DB)

SET_CHANNEL_ID(bdaymsg, 3)

SET_TIME_CREATED (bdaymsg, CLOCK24(12, 00))

SET_PRECEDENCE(bdaymsg, R)

SET_ORIGIN_MEDIA(bdaymsg, T)

SET_DEST_MEDIA (bdaymsg, T)

SET_CLASSIFICATION (bdaymsg, U)

SET_CONTENT_ACTION(bdaymsg, ZYUW)

SET_SENDER_ORIG_ROUTE(bdaymsg, RUCLDBA)

SET_SERIAL(bdaymsg, 2355)

SET_DATE_CREATED (bdaymsg, JULIAN(76, 186))
    or SET_DATE_CREATED(bdaymsg, DAYMOYR(4, JULY, 76))

SET_ADDRESSEE_ROUTE(bdaymsg, RUHHLFA)

SET_ORIGINATOR(bdaymsg, COMNAVTELCOM WASHINGTON DC)

SET_TO_LIST (bdaymsg, RUHHLFA, ALCOM)

SET_SUBJECT_CODE(bdaymsg, N09999)

SET_TEXT(bdaymsg, HAPPY BIRTHDAY)
```

COMMENTS

1. The "SET" functions may be called in any order; the abstract interface programs arrange the information in the order required by the message format.

2. No function calls are needed for control characters such as "VZCZC"; these are inserted by the abstract interface programs.

3. A given "SET" function need be called only once, even if the information appears in the message several times. For example, even though "precedence" is inserted in both FL1 and FL5, "SET_PRECEDENCE" need only be called once. The abstract interface programs take care of the repetition.

4. Even though the date appears in two forms (see FL3 and FL5), "SET_DATE_CREATED" need only be called once. The abstract interface programs can compute the information required for both forms from the date variable it receives as the "SET_DATE_CREATED" parameter. It doesn't matter which "date" function (JULIAN or DAYMOYR) is used to create the date variable.

HIE.1 Hierarchy Survey

LECTURE

I. Introduction

A. Much disagreement about benefits and disadvantages of hierarchical structures for computer software

B. Many different things meant by "hierarchical structure"

C. Nontrivial hierarchical structures <u>always</u> imply restrictions placed on the programmer

1. Restrictions may result in disciplined programming and a quality product

2. A given set of restrictions may not be appropriate for all situations

D. Purpose of lecture: Survey of several well-known hierarchical structures

II. Definition of structure

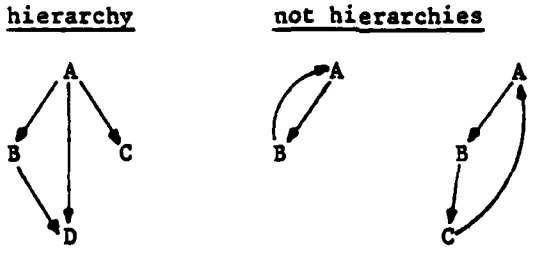
A. Division into parts

B. Relation between parts

C. Structure graphs

III. Definition of hierarchical structure

A. A structure with no loops in its relation



B. Before you know what someone means by a hierarchical structure, you must know the parts and the relation

C. Hierarchies not necessarily trees

IV. The uses hierarchy

A. Parts: programs

Relation: uses

Time: late design time

B. Definition of uses:

Given program A with specification S_a and program B, we say that A uses B if A cannot satisfy S_a unless B is present and satisfies some nontrivial specification S_b. The assumed specification S_b may differ for different users of B

C. Differences between call and use

1. Calls that are not uses

2. Uses that are not calls

3. Example: hardware for division
   uses power supply
   but calls divide by 0 routine

D. Virtual-machine analogy

E. Found in T.H.E., also in many examples of structured programming

F. Advantages

1. Availability of tailored subsets

2. Fail-soft capabilities when UEs occur

3. Incremental development

   Counter example: Multics file system

4. Code duplication avoided

V. The Gives Work Hierarchy

A. Parts: processes

Relation: gives an assignment to

Time: run time

B. Found in T.H.E.

C. Useful in guaranteeing termination and preventing deadlock; neither necessary nor sufficient

D. In T.H.E. system uses and gives work hierarchies coincide

VI. The Resource Allocation Hierarchy

A. Parts: processes

Relation: allocates a resource to, or owns the resources of

Time: run time

B. Applicable with dynamic resource administration only

C. "Allocates to" vs. "controls": The question of preemption

D. Advantages

1. Interference reduced or eliminated

2. Deadlock possibilities reduced

E. Disadvantages

1. Poor utilization when load unbalanced

2. High overhead when resources are tight (especially with many levels)

VII. The Courtois Hierarchy

A. Parts: operations

Relation: takes more time and occurs less frequently than

Time: run time

B. Economics analogy

C. Approximately decomposable systems

D. T.H.E. comparison

VIII. The Module Decomposition Hierarchy

A. Parts: modules

Relation: part of

Time: early design time

B. All of a module's functions are not on the same level of the uses hierarchy. Not all functions need be offered in all system subsets.

C. Never a loop in "part of" — module decomposition always a hierarchy

IX. The Created Hierarchy

A. Parts: processes

Relation: created

Time: run time

B. Must be a hierarchy (father is older than son)

C. Why a tree? — team work in creating progeny is accepted practice

D. Sometimes implies unnecessary restrictions

1. Father cannot die until all progeny die

2. Progeny die when father dies

X. The Protection Hierarchy (Multics)

A. Parts: system components

Relation: can access the data of

Time: design time and run time
(decisions made at design time, enforced at run time)

B. Disadvantage: Violate need to know principle

XI. Conclusions

A. When someone tells you "the software is hierarchically structured"

1. Find out what they mean (What are the parts? What is the relation?)

2. Evaluate appropriateness for particular application

B. Forcing different structures to coincide may lead to an unrealistic design

XII. References

A. General:

Parnas, D. L. 1974. "On a 'Buzzword': Hierarchical Structure." Proceed. of IFIP Congress 74, pp. 336-339.

B. Uses:

Dijkstra, E. W. 1968. "The Structure of the 'T.H.E.' Multiprogramming System." Comm. ACM, vol. 11, no. 5, pp. 341-346.

Parnas, D. L. 1976. Some Hypotheses About the "Uses" Hierachy for Operating Systems. Technical Report. Darmstadt, W. Germany: Technische Hochschule Darmstadt.

Parnas, D. L. 1979. "Designing Software for Ease of Extension and Contraction." IEEE Trans. on Software Engineering, vol. SE-5, no. 2, pp. 128-137.

C. Gives Work:

Habermann, N. A. 1969. "Prevention of System Deadlocks." Comm. ACM, vol. 12, no. 7, pp. 373-377.

Dijkstra, E. W. 1968. "The Structure of the 'T.H.E.' Multiprogramming System." Comm. ACM, vol. 11, no. 5, pp. 341-346.

D. Owns Resources:

Brinch Hansen, P. 1970. "The Nucleus of a Multiprogramming System." Comm. ACM, vol. 13, no. 4, pp. 238-241, 250.

E. Shorter Duration -- Higher Frequency:

Courtois, P. J. 1975. "Decomposability, Instabilities and Saturation in Multiprogramming Systems." Comm. ACM, vol. 18, no. 7, pp. 371-377.

Courtois, P. J. 1977. Decomposability: Queuing and Computer System Applications. New York: Academic Press.

F. Module Decomposition:

Parnas, D. L. 1972. "On the Criteria to be Used in Decomposing Systems into Modules." Comm. ACM, vol. 15, no. 12, pp. 1053-1058.

G. Created:

Brinch Hansen, P. 1970. "The Nucleus of a Multiprogramming System." Comm. ACM, vol. 13, no. 4, pp. 238-241, 250.

HIE.2 Designing a Uses Hiearchy

LECTURE

I. Goals

A. Program families: Different installations require different capabilities

1. Systems with different capacities

2. Systems with different degrees of flexibility

3. Spectrum: ONE to FIXED to VARYING

B. Adjustable systems: Easy to extend or subset

1. Ability to remove functions to make room for other functions

2. Fail-soft response to undesired events

II. Alternatives available to the software procurer

A. The super system: Generality costs!

B. A system for the "average" user

C. A set of independently developed systems

** D. A subsettable super system — each family member offers a subset of the services provided by the largest member

1. Individual installations only pay for what they need

2. Ability to extend by adding programs, without changing existing programs

3. Incremental implementation possible

III. Uses hierarchy, reviewed

A. Parts: Programs, not modules

B. Relation: "Requires correct operation of"

C. When defined: Late design time

D. Purpose: Additional specifications for programmers

E. Result: A heirarchy representing controlled interdependencies among programs

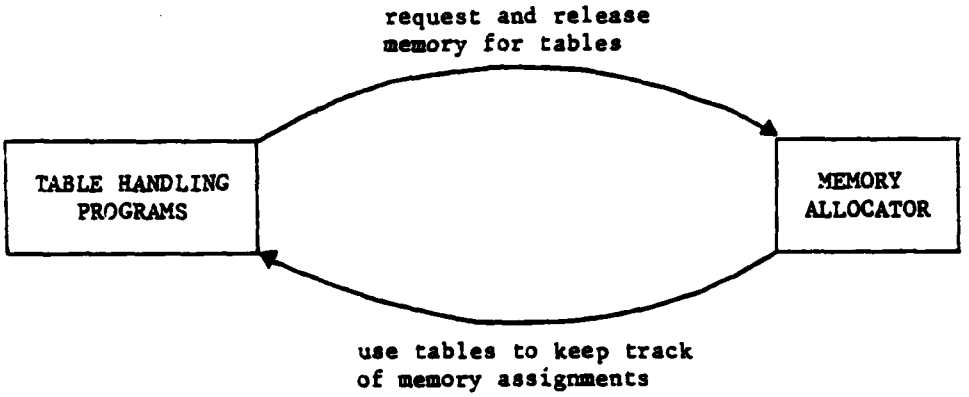
F. Why important

1. Determines possible subsets

2. Determines possible fail-soft modes

3. Affects order of program integration

G. Design error: Loops in uses hierarchy



Two dangers:

1. Memory allocator and table generator use each other

   — Neither works until both work

   — If either is removed, system no longer works

2. Memory allocator builds own tables

   — Code duplication

IV. Basic steps in the design of a subsettable system

A. Requirements definition: Identify the subsets first

B. List programs belonging to each module

1. Access programs

2. Internal programs — cannot be used directly by programs outside the module

3. Main programs — cannot be used — top level in uses hierarchy

C. For every pair of programs, three possibilities

A may use B

B may use A

Neither may use the other

D. List programs at level 0: Programs that use no other programs

E. Work up from there

— Level 1 programs use only level 0 programs

— Level 2 programs use only level 0 or level 1 programs, etc.

F. Four conditions for allowing program A to use program B

1. A is simpler because it uses B

   — Information Retrieval programs in MP simpler because they use Page Storage programs: Don't have to know details of memory handling

2. B is not much more complex because it is not allowed to use A

   — Page Storage programs: Wouldn't be simpler if they used Information Retrieval programs

3. There is a useful subset containing B and not A

   — Page Storage programs useful for other purposes besides Information Retrieval, e.g., for implementing Message Holder programs

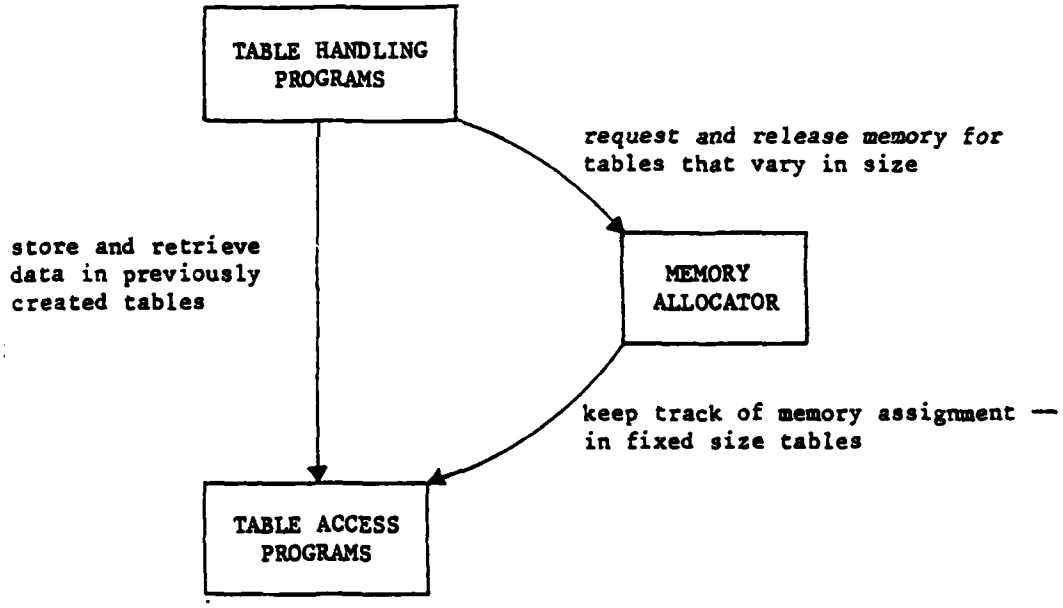
4. There are no useful subsets containing A and not B

   — No reason to have Information Retrieval without memory (Page Storage)

G. Often not clearcut: Must make tradeoff decisions

H. Refinement through sandwiching — what to do if the four conditions don't hold

— If A and B use each other, can one be split into two parts, i.e., a simple version and a complex version?



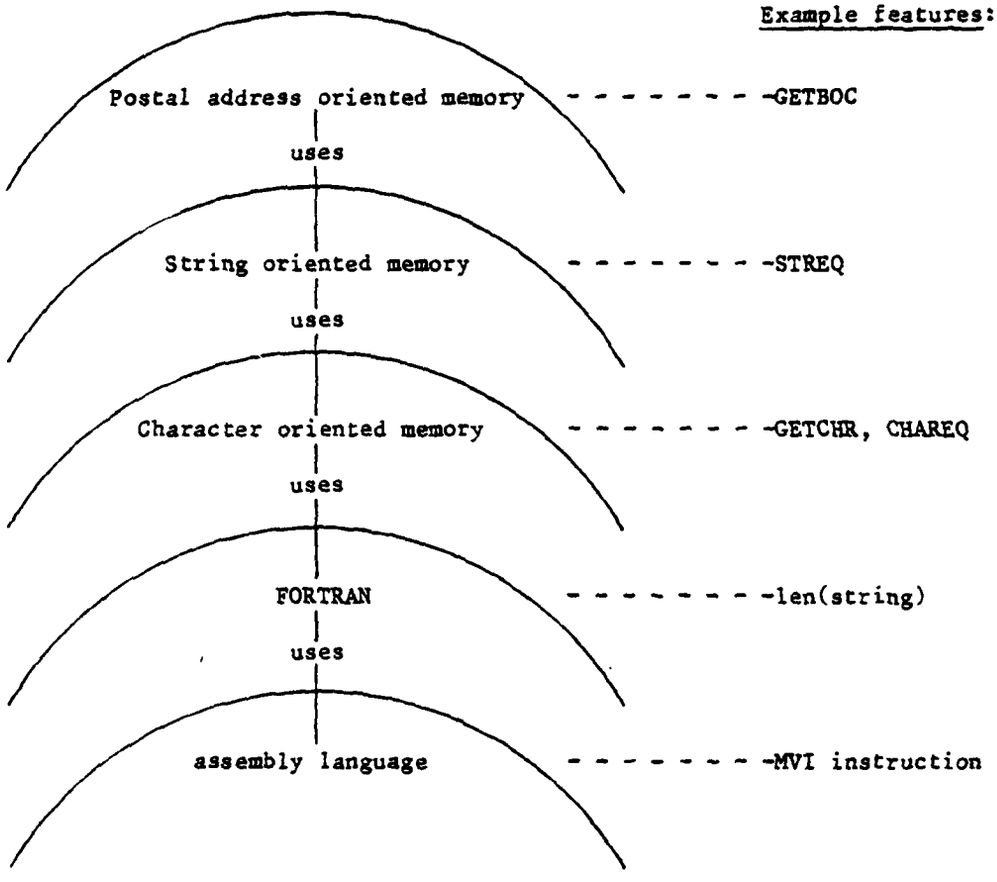
V. Result: Layers of virtual machines

A. Definition: A set of objects and operations, implemented in software, that could conceivably be provided by hardware

B. Applications programs are simpler because they use virtual machine programs

C. Resources used to implement a virtual machine not available to a program that uses it

D. Example from the MADDS example: Layered virtual machines



VI. Deriving subsets from a uses hierarchy

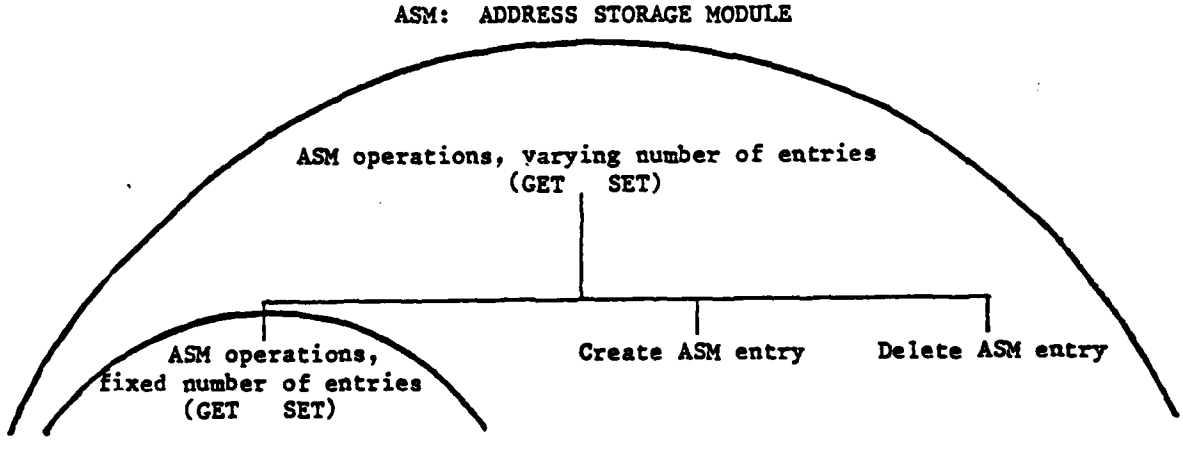
A. Rules

1. Can leave out upper levels

2. Can leave out parts of levels

3. If program A left out, must leave out all programs that use it

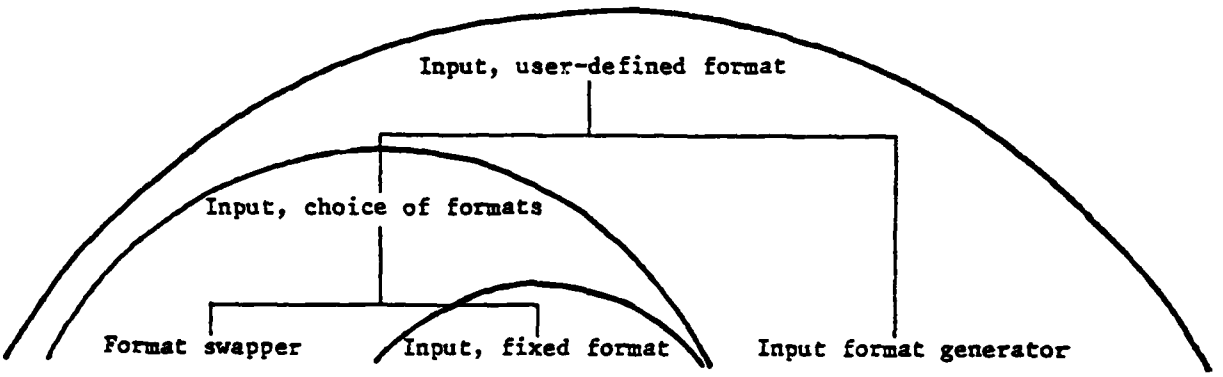
B. Example: Part of hierarchy for family of systems with different capacities



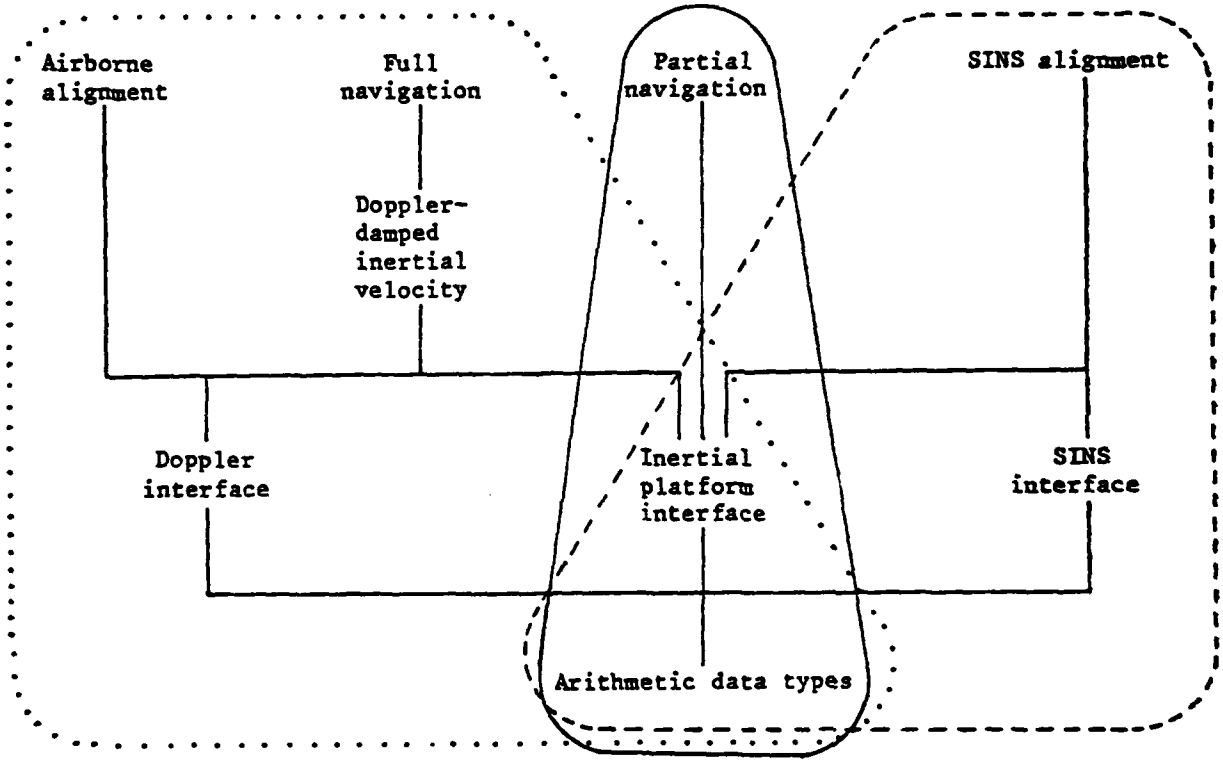
Note: The number of entries must be parameterized even for the simple ASM, so that the variability can be added easily. For systems with only the simple ASM, the capacity can be a system generation parameter that doesn't change at run-time. Thus different installations can trivially have different capacities.

Key: Anticipate future extension.

C. Example: Part of hierarchy for family of systems with different degrees of flexibility



D. Example: Part of hierarchy for single system that can be subsetted easily



VII. Evaluation criteria for a uses hierarchy: What are the goals?

A. Elegance and simplicity

B. Avoid duplication

C. The existence of an appropriate subset for each application situation ("without consideration of subsets, anything goes")

VIII. Observations

Uses hierarchy as a compromise between

A. Letting any program use any other -- excessive dependencies

B. Not letting anything use anything — duplication

IX. References

Parnas, D. L. 1976. Some Hypotheses about the "Uses" Hierarchy for Operating Systems. Technical Report. Darmstadt, W. Germany: Technische Hochschule Darmstadt.

Parnas, D. L. 1979. "Designing Software for Ease of Extension and Contraction." IEEE Trans. on Software Engineering, vol. SE-5, no. 2, pp. 128-137.

Heninger, K. L., Kallander, J., Parnas, D. L., and Shore, J. E. 1978. Software Requirements for the A-7E Aircraft. Naval Research Laboratory Memorandum Report no. 3876. See Chapter 8, entitled "Required Subsets."

HIE.3 Uses Hierarchy for an Address System

EXERCISE

Name:_____

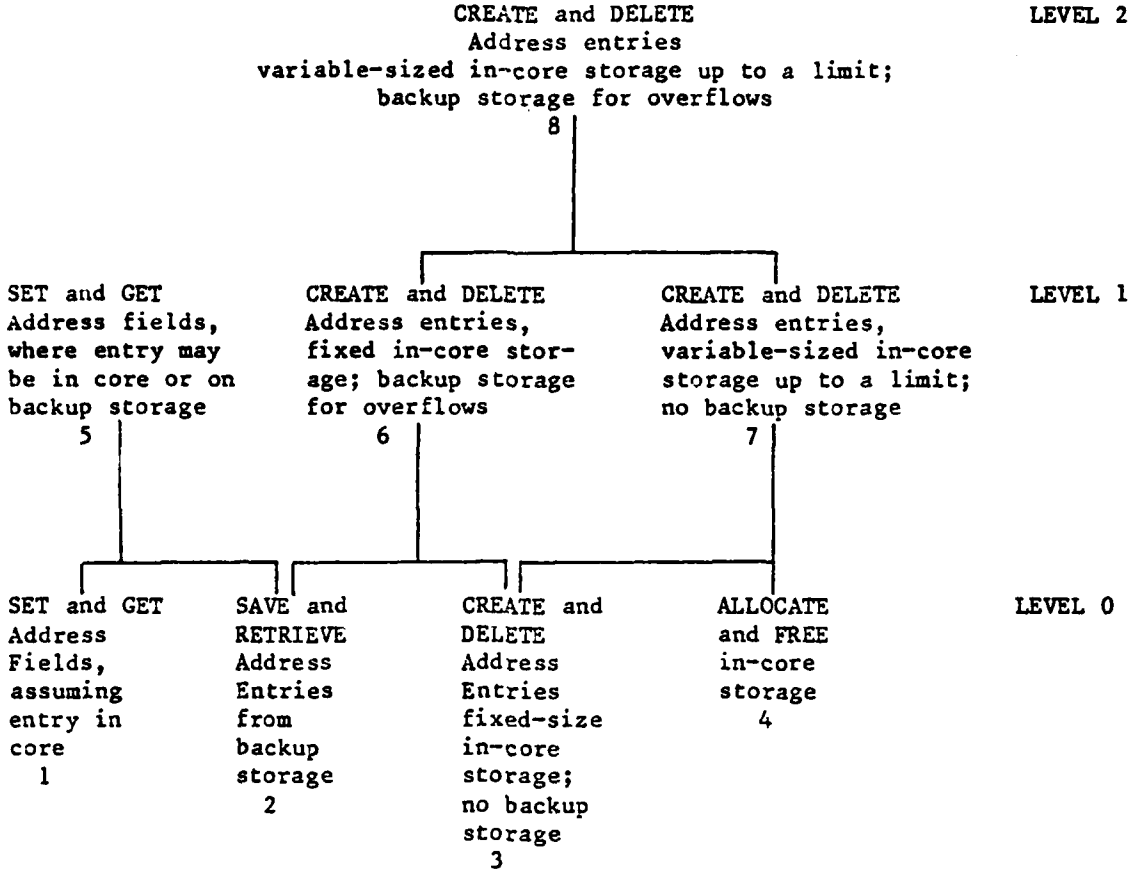
The diagram on the next page shows the uses hierarchy for a hypothetical family of address systems. By selecting components from this hierarchy, several different address systems can be constructed. These systems range from a system with a fixed, in-core memory allocation to the most flexible system, which can both vary its in-core allocation and swap entries in and out of back-up storage.

The components in the uses hierarchy are building blocks from which these systems can be made. GET and SET functions read and fill in fields of already existing address entries. CREATE and DELETE functions change the number of available address entries. SAVE and RETRIEVE swap entries in and out of back-up storage. ALLOCATE and FREE change the amount of in-core storage available to the address system.

Remember that programs on the higher levels use, rather than duplicate, the code in the lower level programs. For example, component 3 assumes that it has a fixed memory space, but the size is either a global or system-generation parameter. If higher level components of the uses hierarchy are not available and the memory allocation is already full when CREATE is called, an undesired event occurs. If this occurs in a system that includes component 7, component 7 uses component 4 to get additional space, then changes the parameter value, and then calls component 3 to create the new entry.

Note: We have combined the pairs GET/SET, SAVE/RETRIEVE, etc. into single components in order to make the example simpler. This is not necessary, since there may conceivably be systems in which only one member of the pair is needed. For example, if the address file is treated as a read-only data structure, GET functions will be needed but SET functions will not.

Study the diagram and answer the questions that follow.


CREATE and DELETE Address entries variable-sized in-core storage up to a limit; backup storage for overflows
8
LEVEL 2
SET and GET Address fields, where entry may be in core or on backup storage
5
CREATE and DELETE Address entries, fixed in-core storage; backup storage for overflows
6
CREATE and DELETE Address entries, variable-sized in-core storage up to a limit; no backup storage
7
LEVEL 1
SET and GET Address Fields, assuming entry in core
1
SAVE and RETRIEVE Address Entries from backup storage
2
CREATE and DELETE Address Entries fixed-size in-core storage; no backup storage
3
ALLOCATE and FREE in-core storage
4
LEVEL 0

Part 1: Subsets with Different Capabilities

For each of the systems described below, figure out which components are required. Circle or underline all the numbers below each description corresponding to required components for that system.

Example

An address system with a fixed number of entries, all in main storage.

(1) 2 3 4 5 6 7 8

System A: An address system with a varying number of entries, all in main storage. The table size, and therefore the amount of storage allocated to the system, is fixed at system initialization.

1 2 3 4 5 6 7 8

System B: An address system with a varying number of entries, all in main storage. Main storage space is dynamically allocated and freed as the number of entries changes. The amount of memory available to the system may not exceed some limit.

1 2 3 4 5 6 7 8

System C: An address system with a varying number of entries. The main storage allotment is fixed at system initialization, so that when it is full, overflow entries must be stored in backup storage.

1 2 3 4 5 6 7 8

System D: An address system with a large constant number of entries which do not all fit in the main storage allotment at once. The overflow entries are stored in backup storage.

1 2 3 4 5 6 7 8

System E: A very large capacity address system, with a varying number of entries. Main storage is dynamically allocated as the number of entries changes. When the number of entries exceeds a certain number, the overflow entries are stored in backup storage.

1 2 3 4 5 6 7 8

Part 2: Degraded Modes

Which of the above systems (A, B, C, D, or E) would still operate fully if the backup device went down?

Which of the above systems (A, B, C, D, or E) would still operate fully if a large area of core went down so that the program can no longer be allocated additional memory for address entries? (Assume the area occupied by the program and the current address file has not gone down.)

Which systems (A, B, C, D, or E) would still operate fully if both these UEs occurred?

HIE.4 Uses Hierarchy for an Address System

EXERCISE SOLUTION

In part 1, first pick all of the components at the highest levels needed for the specified system. At the very least, each system must include a way to read and write fields in the addresses. Therefore either component 5 or component 1 will be the highest-level component required from that branch of the hierarchy, depending on whether the system includes back-up storage. Which CREATE/DELETE component, if any, is the highest-level component required from the other part of the hierarchy depends on the flexibility requirements of the system.

Once you have selected the highest-level components, you must also include all the lower level components that they use, either directly or indirectly.

The solutions for systems A, B, C, D, and E are given on the next page.

Part 1: Subsets with Different Capabilities

System A: An address system with a varying number of entries, all in main storage. The size of the in-core storage, and therefore the table size, is fixed at system initialization.

Answer:

(1) 2 (3) 4 5 6 7 8

System B: An address system with a varying number of entries, all in main storage. Main storage space is dynamically allocated and freed as the number of entries changes.

Answer:

(1) 2 (3) (4) 5 6 (7) 8

System C: An address system with a varying number of entries. The main storage allotment is fixed, so that when it is full, overflow entries must be stored in backup storage.

Answer:

(1) (2) (3) 4 (5) (6) 7 8

System D: An address system with a large fixed number of entries which do not all fit in the main storage allotment at once. The overflow entries are stored in backup storage.

Answer:

(1) (2) 3 4 (5) 6 7 8

System E: A very large capacity address system, with a varying number of entries. Main storage is dynamically allocated as the number of entries changes. When the number of entries exceeds a certain number, the overflow entries are stored in backup storage.

Answer:

(1) (2) (3) (4) (5) (6) (7) (8)

Part 2: Degraded Modes

Which of the above systems would still operate fully if the backup device went down?

None of the systems that use component 2 could be included. Therefore only system A and system B could continue to operate.

Which of the above systems would still operate fully if a large area of core went down so that the program can no longer be allocated additional memory?

If a large area of core went down, the system is essentially restricted to a fixed area. Therefore only system A, system C, and system D would continue to work, since they do not use component 4.

Which systems would still operate fully if both these UEs occurred?

Only systems using neither component 2 nor component 4 would continue to work — system A.

Comment

In each of these UE situations, some capability is left, even though it may be very restricted. For example, if system E were in operation when the backup store went down, an appropriate UE response might be to continue operation using system B. System B could process the addresses that were in core when the UE occurred, and print a message whenever requested to access an address not in core.

Thus, this software allows for fail-soft operation when resources go down.

LANG.1 Language Selection

LECTURE

I. Introduction

A. Most of ideas discussed in this course are independent of language

B. But languages can help or hurt; the choice is significant

II. Four views of a programming language

A. A notation for describing classes of computations

B. A convenient way to instruct computers

C. "VIGILANTE"; an enforcer of rules of good practice

D. An efficient mechanism for invoking special, previously written programs

III. Four corresponding language evaluation criteria

A. How easy is it to tell which computations are possible (verification)?

B. How well can you control the machine?

C. How "structured" (restrictive) is it? Does it allow bad practices?

D. How "rich" is it?

IV. The four views and evaluation criteria conflict

A. Examples

B. Many choose one view and ignore all others

C. Choosing is reasonable for R&D (separation of concerns)

D. But system developers must resolve the conflicts

1. All views have some validity and should be weighed

2. Sometimes one can accomplish an objective with a non-language means

Example: Allow designers to determine coding restrictions

Example: Obtain an efficient library mechanism

Example: Put machine dependent "features" in library

V. System designers and developers have additional evaluation criteria

A. In what ways does the language help the designer?

1. Be free of surprises

2. Allow straightforward translation to efficient code

3. Help the designer enforce his information-access policies

4. Make it easy and efficient to use programs written by others

5. Make no assumptions about the desired response to run-time errors

6. Don't constrain implementation of parallel processes

7. Provide nonrestrictive looping structures

8. Help in confining assumptions and decisions

9. Facilitate user-defined data types and abstract data types

B. In what ways do the translator and related support help the designer/developer?

1. Give user-level diagnostics in terms of "write-time" structure

2. Debugging aids

3. Preprocessor systems

   a. Advantages

   b. Disadvantages

4. Library of useful programs

5. System management and integration tools

6. Manuals, texts, etc.

VI. What to do if no supportive language is available?

A. Use naming conventions in place of scope rules

B. Include run-time access-restriction code that can be removed later

C. Use the surprise-free subset

D. Use the efficient subset

E. Use the transportable subset

F. Postpone coding and debugging; spend more time on detailed design and evaluation

G. Use coding specifications

H. Write support software — see section V.B.

VII. General remarks about the selection process

A. Look at all of the code that will be in the system

B. List the implied design decisions explicitly

C. Beware of productivity arguments

D. How well is the language supported?

VIII. Conclusions

A. "Right language" is not possible, necessary, or sufficient

B. Some help more than others

C. Some hurt more than others. Language designers make assumptions on how their language will be used. Check the implicit assumptions.

D. Discipline in program design is what matters

IX. References

Parnas, D. L. 1971. "Information Distribution Aspects of Design Methodology." Proceed. of IFIP Congress 71, pp. 339-344.

Parnas, D. L. 1972. "On the Criteria to be Used in Decomposing Systems into Modules." Comm. ACM, vol. 15, no. 12, pp. 1053-1058.

Brinch Hansen, P. 1975. "The Programming Language CONCURRENT PASCAL." IEEE Trans. on Software Engineering, vol. 1, no. 2, pp. 199-207.

Parnas, D. L.; Shore, J. E.; and Elliot, W. D. 1975. On the Need for Fewer Restrictions in Changing Compile-Time Environments. Naval Research Laboratory Report no. 7847.

Linden, T. A. 1976. "The Use of Abstract Data Types to Simplify Program Modifications." Proceed. of Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Special Issue, vol. 11, pp. 12-23.

Parnas, D. L. 1976. "On the Design and Development of Program Families." IEEE Trans. on Software Engineering, vol. SE-2, no. 1, pp. 1-9.

Parnas, D. L.; Shore, J. E.; and Weiss, D. M. 1976. "Abstract Data Types Defined as Classes of Variables." Proceed. of Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Special Issue, vol. 11, pp. 149-154. Also Naval Research Laboratory Report no. 7998.

Parnas, D. L.; and Wuerges, H. 1976. "Response to Undesired Events in Software Systems." Proceed. of Second International Conf. on Software Engineering, pp. 437-446.

Dijkstra, E. W. 1977. A Discipline of Programming. Englewood Cliffs: Prentice Hall.

Wirth, N. 1977. "MODULA: A Language for Modular Multiprogramming." Software — Practice and Experience, vol. 7, no. 1, pp. 3-35.

Wirth, N. 1977. "The Use of MODULA." Software — Practice and Experience, vol. 7, no. 1, pp. 37-65.

Wirth, N. 1977. "Design and Implementation of MODULA." Software -- Practice and Experience, vol. 7, no. 1, pp. 67-84.

Wirth, N. 1977. "Towards a Discipline of Real-Time Programming." Comm. ACM, vol. 20, no. 8, pp. 577-583.

Liskov, B.; Snyder, A.; Atkinson, R.; and Schaffert, C. 1977. "Abstraction Mechanisms in CLU." Comm. ACM, vol. 20, no. 8, pp. 564-576.

Dahl, O. J.; Dijkstra, E. W.; and Hoare, C. A. R. 1972. Structured Programming. London: Academic Press.

Liskov, B.; and Zilles, S. 1974. "Programming with Abstract Data Types." SIGPLAN Notices, vol. 9, no. 4, pp. 50-59.

Elson, M. 1973. Concepts of Programming Languages. Chicago: Science Research Associates.

LANG.2 Ada

LECTURE

I. Ada history

A. Calendar of events

Strawman Requirements ..... June, 1975

Steelman Requirements ..... June, 1978

Preliminary Design Competition .... February, 1978

Selection of "Green" Language ..... April, 1979

Test & Evaluation of Design ..... November, 1979

Compiler Development Start ..... January, 1980

Language Reference Manual ..... July, 1980

MIL-STD-1815 ..... December, 1980

B. Pascal based (Jensen and Wirth 1974) -- many features added

1. Separation of specification and implementation

2. Multitasking

3. Machine-dependent coding

4. Representation specifications

5. Generics

Etc.

II. Ada basics

A. Programs as collections of Ada modules

B. Module organization

1. Specification

2. Body

C. Packages and tasks

1. Package = information-hiding module

2. Task = process

D. Package specification = information-hiding module interface

```
package Chm is

    -- The specification contains all package entities that the user
    -- has access to, including procedures, functions, types,
    -- variables, and constants.

    function CharEq (Ch1, Ch2 : Character) return Boolean;

    function CharLt (Ch1, Ch2 : Character) return Boolean;

end  Chm;
```

E. Types

1. A type defines value space and operations

   a. Numeric

      Integer, Real builtin

```
      type I is range 0..100;

      FieldLength: constant := 30;
```

b. Enumeration

Boolean, Character builtin

```
type Field is (Boc, Cit, Coa, Gn, Gsl, Sop, St, Tit, Zip);

type MaddsModules is (Ssm, Asm, Apm, Ipm, Opm, Idm, Odm,
                      Chm, Mcm, Ueh);
```

c. Array

```
type Table is array(1..10) of integer;

type Address is array(1..NumFields) of string(FieldLength);

type Module_Names is array(MaddsModules) of string(1..3) :=

    ("SSM", "ASM", "APM", "IPM", "OPM", "IDM", "ODM",
     "CHM", "MCM", "UEH");

Num_Errors: constant := 23;

type MsgLength is range 1..61;

type Msgs is array(1..Num_Errors) of string(MsgLength);
```

d. Record

```
type StatusValue is (Undefined, Defined);
type StatusArray is array(1..NumFields) of StatusValue;
type Addr_Form is
    record
      Status: StatusArray;
      Value: array(1..NumFields) of string(FieldLength);
    end record;
```

2. Distinguishing among types

a. Textual distinction

```
type Selector is (Boc, Cit, Zip);
type Field is (Boc, Cit, Zip);
```

b. Derived types

```
type Addr is new integer;
```

F. Variables and constants

```
MaxAds: constant integer := 100;
Addresses: array(1..MaxAds) of Addr_Form;
```

G. Procedures and functions

```
procedure SetBoc(A: Addr; S: string) is
    begin
        if A not in AddressNumber then Ueaida("ASM", "BOC");
        else
            Addresses(A).Status(1) := Defined;
            Addresses(A).Value(1) := S;
        end if
end SetBoc;
function GetBoc(A: Addr) return string is
    if A not in AddressNumber then Ueaida("ASM", "BOC");
    else
        return Addresses(A).Value(1);
    end if;
end GetBoc;
```

H. Example — the address storage module as an Ada package

```
package ASM is

    -- Specifications for procedures, types, variables, constants,
    -- etc. needed by users.

end ASM;

separate package body ASM is

    -- Implementation of procedures, types, variables, constants,
    -- etc. declared in package specification.

end ASM;
```

I. Tasks

1. Task specification defines communication and synchronization operations

2. The rendezvous

3. Specifying entry points

```
task semaphore is
    entry P;
    entry V;
end;
task body semaphore is
    begin
      loop
          accept P;
          accept V;
      end loop;
    end;
```

4. Entry calls and procedure calls

5. HAS location calculator as an Ada example

```
task buffer is
    entry withdraw(Data: out Item);
    entry deposit(Data: in Item);
end buffer;
task Location_Calculator;
task body Location_Calculator is
    -- Calculate location from an Omega reading
    temp, location: integer;
    procedure OmegaCalculation(Reading: in integer) is separate;
    begin
        loop
          -- Obtain Omega reading, calculate location, and
          -- deposit location in the location update buffer.
          Omobsbuf.withdraw(temp);
          location := OmegaCalculation(temp);
          Locupbuf.deposit(location);
        end loop;
end Location_Calculator;
```

6. Task proliferation

   a. One task per buffer monitor, one task per semaphore

b. cannot pass tasks as parameters when desired

III. Ada and software engineering

A. Information-hiding modules

1. Direct correspondence to packages

2. Peepholes into the interface

B. Abstract interfaces: Can be represented as package

C. Processes

1. Process representable as task

2. Process synchronization based on semaphores, task synchronization based on rendezvous

D. Undesired events

1. Sufficient freedom to define and access appropriate procedures and packages when necessary

2. Situations not requiring parameter passing can make use of exceptions

IV. Ada evaluation (LANG.1 criteria)

A. Help the designer

1. Moderately surprise free

2. Probably will not allow very efficient translation

3. Information access enforceable

4. May allow ease and efficiency of use of programs by others

5. Some assumptions about desired response to run-time errors are built-in

6. Parallel process implementation is constrained

7. Nonrestrictive loops are provided

8. Facilities for confining assumptions and decisions are provided

9. Facilities for user-defined types and abstract types provided moderately well

B. Translator and related support

???????

V. The Address Storage Module (ASM) as an Ada package

```
package ASM is

    -- The ASM consists of all address storage and retrieval routines.
    -- It  hides the representation used to store the addresses.  For
    -- each routine, the parameter A indicates the address whose field
    -- is being stored or retrieved.  GetNca returns the number of
    -- complete addresses.

    use SSM;                                -- need definition of strings

    MaxAds: constant integer := 100;        -- maximum allowable addresses

    type Addr is private;                   -- give users access to type Addr

    procedure InitAs;

    procedure VerAds;

    function GetNca return integer;

    function GetBoc(A: Addr) return string;

    procedure SetBoc(A: Addr; S: string);

    function GetCit(A: Addr) return string;

    procedure SetCit(A: Addr; S: string);

    function GetCoa(A: Addr) return string;

    procedure SetCoa(A: Addr; S: string);

    function GetGn(A: Addr) return string;

    procedure SetGn(A: Addr; S: string);

    function GetGsl(A: Addr) return string;

    procedure SetGsl(A: Addr; S: string);

    function GetLn(A: Addr) return string;

    procedure SetLn(A: Addr; S: string);

    function GetSer(A: Addr) return string;
```

```
        procedure SetSer(A: Addr; S: string);

        function GetSop(A: Addr) return string;

        procedure SetSop(A: Addr; S: string);

        function GetSt(A: Addr) return string;

        procedure SetSt(A: Addr; S: string);

        function GetTit(A: Addr) return string;

        procedure SetTit(A: Addr; S: string);

        function GetZip(A: Addr) return string;

        procedure SetZip(A: Addr; S: string);

        private
            type Addr is new integer;

    end ASM;


    package body ASM is separate;

    restricted(Main, SSM);

    separate package body ASM is

        use SSM, UEH;

        type Selector is (Boc, Cit, Coa, Gn, Gsl, Ln, Ser, Sop, St, Tit, Zip);

        -- Addresses are expected to have all fields either defined or
        -- undefined.  The type StatusArray provides an easy way to mark
        -- fields of an address as either defined or undefined.

        type StatusArray is array(Boc .. Zip) of (Undefined, Defined);

        -- The type Addr_Form provides the storage representation for
        -- addresses.

        type Addr_Form is

            record
                Status: StatusArray;
                Value: array(Boc .. Zip) of string;
            end record;
```

```
-- The variables All_Undefined and All_Defined provide convenient
-- arrays for finding out if addresses are either all defined or all
-- undefined.

All_Undefined: constant StatusArray := (Boc .. Zip =] Undefined);

All_Defined: constant StatusArray := (Boc ..Zip =] Defined);

type AddressNumber is range 1 .. MaxAds;

-- Variable Addresses is the array used to store addresses.

Addresses: array(AddressNumber'first .. AddressNumber'last);

Last: integer range 0 .. MaxAds;


procedure InitAs is

    begin
        Last := 0;
        for I in AddressNumber loop
            Addresses(I).Status := All_Undefined;
        end loop;

end InitAs;

procedure VerAds is

    begin
        Last := 0;
        for I in AddressNumber loop
            exit when Addresses(I).Status /= All_Defined;
            Last := I;
        end loop;
        for I in Last + 1 .. MaxAds loop
            if Addresses(I).Status /= All_Undefined then
                Ueasmi("ASM    ", "VERADS");
            end if;
        end loop;
end VerAds;
```

```
        function GetNca return integer is

            begin

                return Last;

        end GetNca;

        procedure SetBoc(A: Addr; S: string) is

            begin

                if A not in AddressNumber then Ueaida("ASM", "BOC");
                else
                    Addresses(A).Status(1) := Defined;
                    Addresses(A).Value(Boc) := S;
                end if

        end SetBoc;

        function GetBoc(A: Addr) return string is

            if A not in AddressNumber then Ueaida("ASM", "BOC");
            else
                return Addresses(A).Value(Boc);
            end if;

        end GetBoc;


    -- Other Set and Get function implementations are similar to SetBoc
    -- and GetBoc and are not included.

end ASM;
```

VI. References

Jensen, K.; and Wirth, N. 1974. Pascal User Manual and Report. 2nd ed. New York: Springer-Verlag.

ACM SIGPLAN. 1979. "Preliminary Ada Reference Manual." SIGPLAN Notices, vol. 14, no. 6, part A.

PROC.1 Process Structure of Software Systems

LECTURE

I. The problem: Imposing structure on run-time events

A. Examples of run-time events

1. Real-time

read value from angle of attack sensor
calculate new system velocities
output new heading value to display
aircraft becomes airborne
pilot keys in a number

2. Data processing

read new record from tape
extract key
print out a line
disk unit raises interrupt

B. Two ways to view events on a time-shared computer system

1. chaotic unrepeatable sequences

```
. . . read line typed on terminal by user A
fetch FORTRAN compiler into core for user B
compute value for user C
start compiling for user B
decode line typed by user A
output value computed for user C
respond to decoded command from user A . . .
```

2. Set of user jobs proceeding independently

A	B
. . . read line typed on terminal	. . . fetch Fortran compiler into core
decode line	start compiling . . .
respond to decoded command . . .	

C. Two ways to view a real-time system on a dedicated computer

1. First page from A-7 math flow

initialize navigation, if needed
calculate magnetic heading
calculate ground speed and total velocity from inertial north and east velocities
determine whether aircraft airborne, landbased, or seabased
determine if inertial platform ready and reliable
format horizontal velocity and total velocity for panel
output zero to ground track needle
if ground align just selected, zero panel clock and turn on light
compute true heading . . . .

2. Single train of thought

set scale for inertial platform accelerometer pulse	p. N-2
read in accelerometer pulses and calculate N and E vel	p. WD-2
calculate inertial groundspeed from N and E velocities	p. N-1
damp inertial groundspeed with system doppler groundspeed	p. N-12

D. Processes as subsets of events occurring in a system

1. In general purpose systems, each subset is a user's job

2. In real-time systems, determining best subsets is a major design problem

II. Three aspects of process design

A. Deciding on the right subsets (processes), i.e., grouping the events into processes — subject of this lecture

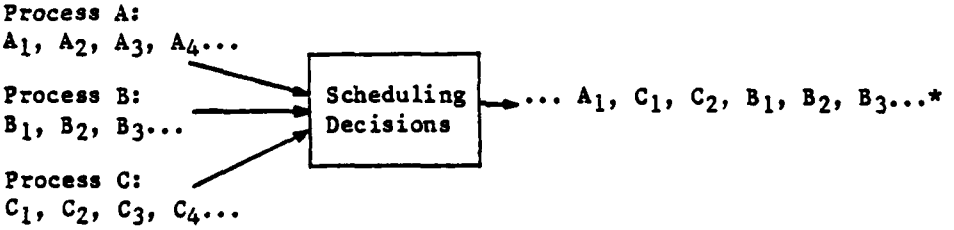
— What are the parts of the structure?

B. How subsets cooperate and communicate — subject of next lecture

— What is the relation between parts?

C. Determining actual run-time order, i.e., scheduling

Scheduling for Single Processor



*One of several acceptable orders.

When scheduling decision made: Alternatives

— Manually, by programmer

— Automatically, at system generation

— Automatically, at run time

III. Sequential processes

A. Operational definition: a unit for processor allocation — i.e., a unit competing for CPU time

B. Sequencing decisions made here

1. Sometimes order of events matters

Example: read value
smooth value

2. Sometimes order of events does not matter

Example: compute magnetic heading
determine whether aircraft airborne

3. When order matters, events belong in same process

4. Order of events in a process is always unambiguously determined

5. Order of events in different processes not well defined

   a. Processes executed on one processor: Interleaved execution

   b. Processes executed on several processors: Depends on speed of processors, allocation strategy, etc.

   c. Unpredictability of interrupts

C. Implications of definitions

1. Process executing on 0 or 1 processor at a time, never more

   — cannot be worked on simultaneously by more than one CPU

2. Two events in same process can never occur simultaneously

   — parallelism restricted

3. Speeds of processes unknown, i.e., time between events within a process unknown

   — rate of one process affected by other processes

IV. Advantages of a well-designed process structure

A. Each process makes sense by itself

B. "What" a process does is separated from "when" it does it

1. "What" is simpler: Can be done by less experienced programmers

2. "When" determined by scheduler: Major timing problems are isolated

C. Easier to make effective configuration changes

1. Each process can be written as if it runs on its own machine

2. Scheduler takes care of interleaving them on the available machines

3. Can add or remove processors without changing anything but scheduler

V. Rules for designing a good process structure

A. Initially divide system into maximum number of processes

1. Two events in the same process if they could never overlap in time, i.e., must occur in a particular order

2. Two events in different processes only if they could conceivably overlap or if the order is irrelevant

B. Decide on right granularity based on tradeoff between cost and benefit

1. Cost of smallest division
   — maintaining process records
   — creating and destroying short-lived processes
   — communication between processes

2. Benefits of smallest division
   — no potential parallelism ruled out
   — no potential configuration ruled out
   — programs extremely simple to understand

C. If granularity too fine, combine strongly related processes into one process

1. Still easy to understand

2. Arbitrary sequencing — rules out some parallelism

3. Reduces cost of process switching and communication

D. Separate out extremely time-critical events

Example:

read Angle of attack
filter sensor value and stale value to produce new value

— must read before filtering, but reading may get ahead
— first step time-critical, second not
— allow second step to get behind

VI. Example: A-7 process structure

A. One process for each requirements function

— as if each executed independently on own microprocessor

B. Processes to track mode transitions, history conditions

C. Processes to keep shared data up-to-date

— avoid duplication of effort

D. Processes to hide device characteristics

E. Sample A-7 process

```
program control_Magheading_display;

begin
comment this process executes periodically.  The desk clerk process
        signals the event "update time" whenever it is time for this
        process to run;
event update_time occurs freq times per second
while true do
begin
    wait(update_time);          comment process suspended until it
                                        receives signal to run;

    if magnetic heading sensor turned off
        then output 0 ;
        else begin
            read value from magnetic heading sensor;
            calculate magnetic heading from sensor reading;
            output magnetic heading;
            end;
    end-if;
end;
end-while;
end;
```

VII. Examples of poor process structure

A. Time cycle organization — events organized by how often they must occur

1. extremely sensitive to small changes

2. hard to follow

3. can be thought of as manual scheduling

B. Processes with internal scheduling

— Sign of design error if every time a process runs it must spend several instructions figuring out what to do next

VIII. Reminder: A system has many structures which do not have to coincide with each other

A. Module structure (early design-time)

B. Uses structure (late design-time)

C. Process structure (run-time)

IX. Research area, deterministic scheduling: Scheduling performed at system generation

A. Retain ease of comprehension

B. Avoid overhead of separate processes

X. References

Dijkstra, E. W. 1968. "Co-operating Sequential Processes." Programming Languages, ed. F. Genuys, New York: Academic Press, pp. 43-112.

Dijkstra, E. W. 1968. "The Structure of the 'T.H.E.' Multiprogramming System." Comm. ACM, vol. 11, no. 5, pp. 341-346.

PROC.2 MP Process Structure

EXERCISE

Name:_____

The original and alternative MP systems have different process structures. In the original MP structure, most modules are also processes. Most MP modules described in MP.3 are the units of processor allocation, and the executive module is the scheduler. In the new MP structure, the modules only provide operations that are executed by the processes at run-time; the modules themselves are not the units of processor allocation.

Pages 9-15 and 9-16 of this exercise illustrate the difference between the two process structures. Listed on page 9-15 are the events occurring in the message analysis module of the original MP structure. These events were taken from MP.2 and MP.3. Listed on page 9-16 are the events occurring in the incoming message process and outgoing message process in the new MP structure.

Evaluate the two different MP process structures, based on the considerations outlined in lecture PROC.1. In particular, consider the questions below. Be sure to give reasons or examples to support your opinions.

1. Which structure will have more inter-process communication overhead?

2. Which structure could take better advantage of a multiprocessor configuration with shared memory?

3. Which structure causes processes to spend more time figuring out what to do next?

PROCESS FROM THE ORIGINAL MP PROCESS STRUCTURE

In the original MP structure described in MP.2 and MP.3, there is one process for each module. Thus there is a process to analyze messages (MA), another to screen messages (SC), another to assemble a message to be transmitted (CO), etc. To illustrate these processes, we list below the events occurring in the message analysis module.

Review the common characteristics of MP modules in document MP.2, page 11-7. There are interrupts, but the RUNNING module is always resumed as soon as the interrupt housekeeping is completed. Since the message analysis module requires long processing time, it releases control of the processor after analyzing every six lines of the message, in order to allow other modules to proceed. When it does this, it sends itself a WCB to tell itself where to pick up. Whenever a process releases control of the processor, the executive must determine which process runs next.

Step	Message Analysis Module
1	get next WCB out of message analysis WCB queue
2	analyze WCB to determine a) which message to work on, and b) what to do next (whether to branch to Step 3, 5, 7, or 9)
3	for lines 1 through 6, analyze line, subtracting points for errors and correcting errors where possible
4	queue WCB to self and give up processor
5	for lines 7 through 12, analyze line, subtracting points for errors and correcting errors where possible
6	queue WCB to self and give up processor
7	for lines 13 through 16, analyze line, subtracting points for errors and correcting errors where possible
8	queue WCB to self and give up processor
9	if remaining points lt 80% of total points then message failed
10	prepare MDB for message
11	queue WCB to DC to store MDB
12	queue WCB to LM module to log message status
13	if message failed then a) queue WCB to DC to remove failed message from storage, and b) terminate process
14	queue WCB to DC to store message on disk
15	if incoming message then queue WCB to SC module, nocifying it message is ready to be screened
16	if outgoing message and channel available queue WCB to TO module, notifying it message is ready to be transmitted
17	wait until WCB queue not empty; then start over with step 1

PROCESSES FROM THE NEW MP PROCESS STRUCTURE

In the new MP structure, there is one process for each active message that is being worked on at a time. A single process may use programs from many different modules, and different processes may all use programs from the same module. The following informal descriptions of two of these processes show the sequence of operations and the programs they use from the MP.4 modules.

Note that these processes can be suspended during any step, either because they request unavailable resources or because a hardware interrupt such as a clock interrupt occurs. Unlike the original MP structure, an interrupted process may not necessarily resume immediately after the interrupt housekeeping is completed. The scheduler may choose to start another process instead.

Since the programs provided by the modules are reentrant, they can be used by more than one process simultaneously.

INCOMING MESSAGE PROCESS

Step		Using programs in:
1	Input a string	Communications & Equipment Control
2	Analyze string, storing it in the message holder	External Interface & Message Holder
3	Register message in log	Information Retrieval/Log
4	Check message against the watch list	Screening Module & Message Holder
5	IF any addressees in the message are in watch list, notify operator on terminal ELSE delete message from storage	Terminal Control Paging Module
6	TERMINATE	

OUTGOING MESSAGE PROCESSES

Step		Using programs in:
1	Help the operator create a message, storing the data he types in the message holder	Text Editor & Message Holder
2	Register message in log	Information Retrieval/Log
3	Transform message into the AUTONOYS format getting data for fields from message holder	External Interface & Message Holder
4	Transmit the message	Communications & Equipment Control
5	TERMINATE	

There might be other processes to retrieve and display messages, edit old messages, etc.

PROC.3 MP Process Structure

EXERCISE SOLUTION

1. Which structure will have more inter-process communication overhead?

   The original MP structure (MP.2)

   The original structure uses control block queuing and scheduling to switch between the activities associated with a single message, where the new structure uses subroutine calls. Control block queuing is considerably more time-consuming than parameter passing. The inter-process overhead must be paid in the original structure even if only one message is processed at a time.

2. Which structure could take better advantage of a multiprocessor configuration?

   The new MP structure (MP.4)

   Consider a multiprocessor configuration with several processes in the middle of message analysis and no other work to be done. In the original structure, only one processor would be used because there is only one MA process; the other messages would have to wait, and the other processor would be idle. Use of a single MA process introduces an artificial bottleneck into the system. With the new structure, each processor could be busy analyzing a different message, sharing the reentrant programs in the External Interface and Message Holder modules.

3. Which structure causes processes to spend more time figuring out what to do next?

   The original MP structure (MP.2)

   When a process in the original structure resumes running, it must analyze a Work Control Block (WCB) to determine what to do next. The decision can be arbitrarily complex: which message to work on, which step to do next, whether the message is in core, etc. In contrast, a process in the new MP structure can continue as if it had not been interrupted. What to do next is determined by the next instruction in the process.

   Note that the same work must be done in both structures to cause a process to resume running: the registers must be restored and the processor instruction counter must be loaded with the address of the next instruction in the process.

PROC.4 Process Synchronization

LECTURE

I. Introduction

A. System of cooperating sequential processes

B. Not totally independent

1. Different processes use same resources

Example: Line printer

2. Production and use of information may be in different processes

Example: One process polls sensor and puts data in a buffer
Another process uses this data to control output device

3. Detection and response to an event may be in different processes

Example: One process detects the event target designation
Four other processes respond:
two displays turned on
ballistics calculations started
radar sampling started

II. Three classes of synchronization problems

A. Mutual exclusion problem

1. Example: airline reservation system

```
(A) local_1:= numseats;
(B) if local_1 lt total_seats
       then numseats:= local_1 + 1;
       else refuse reservation
    end-if;
```

```
(X) local_2:= numseats;
(Y) if local_2 lt total_seats
       then numseats:= local_2 + 1;
       else refuse reservation
    end-if;
```

2. Character of solution: Explicitly prevent improper interleaving, i.e., restrict scheduling decisions

Correct Interleaving		Incorrect Interleaving			
A	X	A	A	X	X
B	Y	X	X	A	A
X	A	B	Y	Y	B
Y	B	Y	B	B	Y

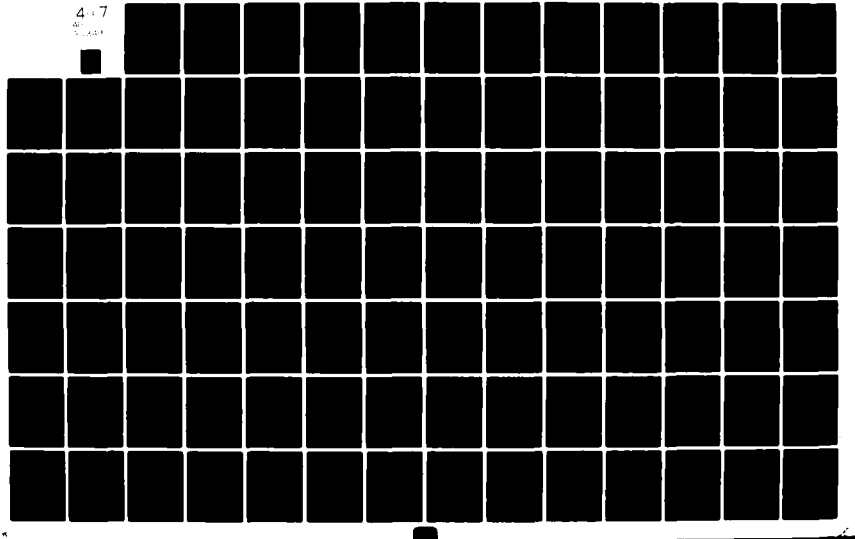
B. Reader-writer problem

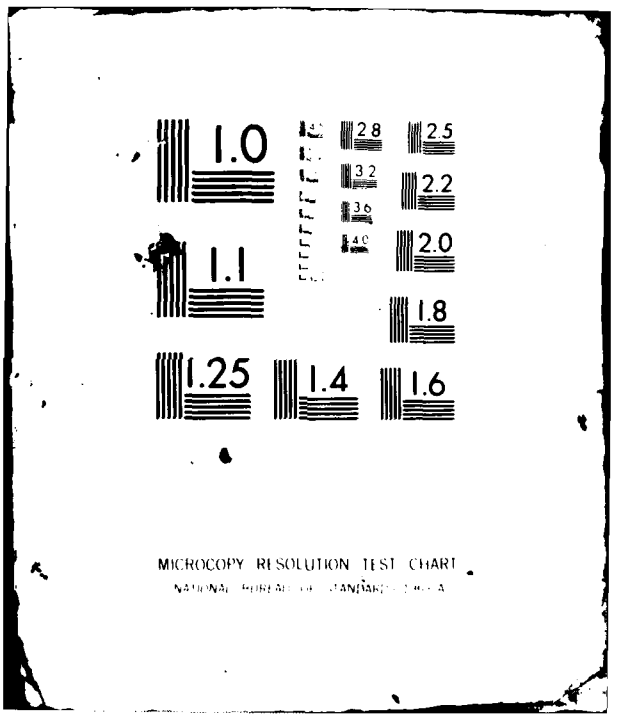
1. Example: Interactive data base
   Readers do not interfere with each other
   Writers must have exclusive access

AD-A113 415 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
SOFTWARE ENGINEERING PRINCIPLES 3-14 AUGUST 1981.(U)
AUG 81 L J CHMURA, P C CLEMENTS

UNCLASSIFIED NL

4 7




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

2. Character of solution: Less restrictive than mutual exclusion

C. Signalling problem

1. Example: The event "target designation"

```
wait (@T(!Desig!))              detect target designation
proceed                         signal (@T(!Desig!))
                                proceed
```

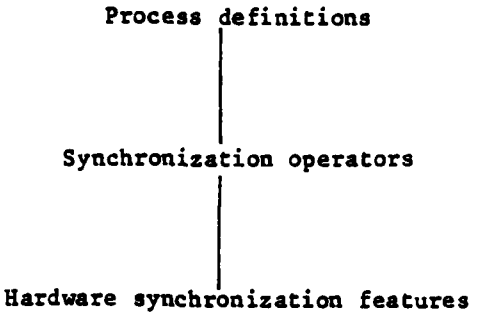
2. Character of solution: All processes that execute "wait" are suspended until another process executes "signal"; all waiting processes become ready simultaneously

III. The need for special synchronization operators

A. Synchronization problems difficult to solve — prone to subtle errors

B. Goal: To solve synchronization problems in a general way, rather than allow each programmer to solve them his own ad-hoc way

C. Uses hierarchy



1. Machine-dependent operations used for synchronization (e.g., disabling interrupts) — confined to the implementation of synchronization routines

2. Synchronization routines: Crucial code. Very carefully programmed and tested — must be correct and fast

D. Choice of right synchronization operations: Design problem

1. Several mentioned in literature

2. Choice may depend on

   — application

   — configuration

   — language

   — synchronization problems to be solved

IV. The rules of the game

A. No assumptions about the relative speeds of processes

1. Cannot solve synchronization problems by assuming

"This takes longer than that"

2. Train analogy: Why we need explicit synchronization

B. Minimize interrupt-disabled time

C. Avoid "busy form of waiting" -- waste of CPU and memory cycles

```
label:  if (busy = true) then go to label; end if;
        busy := true;
        . . .
        busy := false;
```

V. Synchronization operators change the set of processes eligible for scheduling

A. Process states:

Running — currently allocated a processor

Ready — eligible for scheduling

Waiting — not eligible for scheduling

B. Synchronization operator may cause a process to change state

VI. Example: Classic semaphore variables, with P and V operations — Dijkstra

A. Semaphore variable

1. Only accessed by P and V operations

2. Usually implemented as a counter and a list of waiting processes

B. P(semaphore) — "try" in Dutch

1. Process asks for permission to proceed

2. P-operation may affect state of process that calls it: Change state from "running" to "waiting"

3. When process resumes, there is no record of interruption

4. Example implementation for P(semaphore)

```
        begin
indivisible{   semaphore.ctr:= semaphore.ctr - 1;
           if semaphore.ctr lt 0 then
              begin
              process status changed to "waiting";
              process put in the waiting list for the semaphore;
              end;
           end-if;
        end;
```

C. V(semaphore) — "increase" in Dutch

1. Running process may change state of another process from "waiting" to "ready"

2. Example implementation for V(semaphore)

```
        begin
indivisible{   semaphore.ctr:= semaphore.ctr + 1;
           if semaphore.ctr le 0 then
              begin
              one process removed from waiting list;
              status of that process changed to ready;
              end;
           end-if;
        end;
```

VII. Solving synchronization problems with P and V

A. Mutual exclusion problem (initial value of mutex.ctr = 1)

```
begin                                   begin
   global semaphore mutex;                 global semaphore mutex;
   P(mutex);
   local_1:= numseats;
                                           P(mutex);
   if local_1 lt total_seats then
      numseats:=local_1+1; end-if;
   V(mutex);
end;
                                           local_2:= numseats;
                                           if local_2 lt total_seats then
                                           numseats:=local_2+1; end-if;
                                           V(mutex);
                                        end;
```

B. Signalling events (initial value of desig.ctr = 0)

```
begin                                   begin
   global semaphore desig;                 global semaphore desig;
   P(desig);
                                           detect target designation;
                                           V(desig);
   start radar sampling;
end;                                    end;
```

— only one process started

— not really suitable for broadcast

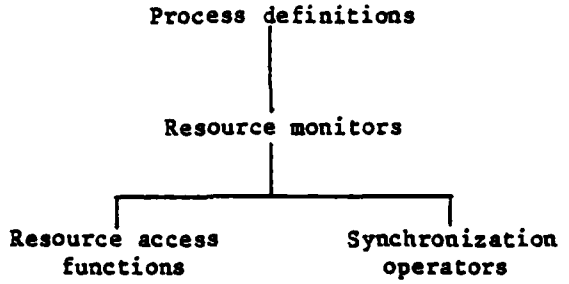
VIII. Coordinating access to resources using P and V

A. Monitors

1. Set of functions assuring resources accessed correctly, according to a particular set of rules

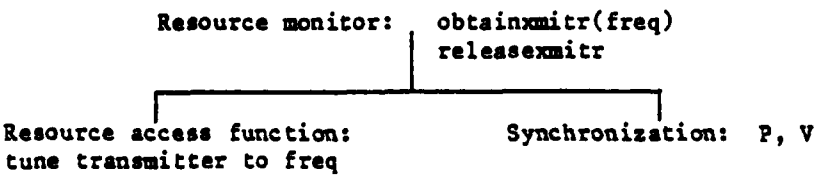
2. Resource can only be accessed through monitor functions

B. Uses hierarchy



C. Monitor for a transmitter (see HAS.4, p. 13-60)

1. Uses hierarchy



2. Rules: To avoid interleaved transmission, mutually exclusive access to a particular transmitter

3. Monitor: Only program that knows how many transmitters

4. Processes: Written as if each has own transmitter

5. If transmitters added, only monitor needs to change

D. Example: Monitor controlling access to a buffer

1. Initial condition -- buffer empty

   counter of semaphore "data" = 0 (no data available)

   counter of semaphore "space" = size-of-buffer
   (all spaces in buffer available)

2. No access to buffer allowed except through monitor programs "accept" and "deposit"

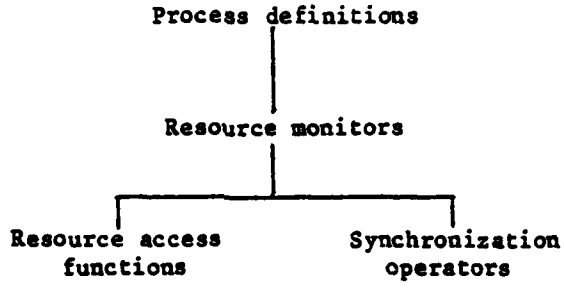
VIII. Coordinating access to resources using P and V

A. Monitors

1. Set of functions assuring resources accessed correctly, according to a particular set of rules

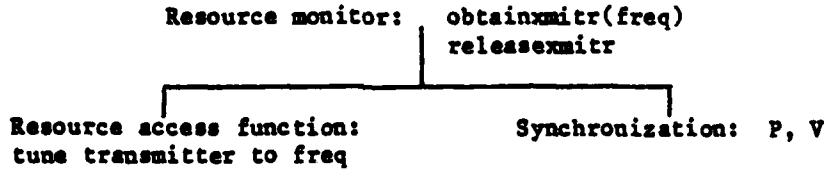
2. Resource can only be accessed through monitor functions

B. Uses hierarchy



C. Monitor for a transmitter (see HAS.4, p. 13-60)

1. Uses hierarchy



2. Rules: To avoid interleaved transmission, mutually exclusive access to a particular transmitter

3. Monitor: Only program that knows how many transmitters

4. Processes: Written as if each has own transmitter

5. If transmitters added, only monitor needs to change

D. Example: Monitor controlling access to a buffer

1. Initial condition -- buffer empty

   counter of semaphore "data" = 0 (no data available)

   counter of semaphore "space" = size-of-buffer
   (all spaces in buffer available)

2. No access to buffer allowed except through monitor programs "accept" and "deposit"

3. Rules implemented by monitor

   — Only one deposit at a time on a particular buffer

   — Only one accept at a time on a particular buffer

   — accept and deposit may occur simultaneously, so long as they are not operating on same buffer slot

   — deposit: process must wait if buffer full

   — accept: process must wait if buffer empty

4. Deposit: Function that puts an item in the buffer
   waits — if another process putting an item in the buffer
   waits — if buffer full

```
begin
    P(in);
    P(space);
    put item in buffer;          comment call buffer access fcn;
    V(data);
    V(in);
end;
```

5. Accept: Function that takes an item out of a buffer
   waits — if another process taking an item out of the buffer
   waits — if buffer empty

```
begin
    P(out);
    P(data);
    take item out of buffer;      comment call buffer access fcn;
    V(space);
    V(out);
end;
```

VIII. References

Courtois, P. J.; Heymans, F.; and Parnas, D. L. 1971. "Concurrent Control with 'Readers' and 'Writers.'" Comm. ACM, vol. 14, no. 10, pp. 667-668.

Habermann, A. N. 1972. "Synchronization of Communicating Processes." Comm. ACM, vol. 15, no. 3, pp. 171-176.

Cooprider, L. W.; Heymans, F.; Courtois, P. J., and Parnas, D. L. 1974. "Information Streams Sharing a Finite Buffer: Other Solutions." Information Processing Letters, vol. 3, no. 1, pp. 16-21.

Shaw, A. C. 1974. The Logical Design of Operating Systems, Chap. 3. Englewood Cliffs: Prentice Hall.

Parnas, D. L. 1975. "On the Solution to the Cigarette Smoker's Problem (Without Conditional Statements)." Comm. ACM, vol. 18, no. 3, pp. 181-183.

Reed, D. P.; and Kanodia, R. K. 1979. "Synchronization with Eventcounts and Sequences." Comm. ACM, vol. 22, no. 2, pp. 115-123.

Belpaire, G.; and Wilmotte, J. P. 1974. "A Semantic Approach to the Theory of Parallel Processes." International Computing Symposium 1973, A. Guenther, et al. (eds.). New York: N. Holland Publishing Co.

Britton, K. Heninger; and Weiss, D. 1981. Interface Specifications for the A-7E Extended Computer Module, Parallelism section. Naval Research Laboratory Memorandum Report in publication.

DOC.1 Documentation Guidelines

LECTURE

PART 1: GENERAL REMARKS

I. Why documentation is so important

A. Uses during development

1. Communication among designers, users, programmers, etc.

2. Training — makes personnel turnover less disruptive

3. Prevents duplication of effort — if reasons for design decisions recorded, reduces need to rethink them later

4. Basis for design reviews

5. Quality assurance — standard against which software can be judged

B. Uses during maintenance

1. Training

2. Reduces labor of evaluating feasibility of changes

3. Guides programmers as they find and correct errors

4. Repository of design information, which even the original programmers often forget

5. Preservation of program conceptual integrity — maintenance programmers have a way to check consistency of proposed change

II. Common problems with documentation — why is it hard to use?

A. Difficult to understand — assumes reader knows more than he does

B. Difficult to find answers to specific questions

C. Difficult to maintain -- *gets out-of-date all too soon*

D. Wordy, repetitive, and boring

E. Confusing, inconsistent terminology

III. Remedy

A. View documentation as the important product of design, not as a by-product of coding

B. Design the documentation -- objectives, contents, organization, format

1. To be a convenient format for designers to record and exchange ideas

2. To serve as ready reference tools

3. To be maintained -- controlled and kept up-to-date

4. To explain reasons for decisions since reasons cannot be inferred from code

C. General principles for documentation design

1. Determine objectives

- Who will need it?
- What should they already know?
- What should they be able to find out?

2. State questions before trying to answer them

3. Separate concerns

4. Documentation should consist of mutually supportive formal and informal parts

- Informal — easy for anyone to understand; useful for reviewers who are not programmers
- Formal — precise, concise, unambiguous

5. Involve maintainers early — to find out what they need

D. Documentation design techniques

1. Decide what information belongs

2. List questions to be answered

3. Organize into sections according to who needs to know answers, for what purpose

4. Determine units of discourse

5. Design forms, tables, notation, templates

   - Use English only for overviews, narratives, and explanations

   - Use abstract Programs (otherwise known as PDL or coding specifications) for documenting algorithms

6. Plan to revise forms several times as documentation is written

7. Define terms precisely; provide glossary

E. Design documentation reviews and configuration control procedures

1. Design reviews: What questions should reviewers ask themselves to determine if document meets its objectives?

2. Configuration control procedures

- How are changes reported?
- Who decides whether to make them?
- Who reviews them?
- How are updates distributed? To whom?
- What tools are needed? — word processing support invaluable

IV. Three types of documentation

A. Software requirements specification (e.g., Program Performance Specification)

1. Product of overall system design — represents agreement among

- User representatives
- Builders of interfacing equipment or software
- Software builders

2. Questions answered

   - What role does the software play in the whole system?

   - What constraints are placed on the software?

3. Reference document for software designer and programmers, i.e., overall problem statement

4. Guidebook for maintenance programmers

   - Constraints on future improvements

   - Conceptual integrity of software

B. Overall design document (e.g., Program Design Specification)

1. Agreement between designer and programmers about system structure

2. Questions answered

   - How is the software divided into modules?

   - How do the modules work togetner to meet the overall requirements?

   - Specifications for the module interfaces

   - Overall system tradeoffs

3. Reference document for programmers, i.e., their individual problem statements

4. Guidebook for maintenance programmers — where to make changes

C. Detailed design document (e.g., Program Description Document)

1. Program-by-program description

2. Questions answered

- What algorithms and data structures were selected? Why?

- What program implementation tradeoffs were made?

3. First product from programmers, i.e., the algorithms they choose and why

4. Guidebook for maintenance programmers — how to make changes

5. Appropriate place for Abstract Programs or PDL

PART 2: AN EXAMPLE OF CAREFUL REQUIREMENTS SPECIFICATION

I. Background: A-7 Project

A. Purposes

1. To evaluate usefulness of modern software technology for real-time systems with tight constraints

2. To provide an engineering model

B. Basis: Existing flight software for Navy's A-7 aircraft

C. First step: Document requirements of existing system

1. Implementation-independent description of current system

2. Problem statement for NRL A-7 project

II. Documentation objectives

A. Specify external behavior only -- everything one needs to know to design and program the software

- If less, software may not fill purpose in larger system

- If more, software personnel constrained unnecessarily — may not be able to use best approach

B. Specify constraints on implementation

- e.g., timing, accuracy, algorithms, and response time, etc.

C. Be easy to change

D. Serve as a reference tool for experienced designers and maintainers

E. Specify expected changes to software

F. Specify desired responses to undesired events

III. Does NOT contain programs, data structures, flowcharts

IV. Table of contents

V. Hardware interface documentation

A. Unit of discourse: Data item

B. Design standard forms

C. Describe input data items as resources -- no mention of how used

D. Describe output data items in terms of effects on hardware -- no mention of purpose

E. Formal notation for data items

1. Bracketed names

inputs:	/RADALT/,	/IMSMODE/
outputs:	//STERROR//,	//HUDSCUE//
values:	On,	Off

2. Expressions

comparison	/RADALT/ lseq 3000 ft
change value	//HUDSCUE// := Off

F. Example of hardware interface description

Input Data Item: IMS Mode Switch

Acronym: /IMSMODE/

Hardware: Inertial Measurement Set

Description: /IMSMODE/ indicates the position of a six-position rotary switch on the IMS control panel.

Characteristics of Values

Value Encoding:		
	$Offnone$	(00000)
	$Gndal$	(10000)
	$Norm$	(01000)
	$Iner$	(00100)
	$Grid$	(00010)
	$Mags1$	(00001)

Instruction Sequence: READ 24 (Channel 0)

Data Representation: Bits 3-7

Comments: /IMSMODE/ = $Offnone$ when the switch is between two positions.

VI. Software function interface

A. Unit of discourse: Function

B. Distinguish periodic and demand functions

1. Periodic functions: Occur at regular time intervals

2. Demand functions: Occur in response to specific events

C. Output values based on conditions, events, and modes

1. Conditions as predicates

2. Events as changes in condition values

3. Modes as classes of system states

D. Notation

1. Text macros: !Ground range to target!

2. Conditions: /IMSMODE/=$Gndal$

3. Events: @T(/IMSMODE/=$Gndal$)
@F(!Ground range to target! = 30 nmi)

4. Modes *DIG*

E. Example of a special table

Condition Table: Magnetic heading (//MAGHDGH//) output values		
MODES	CONDITIONS	
DIG, *DI*, *I* *Mag sl*,*Grid*	Always	X
IMS fail	(NOT /IMSMODE/=$Offnone$)	/IMSMODE/=$Offnone$
//MAGHDGH// value	angle defined by /MAGHCOS/ and /MAGHSIN/	0 (North)

F. Example of function description

Periodic function name: Update Flight Path Marker coordinates

Modes in which function required:
DIG, *DI*, *I*, *Mag Sl*, *Grid*, *IMS fail*

Output Data Items: //FPMAZ//, //FPMEL//

Initiation and Termination Events:
Start: @T(//HUDVEL// = On)
Stop: @T(//HUDVEL// = Off)

The flight path marker shows the direction of the aircraft velocity vector. The azimuth displacement from HUD center shows the lateral velocity component and elevation displacement shows vertical velocity component.

If the components are calculated from !System velocities!, !System velocities! are first resolved into forward, lateral and vertical components, that is, components along the aircraft Y, X, and Z axes. From these, the HUD coordinates are calculated in the following manner:

//FPMAZ// shows $\frac{\text{Lateral velocity}}{\text{Forward velocity}}$ //FPMEL// shows $\frac{\text{Vertical velocity}}{\text{Forward velocity}}$

Condition Table: Coordinates of the Flight Path Marker

MODES	CONDITIONS		
DIG, *DI*	X	Always	X
I	/ACAIRB/ = No	/ACAIRB/ = Yes	X
Mag sl, *Grid*	/ACAIRB/=No	!ADC Up! AND /ACAIRB/=Yes	!ADC Down! AND /ACAIRB/=Yes
IMS fail	/ACAIRB/=No	X	/ACAIRB/=Yes
FPM COORDINATES	//FPMAZ//:= 0 //FPMEL//:= 0	based on !System velocities!	//FPMAZ//:= 0 //FPMEL//:=/AOA/

VII. References

Heninger, K. L.; Kallander, J.; Parnas, D. L.; and Shore, J. E. 1978. Software Requirements for the A-7E Aircraft. NRL Memorandum Report no. 3876. See readers' guide in preface.

Heninger, K. L. 1980. "Specifying Software Requirements for Complex Systems: New Techniques and their Application." Trans. on Software Engineering, vol. SE-6, no. 1, pp. 2-13.

MIL-STD-1679. 1978. Weapon System Software Development.

Parker, R. A., Heninger, K., Parnas, D. L.; and Shore, J. 1980. Abstract Interface Specifications for the A-7E Device Interface Module. Naval Research Laboratory Memorandum Report no. 4385.

MP.1 The UGH Message Processing (MP) System

EXAMPLE DESCRIPTION

Fapsan Rat
Cognizant Engineer
UGH Corporation

Introduction

The UGH Message Processing System (MP) may use a variety of UGH computers from the large 2PIE to the small UGH-20 to provide an integrated support system for any organization requiring rapid and widespread distribution of messages to organizational units in geographically distributed locations. The UGH MP is designed to assist in every stage of message distribution beginning with the input of the draft message into the system, including automatic control of the communications equipment, the production of periodic reports about the status of the system and the messages that it has processed, and including (optional) an interactive information retrieval facility to allow managers to check on the status of the system or any message that has been submitted to it in the recent past. The UGH MP is highly modularly structured and can be tailored to meet the needs of any organization. As a result, its adaptability to changing needs is assured.

Interfaces and Functions

The UGH MP is designed to interface with and utilize the services of the world-wide AUTONOYS communications network. This network includes direct RF ship-to-shore communications, satellite relayed communications, and high capacity overland channels and is continually being expanded to include the most modern communications techniques. AUTONOYS is an existing communications network which has evolved over the years, starting from a fully manual system using very noisy (error-prone) communications channels, but has been adapted to take advantage of computer control as well as improved communications equipment. Because some of the low-traffic nodes on the AUTONOYS network are still manually controlled and some of the channels are still quite unreliable, all changes to the original AUTONOYS communications conventions have been made upwards-compatible. This has resulted in a rather complex communications protocol. The MP is designed to produce messages in any of the AUTONOYS formats and is designed to be adaptable to the new formats which are expected to be introduced as AUTONOYS is improved.

It is expected that the organizations which UGH MP will serve will each have their own internal message conventions and guidelines. The MP adjustable user interface is designed to assist an operator with converting internal messages to external messages. MP can even assist with internal

message handling by producing the copies of outgoing messages to be sent to various "in house" addressees for information and retention. MP will automatically put addresses on these copies.

A special feature of MP is its ability to interface with UGHTRANS® computer-controlled communications equipment. This highly sophisticated equipment is designed to eliminate the need for a radio operator except for routine maintenance and emergency repairs. The "normal" functions of the operator such as antenna selection, tuning, connecting transmitter to antennae, etc., are all performed by electronically controlled switches and servomechanisms programmable in UGH hardware. The UGH MP software includes an option that will examine the "routing indicators" and select the appropriate communications setup.

An additional feature of MP is its ability to adjust to the security and privacy needs of its users and its ability to make use of the special AUTONOYS message security conventions. AUTONOYS offers a variety of communications channels ranging from "broadcast" (which is easily intercepted) to highly secure narrow beam communications. AUTONOYS communications conventions include highly redundant security codes designed to minimize the probability of a sensitive message being transmitted over an inappropriate channel. MP provides the necessary software and formatting to take advantage of these features. It also checks all input carefully to make sure that the security classifications are legitimate and consistent.

When used in connection with the broadcast channels of AUTONOYS, MP will receive many messages that are not intended for its user organizations. MP can "screen" these messages and select only those which its users are interested in. To prevent lost messages caused by incorrect screening, typing errors, or transmission errors, MP will (if requested) produce a list of rejected messages for operator review.

One of the special features of MP in this regard is its dynamic watch list. Every incoming message has a list of addressees. To screen these incoming messages, MP uses a list of "addressees of interest" or a "watch list." The message is selected if one of the addressees is on the watch list. MP allows the operator to alter this list so that messages for guests may also be received. This is also useful if one unit must temporarily support or replace another and must receive its messages.

The AUTONOYS system requires that each addressee be further identified by a routing designator which allows AUTONOYS to select the intermediate relay stations to be used. This has been a major source of costly errors and lost messages in manual systems. An error in one character of this routing indicator can cause a message to reach an unintended destination. In some organizations, where the addressees are "mobile" (e.g., ships or traveling salesmen), routing indicators must be updated frequently, which results in still more possibilities for making errors. MP automatically supplies the routing indicators for outgoing messages, using an internal routing indicator list. An optional feature is its ability to monitor incoming messages and

report any inconsistencies between the routing indicators on these messages and its own list to the operator. This feature leads to greatly increased reliability if the routing designators change frequently.

Increased communications reliability is in fact one of the major advantages of using MP. Checking for errors and inconsistencies is performed at every stage of message processing. MP really "knows" the AUTONOYS communications conventions and checks more carefully than any human operator would. An incorrectly formatted message, a message with inconsistent descriptors, WILL NOT BE ISSUED. Many transmission errors on incoming channels will be detected. Such messages will be brought to the operator's attention so that the appropriate corrective measures may be taken. It is known to be mathematically impossible to detect all errors in messages over noisy channels, but checking is so widespread in the MP software that the probability of error is reduced to a lower point than with any alternative system.

Naturally MP supports a variety of input devices ranging from basic teletypes and paper tape readers to the most modern of character-oriented graphics consoles. An interactive option provides the most modern prompting and computer support of message input. With this option, the operator will be "prompted" for each item of format information which AUTONOYS requires. He can't forget anything and need not constantly refer to the AUTONOYS operator manuals. This can greatly increase the productivity of the operator and make him feel "supported."

Another option available is called the remote message drafting option (RMD). This allows the individual responsible for composing the text of the message to do so at a terminal with the aid of an advanced text editor. Individuals authorized to release a drafted message are given passwords and/or key controlled terminals so that they can authorize release without the existence of a hard copy which is transmitted to the operator with signature. As soon as release is obtained, the text is already in the system and can be composed and transmitted almost instantaneously. Possible errors in entering the text are eliminated by this option.

UGH MP FAMILIES

The UGH MP software is actually a "family" of systems, formed by the inclusion of optional features (some of which are mentioned above). The hardware support available is also a family, in two senses. First, the permanently located "base" versions of MP are designed to run on the UGH 2PIE computer family, which provides a wide range of upwards-compatible processors that share peripherals. MP requires a certain minimum hardware configuration depending on the supported options, but it will run on any of the 2PIE range, although larger processors are recommended for the base systems to realize the best MP performance. The 2PIE range was not designed for rugged mobile installations, and since MP is of extreme value in this setting, the system is also supported on the large UGH-7 (UGH-VAN) computer and on the smaller UGH-20 minicomputer. Although the full software system is available on UGH-7, the

incorporation of some features of the software family is of doubtful value. Only a subset of features is available on UGH-20. (Neither of these hardware systems is compatible with UGH 2PIE hardware.)

There is a complex interaction between the hardware required to support a given UGH MP feature, other features necessarily included with it, and the capabilities of the resulting system. Some examples will make this interaction clear.

UGH MP can be configured with optical character readers (OCR) for input text. Of course, the software support for this feature is useless without the OCR peripheral device itself. This device is available only on the 2PIE series as a standard option. (Although offered as a field-modification item for UGH-7, no such modifications exist at the present time.) Furthermore, the OCR is used by the system only in connection with the operator's console station. It is assumed that the operator will use the OCR for the input of authorized text which he receives in hard-copy form. This usage therefore alters the remote message drafting option, if it is selected. The hard-copy message is scanned only for comparison with the internal file copy that was created earlier with RMD, and the operator advised of any inconsistencies. Since some machine printing facility is required for OCR, it is likely that the remote message drafting options should be specified whenever OCR is specified.

The test generation and transmission option allows the complete hardware-software system, along with the crucial data stored within it, to be automatically given a real test and its performance evaluated. A message is generated, and addressed to the same unit which originates it. Using the complete facilities of the system with regard to the watch list, automatic routing of internal messages, etc., the message is sent and (if all is well) received. However, the message is flagged so that upon receipt, a complete description of the system performance is created and made available to the operator. In conjunction with another MP system, this testing can actually involve two or more distinct units, but the external system can also be tested by routing an internal message over an arbitrary route and checking it specially when it arrives in the specified way. MP creates such test messages on command without any operator control except the request to perform the test. The selection of this option evidently requires the complete UGHTRANS® hardware interface, and the additional features of automatic selection of communications setup from routing indicators, and monitoring of incoming routing for consistency. Although this enabling capability is available on UGH-20, the test option is not recommended for this small machine because it would degrade other functions.

Although the following table is not complete, it does include all MP options mentioned in this brief description, and gives an idea of the relationship between the hardware and software families according to the capabilities selected. As an example of the use of the table, capability "A", the information retrieval option, requires at least an UGH-VAN hardware system with extended mass storage ("Z") and memory ("Y"), and cannot be selected without also selecting the message retention capability ("B").

Table 1. UGH MP HARDWARE AND SOFTWARE FAMILIES

	Capability	UGH-2PIE	UGH-VAN	UGH-20	Hardware and Other Capabilities
A	Information retrieval	*	*		ZYB
B	Message retention(1)	*	*	*	Z
C	Automatic selection of communications	*	*	*	X
D	Verify incoming routing indicators	*	*	*	
E	Operator prompting	*	*		
F	Remote message drafting	*	*		
G	OCR input	*	*(2)		WAF
H	Test generation and transmission	*	*	*	XABCD

Hardware codes:

Z Extended mass storage (large disk)
Y Extended memory
X UGHTRANS interface
W OCR peripherals

Notes:

(1) Retention period is 1, 3, or 6 months. Only the first is available for UGH-20, and the longer periods require further mass storage extensions.

(2) Requires a field modification to install OCR peripherals.

DELIVERY

UGH MP will be delivered 2-1/2 years after receipt of the first signed purchase contract. Later versions will be available with shorter delivery times.

MP.2 MP Basic Modular Structure

EXAMPLE DESCRIPTION

Introduction

The UGH MP is designed to be a highly modular piece of software that can be easily adapted to meet the needs of individual users as well as to changing AUTONOYS conventions.

To meet the needs for adaptability, the system is divided into a number of modules each of which has a precisely defined task. Communication between modules has been limited to well-defined data structures stored on disk and a sophisticated intermodule communication facility implemented as part of a system "kernel."

This document is designed to provide an overview of the basic system structure together with a brief introduction to the function of each module. Detailed functional specifications of each module as well as interface definitions will be provided in separate documents.

Common Characteristics of Modules and Intermodule Communication

Each module is designed to perform a given step in the processing of a message. Each one is designed to function independently of the others, starting its processing of a message and carrying this processing through to completion. Work is sent to a module through the system kernel in the form of a Work Control Block (WCB). Modules are called READY when they have one or more WCB's waiting for them. At any one time, one module at most is RUNNING. The RUNNING module is allowed to execute until it releases control because it has completed its work or for some other reason (see below). There may be brief periods of interruption of the RUNNING module to handle machine interrupts, but the RUNNING module is always resumed and is not cognizant of the pause. Some modules that require long processing times will relinquish the processor before finishing with a message in order to allow other messages to be processed. In such cases, the module sends itself a WCB before releasing the processor. This WCB contains the information necessary to allow the module to pick up where it left off.

The other form of communication between modules is through data kept on disk. For example, the text of the message being processed and a Message Description Block (MDB) containing the primary information about the message are stored on disk. The DC module will bring these data into core for processing when it receives a request (by means of a WCB).

Overview of the System Modules

The following are the main software components of the system.

1. The System Executive (EX)

The EX controls the scheduling and communication between the remaining modules of the system. When one module wishes to communicate with another, it does so by preparing a WCB for the other module. It does this in a predefined core location. It then calls the EX routine Send Work Control Block (SWB), which queues the WCB against the recipient. If the recipient's queue was previously empty, it marks the recipient module READY and enters it in the queue of modules waiting for scheduling. EX is also called whenever a module signals completion. If the module still has WCBs in its input queue, it is placed in the list of waiting modules. If not, it is marked NOT READY. EX then determines which module will run next.

2. Message Analysis (MA)

The primary task of MA is to examine the raw message text (incoming and outgoing) identifying the principal message components and detecting format errors in the message. During this process, key information is extracted from the message and stored in the MDB. At the same time, a record of message received and channel is made. MA is also responsible for recording the status of channels and will not release a message to a channel that cannot handle it.

3. Screening Module (SC)

Primary purpose of SC is the detection of messages of interest. To this end, SC examines the list of message addressees and compares it with the WATCH LIST (made available by Data Control Module (DC)). SC also adds routing indicators to outgoing messages and can check incoming message routing indicators. Messages that are not of interest are sent by means of a WCB to the terminal control module (TC). Accepted messages result in a WCB being sent to the Log Maintenance Module (LM).

4. Message Composition Module (CO)

This module is responsible for assembly of the complete message as it is to be transmitted. Information is received from the other modules (e.g., routing indicators from screening) and the completed message is then sent to MA where it is checked as if it were an incoming message. The optional prompting package is part of this module if purchased. CO provides text editing facilities allowing modification of text previously input. These facilities keep control until the operator indicates that he is done editing and asks for composition of the final message.

5. Terminal Control (TC)

TC is responsible for all direct communication with consoles, including teletype, printers, and CRT devices. Terminal Control neither interprets the input nor decides what to output — it is simply a standard interface between the other modules and the terminal.

6. Equipment Control (EC)

This module is responsible for control of the UGHTRANS devices. It receives its WCBs from OP.

7. Operator Control (OP)

This module implements the interfaces to the system control officer. It allows him, for example, to request reports, change circuit connections, verify connections, request tuning, and define available frequencies. This module implements a user-oriented interface with mnemonic commands, but relays all commands to other modules for execution.

8. Traffic Output (TO)

Traffic output is in control of a message during its actual transmission. It retrieves portions of the message from disk and causes them to be sent to the I/O devices. It verifies that no line contains more than 69 characters.

9. Data Control (DC)

This module is responsible for the maintenance of various lists of data used by other modules. For example, DC finds and allocates disk storage for the WATCH LIST, MDBs, and routing indicators. START and END addresses for the working disk storage areas of other modules are also kept by this module. DC does not issue control commands to the disk, but simply sends requests to DK (in terms of track and sector) where the actual I/O commands are generated.

10. Disk Control (DK)

All requests to use the disk are queued as WCBs for DK, thus making sure that the disk is not requested to carry out two access requests at once.

11. LOG Maintenance (LM)

This module is responsible for updating all lists used in the information retrieval and report generation functions. Space is allocated by DC and actual data transfers by call to DK. Among the logs kept are:

a. list of all messages received;
b. list per channel of all messages received;
c. list of messages originated;
d. list of messages sent; and
e. list of messages rejected (optional).

All entries in these logs are complete MDBs that include the address of the full text on disk.

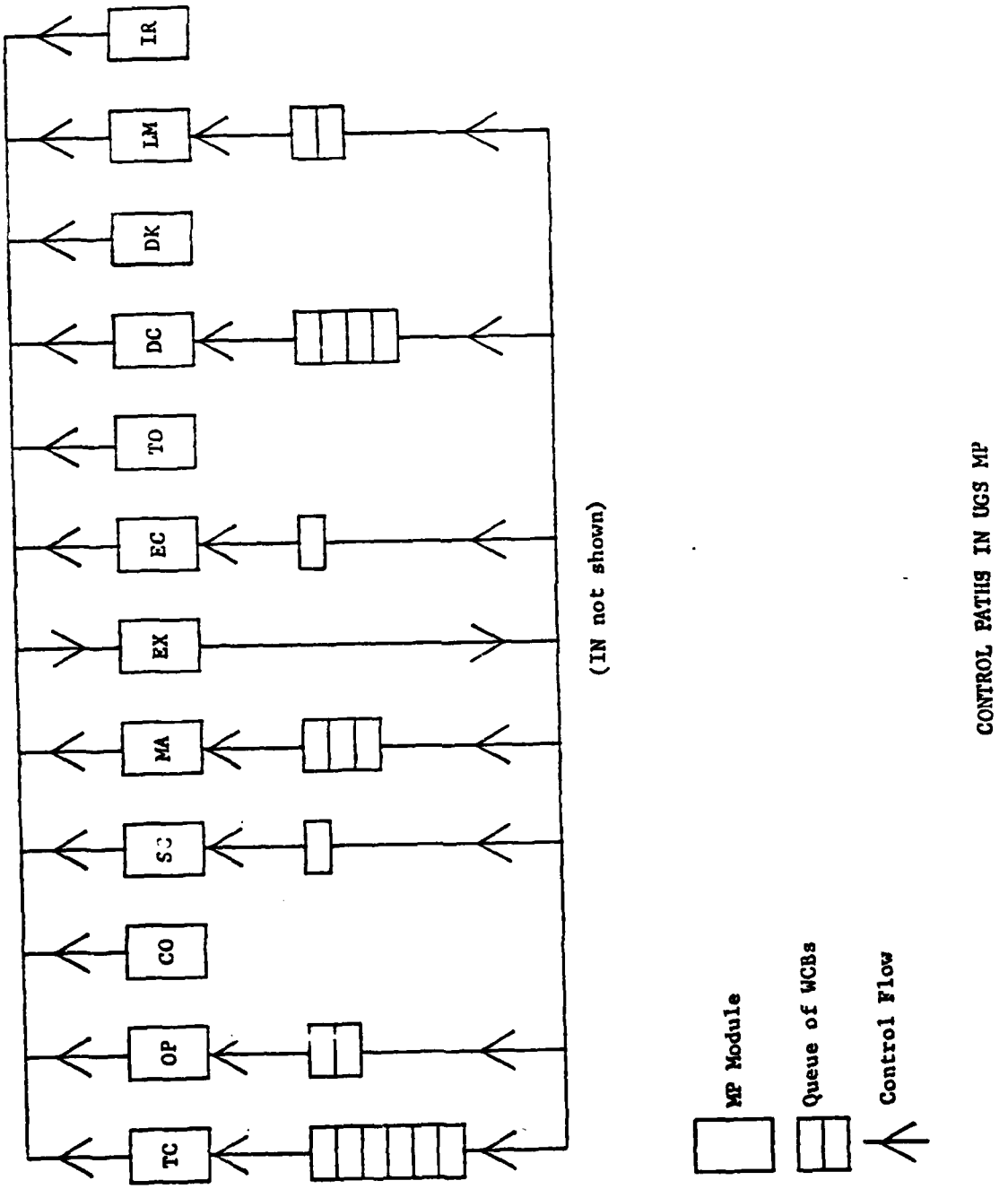
12. Initialization (IN)

This module performs all actions needed at system start. It is normally only called at that time.

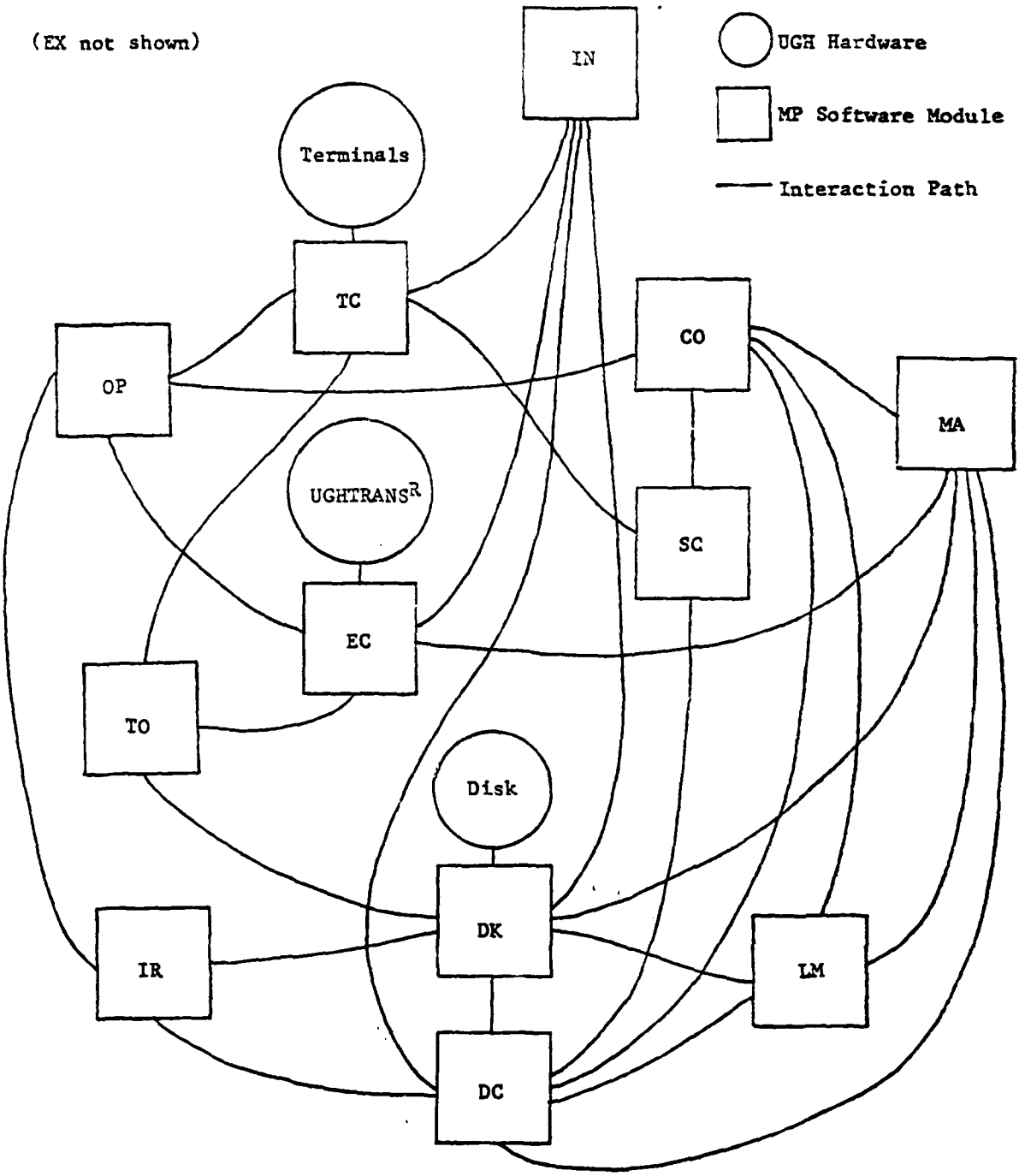
13. Information Retrieval Module (IR)

This module allows officials to obtain information such as:

"*What message came in on channel 3 at 1500?*"
"Was message _____ originated here?"
"Was message _____ transmitted?"



CONTROL PATHS IN UGS MP



BLOCK DIAGRAM OF UGH MP MODULES

MP.3 MP Detailed Modular Structure

EXAMPLE DESCRIPTION

Functional descriptions are attached for some of the primary modules of the UGH MP system. Only the "message-processing" group MA, SC, CO, and the "executive group" EX, DK, DC are included. For other modules the capsule descriptions of MP.2 are sufficient. Following the module descriptions are details of the Work Control Block by which modules communicate with each other. A useful summary of module functions is given below:

The Modules of MP: A Summary

CO: Message Composition — Editing and assembly of messages to be transmitted.
DC: Data Control — Storage allocatic for data, both core and disk.
DK: Disk Control — Controls disk access.
EC: Equipment Control — Controls UGHTRANS devices.
EX: Executive — Handles scheduling, intermodule communication, and interrupts.
IN: Initialization — Initializes system at start.
IR: Information Retrieval — Retrieves information of interest from logs.
LM: Log Maintenance — Maintains logs of MDBs.
MA: Message Analysis — Analyzes potential messages to see if they are real messages.
OP: Operator Control — Handles operator interface.
SC: Screening — Examines incoming traffic for messages of interest.
TC: Terminal Control — Handles communication with system consoles.
TO: Traffic Output — Controls transmission of messages.

MESSAGE FORMAT FOR MP

The description below supplies enough information to understand the actions of MP modules that deal with the headers of messages. It ignores some message features (such as multiple pages), and simplifies some others.

A message consists of a number of Format Lines numbered beginning with one. (These are abbreviated FL1, FL2, etc.) In specifying these, we will use a notation drawn from computer language and control-card manuals.

Uppercase letters represent themselves, and where given, must appear exactly as written. Where information is to be supplied, a lowercase name will appear, to be explained subsequently. Where items are optional, they are enclosed in square brackets; where a choice of items is permitted, these are shown one above the other. The spaces shown are nonrepresentative: the characters begin in the first column and continue to the end of the format line without spacing unless explicit spaces are indicated by the symbol b. Each line ends with a sequence of two-carriage-returns-and-a-line-feed, not

shown. When an item is superscripted, it is repeated that many times; superscript ^{n} means an indefinite repeat (but at least once).

FL1:

VZCZC origin-route-part channel

where "origin-route-part" is a two-letter part of the originating routing code (the 3rd-last and 2nd-last letters of the code), and "channel" is a three-digit channel number.

FL2:

precedence origin-media dest-media class content-action b
sender-orig-route serial b date time b class^{4} routing

where "precedence" is a single letter from a standard list, "origin-media" and "dest-media" each 1-letter language media codes from a standard list, "class" is the security classification letter from a standard list, "content-action" is a four-letter identifier from a standard list, "sender-orig-route" is the seven-letter routing indicator of the sender, "serial" is a four-digit number supplied by the sender, "date" is the three-digit Julian date and "time" the four-digit GCT at which the message was received for transmission, and "routing" is the seven-letter routing indicator for the addressee.

FL3:

DE b sender-orig-route serial date time b year

FL4:

ZNR b class^{5} T [routing]
ZNY

FL5:

precedence b date time Z b JAN ... DEC b year b

where "year" is a two-digit value, e.g., 76 for the bicentennial year.

FL6:

FM b origin

where "origin" may be a routing indicator "sender-orig-route" or may be in plain text.

FL7:

TO b [routing / addressee , n]n.

where "addressee" is the plain text corresponding to the routing indicator it follows. (In the final addressee item, the period replaces the comma, and similarly in FL8, 9.)

FL8:

[INFO b [routing / addressee,]n .]

FL9:

[XMT b [routing / addressee,]n .]

FL11:

BT

FL12:

class subj-code text

where "subj-code" is a six-character code composed of the letter N and five digits, surrounded by double slashes, and "text" is the message text.

FL13:

BT

FL15:

serial

FL16:

null lf^7 NNNN

where "null" is an empty line (but with the usual ending), "lf" is a line-feed.

A sample message in this format:

```
VZCZCDB003
RTTUZYUW  RUCLDBA2355  1861200  UUUURUHHLFA
DE  RUCLDBA23551861200 76
ZNR  UUUUU
R  1861200Z  JUL  76
FM  COMNAVTELCOM WASHINGTON DC
TO  RUHHLFA/ALCOM
BT
U//N09999//
HAPPY BIRTHDAY
BT
#2355
(8 blank lines)
NNNN
```

THE MESSAGE ANALYSIS MODULE (MA)

FUNCTION

The function of the MA module is to analyze a message to make sure that it is a message in the sense of the AUTONOYS standard message syntax rules. Before an incoming message is processed by the rest of the UGH MP system, MA is called upon to make certain that the string being processed is indeed a complete message and not a message fragment or some portion of more than one message. All of the remaining modules in UGH MP may proceed on the assumption that the text that they are processing is indeed a message and they need not perform error checking. To increase reliability, MA is also used to check outgoing messages. MA performs exactly the same analysis on an outgoing message that it performs on incoming messages. This provides an additional check that the other modules and the operator have done their work properly. If MA finds that a message does not conform to the AUTONOYS conventions, it does not release it for transmission.

Non-messages are rejected by MA and then discarded from the system, unless the message retention option with rejected-message log is included.

METHOD

The basic approach is to identify the components (format lines) of a message, thereby making sure that all required components are present and that each contains the information required. The possibility of transmission errors on AUTONOYS channels, together with the redundant nature of the AUTONOYS message conventions, makes it inadvisable and unnecessary to demand that a message be perfect. The MA module uses a sophisticated point system to determine the acceptability of messages. Every time that a message passes a certain requirement, it receives a certain number of points. Potential messages which receive at least 80% of the required points are considered to be messages and are passed on for further processing. MA may also improve a message by making certain obvious corrections.

The message is first checked to determine that it contains Format Line 1, which is required in all messages. In a perfect message, FL1 begins with "VZCZC". MA first checks for the first character being a "V". If the first character is not a "V", it checks to see that the second character is a "Z"; if so, the checking continues, but if the first character is not "V" and the second charcter is not "Z", MA checks to see if the second character is a "V" or the first character is a "Z", which would mean that either the first character of a message was lost or the second character of the message was the actual first character. If either of those conditions are met, the characters in the message are renumbered; otherwise, the checking proceeds as if the first character really was the first character but it is incorrect. In a similar way, MA then goes on to check for the presence of "C", "Z", and "C". Because these characters repeat themselves in this code, no check for misalignment is made. If all five characters are present where expected, the

message is given five points. If by renumbering the characters one can find at least four of the five characters on the proper positions, then four points are given. If only three of the characters can be found, two points are given; in all other cases zero points are computed.

The next component of FL1 should be the 5th and 6th letter of the originator's routing designator. There are only 42 possible combinations in the list of AUTONOYS routing designators, so a check is made to see if the two characters found are on that list of 42 combinations. If so, the message is given two points; if not, it is given no points for this test. The last three characters of FL1 are a three-digit channel designator. A check is performed to make certain that the three digits found here designate one of the channels being used by the system. If so, two points are given. If all the characters are digits but the number is not a possible channel designator, one point is given. If a non-digit is found, no points are given. The remainder of FL1 is spaces.

A search is next made for the beginning of FL2. This must be the message precedence. There are six possible AUTONOYS precedence codes. If the first non-blank character after the end of FL1 is one of these six characters, that position is assumed to be the start of FL2 and two points are given. If the first non-blank character is not a legal precedence code, a search is made for the next non-blank character. If that is a legal precedence code, then it is assumed that FL2 begins at that point (the message is corrected), and one point is credited. If not, FL2 is assumed to be missing and a search is made for the start of FL3. One point is subtracted from the score of the message if this occurs. The next two characters after the precedence are media indicators. These are not checked. The third character, however, is a security class indicator. If this indicates one of the five allowed security classes, three points are given. If not, the two neighboring positions are checked to see if they could be a security class. A match in either position results in one point for the message, and message correction. Four spaces after the security class, a routing indicator for the message originator is expected. This is seven characters long and begins with "R". If the "R" is not present in the expected position, a check is made for the previous or following character being an "R". If so, it is assumed that the routing indicator has been found. Seven points are given for the routing indicator being found where expected, five points if it is found one position off. A check is now made to make sure that characters five and six are on the list of 42 possible 5th and 6th characters for routing indicators. If so, three points accrue. If not, a check is made to see whether the 6th and 7th characters of FL1 passed the test. If so, they are substituted for the corresponding characters at this point. If these characters passed the test, but were not the same as those in FL1, one point is subtracted from the score and the FL1 characters are substituted. The next four characters are a serial number provided by the sending station. If these are all numeric, two points are given. The following three characters must be a Julian date. A possible Julian date receives two points, and today's date one more point. The next four characters represent "time filed"; if all are numeric and a possible time, two points are given. The following four characters must be the

security code repeated four times. If the same character is present four times, and it is the same as the security found on position four of FL2, 10 points are given. If the same character is present four times, but it is not a legitimate security code, then two points are given and the FL2 code is substituted. If it is present four times, is a legitimate code, but is not the same as that found earlier, then (a) seven points are given, and (b) the earlier security code is replaced. If the character is only present three times, but it is a legitimate code, then six points are given. If this character is not the same as that found earlier, the earlier one is replaced and one point is substracted from the score. The next seven characters are intended to be the addressee's routing indicator and must begin with "R". If it is correct, 10 points are given. If the "R" is incorrect, but the remainder is correct, then nine points are given. If the "R" is present, but the remaining code is incorrect in one or more positions, eight points minus the number of incorrect positions are accumulated.

Format Line three is identified by the string "DE", followed by a space, followed by the routing indicator of the originator. If this can be found, the message is given 10 points. If a routing indicator can be found, but it is not that which was found earlier, then seven points are given. If the routing indicator is found, but the "DE" is missing, then six points are given. The next four characters must be the sender's serial number again. If this is found and matches that found earlier, then five points are given. If four numeric characters are found but they do not match those found in FL2, then three points are given. If any non-numeric characters are found, then no points are given. The next characters must be a repeat of the Julian date. A legal date which matches that found earlier brings seven points. A legal date which does not match that found earlier brings four points. The following four characters must be a filing time. If they are all numeric, then three points are credited; if they are not numeric, no points are given.

Format Line four must begin with "ZNR" or "ZNY". If this is found, eight points are credited. If it is found with one or two errors, four points are credited. A search is then made for the classification repeated five times. If the previously determined classification is found five times, 10 points are credited. If it can be found three or four times, five points are credited. If a legitimate code is found five times, but it is not the same as that determined earlier, then five points are credited and the higher classification is used. The other occurrences of the security code are replaced by this higher classification.

MA continues in this fashion until the entire message has been processed.

After determining whether or not the message passes the tests (by obtaining at least 80% of the possible points), WCBs are prepared and sent to other modules. A WCB is sent to the screening module. If this is an outgoing message, a WCB is sent to the TO module. If the message has failed, WCBs are sent to DC to remove the message from the system. In all cases, WCBs are sent to the LM module to record the disposition of the message. A WCB is then sent to the DC module requesting that it allocate disk space to store the corrected

message text. (DC will eventually do so and by a WCB will cause the DK module to transfer the message to disk. DK will, on completion of this transfer, send a WCB to DC which can then release the core space for the storage of an incoming future message.) Before terminating itself, MA prepares an MDB for the message. This is allocated space by DC and stored by DK. The disk address of the MDB for a message is always given in fields 15-17 of a WCB.

THE SCREENING MODULE (SC)

FUNCTION

Once a string of characters, whether incoming or outgoing, has been accepted by MA as an acceptable message, it must be screened to see if it is intended for a recipient served by the installation. Outgoing messages must be screened as well as incoming messages because messages may have multiple addressees and some of those addressees may be at a location served by the installation. The message text, as stored on the disk, is searched for the addressees, and when they are found, a list is prepared in core. This is then compared with the WATCH LIST obtained from the DC module. SC produces an internal routing list, which is the intersection of the two lists (addressees and WATCH). If this intersection is empty, and the message is incoming, SC takes no action. Otherwise, WCBs causing further processing of the message are prepared.

METHOD

Since the addressees are to be found in Format Line seven and Format Line eight, the first step is to find the starting location of Format Line seven. This is done by proceeding stepwise through the message. The start of FL1 is identified by the string "VZCZC". If this string cannot be found anywhere in the first 20 characters, it is assumed to have been destroyed by noise. To assist future modules in their processing, SC corrects the start of the message by inserting VZCZC. The end of FL1 is identified by the three digit channel number. The start of FL2 is identified by searching for the precedence code. The precedence code can be found in field 9 (Byte 27) of the WCB. As a further check that FL2 has been found, the classification code is checked for two characters after the supposed precedence code. The end of FL2 is identified by searching for the four occurrences of the classification code and then the routing indicator. When these are found, it is assumed that we are at the end of FL2 and the search for "DE" which indicates the start of FL3, is begun. Having found this "DE", the end of FL2 is signaled by the occurrence of seven digits in a row. FL4 is identified as the string "ZBR" or "ZNY", followed by five occurrences of the security class. Because FL5 is so short, we search for the end of it as indicated by the clear-text month and date. FL6 must be clearly and certainly identified since the information that we are looking for begins in FL7. FL6 contains "FM" followed by a seven letter routing indicator beginning with "R", or the originator in plain text. The next non-blank character is assumed to be the start of FL7.

The start of FL7 is marked by a "TO", followed by a list of addressees separated by commas. Each addressee consists of a routing code, followed by a "/", followed by the identifier of the addressee. The program searches for the "/", then writes in its working list all characters until it finds a ",". It then searches for the occurrence of either a "/" or a ".". A "." indicates the end of the TO list and FL7. The start of FL8 is indicated by the string "INFO". The addressees which are listed after this are listed in the same

format. Hence, the same algorithm is applied, search for the "/", copy the addressee until "," or "." . The SC module handles information and action (to) addressees identically.

After composing the list of addressees, SC sends a WCB requesting that DC bring the WATCH LIST into its core area. It then terminates until it receives a WCB, which informs it that the WATCH LIST is now in core. It then proceeds to search for each of the addressees in the WATCH LIST. Each one that is found is copied into a list. If this list is empty, then processing of incoming messages stops, unless rejected message retention or the routing indicator check option has been selected. For outgoing messages (and for incoming messages, if one of these options has been chosen), the module next requests that the routing indicator directory be brought into core. This is done by means of a WCB to DC. The routing indicator directory contains a routing indicator for all addressees of interest to the installation. Each addressee is looked up in this directory, and the routing indicator is compared with that found in the message. If it is different, the discrepancy is reported to the operator who is given the opportunity to correct either the message or the directory. An additional option allows the SC module to simply add the routing indicator found in the directory to an outgoing message on the assumption that the routing indicator in the directory is the correct one.

THE MESSAGE COMPOSITION MODULE (CO)

FUNCTION

The Message Composition Module (CO) is one of the most significant new features and innovations of MP. It is designed to take most of the drudgery out of AUTONOYS communication. The characters in an AUTONOYS message can be divided into three categories:

(1) Format characters (e.g., "VZCZC", "TO"), which are present in every message and serve primarily to clearly identify message components.

(2) Redundant characters (e.g., nine repetitions of classification code, second insertion of station serial number, etc.)

(3) Information characters: characters such as the message text, the addressees, etc., which could not be deduced from the remaining text.

The purpose of the CO module is to spare the operator the work of typing in anything but "real" information characters. The format characters are automatically supplied by the CO module; the redundant characters are inserted in the proper portions of the text as soon as the first piece of information has been supplied. For example, once the message classification has been input to the system, it can be inserted in the text automatically wherever the AUTONOYS conventions require it. Even the routing indicators are redundant information; CO requests the routing indicator directory and adds this information to the message as soon as the addressee has been named.

METHOD

OP sends CO a WCB indicating that the operator is ready to input a message. CO's first action is to send a WCB to DC requesting that core space for message composition be supplied. Upon receipt of the WCB from DC, which indicates where this core area is, CO initializes the message by writing the characters "VZCZC" at the start of the core area. It also writes the 5th and 6th characters of the installation's own routing designator in core. It then sends a WCB to the OP module requesting that the operator be prompted to state the channel designator. The response WCB from OP should contain the three-digit channel. If this is not on the list of possible channels, the operator is prompted again (through OP, of course). A proper channel completes Format Line 1 (FL1).

Next, a WCB is sent to OP which requests OP to prompt the operator for message precedence. When this information is received, it is stored in the MDB as well as inserted in the message being composed. Next, the operator is prompted for a media code and this one byte code (either 'T', 'C', 'P', or 'Q') is inserted twice in the message. Similarly, the operator supplies the content-action code and the classification code. A direct request to EX obtains a four-digit station serial number, which is then inserted in the

message text as well as the MDB. The next step is to prompt the operator to supply Julian date and filing time. This information is also checked to make certain that it is feasible (possible date, time less than 2400, etc.), and then inserted in the message text as well as the MDB. CO then goes on to insert the message classification code four times in the text. The operator is then prompted for the name of the first addressee. When this is received, a WCB is sent to DC to bring the routing directory into a specified core area. The addressee is looked up in the routing directory, and the routing indicator (seven characters) is inserted in the text as well as stored in the MDB.

CO now is able to supply FL3 completely automatically, since it consists of format information (DE), the originator's routing indicator (constant for the system and already inserted in FL2), the station serial number, filing date and time, which can be obtained from the MDB, having been supplied earlier, and year. To obtain FL4, the operator is prompted for a three-alphabetic-character transmission (which must be "ZNR"), and then the classification is inserted five times. FL5 is composed of the precedence, date, and time once more, and then the month and year in clear text together following a "Z". This is all supplied on the basis of previously stored information (once a month the system must be reloaded with a new month and year). FL6 is also supplied without bothering the operator since it consists of the format information "FM" together with the name of the originating station and/or its routing indicator.

CO continues in a similar way using information stored inside the system together with a knowledge of the format to produce the message with the minimum amount of operator intervention. Text editing facilities are provided for the message text itself, but not for the fixed format information.

When the message has been completely composed, the operator is prompted for a release number. When this is supplied and verified, a WCB is sent to MA to check the message. Release by MA will result in a WCB to TO which will control the actual transmission. WCBs are also sent to DC (to release storage areas no longer needed), to LM (to make the appropriate journal entries), and to SC, which checks the message to see if there are internal addressees.

EXECUTIVE GROUP (EX, DK, DC)

These modules (along with TC and EC, not described here) are responsible for control of UGH hardware devices. The primary device is the UGH processor, which EX schedules; DK and DC manage disk operations and allocation of disk and core.

THE SYSTEM EXECUTIVE (EX)

FUNCTION

The System Executive Module (EX) is central to the UGH MP system, yet itself takes only a supporting role in that system. EX arranges for the proper functioning of the modules that actually carry out the MP tasks. Several MP resources are controlled by EX and dispensed as needed. Among the EX resources, the UGH memory and logical and arithmetic processing units are primary, and all others are of secondary importance. Examples of secondary resources are the system clock and other centrally stored values such as the date, and unique sequence numbers. In controlling the UGH processors, EX also has control of hardware interrupts, and may itself perform a small amount of processing as the initial part of interrupt service. An important special class of interrupt service is response to software-initiated interrupts from other modules requesting service.

METHOD

Nothing happens within the UGH MP system until an interrupt occurs. Then EX takes control, and deals with the situation directly, or arranges that it will be handled by another module, which EX schedules with a proper Work Control Block (WCB).

The simplest kind of interrupt is a request from a running module. In each such case, a code is provided to describe the necessary action. Simple requests (such as for the time, or a unique sequence number) are handled and dismissed immediately. The requesting module resumes as if it had merely called a subroutine. Of course, the effect of the request often extends beyond the requesting module; for example, the sequence number is updated. More complex requests result in EX passing work to another module (not EX itself). This is accomplished by generating an appropriate WCB, queuing it against the necessary module, and returning to the requesting module. If the requestor must await the completion of the other module's work, it must request termination (an immediate EX service) following return from sending the WCB; when the other module is done, it must send another WCB to restart the original requestor.

The MP modules operate as independent processes, but they differ from arbitrary processes in a general-purpose operating system in that EX has full information on each one and can predict the resource needs of each. This information makes many of EX's tasks easier to perform. Further, preemption of the processor is usually difficult, but EX never preempts a running process.

EX uses a first-come-first-served (FCFS) queuing scheme, with emergency override. The queue order is determined by the list of pending WCB's, which are kept in the order of request. When one module requests work from another, a new WCB is appended to the end of this list. When a module terminates, the first WCB in the list is examined, and if the needed module is in core, it

assumes control. If not, the queues of pending WCBs are examined, and the core resident module which will be invoked last is removed from core. There is one exception to FCFS queueing. A module may request "priority" service, and have a WCB placed at the very head of the queue.

THE DISK CONTROL MODULE (DK)

FUNCTION

Whenever a module requires the use of UGH mass storage (typically disk packs, although this is dependent on the configuration), it sends a WCB to the disk control module (DK), specifying the track and sector, the length (in sectors), and read/write. DK does not check the address for validity for the requesting module because addresses are obtained from the data control module which verifies them (DC is the primary source of work for DK in any case). Two modes of DK operation are available. First, the request can be an I/O and wait. The operation is started, and EX terminates the requesting module. When the completion is signaled to DK, it sends a WCB to the requestor to continue. Second, the request can merely start the disk operation by sending a WCB to DK, but the requesting module continues to be READY. Later, the module may ask EX if the request is yet finished and itself take appropriate action. Use of the second kind of operation realizes a considerable saving when a module can start its requests ahead of time or has other work to do while the request is honored.

METHOD

DK uses the "scan" technique to manage multiple requests and minimize *positioning contention*. While *one* request is in progress on the disk, a number more may be queued. DK accepts their WCBs and forms a list of the needed addresses in order of arm position. It then freezes this list (additional arriving requests start a new one) and makes a "scan" across the disk servicing the requests as their addresses pass under the heads. This operation may reorder the requests slightly, but DK takes care to treat multiple requests from the same module in the order they arrive.

THE DATA CONTROL MODULE (DC)

FUNCTION

The Data Control Module (DC) has two related functions. First, it allocates space on the disk for all modules. In this, it is something like part of a general-purpose file system, although the spaces are not named permanently. To obtain space, a module requests the size needed, and DC returns the address of such a block. Deallocation of the space requires another request. Similarly, DC allocates core memory for data. (EX allocates core for module code.)

The second DC function is also similar to part of a file system: DC knows about certain data sets which are important to all phases of MP operations, and aids modules in accessing these data sets. For example, the WATCH LIST is a permanent part of any MP installation, and DC allocates the space for it. But DC also allows modules to use the WATCH LIST without knowing its format or disk address, as if it were a sort of "file" with variable blocking. A module can thus simply "read" WATCH by sending a WCB to DC, and receive a buffer of data in response. In a sense, each message within the system is a "file" of this kind, since DC will access it for a module from its MDB.

METHOD

Space allocation on disk is done in variable-size segments that are a multiple of a minimum unit. DC controls the available space by keeping a list of addresses of the beginning and size of all in-use space. When a request is made for space DC returns the first block (or part thereof) which is large enough to meet the request. Although this operation can cause disk "checker-boarding," the transient requests for space are not a significant fraction of the total space in use and are of short duration. The space allocated to longer-lasting items like the WATCH LIST, message texts, and logs is allocated contiguously to avoid "holes." The two kinds of space work towards each other from opposite ends of the address space — temporary allocations from the low end, permanent allocations from the high end.

DC may also be called upon to allocate core memory for purposes that are not clearly connected with a particular module (otherwise the core is part of the module itself, allocated by EX). The primary such usage is the memory for messages themselves, allocated from main memory in a manner similar to that of temporary disk allocation. For the "permanent" data sets such as the WATCH LIST, DC does not allocate the memory into which they are read. That is the function of the module requesting the data, and DC merely calls upon DK to perform the transfer. The requesting module must verify that the memory area provided is adequate.

No actual disk operations are performed by DC, but rather by DK upon receipt of a WCB containing a description of the operation. However, DC does call itself to obtain memory space for buffers, and to allocate temporary disk space (in particular to hold the available space list itself).

WORK CONTROL BLOCKS (WCBs)

The communication mechanism between modules is the Work Control Block, which any module can pass to any other (including itself) by means of an EX request.

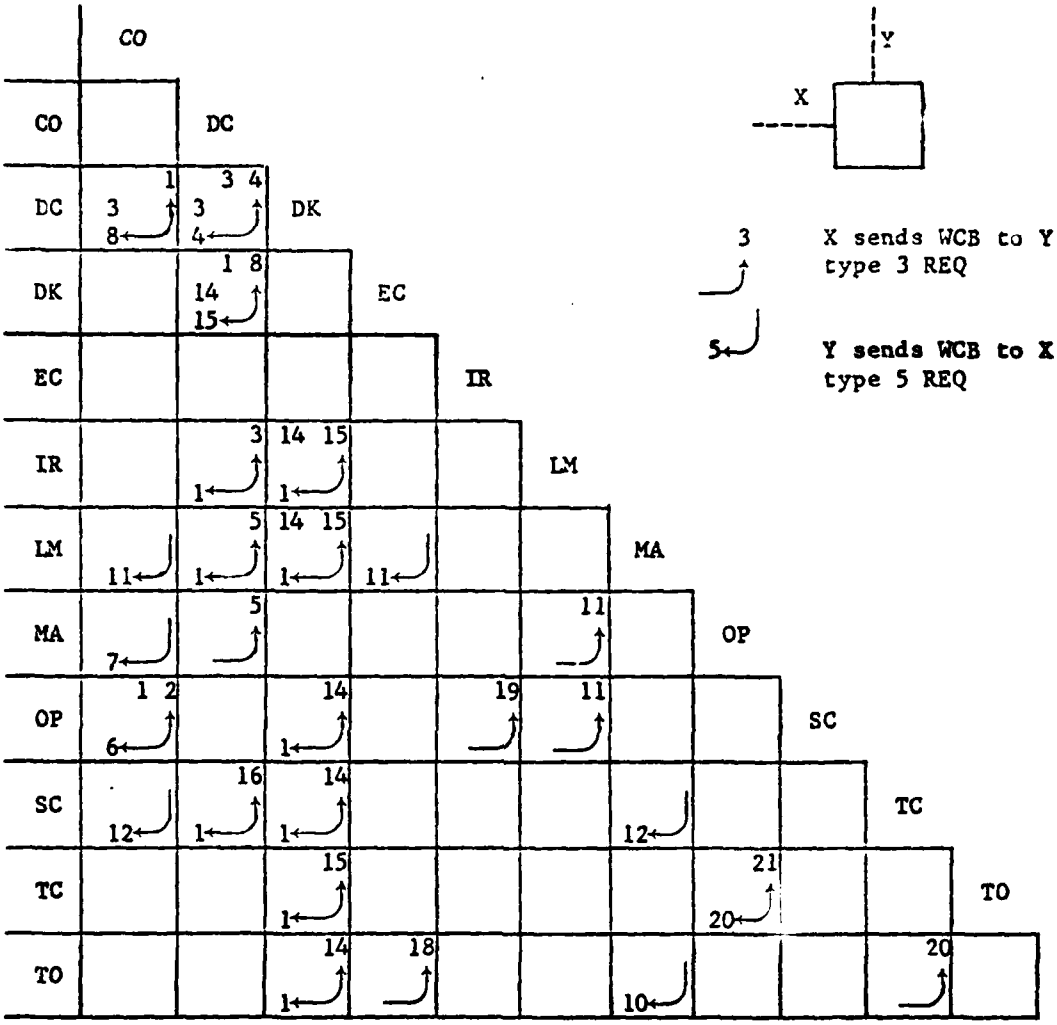
FORMAT OF A WCB (WORK CONTROL BLOCK)

WCBs are the primary means of communication between the modules of MP. All requests from one module to another are made by composing a WCB in a predetermined core area, then calling EX to send it to the recipient. Each WCB concerns a specific message, and the WCB identifies the message of concern both by means of the message identifier and by means of the disk address of the message text.

In the following, the length of each WCB field is given in bytes.

FIELD #	NAME	LENGTH	PURPOSE
0	SIZE	2	Size in bytes of this WCB
1	SENDER	2	Identifies the sending module
2	RECIP	2	Identifies the intended recipient
3	REQ	3	Identifies the function requested of recipient
4	DSKTR	4	Determines the track # on disk where message is
5	DSKST	2	Determines position on track where message starts
6	DSKLN	3	Length of message text in sectors on disk
7	INNO	4	Serial number of message
8	CLASS	6	Classification # of message
9	PREC	1	Message precedence
10	AROUT	7	Routing indicator of originator
11	SEC	1	Security code of message
12	DIRECT	1	1 = incoming, 0 = outgoing
13	IRDSUTC	4	Starting position of internal routing list on disk
14	DSULN	2	Length of internal routing list in sectors
15	MDBTR	4	Disk address of MDB (track)
16	MDBST	2	Track position of MDB
17	MDBLN	1	Length of MDB in disk sectors
18	PRIO	1	Queue priority of WCB
19	SEQ	4	Sequence number of WCB
20	REQDAT	0-10	Request-dependent data

REQ #	Description	REQDAT Contents
1	Response to a previous WCB request	SEQ of request and response data
2	Create new message	None
3	Request core	Size
4	Request temporary disk space	Size
5	Request permanent disk space	Size
6	Prompt operator	Prompt query
7	Test and verify message	None
8	Release core	Address
9	Release disk	Address
10	Transmit message	None
11	Log data	Log identification and data to logged
12	Screen message	None
13	Direct EX request	Desired information
14	Disk operation (wait)	Description of operation
15	Disk operation (proceed)	Description of operation
16	"Read" permanent data set	Data set identification and buffer to use
17	"Write" permanent data set	Data set identification and buffer to use
18	Drive UGHTRANS	UGHTRANS function
19	Retrieval request	None
20	Terminal read/write	Data or pointer to it on temporary disk
21	Terminal attention request	None



The REQ functions are listed below by number.

MP WCB ROUTING DIAGRAM

MP.4 MP Improved Modular Structure

EXAMPLE DESCRIPTION

Introduction

The original modular structure of the MP demonstrates a number of serious and fundamental violations of the information-hiding criterion for dividing systems into modules. This results in the excessive complexity of the system, as well as the fact that changes tend to involve many modules. Among the most significant errors are:

1. Far too many modules are sensitive to changes in the external message format. The descriptions of CO, MA, SC and TO all show a dependence on the AUTONOYS conventions; these conventions are both complex and subject to change.

2. Several modules have direct knowledge of disk characteristics and use disk addresses. This makes it difficult to use another type of storage device, should a better one become available.

3. Two different modules must know the data organization used in the logs. Changes in the queries possible can have major effects on the log maintenance required, and changes in the log organization will in turn have major effects on the IR module.

4. The fact that resources are allocated by several modules without communication will make deadlock recognition and prevention quite difficult.

5. The fact that the incoming data is modified by several modules during their attempts to analyze it may result in subtle, hard-to-find errors.

The following is a proposal for an improved structure.

Modules

MH: Message Holder

This module is responsible for storage and retrieval of all messages. All direct accesses to the internal representation of a message are serviced by functions belonging to this module. The original storage of the message and any subsequent modifications are performed by the functions belonging to this module. The interface to this module is a set of functions allowing other programs to store and access elements of a message, e.g., SET_CHANNEL and GET_CHANNEL to store and read the channel number in the message. The special character sequences at the start of various format lines are no longer considered part of the message and are not made available. Additionally, from

the set of functions available on the interface, one can no longer recognize the order in which the various components appeared in the original text. Further, one cannot test to see if a given item was present several times to provide redundancy.

If messages must be stored on backup-store devices, the data is organized in "pages" that are then stored and retrieved by PS.

EI: External Interface Module(s)

This module is responsible for conversions between the actual message format and the abstract format. If there are several external message formats in use, there will be separate versions or submodules for each one.

Programs in this module analyze the incoming text, which they find stored in buffers, identify the components of the message and call programs in the message holder module to store the information, making it available to other modules.

When a message is being output, programs in this module call the functions of the message holder module to get the contents of the message fields and then arrange the information in the proper order with the appropriate delimiters, storing the completed message in an output buffer.

If the transmission is noise free, the input programs and the output programs are essentially complementary and have the same "secret". If the input data is noisy, then the input programs require additional information that is not needed by the output programs. The input programs must know the expected frequency and nature of errors, in order to detect and correct errors on the basis of redundancy.

CM: Communication Modules

These modules know the communication protocols, including handshaking and timing. Although they control transmission of messages on devices, they know neither the structure of messages nor the details of device control. Incoming messages go to, and outgoing messages come from, the external interface (EI) module. Programs in the equipment control (EC) module are called to tune the device, change the frequency, etc.

SC: Screening

This module fulfills the same function as the SC module in the old MP structure, but it no longer requires detailed knowledge of the format, since it uses the message holder to get the contents of the addressee lists of incoming messages.

The "watch list" is a secret of a submodule of this module. The submodule contains programs to insert and delete watch list entries and to search the watch list for a specific entry.

EC: Equipment Control

Controls UGHTRANS devices.

TC: Terminal Control

This module controls the terminal devices. It knows how to read and write characters, how to generate line feeds, etc. The module includes separate submodules for each terminal type.

DS: Display Module

This module displays messages for the operators. A message can be either received over the UGHTRANS device or created with the text editor module. The module knows how the fields of a message should be arranged in the display, and it uses the message holder interface to get the contents of the fields. It uses a terminal control module to write the characters to the device.

The module includes a separate submodule for each different display format. The display formats will probably be quite different from the format known by the external interface module.

TE: Text Editor

This module implements the command language the operator uses to create messages. It recognizes commands and generates prompts. When the operator inputs a message field, the text editor stores the contents using the message holder interface.

IR/LOG: Information Retrieval and Log Storage

We have combined the information retrieval and log modules into a single module that understands the organization of the data that has been stored about incoming and outgoing messages. This module does not deal directly with background memory but uses pages that are stored and retrieved by PS.

PS: Page Storage

All memory and backup-store access is centralized in this module. It keeps files in terms of pages.

*IC: Intermodule Communication

This module is responsible for keeping the queues of WCBs between components as they are needed. It was formerly a part of EX.

---

* These modules will be discussed in more detail later in the course.

*AL: Allocator and Scheduling

This module is the central allocator of all resources including core and processors. It includes a "banker" to help prevent deadlocks.

*IH: Interrupt Handler

This module translates interrupts into signals for the various system components.

Relation between the "old" modules and the new

1. The work of EX has been divided among IC, AL and IH.
2. MA work is now done by EI, using MH SET functions to store the resulting information.
3. SC work is still done by SC, but the new SC is considerably simpler because it uses the MH GET functions.
4. CO work is now done in the output submodules of EI, which use the GET functions of MH to get the information to put in the messages. The text editor is now separate (TE).
5. TC's duties are carried out by the DS module, which uses the new TC module to write the characters. Thus, the display format is separated from the device characteristics. DS is simpler than the old TC because it gets information for the display using the GET functions of MH.
6. EC now receives commands from the CM module. The new EC is simpler than the old because it knows nothing about when and why things are done, only how they are done.
7. The duties of OP are divided among several modules, including DS, CM, and TE.
8. The work of TO is now performed by parts of MH, EI and CM.
9. DC has been subsumed in PS and AL.
10. DK is now in PS.
11. LM and IR are now the single module IR/LOG, which uses PS.
12. Initialization is not a separate module, but is performed by the initialization routines for the individual modules.

---

* These modules will be discussed in more detail later in the course.

MP.5 MP Message Holder Module

EXAMPLE DESCRIPTION

The following is an informal functional specification of the message holder module.

Index of Function Descriptions

FUNCTION CALLING FORM: BIND(mn)

INPUT PARAMETERS:

Name	Type	Description
mn	message identification	message to be accessed next

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: message designated by mn is bound. Future calls of message information functions refer to this message

---

FUNCTION CALLING FORM: BLANKIT(i)

INPUT PARAMETERS:

Name	Type	Description
i	integer	position to be blank

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: the i^{th} character is removed

FUNCTION CALLING FORM: GET_ACTION_OR_INFO

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: boolean

FUNCTION VALUE: 0 = action required
1 = information only

EFFECTS: none

FUNCTION CALLING FORM: GET_ADDEE

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: string — 6 characters

FUNCTION VALUE: message addressee

EFFECTS: none

FUNCTION CALLING FORM: GET_CHANNEL

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: integer

FUNCTION VALUE: channel on which message was received will be sent

EFFECTS: none

FUNCTION CALLING FORM: GET_CLASSIFICATION

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: character

FUNCTION VALUE: classification of message ("T" = TOP SEC, "S" = SEC, etc.)

EFFECTS: none

FUNCTION CALLING FORM: GET_DAY

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: date

FUNCTION VALUE: day message was received for transmission as indicated in message text

EFFECTS: none

FUNCTION CALLING FORM: GET_ORIGINATOR

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: string

FUNCTION VALUE: code of originating organization in message

EFFECTS: none

FUNCTION CALLING FORM: GET_ORIGINATOR_ROUTING_INDICATOR

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: string — 7 characters

FUNCTION VALUE: routing indicator for originating organization in message

EFFECTS: none

FUNCTION CALLING FORM: GET_PRECEDENCE

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: integer

FUNCTION VALUE: precedence of message

EFFECTS: none

FUNCTION CALLING FORM: GET_ROUTING_INDICATOR

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: string — 7 characters

FUNCTION VALUE: routing indicator in message

EFFECTS: none

FUNCTION CALLING FORM: GET_SERIAL

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: integer

FUNCTION VALUE: serial number in message

EFFECTS: none

FUNCTION CALLING FORM: GET_TEXT(i,j)

INPUT PARAMETERS:

Name	Type	Description
i	integer	starting location
j	integer	ending location

FUNCTION VALUE TYPE: string

FUNCTION VALUE: the string of characters between positions i and j in text

EFFECTS: error call if no such characters in text

FUNCTION CALLING FORM: GET_TIME

INPUT PARAMETERS:

Name	Type	Description
none		

FUNCTION VALUE TYPE: time of day

FUNCTION VALUE: time at which message was received and filed according to message

EFFECTS: none

FUNCTION CALLING FORM: NEW_MESSAGE(mn)

INPUT PARAMETERS:

Name	Type	Description
mn	integer	unused integer identifier to be associated with new message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: bound message is now mn — all functions are reset

FUNCTION CALLING FORM: SET_ACTION_OR_INFO(bit)

INPUT PARAMETERS:

Name	Type	Description
bit	boolean	type of addressee to be stored in message 0 = ACTION 1 = INFORMATION

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: ACTION_OR_INFO = bit

FUNCTION CALLING FORM: SET_ADDEE(ade)

INPUT PARAMETERS:

Name	Type	Description
ade	string	addressee to be stored in message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: addressee is added to message stored

FUNCTION CALLING FORM: SET_CHANNEL(c)

INPUT PARMETERS:

Name	Type	Description
c	integer	channel # to be stored in the message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: channel stored in the message

FUNCTION CALLING FORM: SET_CLASSIFICATION(c)

INPUT PARAMETERS:

Name	Type	Description
c	character	security classification to be assigned to message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: security classification c is stored in message text

FUNCTION CALLING FORM: SET_DAY(d)

INPUT PARAMETERS:

Name	Type	Description
d	date	date to be stored in message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: day of filing is stored in the message

FUNCTION CALLING FORM: SET_ORIGINATOR(c)

INPUT PARAMETERS:

Name	Type	Description
c	string	originator to be stored in message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: originator c is stored in message

FUNCTION CALLING FORM: SET_ORIGINATOR_ROUTING_INDICATOR(r)

INPUT PARAMETERS:

Name	Type	Description
r	string	routing indicator of originator to be stored in message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: the routing indicator is stored in the message

FUNCTION CALLING FORM: SET_PRECEDENCE(p)

INPUT PARAMETERS:

Name	Type	Description
p	integer	precedence to be assigned to message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: message precedence is set

FUNCTION CALLING FORM: SET_ROUTING_INDICATOR(s)

INPUT PARAMETERS:

Name	Type	Description
s	string—7 characters	routing indicator to be in message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: routing indicator is inserted in message

FUNCTION CALLING FORM: SET_SERIAL(n)

INPUT PARAMETERS:

Name	Type	Description
n	integer	serial number to be set in message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: serial number is inserted in message

FUNCTION CALLING FORM: SET_TEXT(i,j,s)

INPUT PARAMETERS:

Name	Type	Description
i	integer	starting point for insertion of new text
j	integer	end point of new text
s	string	text to be inserted in message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: s will be inserted between the i^{th} and j^{th} character of TEXT

FUNCTION CALLING FORM: SET_TIME(t)

INPUT PARAMETERS:

Name	Type	Description
t	time of day	time to be stored in message

FUNCTION VALUE TYPE: none

FUNCTION VALUE: none

EFFECTS: time of filing is stored in message

MP.6 MP Abstract Interface Module

EXAMPLE DESCRIPTION

Introduction

This paper describes an abstract interface module for the MP system. The format of messages transmitted over the AUTONOYS communications network is defined by the AUTONOYS designers; the MP implementors will probably not be consulted about future format changes. The abstract interface module is intended to insulate the rest of the MP system from format changes: if the message format changes, only the code in the abstract interface module should need to change.

Designing an abstract interface module consists of two phases:

(1) compiling a list of assumptions about the information that will be transmitted through the interface, and getting the list reviewed;

(2) designing the access programs provided by the abstract interface module.

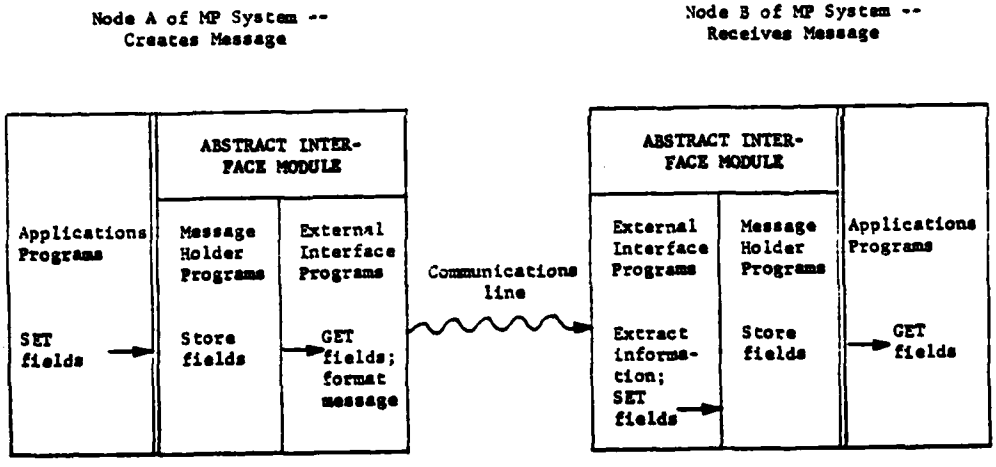
The access programs provide a syntax for communication based on the assumptions. Since the two are closely related, a change in the assumptions will require a corresponding change in the access programs.

The abstract interface module for MP should be based on assumptions that are not likely to change. Consequently it should not be based on specific characteristics of AUTONOYS, such as the order of message fields, the control characters separating fields, or redundancy included for error checking. The interface should be sufficiently general to apply to any message transmission protocol that might reasonably be used to transmit messages to and from the MP system. Care must also be taken that each assumption on the list is both a necessary requirement for the interface and not unduly restrictive.

Description of the MP Abstract Interface Module

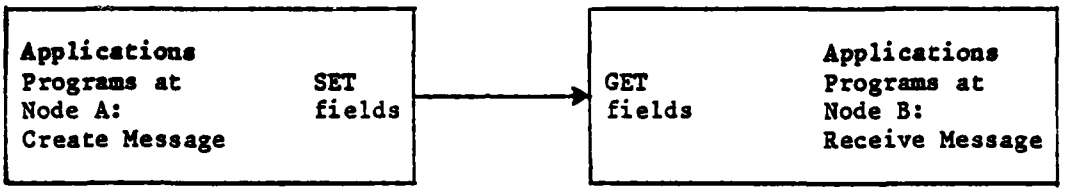
Figure 1 shows how an abstract interface module is used in the MP system. The applications programs in node A construct a message by calling SET access functions provided by the MP abstract interface module. Programs within the abstract interface module in node A arrange the fields in the correct order, duplicate fields that appear more than once in the message, insert control characters, etc. After the message has been transmitted to node B, programs wichin the abstract interface module in node B extract the fields from the stream of characters received over the communications line. Node B applications programs can read fields in the message by calling GET access functions provided by the abstract interface module.

Figure 1. Flow of Information Between MP Modes



As shown in figure 2, the two applications programs can be written as if they could exchange information directly; the fact that the information must be formatted into AUTONOYS messages, sent over a communications line, and extracted from the message is hidden from them.

Figure 2. Message Communication as it Appears to Applications Programs



The same abstract interface module is used to format outgoing messages and to extract information from incoming messages, since both activities need the same secret; i.e., the message format. This module contains two submodules: the message holder (MH) and external interface (EI). Both submodules are described in the paper about the improved MP module structure (MP.4).

Semantic Information in the Assumption List

Semantic knowledge about messages should be an agreement between the applications programs that put information into the message and the applications programs that take information out of the message. However, the semantics of the message are of no concern to the implementor of the abstract interface module; he should not infer any additional assumptions by believing he understands message semantics. Therefore there should be two assumption lists, a semantic list for the applications programs, and a syntactic list for both the abstract interface module and the applications programs. For example, the following assumption is not a proper assumption for the implementor to make, even though it must appear in the semantic list: "Messages may contain declassification information that indicates when the message should be downgraded." The fact that the declassification information states when the message should be downgraded is not a concern of the abstract interface module implementor; he need only know whether his module needs to enforce any restrictions on the declassification values. Leaving the semantics out of the assumption list for implementors will discourage them from inferring such assumptions as:

> declassification information is represented as dates
> and all such dates will be future dates.

However, if it is a requirement that the interface module verify that all declassification dates are in the future, then an assumption to that effect should be explicitly included in the assumption list for the implementor.

Semantic information should be included in a separate assumption list meant only for the applications programs; this list defines terms, relates the assumptions to the environment of the applications programs, and shows how each field should be interpreted. The semantic assumption list is omitted from this document.

Use of Data Types to Simplify Specifications

To specify access functions provided by a module, it is often necessary to place restrictions on the variables that are passed as parameters to or returned by access functions. We define a set of data types in order to express these restrictions concisely and precisely. By associating each value with a data type, we can define the set of legal and meaningful access function calls in a simple and compact manner. For example, the MP abstract interface specifications refer to variables that have the data type time. Restrictions on variables of type time are defined in one place; all variables of type time that are passed as parameters or returned as function values must conform to these restrictions.

If the module is implemented in a modern programming language with user-defined data types, data types used in the specifications can also be used in programs so that the programmer can take advantage of type-checking capabilities in the compiler. If we use a more conventional language such as FORTRAN, we can rely on the programmers or we can provide type-checking by means of preprocessing and/or run-time checking with calls to error routines. The use of data type references in specifications does not imply an implementation requirement.

The data type definitions must provide a way for the applications programs to create and refer to variables of specific types. These are simple for conventional data types: character strings are represented by strings of characters and integers by integers. For more novel data types like date and time, we provide functions that convert integers or character strings into variables of these types.

The three functions in figure 3 are provided to create date and time variables from integers and strings. Note that two date conversion functions are provided; if the parameters represent the same date, the two date functions produce variables with the same value. Thus a date variable representing July 4, 1976 can be produced either by calling JULIAN (76, 186) or by calling DAYMOYR (4, July, 76).

Figure 3. Data Type Function Specifications

FUNCTION CALLING FORM: JULIAN (year, day)

INPUT PARAMETERS:

Name	Type	Description
year	integer	year in the 20th century
day	integer	day in year

FUNCTION VALUE TYPE: date

FUNCTION VALUE: date represented by the two input parameters

FUNCTION CALLING FORM: DAYMOYR (day, month, year)

INPUT PARAMETERS:

Name	Type	Description
day	integer	day in month
month	string	name of month
year	integer	year in 20th century

FUNCTION VALUE TYPE: date

FUNCTION VALUE: date represented by the three input parameters

---

FUNCTION CALLING FORM: CLOCK24 (hour, min)

INPUT PARAMETERS:

Name	Type	Description
hour	integer	hour of day in 24-hour clock
min	integer	minutes after the hour

FUNCTION VALUE TYPE: time

FUNCTION VALUE: time represented by the two input parameters

The table below lists the data types used in the abstract interface specifications that follow.

Type	Meaning	Example
integer	a conventional integer	3
boolean	variable which can take two values: TRUE and FALSE	TRUE, FALSE
character	an alphanumeric character	c, x, z, 1, p
time	variable from which hour, minute can be determined	CLOCK24(13,15)
date	variable from which day, month and year can be determined	JULIAN(76,186) DAYMOYR(4,JULY,76)
string	string of characters	birthday_message, congratulations, please send money
ARchar	single character, two legal values: A and R	A, R

MP ABSTRACT INTERFACE MODULE: ASSUMPTION LIST

In the assumption list that follows, various parameters are used in order to defer decisions that are better made during implementation or at system generation time. These parameters characterize the message holder in a particular node of the MP system; the parameters may take different values in different nodes. These parameters are:

MAX_{msg}	=	the maximum possible number of messages in the message holder
MAX_{line}	=	the maximum number of lines in a message
MAX_{char}	=	the maximum number of characters in one line of a message
$MAX_{addressee}$	=	the maximum number of addressees in a message
MAX_{to_list}	=	the maximum number of addressees that the message may contain in a TO line
MAX_{info_list}	=	the maximum number of addressees that the message may contain in an INFO line
MAX_{xmt_list}	=	the maximum number of addressees that the message may contain in an XMT line

ASSUMPTIONS

1. The message holder can contain both received messages and created messages. Received messages arrived over the communications line; created messages are being contructed at this node. It will be possible to distinguish between them. (A node may transmit a message to itself; the message will be treated as if it were received. The created version of the message will also be accessible until it is destroyed (see assumption 5).)

2. After starting to create a message and before the message is destroyed, other messages may be created or received. There will be a maximum of MAX_{msg} messages in the message holder at a time.

3. Information in received messages cannot be altered (only GET functions are permitted).

4. Information in created messages may be read or written (both GET and SET functions are permitted).

5. Message information is accessible until the message holder module is given a command to destroy the message. (The message holder makes no assumptions about how long to retain messages or whether to delete messages when they are transmitted.)

6. Received message information is not accessible from the message holder until the message holder is given a name to associate with the message. The message holder indicates whether or not there are any received messages waiting to be named.

7. Each message contains at most MAX_{line} number of lines.

8. Each message line has at most MAX_{char} items of type character.

9. The following information may be found or placed in a message.

Item name	Number in completed message	Type
addressee	at least one at most $MAX_{addressee}$	string
origin_route_part	exactly one	string
channel_id	exactly one	integer
precedence	exactly one	string
origin_media	exactly one	string
dest_media	exactly one	string
classification	exactly one	string
content_action	exactly one	string
sender_orig_route	exactly one	string
serial	exactly one	integer
date_received	exactly one	date
time_received	exactly one	time
addressee_route	exactly one	string
to_list	at least one at most MAX_{to_list}	string
info_list	at most MAX_{info_list}	string
xmt_list	at most MAX_{xmt_list}	string
subject_code	exactly one	string
text	exactly one	string
originator	exactly one	string

MP ABSTRACT INTERFACE MODULE: ACCESS FUNCTION SPECIFICATIONS

This section specifies the access functions provided by the MP abstract interface module. The function specifications will require alteration if the assumption list is changed.

Undesired events are handled in the following specifications via calls to user-supplied trap routines. The routine names suggest the nature of the undesired event that occurred. This mechanism allows us to delay specifying the action that should be taken in exceptional circumstances until the trap routines are designed.

FUNCTION CALLING FORM: GET_NUMBEROF_MESSAGES MODULE: MH

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: integer

FUNCTION VALUE: Number of messages currently in the message holder. This number is at most MAX_{msg}.

INITIAL VALUE: 0

EFFECTS: None

FUNCTION CALLING FORM: IS_CREATED_STATUS(msg) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	an identifier associated with a message

FUNCTION VALUE TYPE: boolean

FUNCTION VALUE: TRUE if msg is a created message; FALSE if msg is a received message

EFFECTS: If msg is not associated with a message in the message holder then UE_NO_MSG is called.

FUNCTION CALLING FORM: CREATE(msg) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	an identifier to be associated with the newly created message

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: If GET_NUMBEROF_MESSAGES = MAX_{msg} then UE_TOO_MANY_MSG is called.

Otherwise, a new message is created and associated with the string msg. GET_NUMBEROF_MESSAGES is incremented by 1. IS_CREATED_STATUS(msg)=TRUE

FUNCTION CALLING FORM: IS_WAITING_MESSAGE MODULE: MH

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: boolean

FUNCTION VALUE: TRUE if any messages have been received but not named.
FALSE if no messages are waiting to be named.

EFFECTS: None

FUNCTION CALLING FORM: NAME_MESSAGE(msg) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	an identifier to be associated with the newly received message

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: If GET_NUMBEROF_MESSAGES = MAX_{msg} then UE_TOO_MANY_MSG is called. If IS_WAITING_MESSAGE = FALSE then UE_NO_WAITING_MESSAGE is called

Otherwise, a received message is associated with the name msg. GET_NUMBEROF_MESSAGES is incremented by 1. IS_CREATED_STATUS(msg)=FALSE

FUNCTION CALLING FORM: DESTROY(msg) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	the identifier of a message in the message holder

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: If msg is not associated with a message in the message holder then UE_NO_MSG is called.

Otherwise, msg is no longer associated with a message in the message holder. GET_NUMBEROF_MESSAGES is decremented by 1.

The notation GET_@ and SET_@ denotes a set of access functions where each string in the following list may be substituted for @ to obtain a particular function. Additionally, the parameter type for each access function may differ depending on the value of @. These differences are indicated in the #-type list below.

In the following function specifications substitute the following values for @ and #. Whenever @ is used and a # symbol appears in the specification, substitute the corresponding type. For example, in SET_@(msg,parm), if @ is SERIAL, then # is integer; the resulting function is SET_SERIAL(msg,parm) and parm must be an integer.

@	#-type
ORIGIN_ROUTE_PART	string
CHANNEL_ID	integer
PRECEDENCE	string
ORIGIN_MEDIA	string
DEST_MEDIA	string
CLASSIFICATION	string
CONTENT_ACTION	string
SENDER_ORIG_ROUTE	string
SERIAL	integer
DATE_CREATED	date
TIME_CREATED	time
ADDRESSEE_ROUTE	string
SUBJECT_CODE	string
TEXT	string
ORIGINATOR	string

FUNCTION CALLING FORM: SET_@(msg,parm) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	the identifier of a message in the message holder
parm	#	the item to be stored in the @-field of msg

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: If msg is not associated with any message in the message holder, then UE_NO_MSG is called. If IS_CREATED_STATUS(msg) = false, then UE_GET_ACCESS_ONLY is called.

Otherwise, the @-field of msg is set to parm. If the @-field has previously been set then the @-field is overwritten with the new value.

FUNCTION CALLING FORM: GET_@(msg) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	the identifier of a message in the message holder

FUNCTION VALUE TYPE: #

FUNCTION VALUE: the @-field of the message associated with msg

EFFECTS: If msg is not associated with a message in the message holder then UE_NO_MSG is called. If this field has not been set in the message associated with msg then UE_NO_@ is called.

In the following function specifications substitute the following values for $. Each list contains up to $MAX_{\$}$ pairs of variables; one variable is the routing indicator, the other an addressee identifier.

$
INFO_LIST
XMT_LIST
TO_LIST

FUNCTION CALLING FORM: GET_NUMBEROF_$(msg) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	the identifier of a message in the message holder

FUNCTION VALUE TYPE: integer

FUNCTION VALUE: the number of $-list pairs in msg. This number is at most $MAX_{\$}$.

EFFECTS: If msg is not associated with a message in the message holder, then UE_NO_MSG is called

---

FUNCTION CALLING FORM: SET_$(msg,route,addressee) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	the identifier of a message in the mesage holder
route	string	the routing indicator to be stored in the next $ pair of the message
addressee	string	the addressee identifier to be stored in the next $ pair of the message

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: If msg is not associated with any message in the message holder, then UE_NO_MSG is called. If IS_CREATED_STATUS(msg) = FALSE, then UE_GET_ACCESS_ONLY is called. If GET_NUMBEROF_$(msg) = $MAX_{\$}$, then UE_TOO_MANY_$ is called.

Otherwise, route and addressee are set in a pair appended to the contents of the $-field of msg and GET_NUMBEROF_$(msg) is incremented. If there were no previous items in the $-field, then this pair is the first.

FUNCTION CALLING FORM: GET_$(msg,infotype,i) MODULE: MH

INPUT PARAMETERS:

Name	Type	Description
msg	string	the identifier of a message in the message holder
infotype	ARchar	whether addressee or route component of the pair
i	integer	the number of the pair in $-list

FUNCTION VALUE TYPE: string

FUNCTION VALUE: if infotype = A then the ith addressee in the $-list
if infotype = R then the ith route in the $-list

EFFECTS: If msg is not associated with any message in the message holder, then UE_NO_MSG is called. If i lt 0 or i gt GET_NUMBEROF_$(msg), then UE_BAD_$_NO is called.

MADDS.1 The Military Address System (MADDS)

EXAMPLE DESCRIPTION

MOTIVATION

Many organizations maintain lists of names and postal addresses in a computer. In simple applications, the whole list is used to generate a set of mailing labels or "personalized" letters. In other applications, a subset of the list is selected according to criteria believed to identify individuals most likely to be interested in the contents of certain mailings. For example, a publisher who wishes to offer a new magazine called Tax Loopholes might want to select addresses for persons with medical degrees. Others might want to select all persons within a particular geographic area (consider a magazine like Southern Living, for example), while still others might be interested in persons with specific first or last names.

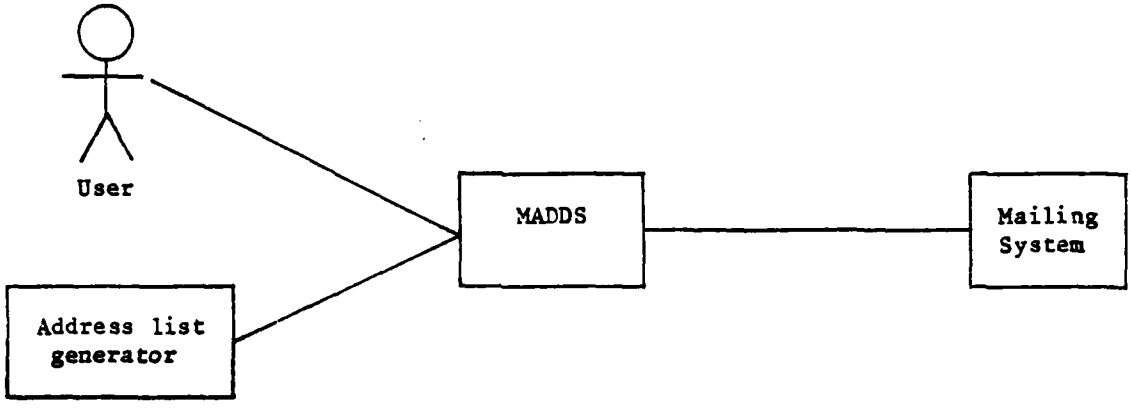
The address lists can be obtained from various sources, such as magazine subscription departments, and are generally delivered on a medium such as magnetic tape. Data from different sources are likely to appear in different record formats.

The general task for any software system that processes such a list is to read the input data in a specified format, extract the desired subset of the list according to specified criteria, and print that subset in a specified format.

A general address-list-processing system is an example of an embedded system; that is, one which is subject to arbitrarily changing constraints, outside the designer's control. The input format is determined by the designers of the system that produced the tape, and the output format is constrained by the requirements of the postal system in which the mail will be deposited.

THE MILITARY ADDRESS SYSTEM (MADDS)

MADDS is a simple address-processing system. MADDS has three system interfaces pictured below:



An important element in understanding MADDS functioning is the data object ADDRESS STORAGE. It is a data base of addresses that MADDS processes. MADDS builds this data base by reading a list of addresses supplied by the address list generator.

User Interface

Inputs. None.

Outputs. There is one possible output from MADDS to the user — a warning that the system has terminated execution because an undesired event (UE) was detected. The warning message will identify the UE and where it was detected in the MADDS software.

Address List Generator Interface

Inputs. There is only one input to MADDS — a file or list of addresses of high-ranking DoD civilian personnel and military officers. (See MADDS.2.) Currently, there are at most 30 addresses in the list. MADDS reads this file into ADDRESS STORAGE prior to producing any mailing lists.

Outputs. None.

Mailing System Interface

Inputs. None.

Outputs. There are two outputs: a list of addresses of persons within the ZIP-code area 203__, and a list of addresses of persons at or above the O-grade 6. The following table explains DoD O-grade levels.

O-GRADE LEVELS

Service:	USA USAF USMC	USN	Civilian
O-grade	Title	Title	GS Level
01	2LT	ENS	07
02	1LT	LTJG	08, 09
03	CAPT	LT	10, 11
04	MAJ	LCDR	12
05	LCOL	CDR	13, 14
06	COL	CAPT	15
07	BG	RADM	16
08	MG	RADM	16
09	LG	VADM	17
10	GEN	ADM	18

The format of both output lists is described in MADDS.2. MADDS generates the lists from addresses in ADDRESS STORAGE whenever requested by the user. There is no requirement on which list is output first; that is, there is no requirement that the O-grade application be done after the area application. Indeed, there is no requirement that there be separate applications.

Likely Changes

1. The format of the address list input from the address list generator can change (see MADDS.2).

2. The format of the address lists output to the mailing system can change (see MADDS.2).

3. The actual ZIP-code area and O-grade limit used in producing the address lists output to the mailing system can change.

4. The maximum number of addresses in the address list input from the address list generator can increase or decrease.

5. The input and output devices can be replaced by similar devices.

6. The order in which the output lists are produced may become important.

THE MADDS PROGRAMMING ASSIGNMENT

There are four things that will happen.

1. First, you will identify the modules of MADDS.

2. After that, you will be given a set of informal interface specifications for the MADDS modules identified by us. You will study these specifications and satisfy yourself that they are complete. All questions should be resolved before you continue.

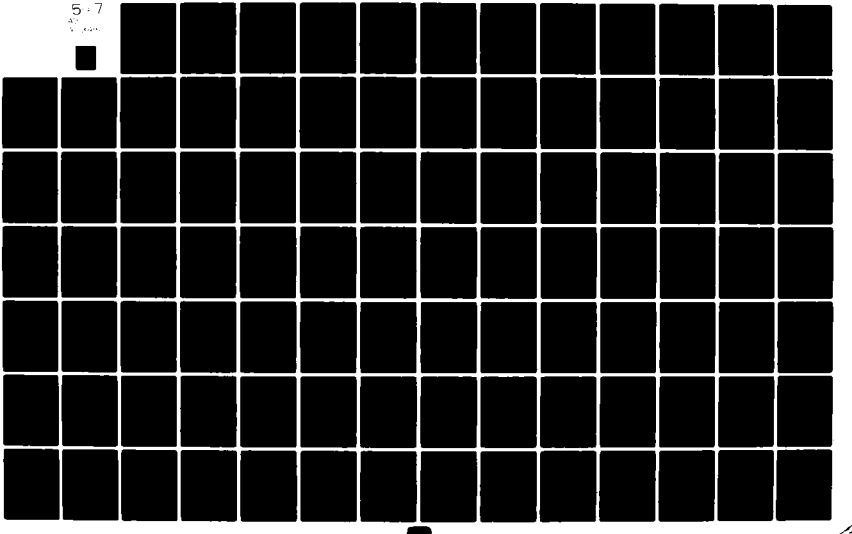
3. The class will be divided into two-person teams. Each team will be assigned the responsibility of implementing one of the modules of the system, as defined in its interface specifications. Implementation information and documentation concerning the local programming environment will be distributed.

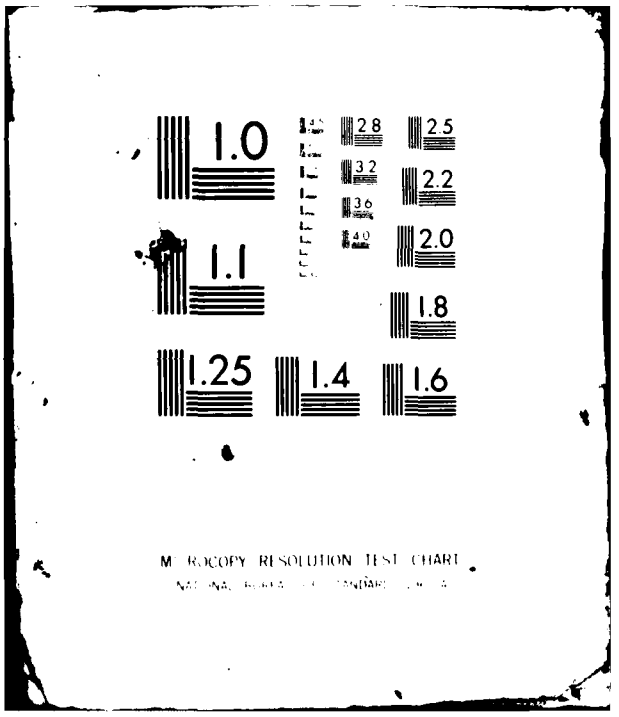
4. Finished modules will be chosen in various combinations to create a running MADDS system, demonstrating the value of carefully chosen and properly specified modules.

AD-A113 415 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
SOFTWARE ENGINEERING PRINCIPLES 3-14 AUGUST 1981.(U)
AUG 81 L J CHMURA, P C CLEMENTS

UNCLASSIFIED NL

5-7




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART

MADDS.2 Input and Output Formats

EXAMPLE DESCRIPTION

MADDS INPUT FORMATS

The list of addresses input to MADDS can be read in character by character. Addresses follow directly after one another without intervening characters. Each address consists of 11 fields. The fields follow directly after one another without intervening characters. The order of the fields is still uncertain at this point. The most likely order is specified by the following table.

Field	Name	Field Size (Number of Characters)	Content(1)
1	Title	4	E.g., "Mr.", "Ms.", "Dr.", "CAPT", "Capt"(2)
2	Last Name	15	
3	Given Names	20	Two strings separated by at least one blank; first string is first name, second is middle
4	Branch or Code	20	
5	Command or Activity	20	
6	Street or P.O. Box	20	E.g., "P.O. Box 208"
7	City	20	
8	State Abbreviation	2	
9	Zip Code, APO code, or FPO code	7	Contiguous decimal digits
10	GS Level	2	"01", "02", ... , "18"(3)
11	Branch of Service	4	"USA", "USMC", "USN", "USAF" or an upper-lowercase variant

The last address of the file is fake, having only a title field consisting completely of asterisks. It is an end-of-file marker.

---

(1) The value of any field can be an all-blank string. If the value of a field is not an all-blank string, then its first character will be non-blank (i.e., values are left justified). Any example field value shorter than the field size should be considered to be padded on the right with blanks.

(2) An officer rank appears when and only when the Branch of Service field is non-blank.

(3) Field is blank when Branch of Service field is non-blank and is non-blank only when Branch of Service field is blank.

Another likely order of the input fields is specified by the table below. The last address of the file is still fake, having only a command or activity field consisting of asterisks.

Field	Name
1	Command or Activity
2	Street or P.O. Box
3	City
4	State Abbreviation
5	Zip Code, APO code, or FPO code
6	Title
7	Given Names
8	Last Name
9	Branch or Code
10	GS Level
11	Branch of Service

MADDS OUTPUT FORMATS

The specifications for the MADDS output format are not fixed. It is likely that each address produced by MADDS should adhere to the following:

Line 1:	Title single blank Given Names single blank Last Name
Line 2:	Branch or Code (BOC)
Line 3:	Command or Activity (COA)
Line 4:	City comma single blank State Abbreviation single blank Zipcode

Trailing blanks in a field should not be printed. Addresses are separated from one another by five blank lines. Each output line is to begin in the first column.

But, it is also possible that the output format specified below will be required:

Line 1:	Command or Activity (COA)
Line 2:	City comma single blank State Abbreviation
Line 3:	Street or Post Office
Line 4:	Last Name

Trailing blanks in a field should not be printed. Addresses are separated from one another by three blank lines. Each output line is to begin in the first column.

MADDS.3 MADDS Modular Structure

EXERCISE

Name: _____

Consider the Military Address System described in MADDS.1 and MADDS.2. **Identify the main modules of the system and describe the information that each module hides.**

Module Name	Secret

MADDS.4 MADDS Modular Structure

EXERCISE SOLUTION

The system consists of nine modules. In addition, there is an undesired event handler.

Module Name	Secret
Applications Program Module (APM)	The APM consists of all applications programs, that is, all programs that perform end-user specified operations on a database of addresses. There are at least two applications programs. One finds and outputs addresses with a specific ZIP-code area. Another outputs addresses with O-grade level less than or equal to a specific value. The APM hides selection algorithms and the order of address entries output.
Address Storage Module (ASM)	The ASM consists of all address storage and retrieval routines. It hides the structure used to store addresses.
Character Module (CHM)	The CHM consists of all character manipulation routines. It hides the internal representation of characters.
Input Device Module (IDM)	The IDM consists of all routines that communicate directly with the physical input device. It hides knowledge of the type of device, the device's representation of characters, the length of input lines, and the protocol used to read input from the device.
Input Module (IPM)	The IPM analyzes the input data and stores the addresses for the ASM. It hides the format (e.g., number, order, and length of the fields) of the input.

Output Device Module (ODM)	The ODM consists of all routines that communicate directly with the physical output device. It hides knowledge of the type of device, the device's representation of characters, the length of output lines, and the protocol used to write output to the device.
Output Module (OPM)	The OPM determines the format in which the address entries are to be output. It hides, for example, order of fields as well as number and content of output lines.
String Storage Module (SSM)	The SSM consists of all string manipulation routines. It hides the internal representation of strings.
Master Control Module (MCM)	The MCM determines the sequence in which applications programs and other programs will be called. Its secret is the way that the functions of other modules are put together to perform useful tasks.
Undesired Event Handler (UEH)	The UEH is a collection of UE handling functions that any module can use to report a UE.

MADDS.5 Using the Computer System

LECTURE

Table of Contents

I. Introduction

During the MADDS exercise, you will be using the IBM Conversational Monitor System (CMS). This tutorial gives you a brief overview of how to use CMS to manipulate files and to compile and execute programs. By treating only a small subset of CMS, we hope to keep the tutorial simple. We also hope to focus your attention on the questions posed by MADDS, rather than on idiosyncrasies of CMS.

In this document, special terminal keys are parenthesized and underlined. For instance, (ENTER) means that you should press the key labeled ENTER, as opposed to typing the five letter word E-N-T-E-R. Also, references will be made to PF keys. To enter PF1-PF12, hold the (ALT) key down and press the key with the appropriate PF number on the front.

If a bold X ever appears in the bottom left corner of the screen, press (RESET). If the X remains, wait a few seconds and press (RESET) again. If the X still remains, ask for assistance.

II. Beginning a Session

A. Locate an IBM screen terminal that is not marked APL. Turn the terminal on by setting the red switch on the lefthand side to 1. The terminal will take about 20 seconds to warm up; when it does, the screen should show the large NPS logo. If this doesn't happen, press the (RESET) key. If the logo still doesn't appear, try another terminal, or ask for assistance. Otherwise, press (RESET), then (ENTER).

B. Type your logon number: L ????P (ENTER)

C. Type your password, followed by (ENTER). The password will not appear on the screen.

D. If you make a mistake, try again.

III. Creating/Editing a File: the XEDIT Editor

You will use the XEDIT editor under CMS to create and handle files. The file you wish to manipulate must be identified when XEDIT is invoked, and no other file may be edited without leaving and re-invoking the XEDIT editor.

A. Identifying a file: All files in CMS are identifed by two things:

1. filename (fn) — any name you choose; up to 8 characters. A good convention is to choose your filename to match the name of the program it contains.

2. filetype (ft) — if the file contains FORTRAN statements to be compiled, the filetype will be WATFIV.

B. Creating a new file: Type

XEDIT fn ft (ENTER)

where fn is the name of a new file and ft is the appropriate filetype. A new and empty file now exists in your directory (i.e., the list of files that CMS has attributed to you; see section IV for further information).

C. Entering data into a file: After invoking the XEDIT editor, type

INPUT (ENTER)

You are now in input mode, and may begin typing lines into the file, one at a time. Should you make a mistake while typing a line, use the cursor keys (marked with arrows) to move back and correct the error. When the line is correct, press (ENTER). Now type the next line. Do not try to change a line after it has been entered.

When you have entered all the lines you want, enter a null line (by pressing (ENTER) twice in a row) to leave input mode. You may now make further changes to the file if you wish.

D. Editing a file:

You may edit a file you just created (by leaving input mode) or edit an old file (by invoking XEDIT and naming the old file). Either way, the contents of the file will be displayed on your screen.

1. Screen editing: The simplest way to change part of a line is to move the cursor (using the arrow keys) to the line and position on that line that you wish to change. Change the line by typing over the old characters; press (ENTER).

2. Prefix subcommands: To make a change involving an entire line or several lines in your file, the simplest method is to invoke XEDIT subcommands. Note that each line on the screen begins with a blank area. To issue a subcommand for a particular line, move the cursor to the blank space (prefix) preceding that line, enter the subcommand, and press (ENTER).

   A brief description of a few important XEDIT subcommands follows. In the descriptions, n stands for an integer of your choice.

Prefix Subcommand	Effect
An	Add n blank lines following this line. After the lines are added, you may use the screen editing technique to fill them in.
Dn	Delete the next n lines, including this one.
Cn	Copy the next n lines, including this one, to somewhere else in the file. Specify that destination by using the P or F subcommand before pressing (ENTER).
Mn	Move the next n lines, including this one, to somewhere else in the file. Specify that destination by using the P or F subcommand before pressing (ENTER).
P or F	Marks the destination location for a move or copy operation. Place the moved or copied block of text preceding or following the line where the P or F subcommand was entered.
/	Makes this line the "current" line. XEDIT always recognizes one line as the current line; it will be displayed on the screen brighter than the rest. Some XEDIT subcommands always have their effect on the current line, and this lets you specify which line that is. The current line is always displayed (along with a screen's worth of surrounding lines) and so redefining the current line also affects what portion of your file is displayed.

3. Other XEDIT commands: To invoke these commands, move the cursor to the bottom of the screen (the "command line").

GET fn ft nl n2	Appends or inserts file fn after the current line in the file your are now editing. Omitting nl and n2 brings in all of fn; otherwise, nl is the beginning line number in fn and n2 is the number of lines you want to move. Before using this subcommand, make sure the current line in the file being edited is the one you want.
TOP	Makes the current line the top of the file; moves the display to the top of the file.

BOTTOM	Makes the current line the bottom of the file; moves the display to the end of the file.
Un	Moves the current line up n lines. This can be used to make the display back up through your file.
Dn	Moves the current line down n lines. This can be used to display lines farther down in your file.
LOCATE /string/	Makes the next line containing the specified string of characters the current line. The / marks are delimiters; they may be any character that does not appear in the string, except @, #, ", ¢, or blank.
CHANGE /s1/s2/	Changes the next occurrence of the string s1 to string s2; the line where s1 was found becomes the current line. Note that the command CHANGE /s1// has the effect of deleting the next occurence of string s1.
ERASE fn ft	Deletes file fn from your directory. You should erase any file you no longer need, but make sure that it's not useful anymore.
RECOVER	In many cases, undoes the last change to the file being edited.

4. Leaving XEDIT:

To leave XEDIT and keep all the changes you made to the file, type:

FILE (ENTER)

To leave XEDIT and cancel all the changes you made to your file (that is, keep your file exactly as it was before you began your XEDIT session), type:

QUIT (ENTER)

XEDIT will ask you if you're sure you want to throw away all the work done in this session. If you are, type

QQUIT (ENTER)

IV. Examining Your Directory: FLIST

When you are not using XEDIT, you may see a list of all of the files that belong to you. Type:

FLIST (ENTER)

You may use FLIST to view (but not change) the contents of any file in your directory. This will be especially helpful when you wish to examine the results of a program execution stored in a LISTING file (explained in section VI). Move the cursor to the line containing the name of the file you wish to examine. Type (PF2). You may page back and forth through the file by typing (PF7) or (PF8). To leave a file, type (PF3).

To leave FLIST, type (PF3).

V. Printing a File

When you are not using XEDIT, you may have the contents of a file printed on a line printer. Type:

PRINT fn ft (ENTER)

where fn and ft are the name and type of the file you wish printed. Output from the printer is placed on the counter or in the pigeonhole boxes outside Ingersoll 147 (the Computation Center) in the section marked VM OUTPUT. Help yourself. The output will have your user number printed in large block letters at the beginning, and END in large block letters at the end.

VI. Running a Program

To compile and execute a FORTRAN program, type:

WATFIV fn (ENTER)

where fn is the WATFIV-type file that you wish to have compiled and executed. Sometimes it may be more convenient to have your source code distributed in more than one file. If that is the case, type WATFIV followed by each filename you wish included (filenames separated by a single blank), followed by (ENTER). The WATFIV FORTRAN processor reads all of the named files as a single continuous file, concatenated in the order they were named.

After execution is complete, you will get a short acknowledgement from CMS. The file(s) you executed will not have changed.

The results of the execution will be placed in a new file created by CMS and placed in your directory. The name of the file is

fn LISTING

where fn is the name of the single file you executed, or the first in the list of files you executed. To view your results, invoke the XEDIT editor on this new LISTING file, or use FLIST as explained in section IV. It will contain a copy of all of the program source code, along with warning/error messages and execution results. If you wish a hard copy of the listing, use the PRINT command as explained in section V.

VII. Running MADDS Programs

You will find that your file directory already contains several files of type WATFIV. You will need these files in order to program and test your MADDS module. Every directory contains the file MADDS. This file consists of the source code for the "rest" of the MADDS system; that is, the system that includes everything except the module that you are writing.

The rest of the files contain lists of sample input data that you can use to test your programs.

File name	File contents
DAT1ADR	Test data file with 1 address
DAT3ADR	Test data file with 3 addresses
DAT7ADR	Test data file with 7 addresses
DATUE	Test data file that will cause a UE
DAT26ADR	Test data file with 26 addresses

To run the complete MADDS system, issue the following command:

WATFIV MADDS yourmodule datafile (ENTER)

where yourmodule is the name(s) of the file(s) containing the module you have written, and datafile is the name of one of the test data files listed above. The results of the run will appear in the file MADDS LISTING.

VIII. Logging Off

To end a terminal session, type:

LOG (ENTER)

IX. Sample CMS Session

Below is a script of the commands and operations that make up the system demonstration held in class. We recommend that you familiarize yourself with CMS by repeating the demonstration on your own.

```
**********************************************************
* Begin session by preparing terminal and logging on.*
**********************************************************
    Turn terminal on; wait for NPS logo.
    (RESET) (ENTER)
    L ????P
    Enter password when prompted.

**********************************************************
* Create a new file; enter a sample FORTRAN program. *
**********************************************************
    XEDIT EXAMPLE WATFIV
    INPUT
    $JOB
    C
    C  Read, store, and print a series of five character strings.
    C
          INTEGER I,J
          CHARACTER*10 WORD, WLIST(5)
    C
          DO 10 I=1,5
             READ (5,30) WORD
             WLIST(I) = WOOF
       10 CONTINUE
    C
          DO 20 J=1,I
             WRITE(6,40) WLIST(I)
       20 CONTINUE
          STOP
    C
       30 FORMAT(A10)
       40 FORMAT(' ', A10)
    C
          END
    $ENTRY
    FIRSTWORD
    SECONDWORD
    THIRDWORD
    FOURTHWORD
    FIFTHWORD
                                          (null line)
    FILE
```

```
*********************************************************
* Run the program; examine the listing.                 *
*********************************************************
    WATFIV EXAMPLE
    FLIST
    Move cursor to file EXAMPLE LISTING; type (PF2).
    Use (PF7) and (PF8) to examine the listing.
    Locate error(s) and diagnose.
    Type (PF3) to leave FLIST.

*********************************************************
* Edit the source code file to fix errors.              *
*********************************************************
    XEDIT EXAMPLE WATFIV
    TOP
    LOCATE/WCOF/
    Move cursor to error; re-type; (ENTER)

*********************************************************
* Re-run the program, and examine the new listing.      *
*********************************************************
    WATFIV EXAMPLE
    FLIST
    Move cursor to file EXAMPLE LISTING; type (PF2).
    Use (PF7) and (PF8) to examine the listing.
    Type (PF3) to leave FLIST.

*********************************************************
* Listing is satisfactory; print it.  End session.      *
*********************************************************
    PRINT EXAMPLE LISTING
    LOG
    Turn terminal off; retrieve printed listing.
```

MADDS.6 Informal Functional Specifications for MADDS Modules

EXAMPLE DESCRIPTION

TABLE OF CONTENTS

TABLE OF CONTENTS (concluded)

INTRODUCTION

This document describes the functions offered by each MADDS module, how to invoke them, and what each does. It does not set performance constraints.

The description of each function has a standard format that is largely self explanatory. The EFFECTS section describes what a function does. The EFFECTS sections of many function descriptions mention the testing for and signaling of undesired events. The function does these UE tests, not the calling program.

The function descriptions are not tied to any one programming language (e.g., FORTRAN, ALGOL, PL/I, or COBOL). In FORTRAN, a function described as returning a value (e.g., GETNCA of the Address Storage Module) could be implemented as a function subprogram and invoked by function reference. A function described as not returning a value could be implemented as a subroutine and invoked by a CALL statement.

APPLICATIONS PROGRAM MODULE (APM)

FUNCTION CALLING FORM: AREA(prezip) MODULE: APM

INPUT PARAMETERS:

Name	Type	Description
prezip	string	a string whose first three characters are digits $d_1d_2d_3$, giving the area part (first three digits) of a set of ZIP-codes, and whose remaining characters are blanks

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: AREA selects and writes out all complete addresses with $d_1d_2d_3$ as the area part of their ZIP-code fields. If no addresses are selected, then no output occurs. A set of complete addresses (for searching and selection) exists after RDADS of the IPM has been called; hence, AREA assumes that RDADS has been called since the last INITAS. AREA also assumes that the output device is open for output.
If $d_1d_2d_3$ is not a three-digit sequence representing an integer $i : 0 \leq i \leq 999$,
then UEZIP is called.

FUNCTION CALLING FORM: RANK(oglim) MODULE: APM

INPUT PARAMETERS:

Name	Type	Description
oglim	string	a string whose first two characters are digits d_1d_2 giving an O-grade level, and whose remaining characters are blanks

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: RANK selects and writes out all complete addresses with O-grade $\leq d_1d_2$. The O-grade is determined by the following table. A set of complete addresses (for searching and selection) exists after RDADS of the IPM has been called; hence, RANK assumes that RDADS has been called since the last INITAS. RANK assumes that the output device is open for output.
If d_1d_2 is not a two-digit sequence representing an integer i: $1 \leq i \leq 10$,
then UEOGL is called.

Table: O-Grade Levels

Service:	USA USAF USMC	USN	Civilian
O-grade	Title	Title	GS Level
01	2LT	ENS	07
02	1LT	LTJG	08, 09
03	CAPT	LT	10, 11
04	MAJ	LCDR	12
05	LCOL	CDR	13, 14
06	COL	CAPT	15
07	BG	RADM	16
08	MG	RADM	16
09	LG	VADM	17
10	GEN	ADM	18

ADDRESS STORAGE MODULE (ASM)

An address is simply a set of strings. If one, but not all, of the fields is undefined (i.e., has not been assigned a string value), then the address is partially defined, and if all fields are undefined, then the address is undefined. If all fields of an address are defined, then the address is complete. An address identifier is a positive integer no larger than maximum address storage capacity MAXADS and is "absurd" if not in this range. Once the number of complete address n: $0 \leq n \leq$ MAXADS has been determined (by VERADS of the ASM), subsequently giving GETNCA = n, we may say that an address identifier adr is assigned if and only if $1 \leq adr \leq$ GETNCA, and unassigned otherwise. There are no operations on addresses as such; only the setting of and retrieving from their fields is allowed. Multiple ownership of addresses (e.g., several instances of the same identifier value occurring in the system) is permitted; therefore, caution must be exercised because changes to an address by one owner, of course, affects the other owners -- possibly adversely.

FUNCTION CALLING FORM: INITAS MODULE: ASM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: INITAS initializes the ASM for the storage of addresses. That is, it sets up the data structures and places the ASM in the initial state (i.e., capable of address storage but with no addresses presently existing and all fields undefined). It must be called prior to any calls to other ASM functions to assure that the ASM operates correctly in all cases. A call to INITAS automatically destroys any currently existing addresses and returns the ASM to maximum address storage capacity available.

FUNCTION CALLING FORM: MAXADS MODULE: ASM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: integer

FUNCTION VALUE: maximum address storage capacity

EFFECTS: None

FUNCTION CALLING FORM: VERADS MODULE: ASM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: VERADS determines the largest integer n: $0 \leq n \leq$ MAXADS, for which all address identifiers adr: $1 \leq adr \leq n$ have all fields defined by previous call to set, and sets an internal counter in the ASM to n. The value of this counter is considered to be the number of complete addresses stored and is returned as the value of GETNCA of the ASM. The ASM is in a correct state only if all fields of all address identifiers adr are undefined, for $n < adr \leq$ MAXADS.
If there is a defined field for an adr: $n < adr \leq$ MAXADS (i.e., an incorrect state for the ASM),
then UEASMI is called.

FUNCTION CALLING FORM: GETNCA MODULE: ASM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: integer

FUNCTION VALUE: The number of complete addresses stored

EFFECTS: GETNCA returns the value of an internal counter in the ASM giving the number of complete addresses stored.
If VERADS hasn't been called since the last INITAS,
then UENCAU is called.

Below is a table of field mnemonics @, their corresponding descriptive phrases #, and the lengths of their string parameters $. For each, there is a field setting (i.e., insertion) and a field getting (i.e., extraction) function in the ASM.

Table: Field Mnemonics for Addresses

@	#	$
BOC	Branch-Or-Code	20
CIT	City	25
COA	Command-Or-Activity	30
GN	Given Names	20
GSL	GS Level	2
LN	Last Name	25
SER	Service	5
SOP	Street-Or-Post-Office-Box	30
ST	State Abbreviation	2
TIT	Title	10
ZIP	ZIP-Code	9

For the field insertion functions, a single informal specification schema suffices, which is identical for each field except for the field mnemonic @, its descriptive phrase #, and its string parameter length $. The same is true for field extraction. Below are given the field insertion and extraction schemas, each followed by an example obtained in this case by substituting "BOC" for @, "Branch-Or-Code" for #, and "20" for $.

Field Insertion Function Schema

FUNCTION CALLING FORM: SET@(adr,str) MODULE: ASM

INPUT PARAMETERS:

Name	Type	Description
adr	integer	identifier of an address
str	string	a string

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: The first $ characters of the string str are stored as the new value of the # field of the address adr. If the length of str is less than $, then the # field is set to str followed by blanks. In either case, the previous value is lost.
If adr is < 1 or > MAXADS,
then UEAIDA is called.

Sample Field Insertion Function

FUNCTION CALLING FORM: SET3OC(adr,str) MODULE: ASM

INPUT PARAMETERS:

Name	Type	Description
adr	integer	identifier of an address
str	string	a string

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: The first 20 characters of the string str are stored as the new value of the Branch-Or-Code field of the address adr. If the length of str is less than 20, then the Branch-Or-Code field is set to str followed by blanks. In either case, the previous value is lost.
If adr is < 1 or > MAXADS,
then UEAIDA is called.

Field Extraction Function Schema

FUNCTION CALLING FORM: GET@(adr) MODULE: ASM

INPUT PARAMETERS:

Name	Type	Description
adr	integer	identifier of an address

FUNCTION VALUE TYPE: string with length $

FUNCTION VALUE: the string stored in the # field of address adr.

EFFECTS: If adr is < 1 or > MAXADS,
then UEAIDA is called.
If the # field of address identifier adr is undefined but adr has a defined field (i.e., partial address),
then UEADRP is called.
If all fields of address adr are undefined,
then UEADRU is called.

Sample Field Extraction Function

FUNCTION CALLING FORM: GETBOC(adr) MODULE: ASM

INPUT PARAMETERS:

Name	Type	Description
adr	integer	identifier of an address

FUNCTION VALUE TYPE: string with length 20

FUNCTION VALUE: the string stored in the Branch-Or-Code field of address adr.

EFFECTS: If adr is < 1 or > MAXADS,
then UEAIDA is called.
If the Branch-Or-Code field of address identifier adr is undefined but adr has a defined field (i.e., partial address),
then UEADRP is called.
If all fields of address identifier adr are undefined,
then UEADRU is called.

CHARACTER MODULE (CHM)

The character module supports an abstract data type character. Much of the module is provided by the local FORTRAN system. For example:

1. The following character constants are provided:

```
'!', '"', '#', '$', '%', '&', ''', '(', ')', '*', '+', ',', '-', '.',
'/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<',
'=', '>', '?', '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', '[', '|', ']', '\', '_', ' ', 'a', 'b', 'c', 'd', 'e', 'f',
'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
'u', 'v', 'w', 'x', 'y', 'z', '{', '}', '~', ' '.
```

2. The assignment operator is the FORTRAN character assignment operator =.

3. Character variables are declared as follows:

```
CHARACTER  X, Y, Z
```

4. Character variables can be initialized in data statements as follows

```
DATA  X / 'A' /
```

Two new functions or operators, CHAREQ and CHARLT, should be used in place of the standard FORTRAN relational operators .EQ. and .LT.. They are defined as follows:

FUNCTION CALLING FORM: CHAREQ(ch1, ch2) MODULE: CHM

INPUT PARAMETERS:

Name	Type	Description
ch1	char	first character to be compared
ch2	char	second character to be compared

FUNCTION VALUE TYPE: boolean

FUNCTION VALUE: if ch1 = ch2 then true else false

EFFECTS: Equality (=) is defined as equality of the internal integer character codes, except in the following cases: upper and lower case alphabetic characters are considered equal (e.g., 'a' = 'A', 'b' = 'B', ..., 'z' = 'Z').

FUNCTION CALLING FORM: CHARLT(ch1, ch2) MODULE: CHM

INPUT PARAMETERS:

Name	Type	Description
ch1	char	first character to be compared
ch2	char	second character to be compared

FUNCTION VALUE TYPE: boolean

FUNCTION VALUE: **if** ch1 < ch2 **then true else false**

EFFECTS: The relation is-less-than (<) is defined by the following:

(a) SPACE (blank) $< \{'A', 'a'\} < \{'B', 'b'\} < \ldots < \{'Z', 'z'\} <$ '0' < '1' < ... < '9'

(b) is restricted to this subset of characters and hence is a partial function.

If ch1 or ch2 is not a blank, a digit, or a letter, then UECHLT is called.

INPUT DEVICE MODULE (IDM)

FUNCTION CALLING FORM: OPENID MODULE: IDM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: OPENID "opens", or initializes, the input device and the buffers, etc., to enable reading. The input device is initially in the closed state. Whenever it is in the closed state, it must be opened by OPENID, prior to reading characters via RDCHAR.
If the input device is open and OPENID is called,
then UEROPN is called.

FUNCTION CALLING FORM: CLOSID MODULE: IDM

INPUT PARAMENTERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: CLOSID "closes" the input device to reading. For each OPENID there must be a corresponding CLOSID.
If the input device is closed and CLOSID is called,
then UERCLS is called.

FUNCTION CALLING FORM: RDCHAR MODULE: IDM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: char

FUNCTION VALUE: the next character from the input device

EFFECTS: If the input device is closed,
then UEWRCL is called.
If no characters are available on the input device,
then UENOCH is called.
If a device error occurs during the read,
then UEDVER is called.

INPUT MODULE (IPM)

FUNCTION CALLING FORM: RDADS MODULE: IPM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: The input addresses are read from an external storage medium or device accessed by the IDM. These addresses are in external form as sequences of characters, which are partitioned into strings according to an input format known to the IPM. RDADS by means of the IDM causes these strings to be read in character-by-character and stored as the fields of a set of addresses. The reading of this set of addresses is terminated at the first "address" whose first field consists of all end-of-file marker characters; this "address" is not stored. No input validation is performed on the input field values. After all input has been read, RDADS calls VERADS of the ASM to verify the addresses as complete and to set the number-of-complete-addresses counter in the ASM. RDADS uses functions of the IDM to open and close the input device (file). If the number of addresses read exceeds MAXADS of the ASM, then UEADOV is called.

MASTER CONTROL MODULE (MCM)

FUNCTION CALLING FORM: MAIN MODULE: MCM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: MAIN is the main driver program. It performs all module initializations by invoking initialization functions of the modules. Thus, no other program should perform initializations. The central task of MAIN is to specify a particular sequence of input, output, and computation actions for which MADDS is designed. Thus, it will typically use the IPM, the APM and possibly the ODM; however, the capabilities of all the modules are available to MAIN, subject only to use rules stated in their interface specifications.

OUTPUT DEVICE MODULE (ODM)

FUNCTION CALLING FORM: OPENOD MODULE: ODM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: OPENOD "opens," or initializes, the output device and buffers, etc., to enable writing. The output device is initially in the closed state. Whenever it is in the closed state, it must be opened by OPENOD prior to writing characters via WRCHAR.
If the output device is open and OPENOD is called,
then UEROPN is called.

FUNCTION CALLING FORM: CLOSOD MODULE: ODM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: CLOSOD "closes" the output device to writing. For each OPENOD there must be a corresponding CLOSOD. The output device is initially in the closed state.
If the output device is closed and CLOSOD is called,
then UERCLS is called.

FUNCTION CALLING FORM: WRCHAR(chr) MODULE: ODM

INPUT PARAMETERS:

Name	Type	Description
chr	char	a character to be written

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: The character chr is written on the output device.
If the output device is closed,
then UEWRCL is called.
If a device error occurs during the write,
then UEDVER is called.

FUNCTION CALLING FORM: NEWLIN MODULE: ODM

INPUT PARAMETERS: None

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: NEWLIN undertakes device-dependent actions which have the effect of writing an end-of-line character for the device (via WRCHAR). No printable character is actually written on the device. Subsequent writes by WRCHAR start on the next line, unless the current line has not had any printable characters written to it, in which case subsequent writes are to the current line.
If the output device is closed,
then UEWRCL is called.
If a device error occurs during the write,
then UEDVER is called.

OUTPUT MODULE (OPM)

FUNCTION CALLING FORM: WRADR(adr) MODULE: OPM

INPUT PARAMETERS:

Name	Type	Description
adr	integer	identifier of an address

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: The address adr is written out to an external device used by the ODM. Certain fields of the address are written character-by-character according to an output format known to the OPM. This format specifies at least the order and identity of fields, spacing, and line contents. It is assumed that RDADS of the IPM has been called since the last INITAS.
If adr < 1 or > MAXADS,
then UERAIDA is called.
If GETNCA < adr ≤ MAXADS,
then UERAIDU is called.

STRING STORAGE MODULE (SSM)

The string storage module supports an abstract data type string. Much of the module is provided by the local FORTRAN system. For example:

1. String constants are characters enclosed in quotes, e.g., "This is a string.".

2. The assignment operator is the FORTRAN assignment operator =.

3. String variables are declared as follows:

```
CHARACTER * 10   X, Y, Z
```

   which declares variables X, Y, Z to be string variables of length 10.

4. Character variables can be initialized as follows:

```
DATA X /"Example"/
```

Besides this capability, the following functions can be used to manipulate strings.

FUNCTION CALLING FORM: SETCHR(str, pos, chr) MODULE: SSM

INPUT PARAMETERS:

Name	Type	Description
str	string	a string
pos	integer	a character position in str
chr	char	a character to be inserted

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: SETCHR replaces the character at position pos of string str by the character chr.
If pos < 1 or pos > length of str,
then UESPOS is called.

FUNCTION CALLING FORM: GETCHR(str, pos) MODULE: SSM

INPUT PARAMETERS:

Name	Type	Description
str	string	a string
pos	integer	a character position in str

FUNCTION VALUE TYPE: char

FUNCTION VALUE: the character at position pos of string str

EFFECTS: If pos < 1 or pos > length of str,
then UESPOS is called.

FUNCTION CALLING FORM: SUBSTR(str, pos, len) MODULE: SSM

INPUT PARAMETERS:

Name	Type	Description
str	string	a string
pos	integer	a character position in str
len	integer	the length of the substring to be extracted

FUNCTION VALUE TYPE: string

FUNCTION VALUE: a string whose first len characters are the len characters of the string str, beginning with position pos, and whose remaining characters are blanks

EFFECTS: If pos < 1 or pos > length of str,
then UESPOS is called.
If len < 0 or pos + len - 1 > length of str,
then UESLEN is called.

FUNCTION CALLING FORM: STREQ(str1, str2) MODULE: SSM

INPUT PARAMETERS:

Name	Type	Description
str1	string	a string
str2	string	a string

FUNCTION VALUE TYPE: boolean

FUNCTION VALUE: if str1 = str2 then true else false

EFFECTS: Let c_i and c'_i denote the characters at position i of strings str1 and str2, respectively. Let L be the length of the shorter of the two strings. Then str1 and str2 are equal (=) if and only if,

a) for $1 \leq i \leq L$, CHAREQ(c_i, c'_i), and

b) all remaining characters of the longer string are blanks.

If both str1 and str2 are the same length, then the second condition is, of course, unnecessary.

UNDESIRED EVENT HANDLER (UEH)

The UEH consists of the UE handling functions, one for each UE. The list of UEs that can occur are summarized in the following table, where $ is the UE mnemonic and ¢ is the corresponding UE description.

Table: Undesired Events (UEs)

$	¢
ADOV	Address storage capacity overflow
ADRP	Partially defined address (at least one field undefined)
ADRU	Undefined address (no fields defined)
AIDA	Absurd address identifier (i.e., < 1 or > max capacity)
AIDU	Unassigned address identifier (i.e., > GETNCA and ≤ MAXADS)
ASMI	State of the ASM incorrect (i.e., defined fields beyond GETNCA)
CHLT	Undefined character comparison
DVER	Device error
MIDU	Non-existent module identifier
NCAU	Undefined number of complete addresses
NOCH	No characters available on input device
OGL	O-grade level is < 1 or > 10
RCLS	Redundant device closing
ROPN	Redundant device opening
SLEN	Substring length is < 0 or too large
SPOS	Character position in string is < 1 or > string length
WRCL	Writing or reading on closed device
ZIP	ZIP-code area part not three decimal digits

All of the UE handling functions can be represented by a single function schema. This is given below, using UE mnemonic $ and description ¢, and is followed by a sample UE function. Note that there is a UE handler for UEs in the UE handlers (i.e., MIDU; see next page).

The module identifiers used as arguments in the UE handler calls are simply the module abbreviations, "APM", "ASM", "CHM" .... The function identifiers are the full function names, e.g. "AREA".

UE Function Schema

FUNCTION CALLING FORM: UE$(mdid,fnid) MODULE: UEH

INPUT PARAMETERS:

Name	Type	Description
mdid	string	module identifier
fnid	string	function identifier

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: A message is written out to the effect that the UE "ç" has been detected in function fnid of module mdid and the run is aborted. Then execution is terminated.
If mdid is not the identifier of a known module,
then UEMIDU is called (except when $ is MIDU; that is, UEMIDU will not call itself).

Sample UE Function

FUNCTION CALLING FORM: UEAIDU(mdid,fnid) MODULE: UEH

INPUT PARAMETERS:

Name	Type	Description
mdid	string	module identifier
fnid	string	function identifier

FUNCTION VALUE TYPE: None

FUNCTION VALUE: None

EFFECTS: A message is written out to the effect that the UE "Unassigned address identifier" has been detected in function fnid of module mdid and the run is aborted. Then execution is terminated.
If mdid is not the identifier of a module,
then UEMIDU is called.

MADDS.7 MADDS Program Listing

EXAMPLE DESCRIPTION

Table of Contents

```
C
C   apm
C
C   output addresses with zip area part prezip
C
       subroutine area(prezip)
       implicit complex (a-z)
       character*1 digit(10),chr
       character*3 prezip,zarea
       character*9 newzip
       integer adr, getnca, i, j, n
       logical chareq,streq
       data digit/'1','2','3','4','5','6','7','8','9','0'/
       do 20 j=1,3
         call getchr(chr,prezip,j)
         do 10 i=1,10
           if (chareq(chr,digit(i))) go to 20
10     continue
         call uezip('apm','area')
20     continue
       n=getnca()
       if (n.eq.0) return
       do 30 adr=1,n
         call getzip(newzip,adr)
         call substr(area,newzip,1,3)
         if (streq(zarea,prezip)) call wradr(adr)
30     continue
       return
       end
C
C   apm
C
C   output addresses with 0-grade at most oglim
C
       subroutine rank(oglim)
       implicit complex (a-z)
       character*1 ch1,ch2,ch3,ch4,ch5,ch6,digit(10)
       character*2 oglim,gslev(10),gs1
       character*4 serv,usatit(10),usntit(10)
       character*10 title
       integer lim,nca,getnca,adr,j
       logical chareq,charlt,streq
       data usatit/'2lt','1lt','capt','maj','lcol','col','bg','mg',
     &'lg','gen'/
       data usntit/'ens','ltjg','lt','lcdr','cdr','capt','radm',
     &'radm','vadm','adm'/
       data gslev/'07','09','11','12','14','15','16','16','17','18'/
       data digit/'1','2','3','4','5','6','7','8','9','0'/
       call getchr(ch1,oglim,1)
       call getchr(ch2,oglim,2)
```

```
      if (chareq(ch1,'0')) go to 10
      if (.not.chareq(ch1,'1')) go to 30
      if (.not.chareq(ch2,'0')) go to 30
      lim = 10
      go to 40
10    do 20 lim=1,9
            if (chareq(ch2,digit(lim))) go to 40
20    continue
30    call ueog1('apm','rank')
40    nca=getnca()
      if (nca.eq.0) return
      do 80 adr=1,nca
        call getser(serv,adr)
        call gettit(title,adr)
        if (streq(serv,' ')) go to 70
        call getchr(ch1,serv,1)
        call getchr(ch2,serv,2)
        call getchr(ch3,serv,3)
        call getchr(ch4,serv,4)
        if (.not.chareq(ch1,'u') .or. .not.chareq(ch2,'s')) go to 80
        if (chareq(ch3,'n')) go to 60
        if (chareq(ch3,'a')) go to 45
        if (.not.chareq(ch3,'m')) go to 80
        if (chareq(ch4,'c')) go to 50
        go to 80
45    if (.not.(chareq(ch4,' ').or.chareq(ch4,'f'))) go to 80
50    do 57 j=1,lim
            if (.not.streq(title,usatit(j)))go to 57
            call wradr(adr)
            go to 80
57    continue
        go to 80
60    do 67 j=1,lim
            if (.not.streq(title,usntit(j)))go to 67
            call wradr(adr)
            go to 80
67    continue
        go to 80
70    call getgs1(gs1,adr)
        call getchr(ch1,gs1,1)
        call getchr(ch2,gs1,2)
        call getchr(ch5,gslev(lim),1)
        call getchr(ch6,gslev(lim),2)
        if (charlt(ch5,ch1)) go to 80
        if (charlt(ch1,ch5)) go to 75
        if (charlt(ch6,ch2)) go to 80
75    call wradr(adr)
80    continue
      return
      end
```

```
C
C   asm
C
C   initialize the asm
C
        subroutine initas
        implicit complex (a-z)
        character*30 sop,coa
        character*25 cit,ln
        character*20 boc,gn
        character*10 tit
        character*9  zip
        character*5  ser
        character*2  gsl,st
        integer  nca,mad,nflds,adr,i,j
        logical adflag
        parameter (mad=26,nflds=11)
        common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
        nca=-1
        do 10 j=1,nflds
          do 10 i=1,mad
            adflag(i,j)=.false.
10     continue
        return
        end
C
C   asm
C
C   maximum number of addresses
C
        integer function maxads
        implicit complex (a-z)
        integer mad,nflds
        parameter (mad=26,nflds=11)
        maxads=mad
        return
        end
C
C   asm
C
C   determine number of consecutive complete addresses
C
        subroutine verads
        implicit complex (a-z)
        character*30 sop,coa
        character*25 cit,ln
        character*20 boc,gn
        character*10 tit
        character*9  zip
        character*5  ser
        character*2  gsl,st
        integer  nca,mad,nflds,adr,i,j,n
        logical adflag
```

```
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      do 20 n=1,mad
        do 10 j=1,nflds
          if (.not.adflag(n,j)) go to 30
10    continue
20    continue
      nca=mad
      return
30    nca=n-1
      do 50 i=n,mad
        do 40 j=1,nflds
          if (adflag(i,j)) call ueasmi('asm','verads')
40    continue
50    continue
      return
      end
C
C
C   asm
C
C   get number of consecutive addresses
C
      integer function getnca
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      integer  nca,mad,nflds
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca, sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (nca.lt.0) call uencau('asm','getnca')
      getnca=nca
      return
      end
```

```
C
C   asm
C
C   set branch-or-code field of adr to str
C
      subroutine setboc(adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setboc')
      boc(adr)=str
      adflag(adr,1)=.true.
      return
      end
C
C   asm
C
C   set city field of adr to str
C
      subroutine setcit(adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setcit')
      cit(adr)=str
      adflag(adr,2)=.true.
      return
      end
```

```
C
C   asm
C
C   set command-or-activity field of adr to sir
C
      subroutine setcoa(adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common   /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setcoa')
      coa(adr)=str
      adflag(adr,3)=.true.
      return
      end
C
C   asm
C
C   set given-names field of adr to str
C
      subroutine setgn (adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setgn')
      gn(adr)=str
      adflag(adr,4)=.true.
      return
      end
```

```
C
C   asm
C
C   set gs-level field of adr to str
C
      subroutine setgsl(adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setgsl')
      gsl(adr)=str
      adflag(adr,5)=.true.
      return
      end
C
C   asm
C
C   set last-name field of adr to str
      subroutine setln (adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setln')
      ln(adr)=str
      adflag(adr,6)=.true.
      return
      end
```

```
C
C   asm
C
C   set service field of adr to str
C
       subroutine setser(adr,str)
       implicit complex (a-z)
       character*30 sop,coa
       character*25 cit,ln
       character*20 boc,gn
       character*10 tit
       character*9  zip
       character*5  ser
       character*2  gsl,st
       character*(*) str
       integer  nca,mad,nflds,adr
       logical adflag
       parameter (mad=26,nflds=11)
       common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
       if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setser')
       ser(adr)=str
       adflag(adr,7)=.true.
       return
       end
C
C   asm
C
C   set street-or-post-office-box field of adr to str
C
       subroutine setsop(adr,str)
       implicit complex (a-z)
       character*30 sop,coa
       character*25 cit,ln
       character*20 boc,gn
       character*10 tit
       character*9  zip
       character*5  ser
       character*2  gsl,st
       character*(*) str
       integer  nca,mad,nflds,adr
       logical adflag
       parameter (mad=26,nflds=11)
       common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
       if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setsop')
       sop(adr)=str
       adflag(adr,8)=.true.
       return
       end
```

```
C
C   asm
C
C   set state field of adr to str
C
      subroutine setst (adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
       if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setst')
      st(adr)=str
      adflag(adr,9)=.true.
      return
      end
C
C   asm
C
C   set title field of adr to str
C
      subroutine settit(adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','settit')
      tit(adr)=str
      adflag(adr,10)=.true.
      return
      end
```

```
C
C   asm
C
C   set zip-code field of adr to str
C
      subroutine setzip(adr,str)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      character*(*) str
      integer  nca,mad,nflds,adr
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','setzip')
      zip(adr)=str
      adflag(adr,11)=.true.
      return
      end
C
C   asm
C
C   get branch-or-code field from adr
C
      subroutine getboc(result,adr)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn,result
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      integer  nca,mad,nflds,adr,j
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getboc')
      if (adflag(adr,1)) go to 20
      do 10 j=1,nflds
        if (adflag(adr,j)) call ueadrp('asm','getboc')
10    continue
      call ueadru('asm','getboc')
20    continue
      result=boc(adr)
      return
      end
```

```
C
C   asm
C
C   get city field from adr
C
      subroutine getcit(result,adr)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln,result
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      integer  nca,mad,nflds,adr,j
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getcit')
      if (adflag(adr,2)) go to 20
      do 10 j=1,nflds
        if (adflag(adr,j)) call ueadrp('asm','getcit')
10    continue
      call ueadru('asm','getcit')
20    continue
      result=cit(adr)
      return
      end
C
C   asm
C
C   get command-or-activity field from adr
C
      subroutine getcoa(result,adr)
      implicit complex (a-z)
      character*30 sop,coa,result
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      integer  nca,mad,nflds,adr,j
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
```

```
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getcoa')
      if (adflag(adr,3)) go to 20
      do 10 j=1,nflds
        if (adflag(adr,j)) call ueadrp('asm','getcoa')
10    continue
      call ueadru('asm','getcoa')
20    continue
      result=coa(adr)
      return
      end
C
C   asm
C
C   get given-names field from adr
      subroutine getgn(result,adr)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn,result
      character*10 tit
      character*9  zip
      character*5  ser
      character*2  gsl,st
      integer  nca,mad,nflds,adr,j
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getgn')
      if (adflag(adr,4)) go to 20
      do 10 j=1,nflds
        if (adflag(adr,j)) call ueadrp('asm','getgn')
10    continue
      call ueadru('asm','getgn')
20    continue
      result=gn(adr)
      return
      end
```

```
C
C   asm
C
C   get gs-level field from adr
C
       subroutine getgsl(result,adr)
       implicit complex (a-z)
       character*30 sop,coa
       character*25 cit,ln
       character*20 boc,gn
       character*10 tit
       character*9  zip
       character*5  ser
       character*2  gsl,st,result
       integer  nca,mad,nflds,adr,j
       logical adflag
       parameter (mad=26,nflds=11)
       common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
       if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getgsl')
       if (adflag(adr,5)) go to 20
       do 10 j=1,nflds
         if (adflag(adr,j)) call ueadrp('asm','getgsl')
10     continue
       call ueadru('asm','getgsl')
20     continue
       result=gsl(adr)
       return
       end
C
C   asm
C
C   get last-name field from adr
C
       subroutine getln(result,adr)
       implicit complex (a-z)
       character*30 sop,coa
       character*25 cit,ln,result
       character*20 boc,gn
       character*10 tit
       character*9  zip
       character*5  ser
       character*2  gsl,st
       integer  nca,mad,nflds,adr,j
       logical adflag
       parameter (mad=26,nflds=11)
       common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
```

```
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getln')
      if (adflag(adr,6)) go to 20
      do 10 j=1,nflds
        if (adflag(adr,j)) call ueadrp('asm','getln')
10    continue
      call ueadru('asm','getln')
20    continue
      result=ln(adr)
      return
      end
C
C   asm
C
C   get service field from adr
C
      subroutine getser(result,adr)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit
      character*9  zip
      character*5  ser,result
      character*2  gsl,st
      integer  nca,mad,nflds,adr,j
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getser')
      if (adflag(adr,7)) go to 20
      do 10 j=1,nflds
        if (adflag(adr,j)) call ueadrp('asm','getser')
10    continue
      call ueadru('asm','getser')
20    continue
      result=ser(adr)
      return
      end
```

```
C
C   asm
C
C   get street-or-post-office-box field from adr
C
       subroutine getsop(result,adr)
       implicit complex (a-z)
       character*30 sop,coa,result
       character*25 cit,ln
       character*20 boc,gn
       character*10 tit
       character*9  zip
       character*5  ser
       character*2  gsl,st
       integer  nca,mad,nflds,adr,j
       logical adflag
       parameter (mad=26,nflds=11)
       common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
       if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getsop')
       if (adflag(adr,8)) go to 20
       do 10 j=1,nflds
         if (adflag(adr,j)) call ueadrp('asm','getsop')
10     continue
       call ueadru('asm','getsop')
20     continue
       result=sop(adr)
       return
       end
C
C   asm
C
C   get state field from adr
C
       subroutine getst(result,adr)
       implicit complex (a-z)
       character*30 sop,coa
       character*25 cit,ln
       character*20 boc,gn
       character*10 tit
       character*9  zip
       character*5  ser
       character*2  gsl,st,result
       integer  nca,mad,nflds,adr,j
       logical adflag
       parameter (mad=26,nflds=11)
       common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
```

```
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getst')
      if (adflag(adr,9)) go to 20
      do 10 j=1,nflds
        if (adflag(adr,j)) call ueadrp('asm','getst')
10    continue
      call ueadru('asm','getst')
20    continue
      result=st(adr)
      return
      end
C
C   asm
C
C   get title field from adr
C
      subroutine gettit(result,adr)
      implicit complex (a-z)
      character*30 sop,coa
      character*25 cit,ln
      character*20 boc,gn
      character*10 tit,result
      character*9  zip
      character*5  ser
      character*2  gsl,st
      integer  nca,mad,nflds,adr,j
      logical adflag
      parameter (mad=26,nflds=11)
      common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
      if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','gettit')
      if (adflag(adr,10)) go to 20
      do 10 j=1,nflds
        if (adflag(adr,j)) call ueadrp('asm','gettit')
10    continue
      call ueadru('asm','gettit')
20    continue
      result=tit(adr)
      return
      end
```

```
C
C   asm
C
C   get zip-code field from adr
C
        subroutine getzip(result,adr)
        implicit complex (a-z)
        character*30 sop,coa
        character*25 cit,ln
        character*20 boc,gn
        character*10 tit
        character*9  zip,result
        character*5  ser
        character*2  gsl,st
        integer  nca,mad,nflds,adr,j
        logical adflag
        parameter (mad=26,nflds=11)
        common  /asmblk/nca,sop(mad),coa(mad),cit(mad),ln(mad),boc(mad),
     & gn(mad),tit(mad),zip(mad),ser(mad),gsl(mad),st(mad),
     & adflag(mad,nflds)
        if (adr.lt.1.or.adr.gt.mad) call ueaida('asm','getzip')
        if (adflag(adr,11)) go to 20
        do 10 j=1,nflds
          if (adflag(adr,j)) call ueadrp('asm','getzip')
10     continue
        call ueadru('asm','getzip')
20     continue
        result=zip(adr)
        return
        end
C
C   chm
C
C   define internal character comparison code
C
        block data
        integer mask,intchr
        common /chmblk/intchr(128)
        data intchr/0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10,-11,
     &-12,-13,-14,-15,-16,-17,-18,-19,-20,-21,-22,-23,-24,
     &-25,-26,-27,-28,-29,-30,-31,1,-32,-33,-34,-35,-36,
     &-37,-38,-39,-40,-41,-42,-43,-44,-45,-46,28,29,30,31,
     &32,33,34,35,36,37,-47,-48,-49,-50,-51,-52,-53,2,3,
     &4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
     &23,24,25,26,27,-54,-55,-56,-57,-58,-59,2,3,4,5,6,7,
     &8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,
     &26,27,-60,-61,-62,-63,-64/
        end
```

```
C
C   chm
C
C   character equality test
C
      logical function chareq(ch1,ch2)
      character*1 ch1,ch2
      integer mask,intchr,int1,int2,ichar
      common /chmblk/intchr(128)
      int1=ichar(ch1)
      int2=ichar(ch2)
      chareq=intchr(int1+1).eq.intchr(int2+1)
      return
      end
C
C   chm
C
C   character less-than comparison
C
      logical function charlt(ch1,ch2)
      character*1 ch1,ch2
      integer mask,intchr,int1,int2,ichar
      common /chmblk/intchr(128)
      int1=ichar(ch1)
      int2=ichar(ch2)
      if (intchr(int1+1).le.0) call uechlt('chm','charlt')
      if (intchr(int2+1).le.0) call uechlt('chm','charlt')
      charlt=intchr(int1+1).lt.intchr(int2+1)
      return
      end
C
C   idm
C
C   initialize
C
      block data
      implicit complex (a-z)
      character*60 buffer
      integer bufpos,bufsiz,unit
      logical idop
      common /idmblk/bufpos,idop,buffer
      data bufpos/60/idop/.false./
      end
```

```
C
C   idm
C
C   open input device
C
        subroutine openid
        implicit complex (a-z)
        character*60 buffer
        integer bufpos,bufsiz
        logical idop
        common /idmblk/bufpos,idop,buffer
        if (idop)call ueropn('idm','openid')
        idop=.true.
        return
        end
C
C   idm
C
C   close input file
C
        subroutine closid
        implicit complex (a-z)
        character*60 buffer
        integer bufpos,bufsiz
        logical idop
        common /idmblk/bufpos,idop,buffer
        if (.not.idop) call uercls('idm','closid')
        idop=.false.
        return
        end
C
C   idm
C
C   read character from input stream
C
C   this compiler treats an end-of-file condition as an
C   device error condition. so there is no implementation
C   of the device error ue call.
C
        subroutine rdchar(ch1)
        implicit complex (a-z)
        character*1 ch1
        character*60 buffer
        integer bufpos,bufsiz,unit
        logical idop
        parameter (unit=5)
        common /idmblk/bufpos,idop,buffer
```

```
      if (.not.idop) call uewrcl('idm','rdchar')
      if (bufpos.lt.60) go to 10
      read (unit,100,err=30,end=20) buffer
      bufpos=0
10    bufpos=bufpos+1
      call getchr(chl,buffer,bufpos)
      return
20    call uenoch('idm','rdchar')
      return
30    call uedver('idm','rdchar')
      return
100   format (a60)
      end
C
C   ipm
C
C   read and store input addresses
C
      subroutine rdads
      implicit complex (a-z)
      character*20 coa,rdfld,endstr
      integer nads,maxads,mads
      logical streq
      data endstr /'********************'/
      call openid
      mads=maxads()
      nads=0
 1    coa=rdfld(20)
      if (streq(coa,endstr)) go to 20
      nads=nads+1
      if (nads.gt.mads) call ueadov('ipm','rdads')
      call setcoa (nads,coa)
      call setsop (nads,rdfld(20))
      call setcit (nads,rdfld(20))
      call setst (nads,rdfld(2))
      call setzip (nads,rdfld(7))
      call settit (nads,rdfld(4))
      call setgn (nads,rdfld(20))
      call setln (nads,rdfld(15))
      call setboc (nads,rdfld(20))
      call setgsl (nads,rdfld(2))
      call setser (nads,rdfld(4))
      go to 1
20    call verads
      call closid
      return
      end
```

```
C
C   ipm
C
C   read and create an address field string
C
       character *(*) function rdfld(length)
       implicit complex (a-z)
       character*1 chr, rdchar
       integer length,i
       rdfld=' '
       do 10 i=1,length
         call setchr(rdfld,i,rdchar())
10     continue
       return
       end
C
C   mcm
C
       program mcm
       implicit complex (a-z)
       integer i
       character *1 ch1
       character *2 oglim
       character *3 prezip
       call initas
       call openod
       call newlin
       call wrchar (' ')
       call newlin
       do 106 i=1,23
         call getchr(ch1,'starting address input.',i)
         call wrchar(ch1)
106    continue
       call rdads
       call newlin
       call wrchar(' ')
       call newlin
       do 107 i=1,25
         call getchr(ch1,'address reading complete.',i)
         call wrchar(ch1)
107    continue
       prezip = '203'
       call newlin
       call wrchar(' ')
       call newlin
       do 108 i=1,18
         call getchr(ch1,'output of area is:',i)
         call wrchar(ch1)
108    continue
```

```
      call area(prezip)
      oglim = '10'
      call newlin
      call wrchar(' ')
      call newlin
      do 109 i=1,18
        call getchr(chl,'output of rank is:',i)
        call wrchar(chl)
109   continue
      call rank(oglim)
      call newlin
      call wrchar(' ')
      call newlin
      do 110 i=1,17
        call getchr(chl,'end of madds run.',i)
        call wrchar(chl)
110   continue
      call newlin
      call closod
      stop
      end
C   odm
C
C   initialize output device
C
      block data
      implicit complex (a-z)
      character*60 line
      integer linlen, linpos
      logical odop
      common /odmblk/linpos,odop,line
      data linpos/1/,odop/.false./
      end
C
C   odm
C
C   open output device
C
      subroutine openod
      implicit complex (a-z)
      character*60  line
      integer linlen,linpos
      logical odop
      common /odmblk/linpos,odop,line
      if (odop) call ueropn('odm','openod')
      odop=.true.
      return
      end
```

```
C
C   odm
C
C   close output file
C
       subroutine closod
       implicit complex (a-z)
       character*60  line
       integer linpos,linlen
       logical odop
       common /odmblk/linpos,odop,line
       if (.not.odop) call uercls('odm','closod')
       odop=.false.
       return
       end
C
C   odm
C
C   write character to output stream
C
       subroutine wrchar(chr)
       implicit complex (a-z)
       character*1 chr
       character*60  line
       integer linpos, linlen
       logical odop
       common /odmblk/linpos,odop,line
       if (.not.odop) call uewrcl('odm','wrchar')
       call setchr(line,linpos,chr)
       linpos=linpos+1
       if (linpos.le.60)return
       write(0,100,err=20)line
       linpos=1
       return
20     call uedver('odm','wrchar')
       return
100    format (1x,a60)
       end
```

```
C
C   odm
C
C   write current line unless empty and get new line
C
      subroutine newlin
      implicit complex (a-z)
      character*1 line(60), ch1
      integer linpos,linlen,i
      logical odop
      common /odmblk/linpos,odop,line
      if (.not.odop) call uewrc1('odm','wrchar')
      if (linpos.eq.1) return
      write (0,100,err=20) (getchr(line,i),i=1,linpos-1)
      linpos=1
      return
20    call uedver('odm','newlin')
      return
100   format (1x,60a)
      end
C
C   opm
C
C   write address adr
C
      subroutine wradr(adr)
      implicit complex (a-z)
      character*30 getcoa
      character*20 getcit,getln
      character*20 getgn,getboc
      character*10 gettit
      character*9 getzip
      character*2 getst
      integer maxads,getnca,mads,adr,i
      mads=maxads()
      if (adr.lt.1.or.adr.gt.mads)
     & call ueaida('opm','wrads')
      if (getnca().lt.adr.and.adr.le.mads)
     &call ueaidu('opm','wradr')
      call newlin
      call wrfld(gettit(adr))
      call wrchar(' ')
      call wrfld(getgn(adr))
      call wrchar(' ')
      call wrfld(getln(adr))
      call newlin
      call wrfld(getboc(adr))
      call newlin
      call wrfld(getcoa(adr))
      call newlin
```

```
        call wrfld(getcit(adr))
        call wrchar(',')
        call wrchar(' ')
        call wrfld(getst(adr))
        call wrchar(' ')
        call wrfld(getzip(adr))
        call newlin
        do 10 i=1,5
          call wrchar(' ')
          call newlin
10     continue
        return
        end
C
C    opm
C
C    write a field
C
        subroutine wrfld(str)
        implicit complex (a-z)
        character *(*) str
        character*1 getchr
        integer nblank,i,n
        n=nblank(str)
        if (n.gt.0) go to 5
        do 4 i=1,5
4      call wrchar('*')
        return
5      do 10 i=1,n
          call wrchar(getchr(str,i))
10     continue
        return
        end
C    ssm
C
C    replace character at position pos of str by chr
C
        subroutine setchr(str,pos,chr)
        implicit complex (a-z)
        character *1 chr
        character *(*) str
        integer pos,len
        if (pos.lt.1.or.pos.gt.len(str)) call uespos('ssm','setchr')
        str(pos:pos)=chr
        return
        end
```

```
C
C   ssm
C
C   get character at position pos of str
C
      character *1 function getchr(str,pos)
      implicit complex (a-z)
      character *(*) str
      integer len,pos
      if (pos.lt.1.or.pos.gt.len(str)) call uespos('ssm','getchr')
      getchr=str(pos:pos)
      return
      end
C
C   ssm
C
C   create substring of sir
C
      character *(*) function substr(str,pos,length)
      implicit complex (a-z)
      character *(*) str
      integer  pos,length, len, strlen, uppos
      strlen = len(str)
      uppos = pos+length-1
      if (pos.lt.1.or.pos.gt.strlen) call uespos('ssm','substr')
      if (length.lt.0.or.uppos.gt.strlen)
     &call ueslen('ssm','substr')
      substr = ' '
      if (length.eq.0) return
      substr=str(pos:uppos)
      return
      end
C
C   ssm
C
C   string equality test
C
      logical function streq(str1,str2)
      implicit complex (a-z)
      character*(*) str1,str2
      integer 1,len,11,12,min,i
      logical chareq
      11=len(str1)
      12=len(str2)
      1=min(11,12)
      streq=.false.
```

```
       do 10 i=1,1
         if (.not. chareq(str1(i:i),str2(i:i)))return
10     continue
       if (11-12) 100,200,300
100    do 110 i=11+1,12
              if (.not.chareq(str2(i:i),' '))return
110    continue
       streq=.true.
       return
200    streq=.true.
       return
300    do 310 i=12+1,11
             if (.not.chareq(str1(i:i),' '))return
310    continue
       streq= .true.
       return
       end
C
C   ueh
C
C   data definitions for ue handlers
C
       block data
       implicit complex (a-z)
       character*3 mids
       integer nmods
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       data mids/'apm','asm','chm','idm','ipm','mcm','odm','opm',
     &'ssm'/
       end
C
C   ueh
C
C   handler of ue: address storage capacity overflow
C
       subroutine ueadov(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','adov')
30     write(0,100) fnid,mdid
100    format (/' *** the ue: addresss storage capacity overflow '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
```

```
C
C   ueh
C
C   handler for ue: partially defined address
C
       subroutine ueadrp(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20    continue
       call uemidu('ueh','adrp')
30    write(0,100) fnid,mdid
100   format (/' *** the ue: partially defined address '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
C
C   ueh
C
C   handler for ue: undefined address
C
       subroutine ueadru(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20    continue
       call uemidu('ueh','adru')
30    write (0,100) fnid,mdid
100   format (/' *** the ue: undefined address '/
     &' detected in function 'a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
```

```
C
C   ueh
C
C   handler for ue: absurd address identifier
C
       subroutine ueaida(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','aida')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: absurd address identifier '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
C
C   ueh
C
C   handler for ue: unassigned address identifier
C
       subroutine ueaidu(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','aidu')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: unassigned address identifier '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
```

```
C
C   ueh
C
C   handler for ue: state of asm incorrect
C
       subroutine ueasmi(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','asmi')
30     write (0,100) fnid,mdid
100    format (/' *** ue: state of asm incorrect '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
C
C   ueh
C
C   handler for ue: undefined character comparison
C
       subroutine uechlt(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','chlt')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: undefined character comparison '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
```

```
C
C   ueh
C
C   handler for ue: device error
C
        subroutine uedver(mdid,fnid)
        implicit complex (a-z)
        character*3 mids
        character*(*) mdid,fnid
        integer nmods,j
        parameter (nmods=9)
        common /uehblk/ mids(nmods)
        do 20 j=1,nmods
          if (mdid.eq.mids(j)) go to 30
20     continue
        call uemidu('ueh','dver')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: device error '/
     &' detected in function ',a,' of module 'a,'.'/
     &' execution terminated.')
        call error (' ')
        stop
        end
C
C   ueh
C
C   handler for ue: non-existent module identifier
C
        subroutine uemidu(mdid,fnid)
        implicit complex (a-z)
        character*(*) mdid,fnid
30     write (0,100)fnid,mdid
100    format (/' *** the ue: non-existent module identifier '/
     &' detected in function ',a,' of module 'a,'.'/
     &' execution terminated.')
        call error (' ')
        stop
        end
```

```
C
C   ueh
C
C   handler for ue: number of complete addresses undefined
C
       subroutine uencau(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','ncau')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: number of complete addresses undefined '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
C
C   ueh
C
C   handler for ue: no chars available on input device
C
       subroutine uenoch(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','noch')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: no chars available on input device '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
```

```
C
C   ueh
C
C   handler for ue: 0-grade level , 1 or . 10
C
       subroutine ueogl(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20    continue
       call uemidu('ueh','ogl')
30    write (0,100) fnid,mdid
100   format (/' *** the ue: 0-grade level , 1 or . 10 '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
C
C   ueh
C
C   handler for ue: redundant device closing
C
       subroutine uercls(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20    continue
       call uemidu('ueh','rcls')
30    write (0,100) fnid,mdid
100   format (/' *** the ue: redundant device closing '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
```

```
C
C   ueh
C
C   handler for ue: redundant device opening
       subroutine ueropn(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','ropn')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: redundant device opening '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
C
C   ueh
C
C   handler for ue: substring length illegal
C
       subroutine ueslen(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','slen')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: substring length illegal '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
```

```
C
C   ueh
C
C   handler for ue: string character position illegal
C
       subroutine uespos(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','spos')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: string character position illegal '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
C
C   ueh
C
C   handler for ue: write/read on closed device
C
       subroutine uewrcl(mdid,fnid)
       implicit complex (a-z)
       character*3 mids
       character*(*) mdid,fnid
       integer nmods,j
       parameter (nmods=9)
       common /uehblk/mids(nmods)
       do 20 j=1,nmods
         if (mdid.eq.mids(j)) go to 30
20     continue
       call uemidu('ueh','wrcl')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: write/read on closed device '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
       call error (' ')
       stop
       end
```

```
C
C   ueh
C
C   handler for ue: zip area part not 3 dec digs
C
        subroutine uezip(mdid,fnid)
        implicit complex (a-z)
        character*3 mids
        character*(*) mdid, fnid
        integer nmods,j
        parameter (nmods=9)
        common /uehblk/mids(nmods)
        do 20 j=1,nmods
          if (mdid.eq.mids(j)) go to 30
20     continue
        call uemidu('ueh','zip')
30     write (0,100) fnid,mdid
100    format (/' *** the ue: zip area part not 3 dec digs '/
     &' detected in function ',a,' of module ',a,'.'/
     &' execution terminated.')
        call error (' ')
        stop
        end
```

HAS.1 The Host-At-Sea (HAS) Buoy System

EXAMPLE DESCRIPTION

Introduction

The Navy intends to deploy HAS buoys to provide navigation and weather data to air and ship traffic at sea. The buoys will collect wind, temperature, and location data, and will broadcast summaries periodically. Passing vessels will be able to request more detailed information. In addition, HAS buoys will be deployed in the event of accidents at sea to aid sea-search operations.

Rapid deployment and the use of disposable equipment are novel features of HAS. HAS buoys will be relatively inexpensive, lightweight systems that may be deployed by being dropped from low-flying aircraft. It is expected that many of the HAS buoys will disappear because of equipment deterioration, bad weather conditions, accidents, or hostile action. The ability to redeploy rather than to attempt to prevent such loss is the key to success in the HAS program. In this sense, HAS buoys will be disposable equipment. To keep costs down, government surplus components will be used as much as possible.

Hardware

Each HAS buoy will contain a small computer, a set of wind and temperature sensors, and a radio receiver and transmitter. Eventually, a variety of special purpose HAS buoys may be configured with different types of sensors, such as wave spectra sensors. Although these will not be covered by the initial procurement, provision for future expansion is required.

The HAS-BEEN computer has been chosen for the HAS buoy program. There are more than 3000 of these available as government-surplus equipment. They were originally developed as the standard computer for a balloon force (High Altitude Surveying, or HAS), which is now defunct. Known as the Balloon Internal Navigator, they were originally called HAS-BIN computers; the spelling was corrected in 1976 as part of a presidential program to remove "redneckisms" from government documents.

The HAS-BEEN computer has been found suitable for the new HAS program by virtue of its low weight, low cost, low power consumption, and nomenclature. A preliminary study shows that the capacity of a single BEEN computer will be insufficient for some HAS configurations, but it has been decided to use two or more BEEN computers in these cases. Therefore, provision for multi-processing is required in the software.

The HAS-BEEN computer has a typical complement of full-word integer instructions. Input is performed by a SNS (SENSE) instruction that selects a device and stores the contents of its control register at a designated core

location. Up to 256 different sensors may be connected, and the first 256 core locations are available for depositing the results. The device and corresponding core location are addressed by an 8-bit field in the SNS instruction.

The temperature sensors take air and water temperature (Centigrade). On some HAS buoys, an array of sensors on a cable will be used to take water temperature at various depths.

Because the surplus temperature sensors selected for HAS are not designed for sea-surface conditions, the error range on individual readings may be large. Preliminary experiments indicate that the temperature can be measured within an acceptable tolerance by averaging several readings from the same device. To improve the accuracy further and to guard against sensor failure, most HAS buoys will have multiple temperature sensors.

Each buoy will have one or more wind sensors to observe wind magnitude in knots and wind direction. Surplus propellor-type sensors have been selected because they meet power restrictions.

Buoy geographic position is determined by use of a radio receiver link with the Omega navigation system.

Some HAS buoys are also equipped with a red light and an emergency switch. The red light may be made to flash by a request radioed from a vessel during a sea-search operation. If the sailors are able to reach the buoy, they may flip the emergency switch to initiate SOS broadcasts from the buoy.

Software Functions

The software for the HAS buoy must carry out the following functions:

1. Maintain current wind and temperature information by monitoring sensors regularly and averaging readings.

2. Calculate location via the Omega navigation system.

3. Broadcast wind and temperature information every 60 seconds.

4. Broadcast more detailed reports in response to requests from passing vessels. The information broadcast and the data rate will depend on the type of vessel making the request (ship or airplane). All requests and reports will be transmitted in the RAINFORM format.

5. Broadcast weather history information in response to requests from ships or satellites. The history report consists of the periodic 60-second reports from the last 48 hours.

6. Broadcast an SOS signal in place of the ordinary 60-second message after a sailor flips the emergency switch. This should continue until a vessel sends a reset signal.

7. Accept external update data. Although HAS buoys calculate their own position, they must also accept correction information from passing vessels. The software must use the information to update its internal database. Major discrepancies must cause it to invoke elaborate self diagnostics to attempt to eliminate the errors in future calculations.

8. Perform periodic built-in test (BIT) checks. The software should be able to detect and compensate for memory or computer-function failures. Also, the many sensors of a HAS host are relatively easily damaged and may be providing erroneous data. There should be sufficient sensors to provide reasonableness checks and to allow compensation for those found to be inconsistent or biased. Those found to be nonfunctioning can be ignored in future calculations.

Specifically, the following BIT checks are deemed necessary:

(a) Basic computer function test.

This test is designed to check the most frequently used functions of the computer. It checks arithmetic and control operations and all fast registers. It should be repeated every 350 ms.

(b) Extended computer function test.

This program makes more extensive tests on the basic computer, plus checking less central functions such as I/O and shifts. It should be completed at least once every 5000 ms.

(c) Computer memory function test.

Each word in the memory must be checked by storing and reading all zero, all one, and alternating zero-one bit patterns. A complete check of a 10000 word memory should be completed every 15 minutes.

(d) Sensor consistency tests.

Although each of the sensors provides data independently, there are known constraints on the reasonable relationships that they can have to each other. For example, the many temperature readings can be expected to remain within a few degrees of each other and not to change by more than 20 degrees in 30 minutes. Other sensors such as wind sensors, contain provision for calibration readings. Checks of all wind sensors should be made every 10 minutes. Consistency checks of temperature sensors should be completed every 5 minutes.

Response to Detected Failures

The software is expected to function without noticeable degradation with damage to up to 20% of the sensors. If more than 20% of the sensors are improperly functioning, both periodic and request reports should be marked

"suspect." In the event that the data are considered unusable (e.g., more than 50% of the sensors found malfunctioning), a "defective" report should be sent in place of the suspect data.

In the event that BIT detects malfunctioning of a few specific commands, their simulation by means of sequences of other commands (e.g., simulation of subtraction using addition and negation) should be attempted.

Where areas of memory are found defective, functioning with reduced memory should be attempted. If no more than 10% of memory is defective, relocation without loss of function can be attempted. If more memory is defective, deletion of air temperature calculations should be the first step. Relocation should then allow the performance of the remaining functions.

Software Timing Requirements

In order to maintain accurate information, readings must be taken from the sensing devices at the following fixed intervals:

temperature sensors: every 10 seconds

wind sensors: every 30 seconds

Omega signals: every 10 seconds.

Since the buoy can only transmit one report at a time, conflicts will arise.

If the transmitter is free and more than one report is ready, the next report will be chosen according to the following priority ranking:

SOS	1	highest
Airplane Request	2	
Ship/Satellite Request	3	
Periodic	4	
History	5	lowest

Program Generation

HAS host programs will be generated at the HAS Program Generation Center (NAVHASPGC) located at Chesapeake Beach, Maryland. A NAVHASPGCPAC is also planned for eventual location in Monterey, California. Since different HAS buoys may carry different sets of sensors, HAS-BEEN programs may be different. The software to be procured must include a system generator. To generate a specific program, a configuration (number of sensors of each type) will be described and generation of the program should then be automatic.

HAS.2 HAS Data Acquisition and Transmission Software: Program Design Specification

EXAMPLE DESCRIPTION

COMPUTER SYSTEMS DISTRIBUTORS, INC.

O. U. DeZeeman
Cognizant Software Engineer

R. E. Tired
Contract Liaison Officer

Scope

This document is a detailed description of CSD's proposed design for the HAS system software. The reader is assumed to be familiar with the HAS system functions as described in The Host-At-Sea System (HAS) Buoy System (Document HAS.1).

For a variety of reasons, the document does not assume detailed knowledge of the HAS-BEEN computer, which is GFE for this project. CSD has already expressed its opinion that the HAS-BEEN computer is not ideal for the job. Working together with one of our sister firms, CHIP Corporation, we have proposed a specially designed microprocessor that is ideal for the job. In order to allow the Navy more time for a decision, we have prepared our design in a machine-independent form. However, it has been necessary to recognize two limitations of the HAS-BEEN computer at this early stage of the design process.

(1) HAS-BEEN has no interrupt system. Our design calls for periodic polling of sensors.

(2) HAS-BEEN has no instructions that are particularly useful in subroutine calls. For that reason, we have avoided subroutine calls in many places where we might have used them.

In spite of the effects of these two limitations, we believe that our design is also applicable to the CHIP computer.

Documentation Approach

"Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious." (Brooks 1975, p. 102)

Believing that data structure dictates program structure, we frequently reference a description of the Common Data Base (CDB), the data structure that keeps track of the state of HAS. A complete description of all of the data items in the system appears in Appendix I (p. 13-10) of this document. We find reference to the CDB description to be of great value in understanding the algorithms used in the system. A representative sample of the algorithms are documented in Appendix II (p. 13-23), using a self-evident, ALGOL-like pseudo code that we believe everyone can readily understand.

The proposed design divides the HAS software into functional modules, each to be constructed by a separate group of programmers. The remainder of this document describes each module separately and then discusses intermodule cooperation.

Module Overview

For the moment, we will ignore the time constraints on the HAS software and instead will describe only the system functions performed by each module. The modules are described below:

Sensor Reading

Each sensor attached to the HAS system is controlled by one sensor-reading module. Whenever the module polls the sensor, the value obtained is converted to engineering units and stored in a location in the CDB.

Averaging

Unweighted time averages are computed for all sensors that are prone to large errors. One averaging module exists for each of these sensors. The averages are stored in the CDB.

Multisensor Averaging

These modules compute averages of readings from more than one sensor. Depending on the type of the sensors being averaged, the readings used are either raw sensor readings (after conversion to engineering units) or time averages. In either case, the readings are obtained from the CDB and the computed averages are stored in the CDB.

Omega Location Calculator

This module obtains Omega data from the CDB, uses the data to compute the current location, and stores the location into the CDB for use by other modules.

Record Updating

This module maintains the 48-hour history in the CDB. Each time it is started, sensor, location, and time values are copied from their locations in the CDB into the appropriate history locations in the CDB.

Receiving Module

This module controls the radio receiving equipment. It scans assigned frequencies for indication of a message transmission, receives the message, and stores it into the CDB for later interpretation.

Message Interpretation

Messages stored in the CDB are parsed, and the module responsible for responding to the message is initiated.

Report Generator

One report generator module exists for each of the five types of reports. Each module is aware of the priority of its type of report with respect to the other types and uses this information to ensure that reports are broadcast according to the prescribed priority ranking. Additionally, each module is able to access the readings it needs in the CDB and to control the transmitting equipment used to broadcast the report.

Location Verification

This module is initiated by the Message Interpretation module when a passing vessel supplies location information. The location information is used to validate Omega location calculations stored in the CDB. Error recovery is attempted if any discrepancy is larger than a specified tolerance.

Buoy Device Control

This module reads any external switch settings on the buoy and controls the operational emergency beacon.

Intermodule Cooperation

Owing to the limitations of the HAS-BEEN computer, we have designed the HAS software as a set of cooperative modules. All intermodule communication is through the CDB. Each of the modules keeps track of real time and is aware of the deadlines of the other modules. They transfer control to each other according to the urgency of the situation. Where several modules are able to process data, and none has an urgent deadline, a fair round-robin scheduling strategy is used. Each module performs this task itself because the HAS-BEEN computer does not contain the preemption circuitry that the more desirable CHIP computer would contain.

When a module needs data produced by another module (a sequencing requirement) or the use of some resource, the requesting module tests a variable in the CDB to determine the status of the resource (data). If the resource is not available, the requesting module sets the variable to indicate that it needs the associated resource and transfers control to another module.

The transfer of control is effected by using Module Control Blocks (MCBs) that are linked into a set of FIFO queues. The MCB, contained in the CDB, serves two purposes: (1) it holds state information such as register contents and the PC, and (2) it contains queue pointers. The module relinquishing control inserts its MCB into a FIFO queue associated with the resource it needs, saves its state into its MCB, selects a module to start, and loads the machine state from the selected module's MCB.

At any given time, it is likely that many modules will be selectable, i.e., all of the resources they require will be available. One of these modules must be selected on the basis of the urgency of the task it performs. Urgency is represented by a dynamically changing module priority; the more urgent the task, the higher is the priority. A FIFO queue of MCBs is associated with each priority level; the selection process is simply to select the first MCB from the highest priority nonempty queue.

The alert reader now has two questions: (1) how do the MCBs get into the selection queues in the first place, and (2) what enforces polling and other real-time deadlines? In order to answer these questions, let us consider a "snapshot" of the system in action. There is one currently executing module, a number of modules waiting for a resource to become available, and some modules that are ready to use the CPU. The MCBs for the resource-blocked modules are in the queue associated with the resource (as described above); the MCBs for the modules needing only the CPU are in selection queues. When the currently executing module makes a resource available (by creating data or no longer needing an actual resource), it moves the first MCB on the resource queue to the selection queue for the priority level stored in the MCB and adjusts the variable associated with the resource to indicate that the resource is available.

We now consider the enforcement of real-time deadlines. Since the HAS-BEEN computer has no hardware interrupts with which to signal that an event needs to occur, the code in each module must frequently read the real-time clock and determine whether any modules need to be run at that time. If there are any, those modules' MCBs are moved to the appropriate selection queues. In any case, the MCB for the currently executing module is moved to the end of the selection queue that it currently resides in, and the module transfers control as described above. The movement (deletion and reinsertion) of the MCB for the currently executing module assures a round-robin scheduling strategy for equal priority modules since all queues are FIFO.

The CDB includes a Time Control Table (TCT) containing a list of time deadlines. A queue of MCBs is associated with each deadline. Using these data structures, each module need only compare the clock time with the TCT

deadlines, put the associated MCBs on the proper selection queue, select a module, and transfer control to the selected module.

Reentrant procedures are used to save space by avoiding duplicate code. Therefore, a mechanism is needed for providing a separate copy of all private variables for each invocation of a reentrant procedure. The maximum number of invocations of each procedure is determined when the system is constructed. Therefore, we will use an array for each private variable; one array element corresponds to one invocation of the procedure. The invocation number is assigned by the reentrant procedure call mechanism (by use of a bit string for each procedure), and is stored in the INV# field of the module's MCB. The previous value of INV# is saved in the CINV# array in the Private Variable Area (PVA) for the procedure; it is restored when the procedure returns.

The modules performing background tasks are given a low priority, and therefore never execute unless there is extra CPU time. The report priorities described in the module functional descriptions are enforced by the modules themselves.

Conclusion

We believe the design presented in this document to be the best possible design given the constraints imposed by the limitations of the HAS-BEEN computer. It is also applicable to the more suitable CHIP computer. The CDB provides a clean, precisely specified intermodule interface that is system wide.

APPENDIX I. Common Data Base

Contents

FWDPTR (1) = 0	FWDPTR (2) = 0	FWDPTR (3) = 0	FWDPTR (4) = 0	FWDPTR (5) = 0	FWDPTR (6) = 0	FWDPTR (7) = 0	FWDPTR (8) = 0
FWDPTR (9) = 0	FWDPTR (10) = 0	FWDPTR (11) = 0	FWDPTR (12) = 0	FWDPTR (13) = 0	FWDPTR (14) = 0	FWDPTR (15) = 0	FWDPTR (16) = 0
FWDPTR (17) = 0	FWDPTR (18) = 0	FWDPTR (19) = 0	FWDPTR (20) = 0	FWDPTR (21) = 0	FWDPTR (22) = 0	FWDPTR (23) = 0	FWDPTR (24) = 0
FWDPTR (25) = 0	FWDPTR (26) = 0	FWDPTR (27) = 0	FWDPTR (28) = 0	FWDPTR (29) = 0	FWDPTR (30) = 0	FWDPTR (31) = 0	FWDPTR (32) = 0
FWDPTR (33) = 0	FWDPTR (34) = 0	FWDPTR (35) = 0	FWDPTR (36) = 0	FWDPTR (37) = 0	FWDPTR (38) = 0	FWDPTR (39) = 0	FWDPTR (40) = 0
FWDPTR (41) = 0	FWDPTR (42) = 0	FWDPTR (43) = 0	FWDPTR (44) = 0	FWDPTR (45) = 0	FWDPTR (46) = 0	FWDPTR (47) = 0	FWDPTR (48) = 0
FWDPTR (49) = 0	FWDPTR (50) = 0	FWDPTR (51) = 0	FWDPTR (52) = 0	FWDPTR (53) = 0	FWDPTR (54) = 0	BACKPTR (1) = 0	BACKPTR (2) = 0
BACKPTR (3) = 0	BACKPTR (4) = 0	BACKPTR (5) = 0	BACKPTR (6) = 0	BACKPTR (7) = 0	BACKPTR (8) = 0	BACKPTR (9) = 0	BACKPTR (10) = 0
BACKPTR (11) = 0	BACKPTR (12) = 0	BACKPTR (13) = 0	BACKPTR (14) = 0	BACKPTR (15) = 0	BACKPTR (16) = 0	BACKPTR (17) = 0	BACKPTR (18) = 0
BACKPTR (19) = 0	BACKPTR (20) = 0	BACKPTR (21) = 0	BACKPTR (22) = 0	BACKPTR (23) = 0	BACKPTR (24) = 0	BACKPTR (25) = 0	BACKPTR (26) = 0
BACKPTR (27) = 0	BACKPTR (28) = 0	BACKPTR (29) = 0	BACKPTR (30) = 0	BACKPTR (31) = 0	BACKPTR (32) = 0	BACKPTR (33) = 0	BACKPTR (34) = 0
BACKPTR (35) = 0	BACKPTR (36) = 0	BACKPTR (37) = 0	BACKPTR (38) = 0	BACKPTR (39) = 0	BACKPTR (40) = 0	BACKPTR (41) = 0	BACKPTR (42) = 0
BACKPTR (43) = 0	BACKPTR (44) = 0	BACKPTR (45) = 0	BACKPTR (46) = 0	BACKPTR (47) = 0	BACKPTR (48) = 0	BACKPTR (49) = 0	BACKPTR (50) = 0
BACKPTR (51) = 0	BACKPTR (52) = 0	BACKPTR (53) = 0	BACKPTR (54) = 0	PRIORITY (1) = 0	PRIORITY (2) = 0	PRIORITY (3) = 0	PRIORITY (4) = 0
PRIORITY (5) = 0	PRIORITY (6) = 0	PRIORITY (7) = 0	PRIORITY (8) = 0	PRIORITY (9) = 0	PRIORITY (10) = 0	PRIORITY (11) = 0	PRIORITY (12) = 0
PRIORITY (13) = 0	PRIORITY (14) = 0	PRIORITY (15) = 0	PRIORITY (16) = 0	PRIORITY (17) = 0	PRIORITY (18) = 0	PRIORITY (19) = 0	PRIORITY (20) = 0
PRIORITY (21) = 0	PRIORITY (22) = 0	PRIORITY (23) = 0	PRIORITY (24) = 0	PRIORITY (25) = 0	PRIORITY (26) = 0	PRIORITY (27) = 0	PRIORITY (28) = 0
PRIORITY (29) = 0	PRIORITY (30) = 0	PRIORITY (31) = 0	PRIORITY (32) = 0	PRIORITY (33) = 0	PRIORITY (34) = 0	PRIORITY (35) = 0	PRIORITY (36) = 0
PRIORITY (37) = 0	PRIORITY (38) = 0	PRIORITY (39) = 0	PRIORITY (40) = 0	PRIORITY (41) = 0	PRIORITY (42) = 0	PRIORITY (43) = 0	PRIORITY (44) = 0
PRIORITY (45) = 0	PRIORITY (46) = 0	PRIORITY (47) = 0	PRIORITY (48) = 0	PRIORITY (49) = 0	PRIORITY (50) = 0	PRIORITY (51) = 0	PRIORITY (52) = 0

PRIORITY (53) = 0	PRIORITY (54) = 0	DTIME (1) = 0	DTIME (2) = 0	DTIME (3) = 0	DTIME (4) = 0	DTIME (5) = 0	DTIME (6) = 0
DTIME (7) = 0	DTIME (8) = 0	DTIME (9) = 0	DTIME (10) = 0	DTIME (11) = 0	DTIME (12) = 0	DTIME (13) = 0	DTIME (14) = 0
DTIME (15) = 0	DTIME (16) = 0	DTIME (17) = 0	DTIME (18) = 0	DTIME (19) = 0	DTIME (20) = 0	DTIME (21) = 0	DTIME (22) = 0
DTIME (23) = 0	DTIME (24) = 0	DTIME (25) = 0	DTIME (26) = 0	DTIME (27) = 0	DTIME (28) = 0	DTIME (29) = 0	DTIME (30) = 0
DTIME (31) = 0	DTIME (32) = 0	DTIME (33) = 0	DTIME (34) = 0	DTIME (35) = 0	DTIME (36) = 0	DTIME (37) = 0	DTIME (38) = 0
DTIME (39) = 0	DTIME (40) = 0	DTIME (41) = 0	DTIME (42) = 0	DTIME (43) = 0	DTIME (44) = 0	DTIME (45) = 0	DTIME (46) = 0
DTIME (47) = 0	DTIME (48) = 0	DTIME (49) = 0	DTIME (50) = 0	DTIME (51) = 0	DTIME (52) = 0	DTIME (53) = 0	DTIME (54) = 0
MCBPC (1) = 0	MCBPC (2) = 0	MCBPC (3) = 0	MCBPC (4) = 0	MCBPC (5) = 0	MCBPC (6) = 0	MCBPC (7) = 0	MCBPC (8) = 0
MCBPC (9) = 0	MCBPC (10) = 0	MCBPC (11) = 0	MCBPC (12) = 0	MCBPC (13) = 0	MCBPC (14) = 0	MCBPC (15) = 0	MCBPC (16) = 0
MCBPC (17) = 0	MCBPC (18) = 0	MCBPC (19) = 0	MCBPC (20) = 0	MCBPC (21) = 0	MCBPC (22) = 0	MCBPC (23) = 0	MCBPC (24) = 0
MCBPC (25) = 0	MCBPC (26) = 0	MCBPC (27) = 0	MCBPC (28) = 0	MCBPC (29) = 0	MCBPC (30) = 0	MCBPC (31) = 0	MCBPC (32) = 0
MCBPC (33) = 0	MCBPC (34) = 0	MCBPC (35) = 0	MCBPC (36) = 0	MCBPC (37) = 0	MCBPC (38) = 0	MCBPC (39) = 0	MCBPC (40) = 0
MCBPC (41) = 0	MCBPC (42) = 0	MCBPC (43) = 0	MCBPC (44) = 0	MCBPC (45) = 0	MCBPC (46) = 0	MCBPC (47) = 0	MCBPC (48) = 0
MCBPC (49) = 0	MCBPC (50) = 0	MCBPC (51) = 0	MCBPC (52) = 0	MCBPC (53) = 0	MCBPC (54) = 0	MCBACC (1) = 0	MCBACC (2) = 0
MCBACC (3) = 0	MCBACC (4) = 0	MCBACC (5) = 0	MCBACC (6) = 0	MCBACC (7) = 0	MCBACC (8) = 0	MCBACC (9) = 0	MCBACC (10) = 0
MCBACC (11) = 0	MCBACC (12) = 0	MCBACC (13) = 0	MCBACC (14) = 0	MCBACC (15) = 0	MCBACC (16) = 0	MCBACC (17) = 0	MCBACC (18) = 0
MCBACC (19) = 0	MCBACC (20) = 0	MCBACC (21) = 0	MCBACC (22) = 0	MCBACC (23) = 0	MCBACC (24) = 0	MCBACC (25) = 0	MCBACC (26) = 0
MCBACC (27) = 0	MCBACC (28) = 0	MCBACC (29) = 0	MCBACC (30) = 0	MCBACC (31) = 0	MCBACC (32) = 0	MCBACC (33) = 0	MCBACC (34) = 0
MCBACC (35) = 0	MCBACC (36) = 0	MCBACC (37) = 0	MCBACC (38) = 0	MCBACC (39) = 0	MCBACC (40) = 0	MCBACC (41) = 0	MCBACC (42) = 0
MCBACC (43) = 0	MCBACC (44) = 0	MCBACC (45) = 0	MCBACC (46) = 0	MCBACC (47) = 0	MCBACC (48) = 0	MCBACC (49) = 0	MCBACC (50) = 0

MCBACC (51) = 0	MCBACC (52) = 0	MCBACC (53) = 0	MCBACC (54) = 0	MCBIOR (1) = 0	MCBIOR (2) = 0	MCBIOR (3) = 0	MCBICB (4) = 0
MCBIOR (5) = 0	MCBIOR (6) = 0	MCBIOR (7) = 0	MCBIOR (8) = 0	MCBIOR (9) = 0	MCBIOR (10) = 0	MCBIOR (11) = 0	MCBICB (12) = 0
MCBIOR (13) = 0	MCBIOR (14) = 0	MCBIOR (15) = 0	MCBIOR (16) = 0	MCBIOR (17) = 0	MCBIOR (18) = 0	MCBIOR (19) = 0	MCBICB (20) = 0
MCBIOR (21) = 0	MCBIOR (22) = 0	MCBIOR (23) = 0	MCBIOR (24) = 0	MCBIOR (25) = 0	MCBIOR (26) = 0	MCBIOR (27) = 0	MCBICB (28) = 0
MCBIOR (29) = 0	MCBIOR (30) = 0	MCBIOR (31) = 0	MCBIOR (32) = 0	MCBIOR (33) = 0	MCBIOR (34) = 0	MCBIOR (35) = 0	MCBICB (36) = 0
MCBIOR (37) = 0	MCBIOR (38) = 0	MCBIOR (39) = 0	MCBIOR (40) = 0	MCBIOR (41) = 0	MCBIOR (42) = 0	MCBIOR (43) = 0	MCBICB (44) = 0
MCBIOR (45) = 0	MCBIOR (46) = 0	MCBIOR (47) = 0	MCBIOR (48) = 0	MCBIOR (49) = 0	MCBIOR (50) = 0	MCBIOR (51) = 0	MCBICB (52) = 0
MCBIOR (53) = 0	MCBIOR (54) = 0	INV# (1) = 0	INV# (2) = 0	INV# (3) = 0	INV# (4) = 0	INV# (5) = 0	INV# (6) = 0
INV# (7) = 0	INV# (8) = 0	INV# (9) = 0	INV# (10) = 0	INV# (11) = 0	INV# (12) = 0	INV# (13) = 0	INV# (14) = 0
INV# (15) = 0	INV# (16) = 0	INV# (17) = 0	INV# (18) = 0	INV# (19) = 0	INV# (20) = 0	INV# (21) = 0	INV# (22) = 0
INV# (23) = 0	INV# (24) = 0	INV# (25) = 0	INV# (26) = 0	INV# (27) = 0	INV# (28) = 0	INV# (29) = 0	INV# (30) = 0
INV# (31) = 0	INV# (32) = 0	INV# (33) = 0	INV# (34) = 0	INV# (35) = 0	INV# (36) = 0	INV# (37) = 0	INV# (38) = 0
INV# (39) = 0	INV# (40) = 0	INV# (41) = 0	INV# (42) = 0	INV# (43) = 0	INV# (44) = 0	INV# (45) = 0	INV# (46) = 0
INV# (47) = 0	INV# (48) = 0	INV# (49) = 0	INV# (50) = 0	INV# (51) = 0	INV# (52) = 0	INV# (53) = 0	INV# (54) = 0

CDB - MODULE CONTROL BLOCK AREA (MCBA)

DLIST (1) = 0	DLIST (2) = 0	DLIST (3) = 0	DLIST (4) = 0	DLIST (5) = 0	DLIST (6) = 0	DLIST (7) = 0	DLIST (8) = 0
DLIST (9) = 0	DLIST (10) = 0	DLIST (11) = 0	DLIST (12) = 0	DLIST (13) = 0	DLIST (14) = 0	DLIST (15) = 0	DLIST (16) = 0
DLIST (17) = 0	DLIST (18) = 0	DLIST (19) = 0	DLIST (20) = 0	IAFRCV = 0	HAFRCV = 0	TAFRCV = 0	ISFRCV = 0
HSFRCV = 0	TSFRCV = 0	ISATFRCV = 0	HSATFRCV = 0	TSATFRCV = 0	IATS (1) = 0	HATS (1) = 0	TATS (1) = 0
IATS (2) = 0	HATS (2) = 0	TATS (2) = 0	IATS (3) = 0	HATS (3) = 0	TATS (3) = 0	IWSS (1) = 0	HWSS (1) = 0
TWSS (1) = 0	IWSS (2) = 0	HWSS (2) = 0	TWSS (2) = 0	IWSS (3) = 0	HWSS (3) = 0	TWSS (3) = 0	IWSS (4) = 0
HWSS (4) = 0	TWSS (4) = 0	IWSS (5) = 0	HWSS (5) = 0	TWSS (5) = 0	IWDS (1) = 0	HWDS (1) = 0	TWDS (1) = 0
IWDS (2) = 0	HWDS (2) = 0	TWDS (2) = 0	IWDS (3) = 0	HWDS (3) = 0	TWDS (3) = 0	IWDS (4) = 0	HWDS (4) = 0
TWDS (4) = 0	IWDS (5) = 0	HWDS (5) = 0	TWDS (5) = 0	IOBS = 0	HOBS = 0	TOBS = 0	IWT1S = 0
HWT1S = 0	TWT1S = 0	IWT2S = 0	HWT2S = 0	TWT2S = 0	IWT3S = 0	HWT3S = 0	TWI3S = 0

CDB - TIME CONTROL TABLE (TCT)

NEXTOBS (1) = 0	NEXTOBS (2) = 0	NEXTOBS (3) = 0	NEXTOBS (4) = 0	NEXTOBS (5) = 0	NEXTOBS (6) = 0	SUM (1) = 0	SUM (2) = 0
SUM (3) = 0	SUM (4) = 0	SUM (5) = 0	SUM (6) = 0	AVERAGE (1) = 0	AVERAGE (2) = 0	AVERAGE (3) = 0	AVERAGE (4) = 0
AVERAGE (5) = 0	AVERAGE (6) = 0	NUMOBS (1) = 0	NUMOBS (2) = 0	NUMOBS (3) = 0	NUMOBS (4) = 0	NUMOBS (5) = 0	NUMOBS (6) = 0
TEMP (1) = 0	TEMP (2) = 0	TEMP (3) = 0	TEMP (4) = 0	TEMP (5) = 0	TEMP (6) = 0	CINV# (1) = 0	CINV# (2) = 0
CINV# (3) = 0	CINV# (4) = 0	CINV# (5) = 0	CINV# (6) = 0				

CDB - AVERAGE CALCULATOR PVA

DLIST (1) = 0	DLIST (2) = 0	DLIST (3) = 0	DLIST (4) = 0	DLIST (5) = 0	DLIST (6) = 0	DLIST (7) = 0	DLIST (8) = 0
DLIST (9) = 0	DLIST (10) = 0	DLIST (11) = 0	DLIST (12) = 0	DLIST (13) = 0	DLIST (14) = 0	DLIST (15) = 0	DLIST (16) = 0
DLIST (17) = 0	DLIST (18) = 0	DLIST (19) = 0	DLIST (20) = 0	TCT (1) = 0	TCT (2) = 0	TCT (3) = 0	TCT (4) = 0
TCT (5) = 0	TCT (6) = 0	TCT (7) = 0	TCT (8) = 0	TCT (9) = 0	TCT (10) = 0	TCT (11) = 0	TCT (12) = 0
TCT (13) = 0	TCT (14) = 0	TCT (15) = 0	TCT (16) = 0	TCT (17) = 0	TCT (18) = 0	TCT (19) = 0	TCT (20) = 0
TCT (21) = 0	TCT (22) = 0	TCT (23) = 0	TCT (24) = 0	TCT (25) = 0	TCT (26) = 0	TCT (27) = 0	TCT (28) = 0
TCT (29) = 0	TCT (30) = 0	TCT (31) = 0	TCT (32) = 0	TCT (33) = 0	TCT (34) = 0	TCT (35) = 0	TCT (36) = 0
TCT (37) = 0	TCT (38) = 0	TCT (39) = 0	TCT (40) = 0	TCT (41) = 0	TCT (42) = 0	TCT (43) = 0	TCT (44) = 0
TCT (45) = 0	TCT (46) = 0	TCT (47) = 0	TCT (48) = 0	TCT (49) = 0	TCT (50) = 0	TCT (51) = 0	TCT (52) = 0
TCT (53) = 0	TCT (54) = 0	TCT (55) = 0	TCT (56) = 0	TCT (57) = 0	TCT (58) = 0	TCT (59) = 0	TCT (60) = 0

NOTES:
1. THE STORAGE OCCUPIED BY THE VARIABLES ON THIS PAGE IS IDENTICAL TO THAT OCCUPIED BY THE TCT.

CDB - TIME CONTROL TABLE (TCT)

NEXTOBS (1) = 0	NEXTOBS (2) = 0	NEXTOBS (3) = 0	NEXTOBS (4) = 0	NEXTOBS (5) = 0	NEXTOBS (6) = 0	NEXTOBS (7) = 0	NEXTOBS (8) = 0
NEXTOBS (9) = 0	NEXTOBS (10) = 0	NEXTOBS (11) = 0	NEXTOBS (12) = 0	NEXTOBS (13) = 0	SUM (1) = 0	SUM (2) = 0	SUM (3) = 0
SUM (4) = 0	SUM (5) = 0	SUM (6) = 0	SUM (7) = 0	SUM (8) = 0	SUM (9) = 0	SUM (10) = 0	SUM (11) = 0
SUM (12) = 0	SUM (13) = 0	TEMP (1) = 0	TEMP (2) = 0	TEMP (3) = 0	TEMP (4) = 0	TEMP (5) = 0	TEMP (6) = 0
TEMP (7) = 0	TEMP (8) = 0	TEMP (9) = 0	TEMP (10) = 0	TEMP (11) = 0	TEMP (12) = 0	TEMP (13) = 0	CINV# (1) = 0
CINV# (2) = 0	CINV# (3) = 0	CINV# (4) = 0	CINV# (5) = 0	CINV# (6) = 0	CINV# (7) = 0	CINV# (8) = 0	CINV# (9) = 0
CINV# (10) = 0	CINV# (11) = 0	CINV# (12) = 0	CINV# (13) = 0				

CDB - INTERMEDIATE AVERAGER PVA

OBS (1) = 0	OBS (2) = 0	OBS (3) = 0	OBS (4) = 0	OBS (5) = 0	OBS (6) = 0	OBS (7) = 0	OBS (8) = 0
OBS (9) = 0	OBS (10) = 0	OBS (11) = 0	OBS (12) = 0	OBS (13) = 0	OBS (14) = 0	OBS (15) = 0	OBS (16) = 0
OBS (17) = 0	CINV# (1) = 0	CINV# (2) = 0	CINV# (3) = 0	CINV# (4) = 0	CINV# (5) = 0	CINV# (6) = 0	CINV# (7) = 0
CINV# (8) = 0	CINV# (9) = 0	CINV# (10) = 0	CINV# (11) = 0	CINV# (12) = 0	CINV# (13) = 0	CINV# (14) = 0	CINV# (15) = 0
CINV# (16) = 0	CINV# (17) = 0						

CDB - SENSOR READER PVA

IEBP = 0	MEBP = 0	TEBP = 0	IRBS = 0	MRBS = 0	TRBS = 0	IEMOFF = 0	MEMOFF = 0
TEMOFF = 0	ISOSREPT = 0	MSOSREPT = 0	TSOSREPT = 0	IELON = 0	MELON = 0	TELON = 0	IELOFF = 0
MELOFF = 0	TELOFF = 0	IREPTSCHED = 0	MREPTSCHED = 0	TREPTSCHED = 0	IAIRREPT = 0	MAIRREPT = 0	TAIRREPT = 0
IMISTREPT = 0	MMISTREPT = 0	TMISTREPT = 0	IPERIODICREPT = 0	MPERIODICREPT = 0	TPERIODICREPT = 0	ISHIPREPT = 0	MSHIPREPT = 0
TSHIPREPT = 0	IBCAST = 1	MBCAST = 0	TBCAST = 0	IRCVRTABL = 1	MRCVRTABL = 0	TRCVRTABL = 0	IRTABLCHNG = 1
MRTABLCHNG = 0	TRTABLCHNG = 0	IXMITRTABL = 1	MXMITRTABL = 0	TXMITRTADL = 0	ITTABLCHNG = 1	MTTABLCHNG = 0	TTTABLCHNG = 0
IOEMRPTBUF = 1	MOEMRPTBUF = 0	TOEMRPTBUF = 0	IREMRPTBUF = 0	MREMRPTBUF = 0	TREMRPTBUF = 0	IIEMRPTBUF = 1	MIEMRPTBUF = 0
TIEMRPTBUF = 0	IFEMRPTBUF = 3	MFEMRPTBUF = 0	TFEMRPTBUF = 0	SEMRPTBUF = 3	FNTEMRPTBUF = 1	RREMRPTBUF = 3	EMRPTBUF (1) =
EMRPTBUF (2) =	EMRPTBUF (3) =	IOATOBSBUF (1) = 1	MOATOBSBUF (1) = 0	TOATOBSBUF (1) = 0	IRATOBSBUF (1) = 0	MRATOBSBUF (1) = 0	TRATOBSBUF (1) = 0
IIATOBSBUF (1) = 1	MIATOBSBUF (1) = 0	TIATOBSBUF (1) = 0	IFATOBSBUF (1) = 3	MFATOBSBUF (1) = 0	TFATOBSBUF (1) = 0	SATOBSBUF (1) = 3	FNTATOBSBUF (1) = 1
RRATOBSBUF (1) = 3	IOATOBSBUF (2) = 1	MOATOBSBUF (2) = 0	TOATOBSBUF (2) = 0	IRATOBSBUF (2) = 0	MRATOBSBUF (2) = 0	TRATOBSBUF (2) = 0	IIATOBSBUF (2) = 1
MIATOBSBUF (2) = 0	TIATOBSBUF (2) = 0	IFATOBSBUF (2) = 3	MFATOBSBUF (2) = 0	TFATOBSBUF (2) = 0	SATOBSBUF (2) = 3	FNTATOBSBUF (2) = 1	RRATOBSBUF (2) = 3
IOATOBSBUF (3) = 1	MOATOBSBUF (3) = 0	TOATOBSBUF (3) = 0	IRATOBSBUF (3) = 0	MRATOBSBUF (3) = 0	TRATOBSBUF (3) = 0	IIATOBSBUF (3) = 1	MIATOBSBUF (3) = 0
TIATOBSBUF (3) = 0	IFATOBSBUF (3) = 3	MFATOBSBUF (3) = 0	TFATOBSBUF (3) = 0	SATOBSBUF (3) = 3	FNTATOBSBUF (3) = 1	MRATOBSBUF (3) = 3	IOWSOBSBUF (1) = 1
MOWSOBSBUF (1) = 0	TOWSOBSBUF (1) = 0	IWWSOBSBUF (1) = 0	MWWSOBSBUF (1) = 0	TWWSOBSBUF (1) = 0	IIWSOBSBUF (1) = 1	MIWSOBSBUF (1) = 0	TIWSOBSBUF (1) = 0
IFWSOBSBUF (1) = 3	MFWSOBSBUF (1) = 0	TFWSOBSBUF (1) = 0	SWSOBSBUF (1) = 3	FNTWSOBSBUF (1) = 1	RRWSOBSBUF (1) = 3	IOWSOBSBUF (2) = 1	MOWSOBSBUF (2) = 0
TOWSOBSBUF (2) = 0	IRWSOBSBUF (2) = 0	MRWSOBSBUF (2) = 0	TRWSOBSBUF (2) = 0	IIWSOBSBUF (2) = 1	MIWSOBSBUF (2) = 0	TIWSOBSBUF (2) = 0	IFWSOBSBUF (2) = 3
MFWSOBSBUF (2) = 0	TFWSOBSBUF (2) = 0	SWSOBSBUF (2) = 3	FNTWSOBSBUF (2) = 1	RRWSOBSBUF (2) = 3	IOWSOBSBUF (3) = 1	MOWSOBSBUF (3) = 0	TOWSOBSBUF (3) = 0
IRWSOBSBUF (3) = 0	MRWSOBSBUF (3) = 0	TRWSOBSBUF (3) = 0	IIWSOBSBUF (3) = 1	MIWSOBSBUF (3) = 0	TIWSOBSBUF (3) = 0	IFWSOBSBUF (3) = 3	MFWSOBSBUF (3) = 0
TFWSOBSBUF (3) = 0	SWSOBSBUF (3) = 3	FNTWSOBSBUF (3) = 1	RRWSOBSBUF (3) = 3	IOWSOBSBUF (4) = 1	MOWSOBSBUF (4) = 0	TOWSOBSBUF (4) = 0	IRWSOBSBUF (4) = 0

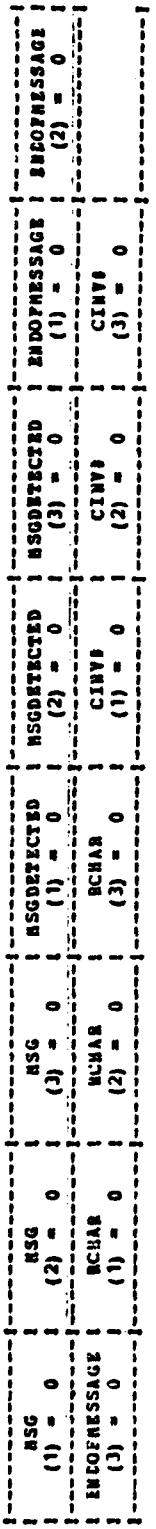
RRWSOBSBUF (4) = 0	TRWSOBSBUF (4) = 0	IIWSOBSBUF (4) = 1	HIWSOBSBUF (4) = 0	TIWSOBSBUF (4) = 0	IFWSOBSBUF (4) = 3	HFWSOBSBUF (4) = 0	TFWSOBSBUF (4) = 0
SWSOBSBUF (4) = 3	FNTWSOBSBUF (4) = 1	RRWSOBSBUF (4) = 3	IOWSOBSBUF (5) = 1	HOWSOBSBUF (5) = 0	TOWSOBSBUF (5) = 0	IRWSOBSBUF (5) = 0	HRWSOBSBUF (5) = 0
TRWSOBSBUF (5) = 0	IIWSOBSBUF (5) = 1	HIWSOBSBUF (5) = 0	TIWSOBSBUF (5) = 0	IFWSOBSBUF (5) = 3	HFWSOBSBUF (5) = 0	TFWSOBSBUF (5) = 0	SWSOBSBUF (5) = 3
FNTWSOBSBUF (5) = 1	RRWSOBSBUF (5) = 3	IOWDOBSBUF (1) = 1	HOWDOBSBUF (1) = 0	TOWDOBSBUF (1) = 0	IRWDOBSBUF (1) = 0	HRWDOBSBUF (1) = 0	TRWDOBSBUF (1) = 0
IIWDOBSBUF (1) = 1	HIWDOBSBUF (1) = 0	TIWDOBSBUF (1) = 0	IFWDOBSBUF (1) = 3	HFWDOBSBUF (1) = 0	TFWDOBSBUF (1) = 0	SWDOBSBUF (1) = 3	FNTWDOBSBUF (1) = 1
RRWDOBSBUF (1) = 3	IOWDOBSBUF (2) = 1	HOWDOBSBUF (2) = 0	TOWDOBSBUF (2) = 0	IRWDOBSBUF (2) = 0	HRWDOBSBUF (2) = 0	TRWDOBSBUF (2) = 0	IIWDOBSBUF (2) = 1
HIWDOBSBUF (2) = 0	TIWDOBSBUF (2) = 0	IFWDOBSBUF (2) = 3	HFWDOBSBUF (2) = 0	TFWDOBSBUF (2) = 0	SWDOBSBUF (2) = 3	FNTWDOBSBUF (2) = 1	RRWDOBSBUF (2) = 3
IOWDOBSBUF (3) = 1	HOWDOBSBUF (3) = 0	TOWDOBSBUF (3) = 0	IRWDOBSBUF (3) = 0	HRWDOBSBUF (3) = 0	TRWDOBSBUF (3) = 0	IIWDOBSBUF (3) = 1	HIWDOBSBUF (3) = 0
TIWDOBSBUF (3) = 0	IFWDOBSBUF (3) = 3	HFWDOBSBUF (3) = 0	TFWDOBSBUF (3) = 0	SWDOBSBUF (3) = 3	FNTWDOBSBUF (3) = 1	RRWDOBSBUF (3) = 3	IOWCOBSBUF (4) = 1
HOWDOBSBUF (4) = 0	TOWDOBSBUF (4) = 0	IRWDOBSBUF (4) = 0	HRWDOBSBUF (4) = 0	TRWDOBSBUF (4) = 0	IIWDOBSBUF (4) = 1	HIWDOBSBUF (4) = 0	TIWCOBSBUF (4) = 0
IFWDOBSBUF (4) = 3	HFWDOBSBUF (4) = 0	TFWDOBSBUF (4) = 0	SWDOBSBUF (4) = 3	FNTWDOBSBUF (4) = 1	RRWDOBSBUF (4) = 3	IOWDOBSBUF (5) = 1	HOWDOBSBUF (5) = 0
TOWDOBSBUF (5) = 0	IRWDOBSBUF (5) = 0	HRWDOBSBUF (5) = 0	TRWDOBSBUF (5) = 0	IIWDOBSBUF (5) = 1	HIWDOBSBUF (5) = 0	TIWDOBSBUF (5) = 0	IFWDOBSBUF (5) = 3
HFWDOBSBUF (5) = 0	TFWDOBSBUF (5) = 0	SWDOBSBUF (5) = 3	FNTWDOBSBUF (5) = 1	RRWDOBSBUF (5) = 3	IOATAVBUF (1) = 1	HOATAVBUF (1) = 0	TOATAVBUF (1) = 0
IRATAVBUF (1) = 0	HRATAVBUF (1) = 0	TRATAVBUF (1) = 0	IIATAVBUF (1) = 1	HIATAVBUF (1) = 0	TIATAVBUF (1) = 0	IFATAVBUF (1) = 3	HFATAVBUF (1) = 0
TFATAVBUF (1) = 0	SATAVBUF (1) = 3	FNTATAVBUF (1) = 1	RRATAVBUF (1) = 3	IOATAVBUF (2) = 1	HOATAVBUF (2) = 0	TOATAVBUF (2) = 0	IRATAVBUF (2) = 0
HRATAVBUF (2) = 0	TRATAVBUF (2) = 0	IIATAVBUF (2) = 1	HIATAVBUF (2) = 0	TIATAVBUF (2) = 0	IFATAVBUF (2) = 3	HFATAVBUF (2) = 0	TFATAVBUF (2) = 0
SATAVBUF (2) = 3	FNTATAVBUF (2) = 1	RRATAVBUF (2) = 3	IOATAVBUF (3) = 1	HOATAVBUF (3) = 0	TOATAVBUF (3) = 0	IRATAVBUF (3) = 0	HRATAVBUF (3) = 0
TRATAVBUF (3) = 0	IIATAVBUF (3) = 1	HIATAVBUF (3) = 0	TIATAVBUF (3) = 0	IFATAVBUF (3) = 3	HFATAVBUF (3) = 0	TFATAVBUF (3) = 0	SATAVBUF (3) = 3
FNTATAVBUF (3) = 1	RRATAVBUF (3) = 3	IOWSAVBUF (1) = 1	HOWSAVBUF (1) = 0	TOWSAVBUF (1) = 0	IRWSAVBUF (1) = 0	HRWSAVBUF (1) = 0	TRWSAVBUF (1) = 0
IIWSAVBUF (1) = 1	HIWSAVBUF (1) = 0	TIWSAVBUF (1) = 0	IFWSAVBUF (1) = 3	HFWSAVBUF (1) = 0	TFWSAVBUF (1) = 0	SWSAVBUF (1) = 3	FNTWSAVBUF (1) = 1

RRNSAVBUF (1) = 3	IONSAVBUF (2) = 1	HONSAVBUF (2) = 0	TONSAVBUF (2) = 0	IRNSAVBUF (2) = 0	HRNSAVBUF (2) = 0	TRNSAVBUF (2) = 0	IINSAVBUF (2) = 1
HINSAVBUF (2) = 0	IINSAVBUF (2) = 0	IFNSAVBUF (2) = 3	HFNSAVBUF (2) = 0	TFNSAVBUF (2) = 0	SRNSAVBUF (2) = 3	FRTNSAVBUF (2) = 1	SRNSAVBUF (2) = 3
IONSAVBUF (3) = 1	HONSAVBUF (3) = 0	TONSAVBUF (3) = 0	IRNSAVBUF (3) = 0	HRNSAVBUF (3) = 0	TRNSAVBUF (3) = 0	IINSAVBUF (3) = 1	HINSAVBUF (3) = 0
IINSAVBUF (3) = 0	IFNSAVBUF (3) = 3	HFNSAVBUF (3) = 0	TFNSAVBUF (3) = 0	SUSAVBUF (3) = 3	FRTNSAVBUF (3) = 1	RRNSAVBUF (3) = 3	IONSAVBUF (4) = 1
HONSAVBUF (4) = 0	TONSAVBUF (4) = 0	IRNSAVBUF (4) = 0	HRNSAVBUF (4) = 0	TRNSAVBUF (4) = 0	IINSAVBUF (4) = 1	HINSAVBUF (4) = 0	TINSAVBUF (4) = 0
IFNSAVBUF (4) = 3	HFNSAVBUF (4) = 0	TFNSAVBUF (4) = 0	SUSAVBUF (4) = 3	FRTNSAVBUF (4) = 1	RRNSAVBUF (4) = 3	IONSAVBUF (5) = 1	HONSAVBUF (5) = 0
TONSAVBUF (5) = 0	IRNSAVBUF (5) = 0	HRNSAVBUF (5) = 0	TRNSAVBUF (5) = 0	IINSAVBUF (5) = 1	HINSAVBUF (5) = 0	TINSAVBUF (5) = 0	IRNSAVBUF (5) = 3
HFNSAVBUF (5) = 0	TFNSAVBUF (5) = 0	SUSAVBUF (5) = 3	FRTNSAVBUF (5) = 1	RRNSAVBUF (5) = 3	IONDAVBUF (1) = 1	HONDAVBUF (1) = 0	TONDAVBUF (1) = 0
IRNDAVBUF (1) = 0	HRNDAVBUF (1) = 0	TRNDAVBUF (1) = 0	IINDAVBUF (1) = 1	HINDAVBUF (1) = 0	TINDAVBUF (1) = 0	IFNDAVBUF (1) = 3	HFNDAVBUF (1) = 0
TFNDAVBUF (1) = 0	SRDAVBUF (1) = 3	FRTNDAVBUF (1) = 1	RRNDAVBUF (1) = 3	IONDAVBUF (2) = 1	HONDAVBUF (2) = 0	TONDAVBUF (2) = 0	IRNDAVBUF (2) = 0
HRNDAVBUF (2) = 0	TRNDAVBUF (2) = 0	IINDAVBUF (2) = 1	HINDAVBUF (2) = 0	TINDAVBUF (2) = 0	IFNDAVBUF (2) = 3	HFNDAVBUF (2) = 0	TFNDAVBUF (2) = 0
SRDAVBUF (2) = 3	FRTNDAVBUF (2) = 1	RRNDAVBUF (2) = 3	IONDAVBUF (3) = 1	HONDAVBUF (3) = 0	TONDAVBUF (3) = 0	IRNDAVBUF (3) = 0	HRNDAVBUF (3) = 0
TRNDAVBUF (3) = 0	IINDAVBUF (3) = 1	HINDAVBUF (3) = 0	TINDAVBUF (3) = 0	IFNDAVBUF (3) = 3	HFNDAVBUF (3) = 0	TFNDAVBUF (3) = 0	SRDAVBUF (3) = 3
FRTNDAVBUF (3) = 1	RRNDAVBUF (3) = 3	IONDAVBUF (4) = 1	HONDAVBUF (4) = 0	TONDAVBUF (4) = 0	IRNDAVBUF (4) = 0	HRNDAVBUF (4) = 0	TRNDAVBUF (4) = 0
IINDAVBUF (4) = 1	HINDAVBUF (4) = 0	TINDAVBUF (4) = 0	IFNDAVBUF (4) = 3	HFNDAVBUF (4) = 0	TFNDAVBUF (4) = 0	SRDAVBUF (4) = 3	FRTNDAVBUF (4) = 1
HRNDAVBUF (4) = 3	IONDAVBUF (5) = 1	HONDAVBUF (5) = 0	TONDAVBUF (5) = 0	IRNDAVBUF (5) = 0	HRNDAVBUF (5) = 0	TRNDAVBUF (5) = 0	IINDAVBUF (5) = 1
HINDAVBUF (5) = 0	TINDAVBUF (5) = 0	IFNDAVBUF (5) = 3	HFNDAVBUF (5) = 0	TFNDAVBUF (5) = 0	SRDAVBUF (5) = 3	FRTNDAVBUF (5) = 1	RRNDAVBUF (5) = 3
IONT10BSBUF = 1	HONT10BSBUF = 0	TONT10BSBUF = 0	IRNT10BSBUF = 0	HRNT10BSBUF = 0	TRNT10BSBUF = 0	IINT10BSBUF = 1	HINT10BSBUF = 0
TINT10BSBUF = 0	IFNT10BSBUF = 3	HFNT10BSBUF = 0	TFNT10BSBUF = 0	SRT10BSBUF = 3	FRNT10BSBUF = 1	RRNT10BSBUF = 3	HT10BSBUF (1) =
HT10BSBUF (2) =	IT10BSBUF (3) =	IONT20BSBUF = 1	HONT20BSBUF = 0	TONT20BSBUF = 0	IRNT20BSBUF = 0	HRNT20BSBUF = 0	TINT20BSBUF = 0

IIWT2OBSBUF = 1	HIWT2OBSBUF = 0	TIWT2OBSBUF = 0	IFWT2OBSBUF = 3	HFWT2OBSBUF = 0	TFWT2OBSBUF = 0	SWT2OBSBUF = 3	FWTWT2OBSBUF = 1
RRWT2OBSBUF = 3	WT2OBSBUF (1) =	WT2OBSBUF (2) =	WT2OBSBUF (3) =	IOWT3OBSBUF = 1	HOWT3OBSBUF = 0	TOWT3OBSBUF = 0	IRWT3OBSBUF = 0
HRWT3OBSBUF = 0	TRWT3OBSBUF = 0	IIWT3OBSBUF = 1	HIWT3OBSBUF = 0	TIWT3OBSBUF = 0	IFWT3OBSBUF = 3	HFWT3OBSBUF = 0	TFWT3OBSBUF = 0
SWT3OBSBUF = 3	FWTWT3OBSBUF = 1	RRWT3OBSBUF = 3	WT3OBSBUF (1) =	WT3OBSBUF (2) =	WT3OBSBUF (3) =	IOATUPBUF = 1	HOATUPBUF = 0
TOATUPBUF = 0	IRATUPBUF = 0	HRATUPBUF = 0	TRATUPBUF = 0	IIATUPBUF = 1	HIATUPBUF = 0	TIATUPBUF = 0	IFATUPBUF = 3
HFATUPBUF = 0	TFATUPBUF = 0	SATUPBUF = 3	FWTATUPBUF = 1	RRATUPBUF = 3	ATUPBUF (1) =	ATUPBUF (2) =	ATUPBUF (3) =
IOWSUPBUF = 1	HOWSUPBUF = 0	TOWSUPBUF = 0	IRWSUPBUF = 0	HRWSUPBUF = 0	TRWSUPBUF = 0	IIWSUPBUF = 1	HIWSUPBUF = 0
TIWSUPBUF = 0	IFWSUPBUF = 3	HFWSUPBUF = 0	TFWSUPBUF = 0	SWSUPBUF = 3	FWTWSUPBUF = 1	RRWSUPBUF = 3	WSUPBUF (1) =
WSUPBUF (2) =	WSUPBUF (3) =	IOWDUPBUF = 1	HOWDUPBUF = 0	TOWDUPBUF = 0	IRWDUPBUF = 0	HRWDUPBUF = 0	TRWDUPBUF = 0
IIWDUPBUF = 1	HIWDUPBUF = 0	TIWDUPBUF = 0	IFWDUPBUF = 3	HFWDUPBUF = 0	TFWDUPBUF = 0	SWDUPBUF = 3	FWTWDUPBUF = 1
RRWDUPBUF = 3	WDUPBUF (1) =	WDUPBUF (2) =	WDUPBUF (3) =	IOWT1UPBUF = 1	HOWT1UPBUF = 0	TOWT1UPBUF = 0	IRWT1UPBUF = 0
HRWT1UPBUF = 0	TRWT1UPBUF = 0	IIWT1UPBUF = 1	HIWT1UPBUF = 0	TIWT1UPBUF = 0	IFWT1UPBUF = 3	HFWT1UPBUF = 0	TFWT1UPBUF = 0
SWT1UPBUF = 3	FWTWT1UPBUF = 1	RRWT1UPBUF = 3	WT1UPBUF (1) =	WT1UPBUF (2) =	WT1UPBUF (3) =	IOWT2UPBUF = 1	HOWT2UPBUF = 0
TOWT2UPBUF = 0	IRWT2UPBUF = 0	HRWT2UPBUF = 0	TRWT2UPBUF = 0	IIWT2UPBUF = 1	HIWT2UPBUF = 0	TIWT2UPBUF = 0	IFWT2UPBUF = 3
HFWT2UPBUF = 0	TFWT2UPBUF = 0	SWT2UPBUF = 3	FWTWT2UPBUF = 1	RRWT2UPBUF = 3	WT2UPBUF (1) =	WT2UPBUF (2) =	WT2UPBUF (3) =
IOWT3UPBUF = 1	HOWT3UPBUF = 0	TOWT3UPBUF = 0	IRWT3UPBUF = 0	HRWT3UPBUF = 0	TRWT3UPBUF = 0	IIWT3UPBUF = 1	HIWT3UPBUF = 0
TIWT3UPBUF = 0	IFWT3UPBUF = 3	HFWT3UPBUF = 0	TFWT3UPBUF = 0	SWT3UPBUF = 3	FWTWT3UPBUF = 1	RRWT3UPBUF = 3	WT3UPBUF (1) =
WT3UPBUF (2) =	WT3UPBUF (3) =	IOOMOBSBUF = 1	HOOMOBSBUF = 0	TOOMOBSBUF = 0	IROMOBSBUF = 0	HROMOBSBUF = 0	TROMOBSBUF = 0
IIOMOBSBUF = 1	HIOMOBSBUF = 0	TIOMOBSBUF = 0	IFOMOBSBUF = 3	HFOMOBSBUF = 0	TFOMOBSBUF = 0	SOMOBSBUF = 3	FWTOMOBSBUF = 1
RROMOBSBUF = 3	OMOBSBUF (1) =	OMOBSBUF (2) =	OMOBSBUF (3) =	IOLOCUPBUF = 1	HOLOCUPBUF = 0	TOLOCUPBUF = 0	IRLOCUPBUF = 0

NRLOCUPBUF = 0	TRLOCUPBUF = 0	IILOCUPBUF = 1	HILOCUPBUF = 0	TILOCUPBUF = 0	IPLOCUPBUF = 3	HPLOCUPBUF = 0	TPLOCUPBUF = 0
SLOCUPBUF = 3	FNTLOCUPBUF = 1	NRLOCUPBUF = 3	LOCUPBUF (1) =	LOCUPBUF (2) =	LOCUPBUF (3) =	IOLOCUPBUF = 1	HOLOCUIBUF = 0
TOLOCUPBUF = 0	IRLOCUPBUF = 0	HRLOCUPBUF = 0	TRLOCUPBUF = 0	IILOCUPBUF = 1	HILOCUPBUF = 0	TILOCUPBUF = 0	IPLOCUPBUF = 3
HPLOCUPBUF = 0	TPLOCUPBUF = 0	SLOCUPBUF = 3	FNTLOCUPBUF = 1	RRLOCUPBUF = 3	LOCUPBUF (1) =	LOCUPBUF (2) =	LOCUPBUF (3) =
IOLOCORBUF = 1	HOLOCORBUF = 0	TOLOCORBUF = 0	IRLOCORBUF = 0	HRLOCORBUF = 0	TRLOCORBUF = 0	IILOCORBUF = 1	HILOCORBUF = 0
TILOCORBUF = 0	IPLOCORBUF = 3	HPLOCORBUF = 0	TPLOCORBUF = 0	SLOCORBUF = 3	FNTLOCORBUF = 1	RRLOCORBUF = 3	LOCORBUF (1) =
LOCORBUF (2) =	LOCORBUF (3) =	IOMSGBUF = 1	HOMSGBUF = 0	TOMSGBUF = 0	IRMSGBUF = 0	HRMSGBUF = 0	TRMSGBUF = 0
IIMSGBUF = 1	HIMSGBUF = 0	TIMSGBUF = 0	IPMSGBUF = 5	HPMSGBUF = 0	TPMSGBUF = 0	SMSGBUF = 5	FNTMSGBUF = 1
RRMSGBUF = 5	MSGBUF (1) =	MSGBUF (2) =	MSGBUF (3) =	MSGBUF (4) =	MSGBUF (5) =	IOARPTBUF = 1	HOARPTBUF = 0
TOARPTBUF = 0	IRARPTBUF = 0	HRARPTBUF = 0	TRARPTBUF = 0	IIARPTBUF = 1	HIARPTBUF = 0	TIARPTBUF = 0	IPARPTBUF = 2
HPARPTBUF = 0	TPARPTBUF = 0	SARPTBUF = 2	FNTARPTBUF = 1	RRARPTBUF = 2	ARPTBUF (1) =	ARPTBUF (2) =	IOHRPTBUF = 1
HOHRPTBUF = 0	TOHRPTBUF = 0	IRHRPTBUF = 0	HRHRPTBUF = 0	TRHRPTBUF = 0	IIHRPTBUF = 1	HIHRPTBUF = 0	TIHRPTBUF = 0
IPHRPTBUF = 2	HPHRPTBUF = 0	TPHRPTBUF = 0	SHRPTBUF = 2	FNTHRPTBUF = 1	RRHRPTBUF = 2	HRPTBUF (1) =	HRPTBUF (2) =
IOPRPTBUF = 1	HOPRPTBUF = 0	TOPRPTBUF = 0	IRPRPTBUF = 0	HRPRPTBUF = 0	TRPRPTBUF = 0	IIPRPTBUF = 1	HIPRPTBUF = 0
TIPRPTBUF = 0	IPPRPTBUF = 2	HPPRPTBUF = 0	TPPRPTBUF = 0	SPRPTBUF = 2	FNTPRPTBUF = 1	RRPRPTBUF = 2	PRPTBUF (1) =
PRPTBUF (2) =	IOSRPTBUF = 1	HOSRPTBUF = 0	TOSRPTBUF = 0	IRSRPTBUF = 0	HRSRPTBUF = 0	TRSRPTBUF = 0	IISRPTBUF = 1
HISRPTBUF = 0	TISRPTBUF = 0	IPSRPTBUF = 2	HPSRPTBUF = 0	TPSRPTBUF = 0	SSRPTBUF = 2	FNTSRPTBUF = 1	RRSRPTBUF = 2
SRPTBUF (1) =	SRPTBUF (2) =	HEADRDYLIST (1) = 0	HEADRDYLIST (2) = 0	HEADRDYLIST (3) = 0	HEADRDYLIST (4) = 0	HEADRDYLIST (5) = 0	TAILRDYLIST (1) = 0
TAILRDYLIST (2) = 0	TAILRDYLIST (3) = 0	TAILRDYLIST (4) = 0	TAILRDYLIST (5) = 0	SOSREPTREQ = 0	PERIODICREPTREQ = 0	SHIPREPTREQ = 0	AIRREPTREQ = 0
MISTREPTREQ = 0	CMM = 0						

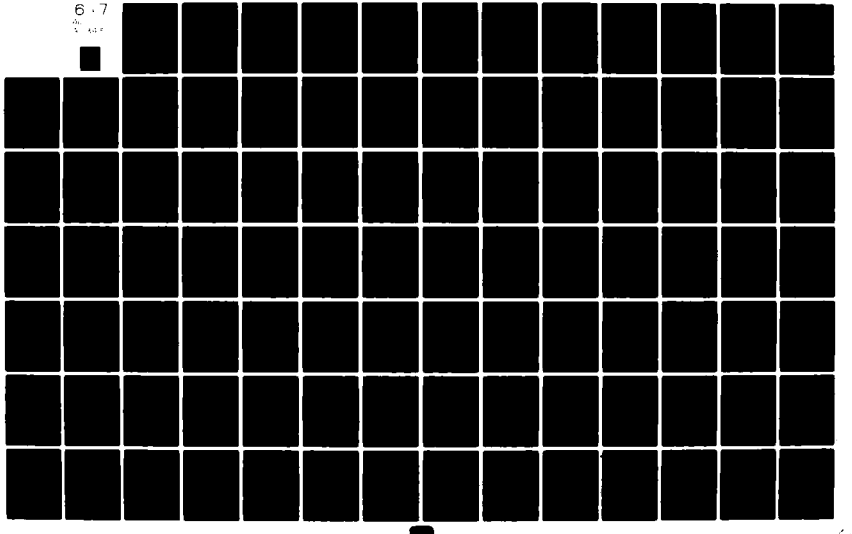
CDB - GLOBAL AREA

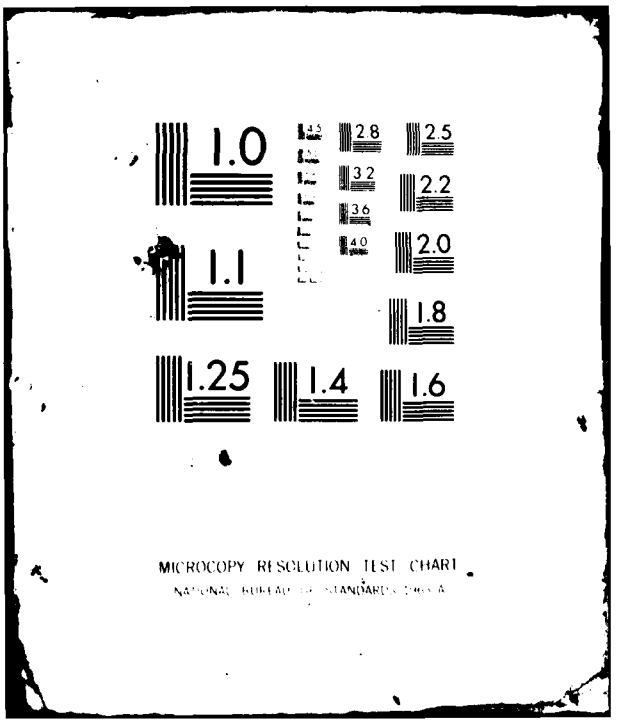


CDB - RECEIVER PVA

AD-A113 415 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
SOFTWARE ENGINEERING PRINCIPLES 3-14 AUGUST 1981,(U)
AUG 81 L J CHMURA, P C CLEMENTS

UNCLASSIFIED NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

APPENDIX II Sample Algorithms

Contents

```
sensor-reader:
reentrant program srdr(sensnum,okfcn,fetfcn,buffer,sem);
comment This program reads a sensor and stores the result
        in an observation buffer.

        sensnum: sensor number
        okfcn: function to test operation of the sensor
        fetfcn: function to retrieve a sample from the sensor
        buffer: observation buffer to insert sample into
        sem: semaphore to wait on for scheduling

     Typical parameter values:
        sensnum   okfcn    fetfcn    buffer     sem
        1 to 3    okat     fetat     atobsbuf   ats
        1 to 5    okws     fetws     wsobsbuf   wss
        1 to 5    okwd     fetwd     wdobsbuf   wds
        1         okom     fetom     omobsbuf   oms
        1         okwt1    fetwt1    wt1obsbuf  wt1s
        1         okwt2    fetwt2    wt2obsbuf  wt2s
        1         okwt3    fetwt3    wt3obsbuf  wt3s;

parameter integer sensnum;
parameter procedure okfcn, fetfcn;
parameter structure buffer of integer(fnt,rr,size)
  of structure o of integer(i, h, t)
  of structure r of integer(i, h, t)
  of structure i of integer(i, h, t)
  of structure f of integer(i, h, t)
  of integer b (s.buffer);
parameter structure sem of integer(i, h, t);
begin private integer obs;
  while true do
  begin
    begin global structure sem(sensnum) of integer(i, h, t);
      i.sem(sensnum) := i.sem(sensnum) - 1;
      if i.sem(sensnum) < 0 then
        begin global integer cmn;
          begin private integer pri; global integer head_ready_list,
                tail_ready_list;
            pri := priority(cmn);
            removep(cmn,head_ready_list(pri),tail_ready_list(pri));
          end;
          insertp(cmn,h.sem(sensnum),t.sem(sensnum));
          processor_allocate;
        end;
      end-if;
    end;
```

```
begin global integer tct, dlist, dtime;
  if dtime(tct(dlist(1)+1)) > "clock time" then
    quick;
  end-if;
end;
if okfcn then
  begin
    obs := fetfcn(sensnum);
    begin global integer tct, dlist, dtime;
      if dtime(tct(dlist(1)+1)) > "clock time" then
        quick;
      end-if;
    end;
    begin global integer fnt.buffer(sensnum), s.buffer(sensnum);
      global integer buffer(sensnum) (s.buffer(sensnum));
      begin global structure i.buffer(sensnum) of integer(i, h, t);
        i.i.buffer(sensnum) := i.i.buffer(sensnum) - 1;
        if i.i.buffer(sensnum) < 0 then
          begin global integer cmn;
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list;
              pri := priority(cmn);
              removep(cmn,head_ready_list(pri),tail_ready_list(pri));
            end;
            insertp(cmn,h.i.buffer(sensnum),t.i.buffer(sensnum));
            processor_allocate;
          end;
        end-if;
      end;
      begin global structure f.buffer(sensnum) of integer(i, h, t);
        i.f.buffer(sensnum) := i.f.buffer(sensnum) - 1;
        if i.f.buffer(sensnum) < 0 then
          begin global integer cmn;
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list;
              pri := priority(cmn);
              removep(cmn,head_ready_list(pri),tail_ready_list(pri));
            end;
            insertp(cmn,h.f.buffer(sensnum),t.f.buffer(sen
            processor_allocate;
          end;
        end-if;
      end;
```

```
            buffer(sensnum)(fnt.buffer(sensnum)) := obs;
            fnt.buffer(sensnum) := mod(fnt.buffer(sensnum),s.buffer(sensnum))+1;
            begin global structure r.buffer(sensnum) of integer(i, h, t);
              i.r.buffer(sensnum) := i.r.buffer(sensnum) + 1;
              if i.r.buffer(sensnum) < 1 then
                begin
                  removep(h.r.buffer(sensnum),h.r.buffer(sensnum),
                        t.r.buffer(sensnum));
                  begin private integer pri; global integer head_ready_list,
                        tail_ready_list, priority;
                    pri := priority(h.r.buffer(sensnum));
                    insertp(h.r.buffer(sensnum),head_ready_list(pri),
                          tail_ready_list(pri));
                  end;
                end;
              end-if;
            end;
            begin global structure i.buffer(sensnum) of integer(i, h, t);
              i.i.buffer(sensnum) := i.i.buffer(sensnum) + 1;
              if i.i.buffer(sensnum) < 1 then
                begin
                  removep(h.i.buffer(sensnum),h.i.buffer(sensnum),
                        t.i.buffer(sensnum));
                  begin private integer pri; global integer head_ready_list,
                        tail_ready_list, priority;
                    pri := priority(h.i.buffer(sensnum));
                    insertp(h.i.buffer(sensnum),head_ready_list(pri),
                          tail_ready_list(pri));
                  end;
                end;
              end-if;
            end;
          end;
          begin global integer tct, dlist, dtime;
            if dtime(tct(dlist(1)+1)) > "clock time" then
              quick;
            end-if;
          end;
        end;
      end-if;
    end;
    end-while;
end;
```

```
emergency_report_generator:
comment Add time dependent information to emergency report and
        put it in the emergency report buffer;
begin private character string;
  while true do
  begin
    begin global structure sosrept of integer(i, h, t);
      i.sosrept := i.sosrept - 1;
      if i.sosrept < 0 then
        begin global integer cmn;
          begin private integer pri; global integer head_ready_list,
                tail_ready_list;
            pri := priority(cmn);
            removep(cmn,head_ready_list(pri),tail_ready_list(pri));
          end;
          insertp(cmn,h.sosrept,t.sosrept);
          processor_allocate;
        end;
      end-if;
    end;
    begin global integer tct, dlist, dtime;
      if dtime(tct(dlist(1)+1)) > "clock time" then
        quick;
      end-if;
    end;
    string := format("(9Hsos from ,A10)",fetemreport);
    begin global integer tct, dlist, dtime;
      if dtime(tct(dlist(1)+1)) > "clock time" then
        quick;
      end-if;
    end;
    begin global integer fnt.emrptbuf, s.emrptbuf;
      global integer emrptbuf(s.emrptbuf);
      begin global structure i.emrptbuf of integer(i, h, t);
        i.i.emrptbuf := i.i.emrptbuf - 1;
        if i.i.emrptbuf < 0 then
          begin global integer cmn;
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list;
              pri := priority(cmn);
              removep(cmn,head_ready_list(pri),tail_ready_list(pri));
            end;
            insertp(cmn,h.i.emrptbuf,t.i.emrptbuf);
            processor_allocate;
          end;
        end-if;
      end;
```

```
      begin global structure f.emrptbuf of integer(i, h, t);
        i.f.emrptbuf := i.f.emrptbuf - 1;
        if i.f.emrptbuf < 0 then
          begin global integer cmn;
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list;
              pri := priority(cmn);
              removep(cmn,head_ready_list(pri),tail_ready_list(pri));
            end;
            insertp(cmn,h.f.emrptbuf,t.f.emrptbuf);
            processor_allocate;
          end;
        end-if;
      end;
      emrptbuf(fnt.emrptbuf) := string;
      fnt.emrptbuf := mod(fnt.emrptbuf,s.emrptbuf)+1;
      begin global structure r.emrptbuf of integer(i, h, t);
        i.r.emrptbuf := i.r.emrptbuf + 1;
        if i.r.emrptbuf < 1 then
          begin
            removep(h.r.emrptbuf,h.r.emrptbuf,t.r.emrptbuf);
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list, priority;
              pri := priority(h.r.emrptbuf);
              insertp(h.r.emrptbuf,head_ready_list(pri),tail_ready_list(pri));
            end;
          end;
        end-if;
      end;
      begin global structure i.emrptbuf of integer(i, h, t);
        i.i.emrptbuf := i.i.emrptbuf + 1;
        if i.i.emrptbuf < 1 then
          begin
            removep(h.i.emrptbuf,h.i.emrptbuf,t.i.emrptbuf);
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list, priority;
              pri := priority(h.i.emrptbuf);
              insertp(h.i.emrptbuf,head_ready_list(pri),tail_ready_list(pri));
            end;
          end;
        end-if;
      end;
    end;
  end;
  end-while;
end;
```

```
transmitter:
reentrant program xmit(freq,rptbuf);
comment This program broadcasts a report.  The report
        contents are obtained from the buffer "rptbuf."

        freq: frequency to broadcast report on
        rptbuf: buffer containing the report contents

     Typical parameter values:
        rptbuf      freq
        prptbuf     5000
        arptbuf     161000
        erptbuf     5100
        srptbuf     300
        hrptbuf     5100;

parameter integer freq;
parameter structure rptbuf of integer(fnt,rr,size)
  of structure o of integer(i, h, t)
  of structure r of integer(i, h, t)
  of structure i of integer(i, h, t)
  of structure f of integer(i, h, t)
  of integer b (s.rptbuf);

begin private character char; private integer xmitrnum;
  while true do
  begin
    begin global integer rr.rptbuf, s.rptbuf;
      global integer rptbuf(s.rptbuf);
      begin global structure o.rptbuf of integer(i, h, t);
        i.o.rptbuf := i.o.rptbuf - 1;
        if i.o.rptbuf < 0 then
          begin global integer cmn;
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list;
              pri := priority(cmn);
              removep(cmn,head_ready_list(pri),tail_ready_list(pri));
            end;
            insertp(cmn,h.o.rptbuf,t.o.rptbuf);
            processor_allocate;
          end;
        end-if;
      end;
```

```
begin global structure r.rptbuf of integer(i, h, t);
  i.r.rptbuf := i.r.rptbuf - 1;
  if i.r.rptbuf < 0 then
    begin global integer cmn;
      begin private integer pri; global integer head_ready_list,
            tail_ready_list;
        pri := priority(cmn);
        removep(cmn,head_ready_list(pri),tail_ready_list(pri));
      end;
      insertp(cmn,h.r.rptbuf,t.r.rptbuf);
      processor_allocate;
    end;
  end-if;
end;
rr.rptbuf := mod(rr.rptbuf,s.rptbuf)+1;
char := rptbuf(rr.rptbuf);
begin global structure f.rptbuf of integer(i, h, t);
  i.f.rptbuf := i.f.rptbuf + 1;
  if i.f.rptbuf < 1 then
    begin
      removep(h.f.rptbuf,h.f.rptbuf,t.f.rptbuf);
      begin private integer pri; global integer head_ready_list,
            tail_ready_list, priority;
        pri := priority(h.f.rptbuf);
        insertp(h.f.rptbuf,head_ready_list(pri),tail_ready_list(pri));
      end;
    end;
  end-if;
end;
begin global structure o.rptbuf of integer(i, h, t);
  i.o.rptbuf := i.o.rptbuf + 1;
  if i.o.rptbuf < 1 then
    begin
      removep(h.o.rptbuf,h.o.rptbuf,t.o.rptbuf);
      begin private integer pri; global integer head_ready_list,
            tail_ready_list, priority;
        pri := priority(h.o.rptbuf);
        insertp(h.o.rptbuf,head_ready_list(pri),tail_ready_list(pri));
      end;
    end;
  end-if;
end;
begin global integer tct, dlist, dtime;
  if dtime(tct(dlist(1)+1)) > "clock time" then
    quick;
  end-if;
end;
```

```
begin private boolean not_found;
  not_found := true;
  while not_found do
  begin
    begin global structure ttablchng of integer(i, h, t);
      i.ttablchng := i.ttablchng - 1;
      if i.ttablchng < 0 then
        begin global integer cmn;
          begin private integer pri; global integer head_ready_list,
                tail_ready_list;
            pri := priority(cmn);
            removep(cmn,head_ready_list(pri),tail_ready_list(pri));
          end;
          insertp(cmn,h.ttablchng,t.ttablchng);
          processor_allocate;
        end;
      end-if;
    end;
    begin global structure xmitrtabl of integer(i, h, t);
      i.xmitrtabl := i.xmitrtabl - 1;
      if i.xmitrtabl < 0 then
        begin global integer cmn;
          begin private integer pri; global integer head_ready_list,
                tail_ready_list;
            pri := priority(cmn);
            removep(cmn,head_ready_list(pri),tail_ready_list(pri));
          end;
          insertp(cmn,h.xmitrtabl,t.xmitrtabl);
          processor_allocate;
        end;
      end-if;
    end;
    comment Look in table for available transmitter of proper type and
            set not_found;
    if not_found then
      begin global structure xmitrtabl of integer(i, h, t);
        i.xmitrtabl := i.xmitrtabl + 1;
        if i.xmitrtabl < 1 then
          begin
            removep(h.xmitrtabl,h.xmitrtabl,t.xmitrtabl);
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list, priority;
              pri := priority(h.xmitrtabl);
              insertp(h.xmitrtabl,head_ready_list(pri),
                    tail_ready_list(pri));
            end;
          end;
        end-if;
      end;
    end-if;
  end;
  end-while;
```

```
    comment Mark selected transmitter as in use;
    begin global structure xmitrtabl of integer(i, h, t);
      i.xmitrtabl := i.xmitrtabl + 1;
      if i.xmitrtabl < 1 then
        begin
          removep(h.xmitrtabl,h.xmitrtabl,t.xmitrtabl);
          begin private integer pri; global integer head_ready_list,
                tail_ready_list, priority;
            pri := priority(h.xmitrtabl);
            insertp(h.xmitrtabl,head_ready_list(pri),tail_ready_list(pri));
          end;
        end;
      end-if;
    end;
    xmitrnum := "selected transmitter number";
    xmitr_tune(xmitrnum,freq);
  end;
  begin global integer tct, dlist, dtime;
    if dtime(tct(dlist(1)+1)) > "clock time" then
      quick;
    end-if;
  end;
  send(xmitrnum,char);
  begin global integer tct, dlist, dtime;
    if dtime(tct(dlist(1)+1)) > "clock time" then
      quick;
    end-if;
  end;
  while (char ne "end of report character") do
  begin
    begin global integer rr.rptbuf, s.rptbuf;
      global integer rptbuf(s.rptbuf);
      begin global structure o.rptbuf of integer(i, h, t);
        i.o.rptbuf := i.o.rptbuf - 1;
        if i.o.rptbuf < 0 then
          begin global integer cmn;
            begin private integer pri; global integer head_ready_list,
                  tail_ready_list;
              pri := priority(cmn);
              removep(cmn,head_ready_list(pri),tail_ready_list(pri));
            end;
            insertp(cmn,h.o.rptbuf,t.o.rptbuf);
            processor_allocate;
          end;
        end-if;
      end;
```

```
begin global structure r.rptbuf of integer(i.h.t);
  i.r.rptbuf := i.r.rptbuf - 1;
  if i.r.rptbuf < 0 then
    begin global integer cmn;
      begin private integer pri; global integer head_ready_list,
            tail_ready_list;
        pri := priority(cmn);
        removep(cmn,head_ready_list(pri),tail_ready_list(pri));
      end;
      insertp(cmn,h.r.rptbuf,t.r.rptbuf);
      processor_allocate;
    end;
  end-if;
end;
rr.rptbuf := mod(rr.rptbuf,s.rptbuf)+1;
char := rptbuf(rr.rptbuf);
begin global structure f.rptbuf of integer(i, h, t);
  i.f.rptbuf := i.f.rptbuf + 1;
  if i.f.rptbuf < 1 then
    begin
      removep(h.f.rptbuf,h.f.rptbuf,t.f.rptbuf);
      begin private integer pri; global integer head_ready_list,
            tail_ready_list, priority;
        pri := priority(h.f.rptbuf);
        insertp(h.f.rptbuf,head_ready_list(pri),tail_ready_list(pri));
      end;
    end;
  end-if;
end;
begin global structure o.rptbuf of integer(i, h, t);
  i.o.rptbuf := i.o.rptbuf + 1;
  if i.o.rptbuf < 1 then
    begin
      removep(h.o.rptbuf,h.o.rptbuf,t.o.rptbuf);
      begin private integer pri; global integer head_ready_list,
            tail_ready_list, priority;
        pri := priority(h.o.rptbuf);
        insertp(h.o.rptbuf,head_ready_list(pri),tail_ready_list(pri));
      end;
    end;
  end-if;
end;
end;
begin global integer tct, dlist, dtime;
  if dtime(tct(dlist(1)+1)) > "clock time" then
    quick;
  end-if;
end;
```

```
    send(xmitrnum,char);
    begin global integer tct, dlist, dtime;
      if dtime(tct(dlist(1)+1)) > "clock time" then
        quick;
      end-if;
    end;
  end;
  end-while;
  begin global structure xmitrtabl of integer(i, h, t);
    i.xmitrtabl := i.xmitrtabl - 1;
    if i.xmitrtabl < 0 then
      begin global integer cmn;
        begin private integer pri; global integer head_ready_list,
              tail_ready_list;
          pri := priority(cmn);
          removep(cmn,head_ready_list(pri),tail_ready_list(pri));
        end;
        insertp(cmn,h.xmitrtabl,t.xmitrtabl);
        processor_allocate;
      end;
    end-if;
  end;
  comment Mark transmitter xmitrnum available in transmitter table;
  begin global structure xmitrtable of integer(i, h, t);
    i.xmitrtabl := i.xmitrtabl + 1;
    if i.xmitrtabl < 1 then
      begin
        removep(h.xmitrtabl,h.xmitrtabl,t.xmitrtabl);
        begin private integer pri; global integer head_ready_list,
              tail_ready_list, priority;
          pri := priority(h.xmitrtabl);
          insertp(h.xmitrtabl,head_ready_list(pri),tail_ready_list(pri));
        end;
      end;
    end-if;
  end;
  begin global structure ttablchng of integer(i, h, t);
    i.ttablchng := i.ttablchng + 1;
    if i.ttablchng < 1 then
      begin
        removep(h.ttablchng,h.ttablchng,t.ttablchng);
        begin private integer pri; global integer head_ready_list,
              tail_ready_list, priority;
          pri := priority(h.ttablchng);
          insertp(h.ttablchng,head_ready_list(pri),tail_ready_list(pri));
        end;
      end;
    end-if;
  end;
```

```
    begin global integer tct, dlist, dtime;
      if dtime(tct(dlist(1)+1)) > "clock time" then
        quick;
      end-if;
    end;
    begin global structure bcast of integer(i, h, t);
      i.bcast := i.bcast + 1;
      if i.bcast < 1 then
        begin
          removep(h.bcast,h.bcast,t.bcast);
          begin private integer pri; global integer head_ready_list,
                tail_ready_list, priority;
            pri := priority(h.bcast);
            insertp(h.bcast,head_ready_list(pri),tail_ready_list(pri));
          end;
        end;
      end-if;
    end;
    begin global integer tct, dlist, dtime;
      if dtime(tct(dlist(1)+1)) > "clock time" then
        quick;
      end-if;
    end;
  end;
  end-while;
end;
```

HAS.3 HAS Improved Modular Structure

EXAMPLE DESCRIPTION

Einar Newhire
Information System Specialist
Computer Software Division
Naval Electronics Research Laboratory (NERL)

Introduction

Last week, the HAS contractor (CSD) sent a memo warning us against making any further changes in the HAS configuration. He complained that the recent decision to use a different kind of transmitter will require such substantial changes to the Computer Program Design Specification (CPDS) that he is not sure he can meet the deadline.

In my opinion, the contractor's reluctance to make any changes is a sign of poorly designed software that will be expensive for the Navy to maintain. It is inevitable that some changes will be needed during the life cycle of the system. The system designer can reduce the cost of future modifications by anticipating areas that are likely to change, and designing the software so that coding changes will be easy to locate and easy to make.

I propose an alternate design for the HAS system, using Information-Hiding Modules. I identify design decisions that are likely to change and limit the knowledge of any one decision to a single module. I contend that a system with this structure will be easier to maintain, since the effects of changes will not ripple through the programs causing unexpected errors.

My proposed design has 17 modules, which are described on the following pages.

BUF: Buffer Maintenance Module

This module knows all the details about the buffers that are used to communicate information between programs, including the storage representation, how large they are, and what to do when a buffer is full or empty.

The programs that accept data from buffers and that deposit data in buffers are part of this module.

If we have several different types of buffers, there may be a separate submodule for each type, since they may have different sizes and behavior when filled up.

CC: Communications Control Module

The transmission frequencies for the various reports and the frequencies to be monitored for incoming messages are secrets of this module.

The module consists of programs that control the transmission and reception of messages, deciding when to reset frequency or change transmitter power. These programs call the TC and RC programs that actually control the devices.

The CC programs take characters to be transmitted out of buffers where they were put by MF programs.

The CC programs put received characters in a buffer for MF programs that handle incoming messages.

EM: Emergency Equipment Control Module

This module turns the emergency light on or off on demand. Its secret is the computer action that controls the light.

IG: Information Gathering Module

This module contains programs that compute the values to be stored in the Record Storage, using data obtained from the Sensor Control Module.

Each type of value is computed by a submodule, whose secret is the algorithm used in the computation.

MEM: Memory Allocator Module

This module knows and enforces the memory usage policy.

It contains a submodule that knows the actual memory size, and how the memory is allocated. The secret of the submodule consists of tables indicating the memory access rights of programs and the operating status of memory areas. The submodule provides programs to obtain or release portions of memory and to mark portions defective. The MEM module uses these programs to implement the memory usage policy.

MF: Message Format Modules

There is one Message Format module for incoming messages and one for each type of report generated by the buoy. All programs that know the structure, format details and information content of any given message belong to these modules.

The report-generation modules contain programs that build a message and put it in a buffer.

The incoming message module consists of programs that determine the message type and find pertinent information in the message.

MO: Monitor Modules

Each module allocates the use of one type of resource, such as buffers, receivers or transmitters. CPU time is not handled by a monitor module.

Monitor interfaces include programs to grant exclusive or shared use of the resource and to allow programs to relinquish use of the resources.

MI: Message Interpretation Module

This module knows the process that should be started in response to any type of incoming message. It is notified of the message type by the MF module.

PA: Processor Allocator Module

This module puts a process in control of a processor, i.e., registers are loaded and control transferred to the task. The secret of the module is the aspect of the architecture and the data structures relevant to task switching.

RC: Receiver Control Module

The receiver characteristics that are visible to the computer are the secrets of this module. The RC programs are used by CC programs as they monitor frequencies and receive messages.

The module includes programs to tune the device to a new frequency, detect a message coming in on a specified frequency, and receive a character.

RS: Record Storage Module

This module holds the buoy database. Its secret is the representation of the recorded values in storage.

This module includes programs used by other modules to update the values and functions used to retrieve the values.

SC: Sensor Control Module

Hidden in this module are the sensor characteristics that might change if we replaced one sensor with another that delivers the same information. The programs that take readings from sensors are in this module; they know the HAS-BEEN instruction sequences that perform sensor input and the hardware defined memory location corresponding to each device.

This module includes programs to get a new value from a sensor and to run a built-in test if the sensor contains self calibration circuitry. It also includes programs to set or to check the operating status of a particular sensor. Programs outside this module refer to sensors by names (e.g., first air temperature sensor); the correspondence between name and the way the actual device is addressed is known only inside this module.

SCH: Scheduler Module

This module schedules processes as they request processor time. It knows processor capacity and the deadlines and priorities associated with different processes. It uses the Processor Allocator Module to get a particular task running.

SOR: System Organization Module

This module knows all the information needed to generate a working HAS system. The values of various parameters, such as the number of sensors of each type, the intervals at which sensor readings are taken, averages computed, locations determined, reports broadcast, self-tests executed, and other periodic functions performed, are hidden in this module. The module

also contains information such as the number of processes of different kinds in the system, the number of processors, and the number of sensors of each type.

The System Organization Module is used to generate specific HAS systems.

TC: Transmitter Control Module

The transmitter characteristics that are visible to the computer are secrets of this module.

The module includes programs to transmit a character, tune the transmitter to a new frequency, or change the power level.

TIM: Timer Module

This module is responsible for keeping track of events that must occur regularly. It knows the time interval associated with each periodic task and how to tell "real time". It notifies the Scheduler when a particular task should be run.

TST: Performance Testing Module

This module knows the tests that must be performed to determine whether the equipment performance is acceptable.

It has separate submodules for sensor checking, memory checking, and computer function checking. Each submodule knows the range of behavior that is acceptable for the corresponding component.

The sensor testing submodule uses Sensor Control and Record Storage functions to get the sensor readings and averages used in its tests. Some sensors have built-in test circuitry that can be activated from the computer and deliver results that can be read by the computer. The control of these devices is a device property, and programs that are dependent on the characteristics of the particular device are part of the sensor module. The test module knows of their availability and uses them, but does so in such a way that, were the device to be replaced by another that could execute similar self tests but had a different computer interface, the test module would be unchanged.

Conclusion

If CSD organizes its documents in accordance with the above structure, both the documents and the software will be less sensitive to change.

Semaphore	P Called by	V Called by
airrept	air_report_generator	report scheduler
ats(i)	sensor_reader	periodic scheduler*
bcast	report_scheduler	transmitter
eloff	emergency_light_off	message_interpreter
elon	emergency_light_on	message_interpreter
emoff	emergency_message_reset	message_interpreter
epb	emergency_button_poller	periodic scheduler
histrept	history_report_generator	report scheduler
oms	sensor_reader	periodic scheduler
periodicrept	periodic_report_generator	regular_report_starter
reptsched	report_scheduler	message_interpreter OR regular_report_starter
rrs	regular_report_starter	periodic scheduler OR emergency_button_poller
shiprept	ship_report_generator	report_scheduler
sosrept	emergency_report_generator	report_scheduler
synch	receiver	periodic scheduler
wds(i)	sensor_reader	periodic scheduler
wss(i)	sensor_reader	periodic scheduler
wt1s(i)	sensor_reader	periodic scheduler
wt2s(i)	sensor_reader	periodic scheduler
wt3s(i)	sensor_reader	periodic scheduler

Buffer	Accept called by	Deposit called by
**&_obsbuf(i)	intermediate_averager	sensor_reader
omobsbuf	location_calculator	sensor_reader
&_avbuf(i)	average_calculator	intermediate_averager
locupbuf	updater	location_calculator
&_upbuf	updater	average_calculator
msgbuf	message_interpreter	receiver
loccorbuff	location_corrector	message_interpreter
emrptbuf	transmitter	emergency_report_generator
arptbuf	transmitter	air_report_generator
hrptbuf	transmitter	history_report_generator
prptbuf	transmitter	periodic_report_generator
srptbuf	transmitter	ship_report_generator

* periodic scheduler not described in this document

** &_ represents at
ws
wd
wt1
wt2
wt3

HAS.4 A Structured View of HAS

EXAMPLE DESCRIPTION

Einar Newhire
Information System Specialist
Computer Software Division
Naval Electronics Research Laboratory (NERL)

INTRODUCTION

Since I reported for duty here at NERL six weeks ago, I have been assigned to review progress on the HAS software procurement. The contractor (CSD) has submitted a Computer Program Design Specification (CPDS), including a set of algorithms in ALGOL-like pseudo-code, which is awaiting formal approval. Because of the length of this approval process, and the short timespan of the project, they are now starting the detailed design of data structures and specifications. It seems certain that the design will be approved since they based it on existing functional aircraft software that CSD developed for the MDADC (Melamine Desert Air Development Center).

During the review process, I heard many complaints about the complexity of the documents and the difficulty of keeping track of what is going on as one traces through the program text. CSD personnel constantly assure us that this is necessary in real-time software using HAS-BEEN computers. They point out that all real aircraft software has these characteristics and that no one can suggest a better way.

The purpose of this memo is to suggest a better way. It is based on a course I took at New Haven University from Professor E. Seawaller on process synchronization. It is also based on structuring concepts, such as stepwise refinement and structured programming.

PROCESSES

Professor Seawaller defines a process as a subset of the events in a system. He is interested in sequential processes within which the ordering of the events is obvious and easily determined.

Seawaller is very fond of trains and often uses the following analogy to clarify the concept of sequential processes. Consider a large railway switching yard with several trains entering and leaving at any given time. The events are cars entering the yard. Since the trains are moving at different speeds, slowing down, speeding up and stopping, the order of events in the whole yard cannot be predicted. However, the order of events is easily predicted for a single train: the first car enters the yard before the second, the second before the third, and so on. Therefore, a train entering the yard

is a sequential process: it is a subset of the events in the system in which the order is easily determined.

Writing programs on a "per-train" basis allows us to take advantage of the ease of predicting the sequence of events. The programs are easy to understand because the order of events makes sense. Programs written on a "process next event" basis are hard to understand because we must deal with an unpredictable sequence of events, where the order is sometimes significant and sometimes not.

The price we pay for the luxury of considering only one process at a time is that each program must include some commands whose only function is to ensure that the processes cooperate harmoniously. We must include commands to make sure that two different processes do not try to update the same variable at the same time because this could result in an erroneous count. These commands may cost us a little execution time, but the benefit of easily understood code is well worth it.

HAS AS A SET OF PROCESSES

The contractor describes HAS as a single process switching its attention from looking at a sensor to preparing part of a report to checking the clock to looking at another sensor to updating an average to .... It struck me that this is much the way a railway yard program would look if we did it on a per-event basis rather than a per-train basis. I think that part of my difficulty understanding the current HAS description is caused by the program being in the middle of so many different things at a time. Also, the order of some of the tests and data modifications is sometimes arbitrary, and sometimes essential for correct functioning. It is hard to tell which is which without very careful analysis, making it even harder to understand the program.

If we are to be able to apply the ideas in Seawaller's course, we must first deal with a problem that he never discussed: we must divide HAS into processes before we can worry about their synchronization. Some of the processes for HAS are described briefly below, and abstract programs for all the processes in the system are included in an appendix.

First we have one process to read each sensor. These sensor reader processes execute the sensing instruction and put the data in core. They repeat that simple sequence forever. The frequency of repetition depends upon the nature of the sensor, but in most cases it will be done at regular intervals and must be executed punctually to achieve accurate time averages.

Instead of raw sensor readings, most of the programs use averages over time to minimize errors caused by noisy readings. I would include another set of processes to read the data stored in core by the sensor readers and compute the tables of average readings used by other programs. I felt rather uncertain about this second set of processes because it seemed as if the single process, "read sensor; compute average" would fit the predictable-sequence criterion for a sequential process: clearly, the sensor had to be read before the average could be calculated. I separated the two because I realized that reading the sensor is time critical, but calculating the average is not. If

HAS gets many requests at once, processor time might get short. It would be important to continue making punctual sensor readings, but it would not be essential to keep up with computing the averages. Putting both actions in a single process forces an all-or-nothing approach to the sensor. In my proposal, the sensor reader processes will place their readings in core buffers; the averagers will empty the buffers. In moments of time pressure, we'll let the averaging processes get behind in their work but keep making the readings on time.

A process can be assigned to calculate each of the required system values. By using a separate process for each value, one can treat some as more urgent than others and avoid making arbitrary sequencing choices when writing the program.

A single process sends the reports that are due every 60 seconds. Since all of the data are prepared by other processes, this process is very simple: it is awakened by the clock, sends its report, and then returns to its resting state.

For each of the requested reports, we will have a report generator process waiting for the request. Since reports are needed quickly, we will have background processes keeping the data up-to-date with whatever computer capacity is available. The actual report process need only work on demand: it is awakened, generates the report, and returns to its resting position.

CONCLUSIONS

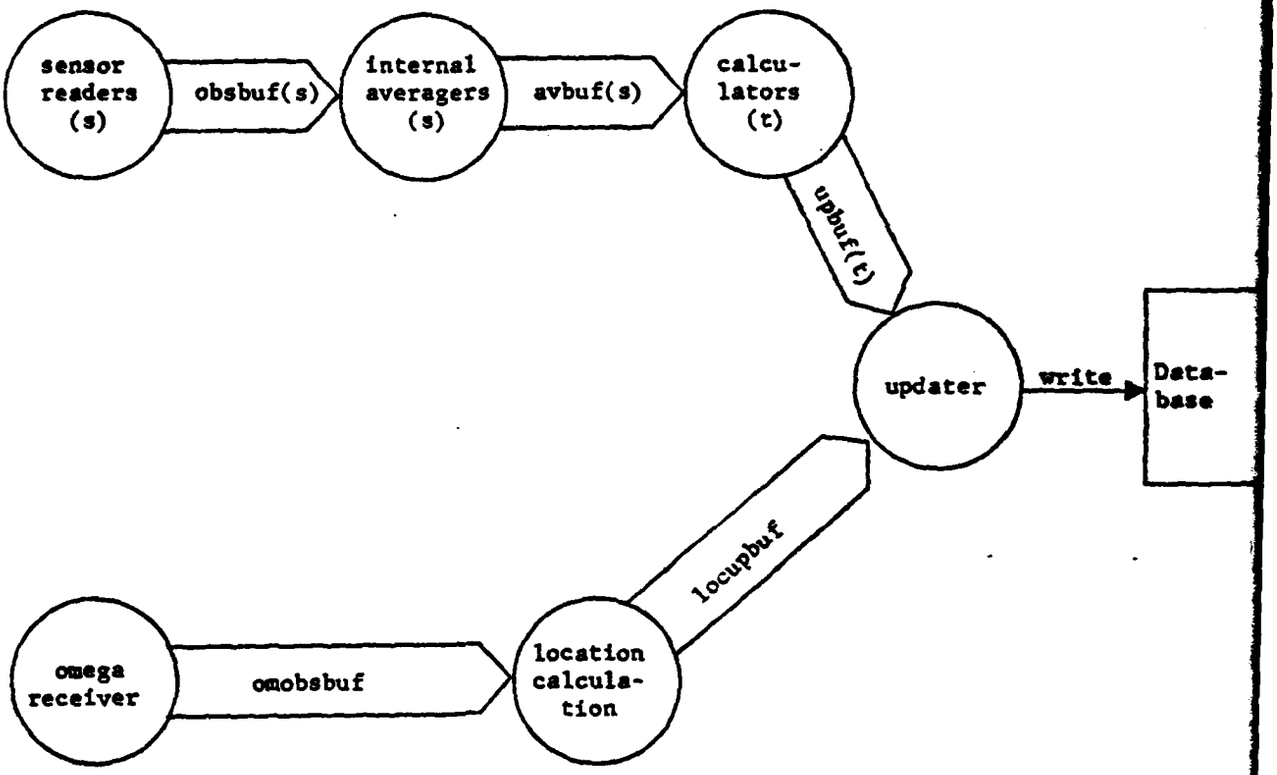
The above discussion is the basis for the enclosed HAS design. The design description includes a diagram of processes and buffers and a set of abstract programs for the individual processes.

As the abstract programs show, the individual processes are controlled by programs that are extremely simple; one might even call them obvious or trivial. Fine: that increases the likelihood that they are correct or at least that we will notice errors. All of the synchronization problems are standard problems dealt with in operating system textbooks (e.g., Shaw 1974). As a result, we can have faith in their correctness.

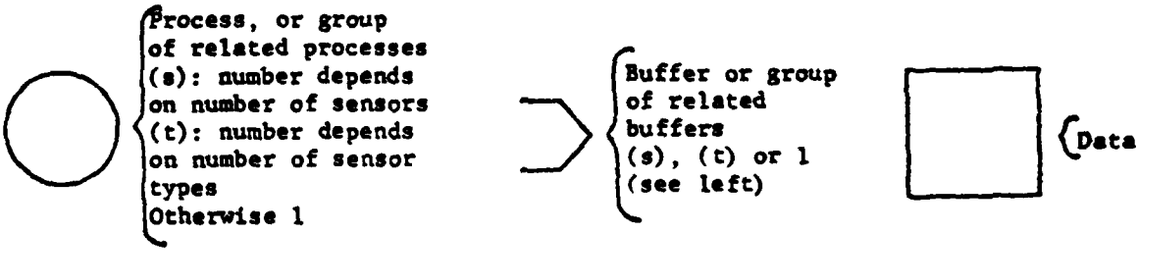
In addition to ease of understanding and verification, there is a side benefit. The design is more easily changed. There are obvious techniques for adding sensors, reports, etc., without changing the existing programs. For example, one can easily add or remove sensor-reader processes if the sensor configuration changes. Seawaller also claims that this type of structure makes it easy to change the number of central processors in a system.

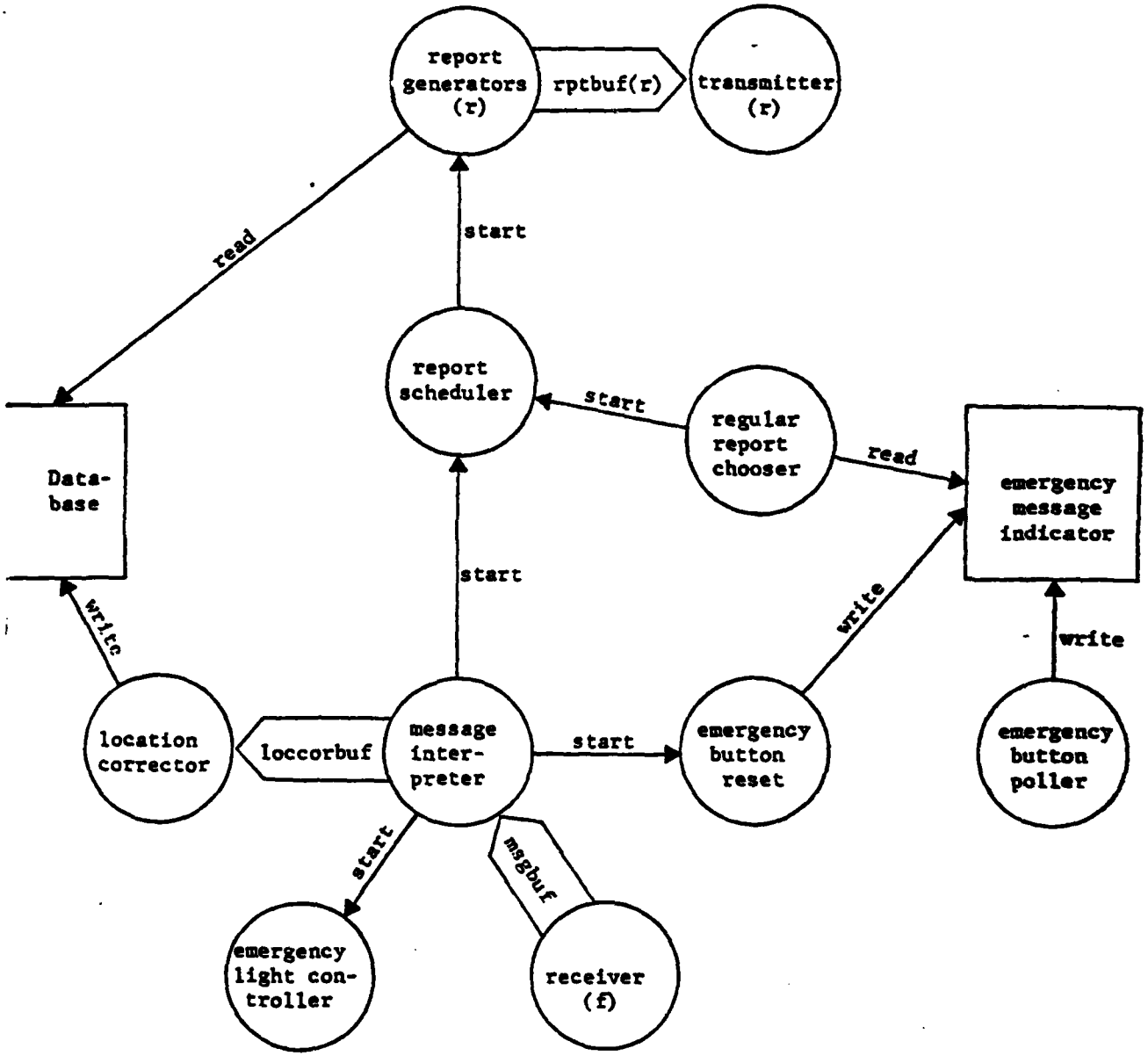
It is my proposal that CSD's CPDS for HAS be rejected, and that they be asked to follow the design in the enclosure.

E. Newhire


sensor readers (s)
obsbuf(s)
internal averagers (s)
avbuf(s)
calcu-lators (t)
upbuf(t)
updater
write
Data-base
locupbuf
omega receiver
omobsbuf
location calcula-tion

Symbol Key:


Process, or group of related processes (s): number depends on number of sensors (t): number depends on number of sensor types Otherwise 1
Buffer or group of related buffers (s), (t) or 1 (see left)
Data


report generators (r)
rptbuf(r)
transmitter (r)
read
start
report scheduler
start
regular report chooser
read
Data-base
emergency message indicator
start
write
write
write
location corrector
loccorbuf
message inter-preter
start
emergency button reset
emergency button poller
start
msgbuf
emergency light con-troller
receiver (f)

```
sensor_reader:
reentrant program srdr(sensnum,okfcn,fetfcn,buff,sem);
comment  This program reads a sensor and stores the result in an observation
         buffer.

         sensnum: sensor number identifying individual sensor of a particular
                  type
         okfcn:   function to test operation of the sensor
         fetfcn:  function to retrieve a sample from the sensor
         buff:    observation buffer to insert sample into
         sem:     semaphore to wait on for scheduling
```

Typical parameter values:

sensnum	okfcn	fetfcn	buff	sem	sensor type
1 to 3	okat	fetat	atobsbuf	ats	air temperature
1 to 5	okws	fetws	wsobsbuf	wss	wind speed
1 to 5	okwd	fetwd	wdobsbuf	wds	wind direction
1	okom	fetom	omobsbuf	oms	Omega
1	okwt1	fetwt1	wt1obsbuf	wt1s	water temperature, depth 1
1	okwt2	fetwt2	wt2obsbuf	wt2s	water temperature, depth 2
1	okwt3	fetwt3	wt3obsbuf	wt3s	water temperature, depth 3;

```
parameter integer sensnum;
parameter procedure okfcn, fetfcn;
parameter buffer array buff [1:5];
parameter semaphore array sem [1:5];

begin private integer obs;
    while true do
    begin
        P(sem[sensnum]);
        if okfcn (sensnum) then
            begin
                obs:= fetfcn(sensnum);
                deposit(obs,buff[sensnum]);
            end;
        end-if;
    end;
    end-while;
end;
```

Note: There is a cross reference table for semaphores and buffers at the end of the document. The semaphore table shows which processes call P or V operations for each semaphore. The buffer table shows which processes call Accept or Deposit for each buffer.

```
intermediate_averager:
reentrant program intavg(sensnum,n,okfcn,obsbuf,avbuf);
comment  This program obtains sensor readings from an observation buffer,
         computes an average, and puts the average into an avbuf.

         sensnum: sensor number
         n: number of readings to average
         okfcn: function to determine the status of a sensor
         obsbuf: buffer containing samples
         avbuf: buffer to insert average into

    Typical parameter values:

    sensnum   n        okfcn          obsbuf      avbuf       sensor type

    1 to 3    4        okat           atobsbuf    atavbuf     air temperature
    1 to 5    4        okws           wsobsbuf    wsavbuf     wind speed
    1 to 5    4        okwd           wdobsbuf    wdavbuf     wind direction;

parameter integer sensnum, n;
parameter procedure okfcn;
parameter buffer array obsbuf [1:5];
parameter buffer array avbuf [1:5];

begin private integer nextobs, temp, sum;
    while true do
    begin
        if okfcn(sensnum) then
            begin
                sum:= 0;
                nextobs:= 0;
                while nextobs lt n do
                    begin
                        nextobs:= nextobs + 1;
                        accept(temp,obsbuf[sensnum]);
                        sum:= sum + temp;
                    end;
                end-while;
                deposit(sum/n,avbuf[sensnum]);
            end;
        end-if;
    end;
    end-while;
end;
```

```
average_calculator:
reentrant program avgcal(numsensors,okfcn,avbuf,upbuf);
comment  This program computes the average reading over all sensors of a given
         type.  The sensor readings are obtained from an avbuf and the
         averages put into an upbuf.
         numsensors: number of sensors
         okfcn: function to determine if sensor is working properly
         avbuf: buffer containing readings
         upbuf: buffer to store averages into

    Parameter values:
         okfcn      avbuf       upbuf     numsensors  sensor type

         okat       atavbuf     atupbuf       3       air temperature
         okws       wsavbuf     wsupbuf       5       wind speed
         okwd       wdavbuf     wdupbuf       5       wind direction
         okwt1      wt1obsbuf   wt1upbuf      1       water temperature, depth 1
         okwt2      wt2obsbuf   wt2upbuf      1       water temperature, depth 2
         okwt3      wt3obsbuf   wt3upbuf      1       water temperature, depth 3

parameter integer numsensors;
parameter procedure okfcn;
parameter buffer array avbuf [1:numsensors];
parameter buffer upbuf;

begin private integer nextobs, sum, average, numobs, temp;
    while true do
    begin
         sum:= 0;
         nextobs:= 0;
         numobs:= 0;
         while nextobs lt numsensors do
         begin
              nextobs:= nextobs + 1;
              if okfcn(nextobs) then
                   begin
                        numobs:= numobs + 1;
                        accept(temp,avbuf[nextobs]);
                        sum:= temp + sum;
                   end;
              end-if;
         end;
         end-while;
         if numobs gt 0 then
              begin
                   average:= sum / numobs;
                   deposit(average,upbuf);
              end;
         end-if;
    end;
    end-while;
end;
```

```
location_calculator:
program loccal;
comment  Calculate location from an Omega reading.  The reading is obtained
         from the Omega observation buffer (omobsbuf) and the calculated
         location is put into the location update buffer (locupbuf);
global buffer omobsbuf,locupbuf;
begin private integer temp, location;
    while true do
    begin
       accept(temp,omobsbuf);
       location:= omega_calculation(temp);
       deposit(location,locupbuf);
    end;
    end-while;
end;


updater:
program updr;
comment  This program updates the database from updated values obtained from
         the update buffers (xxupbuf, where xx is at, ws, wd, loc, wt1, wt2,
         or wt3);
global integer numdepths;
global buffer atupbuf,wsupbuf,wdupbuf,locupbuf,wt1upbuf,wt2upbuf,wt3upbuf;
begin private integer atval, wsval,wdval, locval, wtval[1:numdepths];
    while true do
    begin
       accept(atval,atupbuf);
       accept(wsval,wsupbuf);
       accept(wdval,wdupbuf);
       accept(locval,locupbuf);
       accept(wtval(1),wt1upbuf);
       accept(wtval(2),wt2upbuf);
       accept(wtval(3),wt3upbuf);
       addframe(atval,wsval,wdval,locval,wtval);
    end;
    end-while;
end;
```

```
receiver:
reentrant program rcvr(synch,freq);
comment  This program receives messages and inserts them into the message
         buffer.
         synch: semaphore to signal that it is time to check for a message
         freq : desired monitoring frequency

    Typical parameter values:

         synch        freq

         afrcv        160000
         sfrcv        300
         satfrcv      180000;

comment  Note that obtainrcvr and releasercvr are monitors controlling access
         to the set of receivers.  No other programs even know how many
         receivers there are;

parameter semaphore synch;
parameter integer freq;
global buffer msgbuf;
begin private string msg; private boolean msg_detected,
              end_of_message; private char rchar;
          private integer rcvrnum;
    while true do
    begin
         P(synch);
         rcvrnum:= obtainrcvr(freq);
         msg_detected:= signal_detected(rcvrnum)
         if msg_detected then
           begin
              initmsg(msg);
              end_of_message:= false;
              while not end_of_message do
              begin
                  rchar:= receive(rcvrnum);
                  set_next_char(msg,rchar);
                  if rchar = "eom character" then
                     end_of_message:= true;
                     else end_of_message:= false;
                  end-if;
              end;
              end-while;
              deposit(msg,msgbuf);
           end;
         end-if;
         releasercvr(rcvrnum);
    end;
    end-while;
end;
```

```
message_interpreter:
program msgint;
comment  Examine incoming message to determine desired action.  Messages are
         obtained from the message buffer.  If the message is a report
         request, the appropriate request variable is incremented and the
         report scheduler is signalled.  If the message is a location
         correction, the location is put into the location correction buffer
         (loccorbuf).  If the message is a request to change the emergency
         light or message, the appropriate process is signalled;
global buffer msgbuf, loccorbuf;
global semaphore elon, eloff, emoff, reptsched;
begin private string message; private boolean rept_request;
        global integer ship_rept_req, air_rept_req, hist_rept_req;
    while true do
    begin
      rept_request:= false;
      accept(message,msgbuf);
      case fet_msgtype(message) of
         //shipreq//
            begin
              ship_rept_req:= ship_rept_req + 1;
              rept_request:= true;
            end;
         //airreq//
            begin
              air_rept_req:= air_rept_req + 1;
              rept_request:= true;
            end;
         //histreq//
            begin
              hist_rept_req:= hist_rept_req + 1;
              rept_request:= true;
            end;
         //emlighton//
              V(elon);
         //emlightoff//
              V(eloff);
         //emmsgoff//
              V(emoff);
         //locupdate//
              deposit(findloc(message),loccorbuf);
      end-case;
      if rept_request then
         V(reptsched);
      end-if;
    end;
    end-while;
end;
```

```
regular_report_starter:
program regrepstart;
comment  Decide whether periodic report or emergency report should be
         broadcast based on status of emergency message indicator.  Signal the
         report scheduler;
global semaphore rrs,reptsched;
begin global boolean embc;
    while true do
    begin global integer sos_rept_req, periodic_rept_req;
      P(rrs);
      if embc then
         sos_rept_req:= sos_rept_req + 1;
      else
         periodic_rept_req:= periodic_rept_req + 1;
      end-if;
      V(reptsched);
    end;
    end-while;
end;
```

```
report_scheduler:
program reportsched;
comment  This program ensures that reports are transmitted in the required
         order.  When the report scheduler is signalled, each report request
         variable is checked and its corresponding report generator is
         signalled if a request is outstanding.  The semaphore "bcast" should
         be initialized to the number of simultaneous broadcasts that can be
         made;
begin global boolean sos_rept_req, air_rept_req, ship_rept_req,
       periodic_rept_req, hist_rept_req;
    while true do
    begin
       P(reptsched);
       P(bcast);
       if sos_rept_req gt 0 then
         begin
            sos_rept_req:= sos_rept_req - 1;
            V(sosrept);
         end;
       else if air_rept_req gt 0 then
         begin
            air_rept_req:= air_rept_req - 1;
            V(airrept);
         end;
       else if ship_rept_req gt 0 then
         begin
            ship_rept_req:= ship_rept_req - 1;
            V(shiprept);
         end;
       else if periodic_rept_req gt 0 then
         begin
            periodic_rept_req:= periodic_rept_req - 1;
            V(periodicrept);
         end;
       else if hist_rept_req gt 0 then
         begin
            hist_rept_req:= hist_rept_req - 1;
            V(histrept);
         end;
       end-if;
       end-if;
       end-if;
       end-if;
       end-if;
    end;
    end-while;
end;
```

```
emergency_report_generator:
program emreportgen;
comment  Add time-dependent information to emergency report and put it in the
         emergency report buffer;
global semaphore sosrept;
global buffer erptbuf;
begin private character string;
    while true do
    begin
      P(sosrept);
      string:= format("(9Hsos from ,A10)",fetemreport);
      deposit(string,erptbuf);
    end;
    end-while;
end;


air_report_generator:
program airreportgen;
comment  Generate an air requested report by taking the necessary data from
         the database and enclosing it with any required delimiters.  Store
         strings in arptbuf for the transmitter;
global semaphore airrept;
global buffer arptbuf;
begin private string string;
    while true do
    begin
       P(airrept);
       string:= initmsg("a");
       deposit(string,arptbuf);
       string:= format("(9Hair temp=,I4)",fetatemp);
       deposit(string,arptbuf);
       string:= format("9Hwind dir=,A30",fetwdir);
       deposit(string,arptbuf);
       string:= format("(11Hwind speed=,I4)",fetwspeed);
       deposit(string,arptbuf);
       string:= end_of_message;
       deposit(string,arptbuf);
    end;
    end-while;
end;
```

```
history_report_generator:
program histreportgen;
comment  Generate a history report by taking the necessary data from the
         database and enclosing it with any required delimiters.  Store
         strings in hrptbuf for the transmitter;
global semaphore histrept;
global buffer hrptbuf;
begin private string string;
    while true do
    begin
       P(histrept);
       string:= initmsg("h");
       deposit(string,hrptbuf);
       while not histcomplete do
       begin
         string:= format("(9Hair temp=,I4)",histatemp);
         deposit(string,hrptbuf);
         string:= format("(9Hwind dir=,A3)",histwdir);
         deposit(string,hrptbuf);
         string:= format("(11Hwind speed=,I4)",histwspeed);
         deposit(string,hrptbuf);
         string:= format("(20Hwater temp at depth ,I4,1H=,I3)",
                  histdepth1,histwtemp1);
         deposit(string,hrptbuf);
         string:= format("(20Hwater temp at depth ,I4,1H=,I3)",
                  histdepth2,histwtemp2);
         deposit(string,hrptbuf);
         string:= format("(20Hwater temp at depth ,I4,1H=,I3)",
                  histdepth3,histwtemp3);
         deposit(string,hrptbuf);
       end;
       end-while;
       string:= end_of_message;
       deposit(string,hrptbuf);
    end;
    end-while;
end;
```

```
periodic_report_generator:
program perreportgen;
comment  Generate a periodic report by taking the necessary data from the
         database and enclosing it with any required delimiters.  Store
         strings in prptbuf for the transmitter;
global semaphore periodicrept;
global buffer prptbuf;
begin private integer nexttime; private string string;
    while true do
    begin
       P(periodicrept);
       string:= initmsg("p");
       deposit(string,prptbuf);
       string:= format("(9Hair temp=,I4)",fetatemp);
       deposit(string,prptbuf);
       string:= format("(9Hwind dir=,A3)",fetwdir);
       deposit(string,prptbuf);
       string:= format("(11Hwind speed=,I4)",fetwspeed);
       deposit(string,prptbuf);
       string:= format("(20Hwater temp at depth ,I4,1H= I3)",
                fetdepth1,fetwtemp1);
       deposit(string,prptbuf);
       string:= format("(20Hwater temp at depth ,I4,1H=,I3)",
                fetdepth2,fetwtemp2);
       deposit(string,prptbuf);
       string:= format("20Hwater temp at depth ,I4,1H=,I3)",
                fetdepth3,fetwtemp3);
       deposit(string,prptbuf);
       string:= end_of_message;
       deposit(string,prptbuf);
    end;
    end-while;
end;
```

```
ship_report_generator:
program shipreportgen;
comment  Generate a ship requested report by taking the necessary data from
         the database and enclosing it with any required delimiters.  Store
         strings in srptubf for the transmitter;
global semaphore shiprept;
global buffer srptbuf;
begin private string string;
    while true do
    begin
       P(shiprept);
       string:= initmsg("s");
       deposit(string,srptbuf);
       string:= format("(9Hair temp=,I4)",fetatemp);
       deposit(string,srptbuf);
       string:= format("(9Hwind dir=,A3)",fetwdir);
       deposit(string,srptbuf);
       string:= format("(11Hwind speed=,I4)",fetwspeed);
       deposit(string,srptbuf);
       string:= format("(20Hwater temp at depth ,I4,1H=,I3)",
                fetdepth1,fetwtemp1);
       deposit(string,srptbuf);
       string:= format("(20Hwater temp at depth ,I4,1H=,I3)",
                fetdepth2,fetwtemp2);
       deposit(string,srptbuf);
       string:= format("(20Hwater temp at depth ,I4,1H=,I3)",
                fetdepth3,fetwtemp3);
       deposit(string,srptbuf);
       string:= format("(6Hdrift=,I3)",fetdrift);
       deposit(string,srptbuf);
       string:= end_of_message;
       deposit(string,srptbuf);
    end;
    end-while;
end;
```

```
transmitter:
reentrant program xmit(freq,rptbuf);
comment  This program broadcasts a report.  The report contents are obtained
         from the buffer rptbuf.
         freq  : frequency to broadcast report on
         rptbuf: buffer containing the report contents

    Typical parameter values:

         rptbuf       freq        report type

         prptbuf      5000      periodic report
         arptbuf      161000    aircraft report
         erptbuf      5100      emergency report
         srptbuf      300       ship report
         hrptbuf      5100      history report;

parameter integer freq;
parameter buffer rptbuf;
global semaphore bcast;

begin private char char; private integer xmitrnum;
    while true do
    begin
       accept(char,rptbuf);
       xmitrnum:= obtainxmitr(freq);
       send(xmitrnum,char);
       while (char ne "end of report character") do
       begin
         accept(char,rptbuf);
         send(xmitrnum,char);
       end;
       end-while;
       releasxmitr(xmitrnum);
       V(bcast);
    end;
    end-while;
end;
```

```
location_corrector:
program loccor;
comment  Obtain locations from the locorbuf buffer and the history database.
         Check to see if the difference is greater than tolerance.  If so,
         start the diagnostics.  Update the location in the history file;
global buffer locorbuf;
begin private integer location, dblocation, tol;
   tol:= "maximum acceptable error in location";
   while true do
   begin
       accept(location,locorbuf);
       dblocation:= fetloc;
       if compare(location,dblocation) lt tol
         then locrec(location);          comment  diagnostics program
       end-if;
       setloc(location);
   end;
   end-while;
end;
```

```
emergency_button_poller:
program embuttonpol;
comment  Check to see if the emergency button has been pushed; if so, set
         emergency message indicator so future periodic reports are replaced
         by emergency broadcasts and signal the regular report starter;
global semaphore ebp,rrs;
begin global boolean embc;
   while true do
   begin
       P(ebp);
       if fet_embutton then
         begin
            embc:= true;
            V(rrs);
         end;
       end-if;
   end;
   end-while;
end;
```

```
emergency_message_reset:
program emmsgreset;
comment  Turn off emergency message indicator;
global semaphore emoff;
begin global boolean embc;
    while true do
    begin
       P(emoff);
       embc:= false;
    end;
    end-while;
end;


emergency_light_on:
program emlighton;
comment  Turn emergency light on;
begin global semaphore elon;
    while true do
    begin
       P(elon);
       set_emlight(on);
    end;
    end-while;
end;


emergency_light_off:
program emlightoff;
comment  Turn emergency light off;
begin global semaphore eloff;
    while true do
    begin
       P(eloff);
       set_emlight(off);
    end;
    end-while;
end;
```

Semaphore	P Called by	V Called by
airrept	air_report_generator	report scheduler
ats(i)	sensor_reader	periodic scheduler*
bcast	report_scheduler	transmitter
eloff	emergency_light_off	message_interpreter
elon	emergency_light_on	message_interpreter
emoff	emergency_message_reset	message_interpreter
epb	emergency_button_poller	periodic scheduler
histrept	history_report_generator	report scheduler
oms	sensor_reader	periodic scheduler
periodicrept	periodic_report_generator	regular_report_starter
reptsched	report_scheduler	message_interpreter OR regular_report_starter
rrs	regular_report_starter	periodic scheduler OR emergency_button_poller
shiprept	ship_report_generator	report_scheduler
sosrept	emergency_report_generator	report_scheduler
synch	receiver	periodic scheduler
wds(i)	sensor_reader	periodic scheduler
wss(i)	sensor_reader	periodic scheduler
wt1s(i)	sensor_reader	periodic scheduler
wt2s(i)	sensor_reader	periodic scheduler
wt3s(i)	sensor_reader	periodic scheduler

Buffer	Accept called by	Deposit called by
**&_obsbuf(i)	intermediate_averager	sensor_reader
omobsbuf	location_calculator	sensor_reader
&_avbuf(i)	average_calculator	intermediate_averager
locupbuf	updater	location_calculator
&_upbuf	updater	average_calculator
msgbuf	message_interpreter	receiver
loccorbuff	location_corrector	message_interpreter
emrptbuf	transmitter	emergency_report_generator
arptbuf	transmitter	air_report_generator
hrptbuf	transmitter	history_report_generator
prptbuf	transmitter	periodic_report_generator
srptbuf	transmitter	ship_report_generator

* periodic scheduler not described in this document

** &_ represents at
ws
wd
wt1
wt2
wt3

HAS.5 Academic Poppycock

EXAMPLE DESCRIPTION

O. U. DeZeeman
Computer System Distributors, INC.
Melamine Desert, California

0. Introduction

Every few years those of us who have been toiling at the production of real-time software for relatively small and slow computers are attacked by newly hired youngsters. Still wet behind the ears and fresh from their "alma mater," they accuse us of producing old-fashioned (now its called unstructured) software. Annoyed by their inability to comprehend instantly programs that have taken years to develop, they attack rather than waiting to learn. They interpret the confusion caused by their own lack of experience with real-world software as a confusion caused by muddy thinking on the part of the people who made the software work.

Einar Newhire's memorandum, "A Structured View of HAS" is a perfect example of the phenomenon. Normally, we just ignore such memos and get on with our work. This one, however, is being taken more seriously than most (perhaps because the name Seawaller has become a household word). For that reason, and because it does happen every year, I have decided that it is worthwhile recording the errors in the Newhire paper. This document can be reissued each time that a new youngster arrives from some ivory tower.

There are three basic faults with the "structured view" which I will discuss in depth:

1. It is an oversimplification — important problems are simply omitted,
2. There are technical problems in implementing the concepts -- run-time and memory usage become excessive,
3. It restricts the designer too much — making his work harder when it is hard enough already. Development costs will increase if we go that route.

1. Oversimplifications

1.1. Omissions

One of the easiest ways of making a computer system or language appear simple and obviously correct is to leave out the hard parts. I do not mean eliminating the hard parts by replacing them with a powerful mechanism; I mean ignoring them in the description. This is a ploy commonly used by professors who write textbooks or tutorial papers on complex systems. They are often complimented on their ability to find a simple description of a system previously thought to be too complex for students to understand. It is only when one attempts to apply the knowledge that one gets from such papers that one discovers that essential information has been omitted and that essential problems have been ignored.

In the case of the Newhire paper, the omission technique has been applied in spades. The paper is written as if all that HAS software has to do is read sensors and write reports. The real HAS software is going to be much more complex because it does more. Among other things, the real software must:

a. allocate memory,
b. allocate registers,
c. keep track of the real-time clock,
d. estimate processor time for completion of incomplete tasks,
e. calculate deadlines,
f. make priority decisions for processors and data areas.

None of these problems is even mentioned in the "structured view" document. It is no wonder that the document has an appealing simplicity.

1.2 Unrealistic distribution of emphasis

Another ploy used by academics in the "structured view" game is to emphasize the simple things. The newer operating systems textbooks devote 70% or 80% of their space to discussing perhaps 20% of the actual code in an operating system. These books spend most of their time discussing the easier things in depth (mutual exclusion, producer/consumer) but the actual implementors spend their time on device handlers, device error analysis, data organization, directories, file systems, etc. The academic may say fervently, "We have to stop thinking in terms of unpredictable interrupts and start thinking in terms of cyclic processes," but the programmer spends a lot of his time writing the interrupt-handling routines anyway.

The same phenomenon occurred in the HAS-structured view paper. The HAS-BEEN computer does not have interrupts; if it had, I would have had one more item for my list of omissions. Because of that, a great deal of code is going to be devoted to scanning input registers checking for conditions that would cause interrupts on more modern computers. I am sure that both Seawaller and Newhire would dismiss this with a "We'll do that in our lowest level," and then go on to talk about a new problem in asynchronous programming. Meantime,

we have to write our polling code and make sure that it gets done in all sections of the programs.

1.3 Implementation of processes ignored

Another illustration of the fact that "A Structured View" is an over-simplification is the issue of the processes. The paper is written as though processes existed already. It ignores the fact that by introducing the concept of processes one has added an implementation problem to the set of things to do. Processes have to be represented by data structures; they have to be synchronized; they have to be scheduled. That doesn't happen by magic; it happens by code. Some of that code might not even be needed if we did not think in terms of processes.

1.4 Interprocess interference

Another example of this oversimplification is the way Newhire handwaves about interference between processes. She blithely has some processes writing in a data structure while others read from it. She acts as if the problem with that is easily solved. Here, she is ignoring even those problems that academics know about. Before I got disgusted with the whole thing, I remembered that a heated debate appeared in the literature about various ways of solving just that one problem. If the Government forces us to go the "structured route" on HAS, they'll discover the reader-writer problem later. Newhire didn't think it important enough to bring it up now, before the decision.

2. Technical problems

2.1 Space requirements for many processes

Newhire's approach is based on having many little processes. She overlooks the fact that, in implementing these processes, she is going to have to reserve a large block of space for each of them. Each process is represented by a data structure that describes both the code that controls the process and the data that the process uses, to say nothing of the data needed to schedule it. Lots of processes — lots of space. Lots of similar processes — lots of duplicate data. On a HAS-BEEN computer we can't afford it.

2.2 Unpredictable delays produced by process synchronization

The process synchronization models that Newhire cites were developed for multiprogramming, not real time. Dijkstra clearly considers the speed of a program to be unimportant. Brinch Hansen has dismissed arguments against his conditional critical sections because they relate to "extreme real-time" situations. Well, in our situation, we are in a real-time situation, possibly even an extreme one, given the incredibly slow speed of HAS-BEEN. The process synchronization concepts deal with all processes as if they were the same. They are simple because they do not distinguish between processes. An urgent process and a normal process might be on the same queue and the urgent one

might then be delayed. If all that you care about is the state at the end of the computation, that's fine. But that's not all that we care about.

2.3 Process switching time overhead

Every time a process has to be scheduled or rescheduled, time must be spent in the process scheduler. Because of the slowness of the HAS-BEEN, we can ill afford such switching overhead.

2.4 Size of the minimal system

Seawaller may be the name most commonly associated with the process concept, but Brinch Hansen has actually gone much further. He has written a how-to-do-it book that everyone can follow; he has built systems. By looking at those systems, one can see yet another aspect of the problem. At a recent meeting in San Francisco, Brinch Hansen admitted that the most trivial system required 6000-7000 words. To that amount we must add all of the real code and the data structures described above. Perhaps with a large machine like a PDP-11 that is acceptable, but with the HAS-BEEN computer it is not.

2.5 Procedure call overhead

A general problem with structured approaches to software is their reliance on procedure calls to keep things simple. Procedure calls require a great deal of environment changing (register saving and restoring). In real-time systems with outdated computers we cannot afford that.

2.6 Inability of the program to take actions conditional on real-time

Key to the process concept is that each process continues in its sequence of actions irrespective of the exact rate of progress. One of the "other processes" in our situation is the advance of real time. There are numerous cases in systems like HAS where an action will be taken only if time permits and will be curtailed when time is scarce. The simplest example is the self-test code, which Newhire doesn't even bother to describe. We do it whenever we have spare time. All of the self-test routines check the real-time clock and relinquish control when time does not permit the test to go on. Programs written using Newhire's approach could not do that.

3. Increased development costs

Most of the structured programming missionaries imply or claim that development costs may be reduced by such methods. Strangely enough, there is no solid experimental evidence in that direction. People have published data, but it's like comparing lightbulbs and pears. In some cases productivity increases but the quality of code goes down. In other cases, the cost of the language development is written off in a research budget. Even in cases where the same job has been done twice, we are left without hard evidence because doing a job the second time is not doing the same job. In the sequel, I wish to argue that a structured approach can actually increase development costs.

3.1 Structuring restricts the programmer thereby making his job harder.

Working on real-time software is, in some ways, very much like surgery. One has to work very, very carefully and use one's complete knowledge of the system's anatomy. Just as a surgeon cannot perform an appendectomy without some knowledge of the circulatory system, the real-time programmer cannot work on a program to record sensor data without knowledge of the memory allocation policy. If we asked the surgeon to perform his operations in such a way that it would work even if the patient's circulatory system were changed, we would be making his job much harder — perhaps impossible. The real-time programmer's job is hard enough — why make it harder?

3.2 Reversing early design decisions

Parnas has pointed out that the most critical design decisions are those made early in the project because later decisions are based on the earlier ones. Reversal of the earlier decisions is costly because it implies reconsideration and possible reversal of all later decisions. The Newhire approach represents a design decision that she wants us to make early in the project. Subsequent reversal of that decision will be very costly. The reversal is inevitable because we need the control that she wants to deny us.

Throughout structured programming one makes decisions on the assumption that future decisions can be ignored. Correcting errors will be very expensive.

3.3 There will be extra documentation costs

If we take the Newhire approach, we will start out documenting a fictitious "virtual" system. We will then start to refine that design (following the precept of stepwise refinement). Each refinement will have to be completely documented. If we were to bypass this documentation, no one would understand what was going on in the abstract programs. At each new stage the old information must be included again. If we just write our program, we only have to document the decision once. Moreover, SECNAV INST 3560.1 will cause us to write yet another set of documents because it does not allow "abstraction."

3.4 Repeated testing

Testing is a necessary process in software development. No one in his right mind believes a program if it has not been run. Newhire's memo already contains some programs. They have to be tested. Testing such programs is not easy because we'll have to simulate the missing operations. Worse, this testing process should be repeated with each refinement. With conventional programming, you only have to test when you finish the subprogram and once more at integration.

3.5 More source code will be produced

Even academic papers have shown that structured programming tends to lead to bigger programs. More code, more cost.

Conclusions

When university computer science departments were first proposed, many opposed them on the grounds that the graduates would have no basic training in either fundamental mathematics or engineering techniques. They would have learned theoretical approaches that had not been proven. Nothing personal, mind you, but Einar Newhire is just such a graduate. She would be more useful to us if she had never heard of a computer. Then she would come here without ridiculous ideas, and we could have taught her what she needs to know. She's a bright girl who has been brainwashed. Maybe this paper will cause her, and her ilk, to see the light and to recognize the "structured view" for the academic poppycock that it is.

The CPDS that we submitted for HAS is an unusually complete piece of documentation. It is unfair but typical for Newhire to complain that it is too complex: thorough documentation of real-life programs contains many details that academics tend to "abstract away" in their programs.

In the meantime, the government must look at the track record. Our approach, whatever its faults, has produced programs that fly. The structured approach has not. Until a real project has been successfully completed using the structured approach, no project manager in his right mind will bet on it.

HAS.6 Separation of Concerns

EXAMPLE DESCRIPTION

Eric W. Seawaller
New Haven University
(submitted to SIGOPS)

Introduction

Professor E. W. Dijkstra has introduced two distinct topics into the computer literature:

(1) Process synchronization: how to write programs that control several computations proceeding "in parallel" at unknown relative speeds, given that these computations share variables and other resources (1968a; 1968b).

(2) Separation of concerns: how to organize a program so that programmers need not think about too many things at one time (1968b; 1972; 1976).

The purpose of this note is to relate these two fields of study, showing how "separation of concerns" can help us evaluate synchronization and resource allocation control structures.

This thesis is not new. Dijkstra's original T.H.E. operating system papers (1968b) clearly indicate that he introduced process synchronization to confine the processor allocation policy to one portion of his system. He stated that a change in the number of processors would impact only one level of his hierarchical structure. In this paper, I want to take real-time constraints into account and discuss some limitations they impose on our ability to separate concerns.

Concerns in real-time programs

By examining existing real-time programs one can distinguish seven classes of concerns:

(1) Sensing, i.e., reading input lines and recording the observed values in internal storage.

(2) Initiating sensing.

(3) Decoding, i.e., recognition of an event defined by a predicate on internal variables.

(4) Calculating system values from the input values.

(5) Responding to events that have been decoded and recorded.

(6) Scheduling, i.e., allocating the processor among the processes that are eligible to run.

(7) Coordinating access to shared resources.

Although conventional real-time programs deal with many of these concerns in the same program text, the concerns are independent in the sense that they can change independently. Observe that:

(a) The algorithm involved in sensor reading is largely independent of the period of observation.

(b) Initiating sensing is critical because information may be lost if sensor reading is delayed. The initiating policy depends more on processor speed than on the interface to the input sensors.

(c) If the sensor values have been recorded in internal variables, decoding them to recognize significant events may usually be delayed without deleterious effect. Similarly using them to calculate system values may also be delayed.

(d) Response to an external event is often complex and may extend over a time period that is much longer than the sensor observation period. Sensor observations often must continue throughout the period of response.

(e) Processor scheduling can be performed knowing only the processor demands of the various tasks and their deadlines. It is not influenced by other properties of the tasks.

(f) Coordinating the usage of shared resources is primarily constrained by the number and nature of the resources. For the most part, it can be arranged without keeping track of momentary processor allocation. However, the average allocation to each process cannot be ignored if real-time deadlines are to be met.[1,2]

---

[1] There are two concerns of processor allocation: 1) momentary allocation: which process has the processor at any given instant, and 2) average allocation: how much of the processor time is allocated to each process on the average over a period of time.

[2] Whether one can ignore momentary allocation is a more complex point than it may appear; we will return to it in the section titled "A fundamental limitation of separation of concerns."

The process model

We assume that we have available a number of parallel processes that communicate by means of

(a) shared variables, and

(b) special synchronization variables that are accessed only by special synchronization operations.

We also assume that synchronization operations can block and release processes. A blocked process may not be scheduled to run; when it is released it becomes eligible to run again.

The duration of a process may be long either because it is complex or because it is not allocated the processor by the scheduler. Delays caused by scheduling are hidden from the process, so it cannot affect them. Synchronization operators explicitly block and release processes, constraining the actions that the scheduler may take. Thus, processes are subject to scheduling, while synchronization operations restrict scheduling by defining which processes are eligible to run.

Proposed real-time software organization

I propose that the software be organized into three kinds of units: processes, schedulers, and monitors.

(A) Processes are sequential subsets of the activities of the system. We use the term "sequential" to indicate that the sequence of events within a process can be determined by an examination of the task to be performed and is not influenced by the number or speed of the available processors and devices. The relative order of events within a single process is determined by a conventional program that controls that process, but the relative order of events in different processes is affected by processor speed and resource speed as well as scheduling policies. In order to be able to ignore processor speed and scheduling policies while designing the other algorithms, processes are regarded as proceeding in parallel at unknown relative speeds, but at real speeds sufficient to satisfy the real-time constraints.

There would be four classes of processes:

(1) Cyclic processes that observe external inputs and record their values in internal variables (Sensing)[3].

[3] Names in parentheses key objects in the proposed organization to the seven concerns in real-time programming.

(2) Processes that examine recorded data to recognize events of significance to the system (Decoding).

(3) Processes that process the recorded data to compute the information required (Calculating).

(4) Processes that are awakened whenever an event is noted and carry out the system's response to the event (Responding).

(B) Schedulers allocate the processor to processes, using scheduling policies to decide which process should run next. There would be two schedulers.

(1) A simple scheduler that deals only with the periodic waking of sensing processes (Initiating). It is assumed that the sensing processes use small, fixed amounts of processor time in each observation cycle. That time is effectively reserved for them and is not available for other processes.

(2) A deadline scheduler that allocates the remaining processor time, giving priority to processes with the most imminent deadlines (Scheduling).

(C) A monitor is simply a module or a collection of routines called by other programs in order to obtain access to a shared resource (Coordinating). There would be one for each type of resource. Each monitor hides both the synchronization method used internally and the changeable aspects of the allocation policy. The monitors will use synchronization primitives (Parnas 1978).

Separation of concerns achieved by proposed organization

The following examples illustrate the separation of concerns achieved by the proposed organization:

(a) If a sensor is replaced by one with a different interface to the computer, the program controlling the process that reads and interprets that sensor is the only program that needs to be changed.

(b) If accuracy requirements or other factors dictate a change in the frequency of reading a sensor, the periodic awakener is changed. This may make a change in the amount of time available for demand scheduling, but in most cases there will be no ripple effect.

(c) Replacing a single processor with a faster or slower model (or with a multiprocessor system) will affect the schedulers, but in fairly straightforward ways. If the code was properly parameterized when written, reprogramming will be minimal.

(d) Changes in the way of detecting or responding to a significant event can usually be confined to the program controlling a single process.

(e) Changes in the availability or allocation policy for resources other than the processor result in changes to individual monitors. The process synchronization routines that are used by these monitors are distinct and should not be affected. Unless required processor time is changed drastically, scheduling is not affected. Processes using the resources need not be affected by the changes.

A fundamental limitation of separation of concerns

Unfortunately, separation of concerns in the area of resource allocation is not always possible. The resource allocation strategies may interact strongly through their effects on processes that are using two or more distinct resources. If a process that has been allocated some of resource B is slowed by the monitor that allocates resource A, then the policy used in allocating A may have noticeable effects on the allocation of B.

P. J. Courtois (1975; 1977) has carefully investigated this problem and developed statistical criteria to help recognize situations where there would be too much error caused by ignoring the allocation policy for one resource when designing an allocator for another. Very roughly, if we wish to neglect the dynamics of resource A when concerned with resource B, the actions that change the state of resource A must be of short duration and occur relatively frequently when compared to actions that change the state of resource B. When this is valid, one is justified in considering A to be almost continually available but somewhat slower than the actual A when allocating resource B. For a particular process, we will consider A to perform at a rate equal to its actual rate multiplied by the fraction of the time that it is available for that process.

In designing systems of cooperating processes, we want to be able to neglect the momentary processor allocation, including the time used to accomplish process switching, when allocating other resources. We will only succeed if operations requesting or releasing the processor can be of significantly shorter duration than operations requesting and releasing other resources. If this is so, then we can view the processor as almost continually available for each process, but considerably slower then the actual processor.

Process synchronization primitives

The fundamental limitation suggests that successful application of the process concept will require that the software include several levels of process synchronization primitives (Parnas 1978).

1. The lowest level must include only operations with extremely short execution time. They serve primarily to inform the schedulers of changes in the state of readiness of processes.

2. Using the lower level operations, one can implement monitors or other operations that are convenient for resource allocation. The implementation must keep processes that are waiting for a chance to execute the lower level operations distinct from processes that are waiting for entrance to the monitors and processes waiting for resources.

3. Decoding problems such as those discussed by Patil (1971) and Parnas (1975b) are implemented using the lowest level primitives. The real-time constraints on the duration of the lowest level primitives do not necessarily apply to decoding operators such as those proposed by Patil.

Conclusions

The structure described above seems a plausible way to improve the organization of real-time software. It should be further evaluated by means of prototype software.

HAS.7 Implementing Processes in HAS

EXAMPLE DESCRIPTION

S S S S

SEAWALLER SOFTWARE SYSTEMS SERVICE

our motto

STRUCTURED SOFTWARE SAVES

Introduction

Software Technologist Einar Newhire of the Naval Electronics Research Lab (NERL) has proposed that the Host-At-Sea (HAS) software be implemented using processes as a structuring concept. O. U. DeZeeman of Computer System Distributors (CSD) has written a rather sharply worded paper indicating that practical limitations prevent the implementation of processes for HAS. Under terms of NERL contract NERL-CSS 42089a-216, this paper has been written to resolve the conflict.

SSSS's position on the controversy is clear. The arguments on both sides have definite merit. Newhire's approach leads to a better structured system, but she failed to describe the system completely. The failure to deal with certain issues creates the impression that they have been overlooked. In fact, Newhire simply abstracted. Those issues can be dealt with separately from the ones discussed.

This report is divided into two sections. Section I, a modification of a SIGOPS paper (HAS.6), discusses the motivation for the process concept in HAS. Addressed specifically to HAS, it indicates what can be gained by using processes as a structuring concept. Section II addresses the question of how to implement processes. It includes a set of macros that can be used to translate the processes described in Einar Newhire's proposal into code for the HAS-BEEN computer. Because this contract did not call for study of HAS-BEEN assembly language, we have used a simple, machine-independent notation to describe our code. To avoid confusion, we have used the same notation that CSD used to describe its proposed implementation of HAS.

SECTION I

Professor E. W. Dijkstra has introduced two distinct topics into the computer literature:

(1) Process synchronization: how to write programs that control several computations proceeding in parallel at unknown relative speeds, given that these computations share variables and other resources.

(2) Separation of concerns: how to organize a program so that programmers need not think about too many things at one time.

The purpose of this section is to relate these two fields of study to HAS.

Concerns in HAS

By examining the CSD design for HAS, one can distinguish seven classes of functions to be performed.

(1) Sensing, i.e., reading input lines and recording the observed values in internal variables.

(2) Initiating sensing.

(3) Decoding, i.e., recognition of an event defined by a predicate on several internal variables.

(4) Calculating system values from input values.

(5) Responding to events that have been decoded.

(6) Scheduling i.e., allocating the processor among the processes that are eligible to run.

(7) Coordinating access to shared resources.

Although the CSD design for HAS deals with many of these concerns in the same program text, the concerns are independent because they could change independently. Observe that:

(a) The algorithm involved in sensor reading is largely independent of the period of observation.

(b) Initiating observations is critical because information may be lost if sensor readings are delayed. The initiating policy depends more on processor speed than on the interface to the input sensors.

(c) If the sensor values have been recorded in internal variables, decoding them to recognize significant events may usually be delayed somewhat without deleterious effect. Similarly, using input values to calculate system values may usually be delayed.

(d) Response to an external event is often complex and may extend over a time period that is much longer than the sensor observation period. Sensor observations often must continue throughout the period of response.

(e) Processor scheduling can be performed knowing only the processor demands of the various tasks and their deadlines. It is not influenced by other properties of these tasks.

(f) Coordinating the usage of shared resources is primarily constrained by the number and nature of those resources and may often be arranged without taking the momentary processor allocation into consideration. The average processor allocation cannot be ignored because HAS has real-time constraints.[1], [2]

The process model

Einar Newhire's design assumes that we have available a number of processes that communicate by means of:

(a) shared variables, and

(b) special synchronization variables that are accessed only by special synchronization operations.

We also assume that synchronization operations can block and release processes. A blocked process is not eligible to run until it is released.

The duration of a process may be long either because it is complex or because it is not allocated the processor by the scheduler. Delays caused by scheduling are hidden from it, so that it cannot affect them. Synchronization operators explicitly block and release processes, restricting scheduling by defining which processes are eligible to run.

---

[1] There are two separate concerns of processor allocation that are being distinguished here: 1) momentary allocation: which process has the processor at any given instant, and 2) average allocation: how much of the processor time is allocated to each process on the average over a period of time.

[2] Whether one can ignore momentary allocation is a more complex point than it may appear and we will return to it in the section called "A fundamental limitation of separation of concerns."

Proposed real-time software organization

The HAS software should be organized into two kinds of units to be known as processes and modules. The modules include monitors and schedulers, as described in Newhire's module design (HAS.3).

Processes are sequential subsets of the activities of the system. We use the term "sequential" to indicate that the sequence of events within a process can be determined by an examination of the task to be performed and is not influenced by either the number or the speed of the processors and devices. The relative order of events within a single process is determined by a conventional program that controls that process, but the relative order of events in different processes is affected by processor speed and scheduling policies. To achieve separation of concerns, processes are regarded as proceeding in parallel at unknown relative speeds, but at real speeds sufficient to satisfy the HAS timing constraints.

There are four classes of processes:

(1) Cyclic processes, run periodically, that observe external inputs and record their values in internal variables (Sensing)[3].

(2) Processes that examine recorded data to recognize events of significance to the system (Decoding).

(3) Processes that process recorded data to compute the information desired (Calculating).

(4) Processes that are awakened whenever a significant event is noted and carry out the system's response to the event (Responding).

There are two schedulers:

(1) A simple scheduler that deals only with the periodic starting of sensing processes (Initiating). It is assumed that the sensing processes use small, fixed amounts of processor time in each observation cycle. That time is reserved for them and unavailable to other processes.

(2) A deadline scheduler that allocates the remaining processor time among the other processes (Scheduling).

A monitor module is a collection of routines that are called by the other programs when they need to obtain access to a shared resource (Coordinating). There would be one monitor for each type of resource. Each monitor hides both the synchronization method that is used internally and the changeable aspects of the allocation policy. The monitors use synchronization primitives (Dijkstra 1968; Parnas 1976) to implement coordination.

---

[3] Names in parentheses key these objects to the seven HAS concerns mentioned earlier.

The following examples illustrate the separation of concerns achieved by the proposed organization:

(1) If a sensor is replaced by one with a different interface to the computer, the program controlling the sensing process for that sensor is the only program that needs to be changed.

(2) If accuracy requirements or other factors dictate a change in the frequency of reading a sensor, the simple scheduler that initiates periodic processes is changed. This may make a change in the amount of time available for demand scheduling, but in HAS there will be no ripple effect because the HAS-BEEN has plenty of extra CPU cycles according to our analysis.

(3) Replacing a single processor with a faster or slower model or with a multiprocessor system will affect the schedulers in fairly straightforward ways. Reprogramming will be minimal. The code describing the processes themselves need not change at all.

(4) A change in the algorithm used to detect or to respond to a significant event can be confined to the program controlling a single process.

(5) Changes in the availability or allocation policy for non-processor resources result in changes to individual monitors. The process synchronization routines that are used by these monitors will not be affected. Unless processor time required is changed drastically, scheduling is not affected. Processes need not be affected by such changes.

A fundamental limitation on separation of concerns

Unfortunately, separation of concerns in the area of resource allocation is not always possible. The resource allocation strategies may interact strongly through their effects on processes that are using two or more distinct resources. If a process that has been allocated some of resource B is slowed by the monitor that allocates resource A, then the policy used in allocating A may have noticeable effects on the allocation of B.

P. J. Courtois (1975; 1977) has carefully investigated this problem and developed statistical criteria to help recognize situations where there would be excessive error caused by ignoring the allocation policy for one resource when designing an allocator for another. Very roughly, if we wish to neglect the dynamics of resource A when concerned with resource B, the actions that change the state of resource A must be of short duration and occur relatively frequently when compared to actions that change the state of resource B. When this is valid, one is justified in considering A to be almost continually available but somewhat slower than the actual A when allocating B. For a particular process, we will consider A to be performing at a rate equal to its

actual rate multiplied by the fraction of the time that it is available for that process.

In HAS, we want to be able to neglect the momentary processor allocation when allocating other resources. We will only succeed if operations requesting or releasing the processor (synchronization operations) can be of significantly shorter duration than operations requesting and releasing other resources.

These conditions can be satisfied by the HAS design proposed by Einar Newhire.

SECTION II

In Section I of this paper, we showed that the motivation for using Newhire's approach is to achieve separation of the seven types of concerns encountered in HAS. The processes given in Newhire's proposal take care of four of them: sensing, decoding, calculating, and responding. To make a complete system, it is necessary to write the code for periodic scheduling, processor allocation, and resource coordination. We propose to do this in four stages. In the first stage we will take care of resource coordination by introducing monitors for the shared resources. In the second stage we will take care of the periodic scheduling or "initiating" by introducing a special process analogous to a hotel desk clerk to function as an alarm clock. It waits until a certain time is reached, and then awakens another process. In the third stage, we will implement the synchronization routines that change the set of processes eligible to run. In the final stage, we will implement the scheduler that allocates the real processor(s) among the processes eligible to run, so that all make smooth progress.

We ask Mr. DeZeeman and other readers to be patient with this slow, multistage approach. Our purpose, like Newhire's, is to deal with one problem at a time so that the human brain can handle the degree of complexity required at any given time.

Stage 1: Coordination of Shared Resources.

In HAS resources are shared in two ways: explicitly and implicitly. The explicitly shared resources are primarily data structures, tables, and I/O devices. The implicitly shared resources are the "private" memory areas of the processes. Although we chose to ignore this problem while writing the basic process controller programs, there comes a time when we must recognize that the memory areas of all processes are shared with the memory check process(es), so that no area is really private to a process.

When do we need a monitor?

We need a monitor for any resource such that (a) it is used by more than one process and (b) simultaneous or overlapping attempts to use the resource would result in errors. We need a monitor for any such resource whether it be a single boolean variable, an I/O device, or a mass storage device. For each type of resource there is a usage discipline or protocol that will guarantee that the user processes do not interfere with each other. The monitor is a set of supervisory routines guaranteeing that the discipline is followed.

An example of a shared resource is a buffer that is used to communicate between two processes. Another example is a shared variable, such as embc in HAS.4, that is set by one process and checked by another. All of the data structures used to store the temperatures, wind speeds, and other data are *also considered to be shared variables in this context.*

It is essential to observe that we are talking about variables that are shared among processes, not about variables that are shared among modules. The recorded data are stored in variables that are private to a record storage module, but the module is used by several processes. Those processes may all call the module's access functions, and we must guard against the danger of simultaneous or overlapping access by two or more processes.

What is a monitor?

Some authors (e.g., Brinch Hansen, Hoare) give the term monitor a very narrow meaning as a specific construct in a programming language. At Seawaller Software Systems Service, we use "monitor" in its original and more general sense. Monitor refers simply to the collection of procedures that access the resource directly and hence are able to monitor the accesses. We are not going to propose a specific language construct for monitors for two reasons:

(1) HAS will be implemented using an existing language.

(2) Different types of resources require different types of monitors.

SSSS considers designing the monitors to be a system design problem, not a language design problem. For example, for a data structure that consists of a single word in core, the hardware provides the necessary "monitor" by prohibiting simultaneous reads and writes.

The monitor procedures will use the access procedures of the modules that hide the implementation details of the resource. These access procedures, such as GET and SET functions, hide the data structures and access algorithms. They would suffice if simultaneous calls were not a danger. For more general situations, these procedures will be used by additional programs that synchronize use of the resource by multiple processes. The external interface to the monitor may look like the basic access functions, but it need not. The design criteria and the monitors needed in HAS are discussed below.

Monitors for HAS

The HAS structure proposed by Newhire contains four types of computer resources that are shared among processes: single variables, buffers, the data structures hidden by the record storage module, and the "private" variables of each process. In addition, there are system resources, such as transmitters and receivers.

Buffer Monitors

For buffers, Habermann's ACCEPT and DEPOSIT procedures (1972) can be used as monitors. They use Dijkstra's P and V operators to guard against incorrect simultaneous access by multiple processes. Note that, unlike the monitors built into CONCURRENT PASCAL (Brinch Hansen, 1975), they do not always exclude each other's executions. Mutual exclusion is not necessary for this type of buffer; unless the routines try to operate on the same frame in the buffer, an ACCEPT and a DEPOSIT may occur simultaneously without ill effect.

In the algorithms below, the semaphores "out.xbuf" and "in.xbuf" prevent more than one process from simultaneously accepting from or depositing into a buffer, respectively. The semaphore "space.xbuf" synchronizes the processes, preventing buffer overflow and "data.xbuf" prevents buffer underflow; these two semaphores prevent a simultaneous ACCEPT and DEPOSIT on the same frame. "space.xbuf" must be initialized to the number of initially available frames in the buffer, "data.xbuf" is initialized to zero, and the other semaphores are initialized to 1.

"front.xbuf" is an index to a buffer location; it always points to the first empty buffer slot if no DEPOSIT is in execution. Similarly, "rear.xbuf" points to the frame preceding the first full frame unless an ACCEPT is in progress. "successor" returns a pointer to the next buffer frame succeeding the one pointed to by the parameter.

```
procedure deposit(x, xbuf);
begin global pointer front.xbuf; parameter x;  comment  x is the data to be
                                                          stored
    global buffer xbuf;
    semaphore in.xbuf, space.xbuf, data.xbuf;
    P(in.xbuf);            comment  only one process can deposit at a time;
    P(space.xbuf);         comment  wait if the buffer is full;
    xbuf(front.xbuf):= x;
    front.xbuf:= successor(front.xbuf);
    V(data.xbuf);          comment  signal that the buffer is not empty;
    V(in.xbuf);
end;

procedure accept(x, xbuf);
begin global pointer rear.xbuf; parameter x;  comment x is the data to be
                                                        retrieved
      global buffer xbuf;
      semaphore out.xbuf, space.xbuf, data.xbuf;
    P(out.xbuf);           comment  only one process can accept at a time;
    P(data.xbuf);          comment  wait if the buffer is empty;
    rear.xbuf:= successor(rear.xbuf);
    x:= xbuf(rear.xbuf);
    V(space.xbuf);         comment  signal that the buffer is not full;
    V(out.xbuf);
end;
```

Data Structure Monitors

The record storage functions represent a shared data structure that is a classic instance of the "reader-writer" problem (Courtois 1971). For each such "holder" there is a process that periodically updates the information, and there are several other processes (the report generator processes) that use the information to prepare messages. The latter are "reader" processes and do not interfere[4] with each other. Since the updating processes write

[4] They may slow each other down but they cannot affect the results.

in the data structures, the information in the holder will be inconsistent while one of these processes is in the middle of an update. To design the monitor for each type of record, it is first necessary to decide exactly what is meant by consistent data. If the data items being stored are not to be compared to each other, then the data may be considered consistent after each individual item has been completely updated. On the other hand, if the data are to be compared (e.g., to compute temperature gradients), then it is important that all items taken from the storage represent the same point in time. In that case the data will not be considered consistent between the time that the updating process starts to insert new values and the time that it finishes.

In the first case, where individual data items may be updated and the report is considered consistent, the update access functions provided by the monitor will look like the individual SET functions provided by the basic module. In the second case, since a number of items must be considered consistent, the monitor must provide a single access function for updating all of them.

Example of the Second Case:

If the record storage module provides functions SETTEM1, SETTEM2, and SETTEM3 to store temperature values, and all three temperatures should represent the same moment because differences will be computed, the access monitor will use the three SETTEM functions to implement SETTEM(P1,P2,P3).

SETTEM and the other monitor access functions will contain the necessary synchronization operators to guarantee that after one of the TEMP items has been updated, no data will be used until all three have been updated. At all other times, access to TEMP1, TEMP2, TEMP3 need not be restricted.

Assuming the availability of the three functions above and FETTEM1, FETTEM2, and FETTEM3 to fetch values in a similar way, the monitor procedures would look as follows:

```
INITIALIZATION
begin global integer readcount:=0;
    global semaphore temcs, temw;
    temcs:=1; temw:=1;
end;
```

```
procedure SETTEM(p1, p2, p3);
begin global semaphore temw; parameter p1, p2, p3;
    P(temw);                                    comment  wait if any other process
                                                         operating on data, else
                                                         lock out other processes;
    settem1(p1);
    settem2(p2);
    settem3(p3);
    V(temw);
end;

procedure FETTEM(p1, p2, p3);
begin global integer readcount; parameter p1,p2, p3;
      global semaphore temw, temcs;
    P(temcs);                                   comment  mutually exclusive access
                                                         to readcount;
    readcount:=readcount+1;
    if readcount = 1 then P(temw); end-if;      comment  if first reader, lock out
                                                         writers, wait if writer
                                                         already in progress;
    V(temcs);
    p1:=fettem1;
    p2:=fettem2;
    p3:=fettem3;
    P(temcs);
    readcount:=readcount-1;
    if readcount=0 then V(temw); end-if;        comment  allow waiting writer to
                                                         proceed if no other
                                                         reader;
    V(temcs);
end;
```

Monitor for private memory areas

One of the requirements for the HAS program is that all areas of memory are periodically tested by a memory check process. This includes the areas storing code and data that are private to the other processes. Since memory checking destroys the contents of the memory, the contents must be copied to an area that is private to the memory check process before the test begins. They will be returned after the check is complete. During the test it is not possible to execute the relocated code or access the relocated data.[5] Thus, a process may not run while its memory is being checked.

Each of the processes in Newhire's proposal has a clearly defined homing state, that is, a state in which it remains most of the time and in which it may be safely suspended. To build the monitors for the private code and data of each process, we will introduce one additional semaphore per process. The process does a V on this semaphore before entering the homing state and a P upon starting up again. The memory check process does the P before beginning to relocate the data and a V when the data is returned. Thus, execution of the process and its memory check cannot occur at the same time.

Because shared data areas are already protected by semaphores, the memory check process can use them like any other user.

Shared code areas used by reentrant processes are not already protected because the processes that execute them are "readers": they do not alter the code. In such cases, the shared code must be equipped with reader-writer entry points. The memory check process is the writer.

---

[5] This design for memory checking is based upon the following two assumptions that hold for HAS at present. The assumptions must be noted because the design must be changed if the assumptions no longer hold.

(1) The HAS-BEEN computer is not equipped with hardware that supports run-time relocatable code or data (such as the PDP-11/40 segmentation hardware). If we had such hardware it would be a simple matter to relocate the data and code and perform the memory check without concern for the progress of the process.

(2) The time frame in which sensor readings must be made is relatively long compared to the time required to check the memory belonging to a process. Therefore, not scheduling the process during that time is an acceptable solution. If the time frames involved were shorter, it would be necessary to segment the code and data, thereby reducing the time for any one memory check, or to duplicate the code and data so that progress could be made during the memory check.

There are two areas of memory that are not covered by the above discussions. These are the areas of memory in which the memory-check process resides and the areas of memory devoted to the semaphores. If the memory-check process is to check its own memory, it must have a copy of itself (or a subset) elsewhere. This copy checks the memory-check process as if it were any other process.

The semaphores are a problem because they are not private to any other process and can not be protected by semaphores. Luckily, they represent a very small part of memory, so that all other processes can be suspended while they are being checked.

Stage 2: The "desk clerk" procedure

Even after all of the processes in Newhire's proposal have been refined to include the synchronization implicit in the resource monitors, there is nothing in the code that refers to real time. All of the processes are now synchronized; they will not interfere with each other. This can be proved without making any assumptions about their real speeds or relative speeds. This is a very important property. Designing a system of processes so that their correct cooperation depended on each process proceeding strictly according to a rigid schedule would be like planning a subway system without safety interlocks, on the assumption that the trains would always be on time.

Unlike some multiprogramming systems, the HAS system has real-time deadlines for some of its work that are really "hard" deadlines. By hard deadlines, we mean that the old adage, "better late than never", does not hold. Data that is not read in time is lost forever. Out-of-date data may lead to drastic errors. We must take real time into account somewhere.

Even though many processes may wait for particular points in time, we propose that all observation of the actual time be confined to a single procedure. Each process that needs to wait executes a P operation on a semaphore. The single procedure that observes real time will do the necessary V operation at the proper time. When that happens, the waiting process is marked ready and begins to compete for the processors, making progress in accordance with the scheduling policies. We stated earlier that blocks of time are reserved for these periodic, time-driven processes. This is implemented by assigning them higher priorities than the demand processes.

This approach has certain advantages:

(1) The concern, "What do I do when my time comes?" is separated from "Is it my time to run yet?"

(2) The concern "Which of the ready processes should be run now?" is separated from "Which processes should be ready at this point in time?" (Both of these have been called scheduling and considered one problem in the past.)

(3) Waiting for a given amount of time to pass is only done by one procedure. Certain inaccuracies that can occur because two processes are waiting until the same point in time are more easily avoided.

(4) Changes in the real-time schedule are confined to the desk-clerk program sketched below. When this program runs is discussed later.

```
if time interval elapsed then
  begin
     V(semaphores on list for this interval);
     determine next interval;
  end;
end-if;
```

Stage 3: Implementing the P and V routines

Our code now contains many calls of the synchronization routines invented by E. W. Dijkstra (1968a, 1968b). P is used by a process to try to pass a semaphore and to mark its passage. V is used to allow a semaphore to be passed. The specifications for P and V could be written as shown below:

P AND V WITH TRACES

The specification for P and V recognizes the fact that, when dealing with parallel processes, the events that are described by a trace are not simply calls of routines. We must introduce P_b, which is the event of the start of a P, and P_e, the end of a P. Only by treating these as distinct events can we describe the waiting that occurs during a P. In this specification, illegal traces never occur. A call on P (the event P_b) is always allowed. The P-V module delays the P_e until it is legal. Thus, for this specification, the legal traces might be more accurately called possible traces. These assertions refer only to traces on the events for one semaphore.

SYNTAX:

(1) P: semaphore —) semaphore
(2) V: semaphore —) semaphore

gt: greater than
le: less than or equal to
L: Legal
=: equivalence

SEMANTICS:

Legality:

i = number of P operations that can be completed before any V operation is done (usually one)

(1) (n gt 0) =) $L(V^n.P_b.P_e)$
(2) (n gt 0) =) $L((P_b.P_e)^i.P_b^n.V.P_e)$
(3) (n le i) =) $L((P_b.P_e)^n.V)$

Equivalences:

(1) $L(T.P_b.P_e.V)$ =) $T = T.P_b.P_e.V$
(2) $L(T.P_b.V.P_e)$ =) $T = T.P_b.V.P_e$
(3) $L(T.V.P_b.P_e)$ =) $T = T.V.P_b.P_e$

This does not bind us to any particular implementation.

Dijkstra has published two slightly different implementations of the P and V operations (1968a, 1968b). Both satisfy the above requirements, but we believe that the implementation proposed in this section is appropriate for HAS.

We implement each semaphore using an integer variable and a set variable. The set variable can contain zero, one, or more processes. There are operators to insert a process in the set, to remove a process from the set, and to ask if a specific process is in the set. The integer variable is usually initialized to 1 and the set variable to empty. However, it would be permissible to initialize the integer variable to any non-negative value if the set variable is empty, and to negative values if the number of processes in the set variable is equal to the magnitude of the integer variable.

A P-operation is then implemented by decrementing the integer variable and testing it. If the integer variable is negative when tested, the process enters itself in the semaphore's set variable and releases the processor to ready processes. The process is now blocked. This action corresponds to P_b. If the integer variable is non-negative, the process may continue. This action corresponds to P_e.

The V-operation increments the integer variable. If the result is negative or zero, one of the processes is removed from the semaphore's set variable, and entered in one of the sets of ready processes. This process is now "ready" to run. Removing a process from the semaphore set corresponds to P_e.

In this implementation, if the integer variable is negative, its magnitude always represents the number of processes in the set variable. In this case, the integer value is redundant since we could get the same information by counting the entries in the set variable. When the integer value is positive, it is not redundant.

The operation P followed by a V or the operation V followed by a P will leave the number of the processes in the set unchanged. The fact that the order of these events does not matter, except with regard to the identity of the processes in the set, makes it easier to prove that certain properties of the system will hold even if the speeds of processes change.

There are still two portions of the P and V operation to be refined. If the processor is released in the P operation, a "ready" process must be selected to run. If a process is removed from the semaphore's set in the V operation, one of the members of the set must be selected to be made ready. In neither case have we yet specified which one. If you look back at the specifications for P and V, you will find no help on this question. The choice of a process from the set members is not constrained by the requirements. This has the advantage that any program proven correct using only the specifications of P and V, will work correctly with any policy for selecting set members. For HAS, we suggest two simple policies. We will always remove the longest waiting process from a semaphore set variable. When a process is marked ready, we assign it a priority. We will always select the highest priority process when allocating the processor.

The sets may be represented as First-In-First-Out (FIFO) queues of process descriptors. Each process descriptor will contain the process number, priority, and state information such as register contents and program counter.

Implementations for standard queue manipulation procedures are found in most programming texts; we will not include them here. We will assume we have two procedures, INSERTP (process, queue) to insert a process in the queue, and REMOVEP (process, queue) to remove the longest waiting process from the queue, placing its descriptor in the parameter "process."

The final code for P and V now looks like:

```
procedure P(s);
begin global integer int.s;  global process-descriptor queue set.s;
    int.s:=int.s-1;
    if int.s lt 0 then
       begin
       comment  current_process is a function returning the process descriptor
                of the process running on this processor;
       insertp(current_process, set.s);
       processor_allocate;
       end;
    end-if;
end;

procedure V(s);
begin global integer int.s; global process-descriptor queue set.s;
private process descriptor process;
    int.s:=int.s+1;
    if int.s le 0 then
       begin
       removep(process, set.s);
       make_ready(process);
       end;
    end-if;
end;
```

The "processor_allocate" and "make_ready" routines are described later.

Stage 4: Completing Process Implementation

Adding Preemptive Scheduling

The code that we have now is complete and could be directly translated into running code if we could live with non-preemptive processor scheduling. It has a lot of processes that voluntarily release the processor when they try to execute a P operation and cannot finish it. Because of the mutually cooperative nature of the process structure and the fair scheduling strategy that we have taken, things would go well if we had no real-time deadlines and we really did not care about the relative speeds of the processes. The behavior of the system would be a bit "jerky". One process would run until its input or output buffer emptied or filled, another would then run until a similar event occurred to it, etc.

For HAS, this behavior is not acceptable. We must now add provisions to have the processor preempted between synchronization events. Unfortunately, one of the properties of the HAS-BEEN computer is that it has no interrupt system. Although we may wish to preempt processors, we must really implement things so that the processes release them voluntarily. To ensure smoothness, we will insert instructions every so often to check the priorities of waiting processes and release the processor whenever a waiting process has higher priority than the one currently running. This will be done automatically by postprocessing the code generated by the above expansion and inserting a macro call to the scheduler after every few statements. We are fortunate that the HAS-BEEN computer does have adequate speed for our application even if we insert many instructions in this way.

It is interesting to note that the T.H.E. system (Dijkstra 1968) at one time had a preemptive clock implemented in a similar manner. However, this code was removed permanently after an experiment revealed that the behavior was satisfactory without it. T.H.E. had no real-time demands, but we have the option of trying the same experiment.

The "desk clerk" is given control whenever the processor is reallocated, since checking the real-time clock is the top priority activity.

We will adopt a round-robin policy for processes of equal priority. This strategy is implemented using one queue of ready processes for each priority. When a process gives up running, it inserts itself at the back of the queue with its priority. Then it calls the processor_allocate routine to select and start the next process. The next process is removed from its ready queue before it is run. Code for this routine is shown below:

```
procedure round_robin;
begin
   make_ready(current_process);
   desk_clerk;
   processor_allocate;
end;
```

Process switching routines

The procedure "make_ready" inserts a process into a FIFO queue of ready processes. Code is shown below:

```
procedure make_ready (p);
    begin process-descriptor parameter p; private integer pri;
    global process-descriptor queue array ready_list;
    comment p = process number to insert in ready queue, ready_list
            is an array of queues, one for each process priority level;
    pri:= priority(p);
    insertp (p, ready_list(pri));
    end;
```

The operation of releasing a processor and reassigning it is machine dependent and so will only be sketched here. It involves storing copies of the processor registers in the old process's data area and loading new values into the registers from the new process's data area. The abstract program is shown below:

```
procedure processor_allocate;
begin global process-descriptor queue array ready_list;
    private process-descriptor proc;
    private integer i;
    private constant integer maximum_priority;
    "save registers and PC;"
    comment  find highest-priority, non-empty ready set.  If all the
             other ready sets are empty, there is always a process in the
             0 priority set that consumes time and calls
             processor_allocate again;
    i:= maximum_priority;
    while  ready-list(i) is empty do
        begin i:= i-1; end;
    end-while;
    removep(proc, ready_list(i));  comment  places process descriptor in
                                            proc#;
    load registers and PC from descriptor for process proc;
end;
```

Conclusions

O. U. DeZeeman called Einar Newhire's design unrealistic academic poppycock. He would have been more accurate if he had simply called it incomplete. In this report we have shown how Newhire's design can be refined in a step-by-step way, adding code to implement the missing portions. This code could be added in the form of subroutine calls but we do not recommend that. We propose that the code given here be implemented as macros and inserted in-line so that the many calls will not incur excessive overhead. The resulting code will be a bit hard to read, but no one need read it. The macro expansion process should be automated so that changes can be made to the separate sections rather than to the expanded code. This will save immense labor during the program maintenance part of the HAS system's life cycle.

EVAL.1 Comment Sheets

Name (optional):_____

Instructions

Please use the following sheets to record your observations as we progress through the course. Identify material under discussion by using the relevant document identifier. Highlight what you liked as well as the problems you found. Lengthy comments may be extended into the following entry and should be so indicated by crossing out the intervening typing. Examine the example below.

Collect Comment Sheets and attach to Course Evaluation (EVAL.2); turn in all comments at the end of the course.

Example

Relevant document: PF.2

Not enough time allowed for exercise, but good example of program families.
Typo: "propretiss" on next to last line on first page.

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

Relevant document:_____

EVAL.2 Course Evaluation

Name (optional)_____

Instructions

1. There are eight questions or requests for information. Please respond to all eight.

2. For questions followed by a rating scale, mark the scale with a bar, | , to indicate your answer to the question. Scale calibrations appear below the rating bars. You may rewrite scale calibrations you dislike or do not understand.

   If you are uncertain about your rating, mark a lower bound with a left parenthesis, (, and an upper bound with a right parenthesis,) .

   We encourage you to elaborate on your rating in the space provided below the scale.

   Your rating might look like this .....(.:....|.....:....).....
   XXX XXXX
   I think that this material should...

3. Please attach any Comment Sheets (EVAL.1) you completed during the course.

Questions

1. What percentage of the time that you devote to software do you spend on the following software activities? The sum should approximate 100%.

	0%	25%	50%	75%	100%
Project or Acquisition Management (including contracting, financial management, data item management)	:	:	:	:	:
Software Construction (including analysis, design, code, debug, maintenance, and documentation)	:	:	:	:	:
Software Testing or System Evaluation	:	:	:	:	:
Configuration Management or Quality Assurance	:	:	:	:	:
Software Engineering Research or its Funding	:	:	:	:	:
Teaching	:	:	:	:	:
Other (_____) (specify)	:	:	:	:	:

2. How well were the course goals met? (See section VIII of GEN.1.)

not at all	one-fourth	halfway	three-fourths	totally
:	:	:	:	:

3. What is your overall understanding of the course material?

```
   :.............:.............:.............:.............:
 lost           vague         basics          good          total
```

4.a. How useful will the course be to your work?

```
     :.............:.............:.............:.............:
   fatal          harmful       neutral         helpful       vital
```

b. If you feel it will be useful, how often will it be so?

```
     :.............:.............:.............:.............:
   once           yearly        monthly         weekly        daily
```

5. How was the overall performance of the instructors?

```
   :.............:.............:.............:.............:
terrible        poor          fair            good        excellent
```

6. How do you rate the course overall?

```
   :.............:.............:.............:.............:
terrible        poor          fair            good        excellent
```

7. For each course topic, rate how useful the material will be to your work and rate the quality of presentation. If a bad presentation ruined otherwise useful material, please note this.

	Utility				Presentation				
PROGRAM FAMILIES									
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent
UNDESIRED EVENTS									
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent
INFORMATION-HIDING MODULES									
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent
SPECIFICATIONS									
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent
ABSTRACT INTERFACE MODULES									
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent
HIERARCHICAL STRUCTURES									
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent
LANGUAGE CONSIDERATIONS									
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent

(Question 7 continued)

	Utility					Presentation				
PROCESS STRUCTURE										
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent
DOCUMENTATION										
	fatal	harmful	none	helpful	vital	terrible	poor	fair	good	excellent

8. A list of questions on miscellaneous topics follows. Please add topics of your own if you wish.

   a. What were the good and bad aspects of the programming assignment (MADDS)?

   b. What were the good and bad aspects of the MP and HAS examples?

c. What problems, if any, are there with the pseudo-code (GEN.5)?

d. What definitions should be improved or should be added to the glossary (GLOS.1)?

e. What new topics should be added to the course; which current topics should be dropped?

f. How will you use some of the course ideas in your future work?

g.

GLOS.1 Glossary

A

Abstract — a) as a verb (e.g., "to abstract from a representation") — to ignore certain details in making a description or model of some object.
b) as an adjective (e.g., abstract interface) — an abstract "X" is a model of X that omits certain details of X. Because certain details are not taken into consideration the abstraction represents many possible versions of the object that differ in the ignored details.

Abstract data type — (First, see Data type.) A class of variables that includes more than one data type. A description of an abstract data type describes the common properties of several data types. For example, 'arithmetic' is an abstract data type that includes 15 bit and 30 bit integers as well as floating point numbers. Some authors consider an abstract data type to be a data type whose implementation is hidden from users, who see only the type's *abstract behavior*.

Abstract interface — A model of an interface that is valid for more than one actual interface. All statements made about the abstract interface must be true of all of the actual interfaces that it models.

Abstract program — A program that is incomplete; some details necessary to have it run are omitted. It represents all programs that could be obtained by supplying the missing details in a way consistent with the incomplete description.

Access functions or Access programs — A program in a module that may be called by programs outside of that module. Most run-time communication between modules is effected by invocation of access programs. There are several different sorts of access functions: some return information to the caller, some change the state of the module to which they belong, and some do both.

Address space — The set of data addresses that a program can use.

Algorithm — A precise description of possible sequences of operations. The algorithm is conditional if the sequence of actions depends on the data provided as input to the algorithm. The algorithm is non-deterministic if more than one sequence of operations is allowed for fixed values of the input data.

B

Buffer — A storage device used to transfer information between system components. The buffer allows the information producers and the information consumers to proceed asynchronously. Producers insert items into the buffer; consumers remove items from the buffer. The system components need not wait for the others unless the buffer is full or empty. Sometimes the word "buffer" is used to mean a first-in-first-out (FIFO) buffer.

C

Coding specification — A coding specification for a given program is a document in which "pseudo-code" or abstract programs are used to constrain the selection of algorithms and data structures or to specify them completely. Whatever the extent of the constraints imposed, the coding specification should contain all information (or references) required to write complete and correct code for the program.

Critical section — A portion of a program that should not be executed simultaneously by several processes. One process entering a critical section must exclude other processes from entering the same section until the first process has left (known as mutual exclusion).

D

Data type — A class of variables (information holders) that can be used as operands for a common set of operators. For example, "8 digit integer" is a data type that can be used as an array index. Some authors define data type as a set of values together with a set of allowed operations. (See also Abstract data type.)

Deadlock — A system state in which a set of processes ceases to make progress because each member of the set is waiting for some other member of the set to complete some action.

Decision postponement — Progress in design is made by making decisions. Because early decisions are harder to reverse than later decisions, making decisions that are likely to be reversed later should be avoided. In decision postponement, decisions unlikely to be reversed are made to allow progress while waiting for the resolution of uncertainties. Abstraction is one method of decision postponement.

Design decision — At the start of a software design project, many programs are possible. As each interface is defined, statements written, etc., the set of possible products is reduced until, at the end, only one program remains. Each act that reduces the set of possible products is a design decision.

E

Embedded computer system — A computer system that is part of a larger system and must meet interfaces that are primarily determined by characteristics of the system in which it is embedded.

Extensible languages — Conventional languages provide a set of built-in features such as boolean variables and while statements. By means of subroutines or procedures one user can extend the facilities available to another user. These extensions are easily distinguished (in syntax, efficiency, and "safety") from those features that are built into the language. An extensible language allows a user of the compiler to provide new features that can be used with the same efficiency and safety as those that were built into the compiler.

F

Fail-soft — A system is referred to as having fail-soft features if the occurrence of an undesired event (such as a hardware failure) results in a partial reduction of services rather than a total failure.

Family — A hardware family is a set of computers with enough in common that it pays to study their common properties before looking at individual models. A program family is a set of programs with enough in common that one begins by studying the common properties and then proceeds to look at one or more of the family members. An abstract program is one way of representing a program family.

FIFO — A first-in-first-out (FIFO) queuing discipline is one in which items inserted into the storage device first are extracted from the device in the order inserted.

Formal specification — A statement, by means of mathematical axioms in a well understood mathematical notation, of the requirements that a module must meet.

Function — Used in three distinct senses in this course: a) the role that a system fulfills is often termed the system's function; b) the access programs of a module are often called functions, after FORTRAN (see also Access function); and c) a function is a mathematical mapping from a domain into a range. The syntax portion of formal specifications describes the domain and range of each access function.

H

Hierarchy — A binary relation on a set of objects defines a hierarchy of those objects if the relation is loop free. (See also Level.)

I

Information-hiding module

A set of programs that allows other programs to use a data structure or algorithm without having those other programs be sensitive to changes in the data structure or algorithm. The other programs use access functions that can be implemented in a compatible way for all allowable changes to the data structure or algorithm. The data structure or algorithm is termed the secret of the module. Equivalently, a module refers to a set of programs written by part of a programmer team. Modules being built by more than one person will themselves be divided into modules. In other discussions you may find "module" being used to mean separately compilable portions of a program, separately callable portions, or separately loadable portions of a program. We reserve the use of the word for portions of a program that are written independently. A module may consist of more than one subroutine or macro, the usual case in this course.

Interface

The set of assumptions that one program makes about another program. If program A violates the assumption(s) that program B makes about it, program B will not work properly. An interface may include assumptions about data structures, entry points, calling sequences, etc., as well as more subtle assumptions about the effects of the programs involved.

L

Level

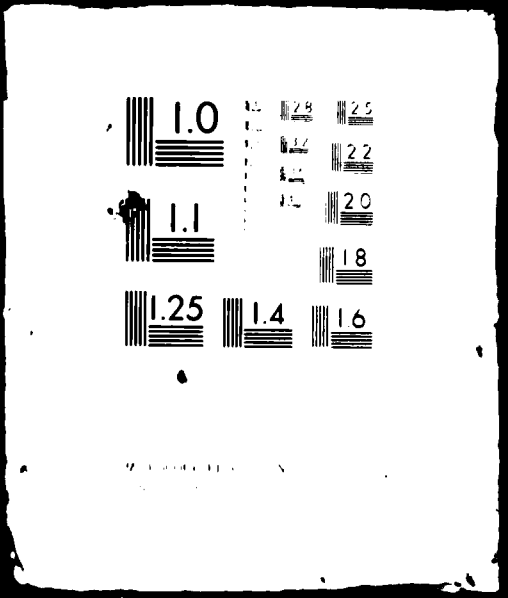
Proper use of the word "level" in describing a software system depends on the definition of a loop-free binary relation between the components of the system. Call that relation R. If the relation R holds between c1 and c2 then R(c1,c2) = true. Level 0 is the set of components c such that there is no component d such that R(c,d) = true. Level i is the set of components c such that a) there is at least one component d in Level i-1 such that R(c,d), and b) if R(c,d) then d is in a lower level than i. The "real meaning" of level depends on the relation R, which should always be specified before using the terms "level" or "hierarchy."

AD-A113 415 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
SOFTWARE ENGINEERING PRINCIPLES 3-14 AUGUST 1981,(U)
AUG 81 L J CHMURA, P C CLEMENTS

UNCLASSIFIED NL

7 7

END
DATE FILMED
5 82
DTIC



M

Macro definition — Associating a name with a program segment known as the body of the macro. The macro name can be used as an abbreviation for the body. Some macro systems allow the definition of the macro body to be conditional.

Macro expansion — The process of taking a program text containing occurrences of the name of the macro (also called calls on the macro) and replacing those calls with the associated bodies. This process is called expansion because usually the body consists of more characters than the name so that the text becomes longer. That need not be the case. Sometimes a macro name may be 'expanded' to an empty string.

Memory allocator — A component of a software system that determines the location of sections of code and data in main memory and mass storage devices.

Module — See Information-hiding module.

Monitor — An information-hiding module in a software system that supervises the use of a given resource. All resource requests and resource releases are made by calling one of the module's access functions. If parallel processes are sharing a resource the monitor synchronizes their activities.

Multiprocessing — Computing on a system in which there are several programmable hardware units in use simultaneously. In normal usage, the term means that there are several CPU's although it can also be applied to systems in which there is one CPU and a set of peripheral programmable units such as channels or front-end processors.

Multiprogramming — Computing on a system in which a single programmable unit is used to execute a number of tasks that proceed independently of each other. In normal usage this refers to systems in which several user jobs may be in the midst of execution simultaneously by intermittent use of a single processor; the term may also be applied to systems in which several processes belonging to the same user job can be in the midst of execution simultaneously. (See also Process.)

Mutual exclusion — See Critical section.

O

O functions/ V functions — The access functions of a module may either change the information in the module or reveal (return) information stored by a module. Those that change the stored information are called operator functions or O-functions. Those that return information are called value functions, or V-functions. O-V functions that do both types of services are possible.

P

P & V — The operators defined by Dijkstra for the data type semaphore (see definition of semaphore). The P operation is used in a Pass attempt. A process may have to wait until the semaphore is passable. The V operation signifies that a Pass attempt should be allowed to complete.

Predicate — A property that may be true or false of some object or ordered tuple of objects. For example, green is a predicate that may be true or false about some object. Greater-than is a predicate that can be defined on pairs of numbers, so that for any specific pair of numbers it will be true or false. Complex predicates are defined in terms of boolean expressions and simpler predicates.

Procedure — The Algol term for a closed subroutine. Some procedures correspond to FORTRAN function subroutines and have a value so that they may appear in expressions. (See also Function.)

Process — A subset of the events in a system. We may describe one or more processes by means of a program that determines the sequence of events. If the system consists of more than one process, the sequence of events in different processes may be determined by the timing of outside events, the relative speeds of devices, scheduling algorithms, etc. This leads us to say that the relative speeds of processes should be considered unknown. (See also Sequential process.)

Program

A specification of an algorithm in a form sufficiently complete either to be executed directly on a computer or to be translated mechanically into a directly executable form. The notation used to specify the algorithm is called the programming language. If programs written in the language can be executed directly on the computer without translation, the language is called a machine language, and the programs are called machine language programs. For languages that are not machine languages, a program, called a translator, that translates from the language into machine language is usually supplied. Most programming languages allow the programmer to divide a program into independently executable components. The component in which execution starts is called the main program. Other components are called subprograms or subroutines. A method of executing subroutines is supplied by the programming language (the call statement in FORTRAN, the call and perform statements in COBOL). A subprogram is executed by calling (or invoking) it. In process structured systems, a program is used to determine the sequence of events for a process, and execution may begin concurrently in several programs (see also Process). A set of programs grouped together for the purpose of concealing a design decision is called an information-hiding module. (See also Information-hiding module.)

Pseudo-code

A program that is not machine executable but is intended to describe the main steps in an algorithm. The software designer can concentrate on the design of an algorithm because the pseudo-code is not as tightly constrained as a real programming language might be.

Q

Queue

A first-in, first-out storage device. In some documents, queue is used in a more general sense to refer to any mechanism capable of storing a set of objects. In that case, the [illegible] or writer must identify the discipline used [illegible]ess.

R

Reader/writer — The "reader/writer problem" is one of the standard problems in process synchronization. It refers to a situation in which several processes wish access to the same data item. Readers do not interfere with each other and may use the item simultaneously. Writers are updating the item and require exclusive access so that they do not interfere with each other or with readers. Computer science literature contains numerous discussions of this problem.

Ready/running/ blocked — The set of states of a process from the point of view of a processor allocator (scheduler). If a process has been allocated by a processor it is running. If it is waiting for some event that will be caused by another process, e.g., a resource becoming available, it is blocked. Processes that are not running or blocked and are waiting for a suitable processor to become available are called ready.

Real-time software — Software in which the programming must take "hard" real-time deadlines into account. A deadline is considered "hard" if the system will be considered to have failed if it delivers the needed results after the deadline.

Redundancy — The use of more information than the minimum needed to describe some situation fully. The extra information is redundant in that it can be computed from other information already supplied. Redundancy is necessary to check for the existence of errors. The more redundancy the greater the class of errors that can be detected and corrected.

Reentrant procedures — A procedure written in such a way that several processes or jobs may use it simultaneously. To accomplish this, the code must be separate from all data that is changed during execution. The code is shared but each process or job has its own copy of changeable variables.

S

Secret of a module — See Information-hiding module.

Semantics — The effect of executing a program or construct. Operational Semantics describes the effect by describing a possible implementation using programs that are assumed to be understood. Abstract Semantics describes the effect in terms of externally visible changes in the values of variables or the behavior of other programs.

Semaphore — A type of variable designed to facilitate the synchronization of processes that are proceeding in parallel at unknown speeds. Just as semaphores in railway systems are used to inform one train of the activities of another, semaphores in computer systems are used to inform one process of the activities of another. See P & V.

Separation of concerns — Refers to a method of simplifying the work of a designer or analyst by having him concentrate on just one aspect of a problem rather than try to deal with all aspects at once. For example, one would want the development of numerical algorithms to be separate from concern with memory allocation policies.

Sequential process — A process is a sequential process if the sequence of events in the process is determined by the algorithm describing the process rather than by the relative speed of other processes.

Sequencing decisions — A subset of design decisions. A design decision is termed a sequencing decision when it reduces the possible sequences of events that could occur in the system.

Specification — A statement of the requirements that a module must satisfy.

Stack — A last-in, first-out storage device. Only the most recently inserted item can be read. When the most recently inserted item is removed, the item that could be read previously can be read again. The name is derived from the analogy of a stack of trays in a cafeteria. Only the top-most tray is visible.

Stepwise refinement — The process of programming by first writing abstract programs in which parts are named but not implemented, and then implementing those parts. The implementation may call on programs that are named but not yet implemented. The process stops when all unimplemented programs have been refined to calls on implemented programs or machine instructions.

Synchronization — The enforcement of timing constraints on parallel processes whose relative speed is unknown. If an event in process A cannot properly occur before an event in process B, A and B must be synchronized.

Syntax — The set of rules that determine what is a legal program in a language. Knowing the syntax of a language, one can tell whether or not a given program has a meaning but one cannot tell what the program does. The description of the effect of the legal programs is termed the semantics of the language.

T

Trap — A deviation from the normal flow of control of a program caused by the detection of some error (undesired event) in the execution of the program. For example, if the execution of a divide command in a program results in overflow, a trap occurs and a special routine for responding to that situation is invoked.

U

Undesired event (UE) — A moment in time that an undesired situation arises. An undesired situation is a binary condition that is unfortunately true. Examples of undesired situations are: (1) data was not received when expected, (2) data was received when not expected, (3) data does not meet specification, and (4) a device or function has malfunctioned.

Uses relation — Given program A with specification S_a and program B, we say that A uses B if A cannot satisfy S_a unless B is present and satisfies some non-trivial specification S_b. The assumed specification S_b may differ for different users of B.

V

Variable — An information holder. The information held is stored and retrieved by means of access operators. Variables are often referred to by means of identifiers.

Virtual machine — A set of programs and data structures that can be used as if they were implemented in hardware. To meet this requirement it must be impossible for programs that use the virtual machine "instructions" to alter the programs that implement those instructions or subvert the resources used in their implementation.

Virtual memory — A mechanism that makes it possible for programs to use addresses that are different from physical memory addresses. The mechanism must function in such a way that the behavior of the program is absolutely independent of the actual memory address except for possible delays in time.

BIB.1 Bibliography

ACM SIGPLAN. 1979. "Preliminary Ada Reference Manual." SIGPLAN Notices, vol. 14, no. 6, part A.

ACM SIGPLAN. 1979. "Rationale for the Design of the Ada Programming Language." SIGPLAN Notices, vol. 14, no. 6, part B.

Anderson, R. B. 1979. Proving Programs Correct. New York: John Wiley & Sons.

Baker, F. T. 1972. "Chief Programmer Team Management of Production Programming." IBM Systems Journal, vol. 11, no. 1, pp. 56-73.

Bartussek, W.; and Parnas, D. L. 1977. Using Traces to Write Abstract Specifications for Software Modules. University of North Carolina Report no. TR 77-012.

Basili, V.; and Weiss, D. 1981. "Evaluation of a Software Requirement Document by Analysis of Change Data." Proceed. Fifth International Conference on Software Engineering (March), pp. 314-323.

Belpaire, G.; and Wilmotte, J. P. 1974. "A Semantic Approach to the Theory of Parallel Processes." International Computing Symposium 1973, A. Guenther et al. (eds.), New York: North Holland Publishing Co.

Bloch, A. 1977. Murphy's Law and Other Reasons Why Things Go Wrong! Los Angeles: Price/Stein/Sloan.

Boehm, B. W. 1973. "Software and Its Impact: A Quantitative Assessment." Datamation, vol. 19, no. 5, pp. 48-59.

Brinch Hansen, P. 1970. "The Nucleus of a Multiprogramming System." Comm. ACM, vol. 13, no. 4, pp. 238-241, 250.

---------. 1973. Operating Systems Principles. Englewood Cliffs: Prentice-Hall.

---------. 1975. "The Programming Language CONCURRENT PASCAL." IEEE Trans. on Software Engineering, vol. 1, no. 2, pp. 199-207.

Britton, K. Heninger; and Weiss, D. 1981. Interface Specifications for the A-7E Extended Computer Module. Naval Research Laboratory Memorandum Report in publication.

---

* Recommended reading

* Brooks, F. P., Jr. 1975. The Mythical Man-Month: Essays on Software Engineering. Reading, Mass.: Addison-Wesley.

Cooprider, L. W.; Heymans, F.; Courtois, P. J.; and Parnas, D. L. 1974. "Information Streams Sharing a Finite Buffer: Other Solutions." Information Processing Letters, vol. 3, no. 1, pp. 16-21.

Coppola, A. 1979. A Design Guide For Built-In Test (BIT). Report RADC-TR-78-224. Available as DTIC Document ADA069384.

Courtois, P. J. 1975. "Decomposability, Instabilities, and Saturation in Multiprogramming Systems." Comm. ACM, vol. 18, no. 7, pp. 371-377.

Courtois, P. J. 1977. Decomposability: Queueing and Computer System Applications. New York: Academic Press.

Courtois P. J.; Heymans, F.; and Parnas, D. L. 1971. "Concurrent Control with 'Readers' and 'Writers.'" Comm. ACM, vol. 14, no. 10, pp. 667-668.

* Dahl, O. J.; Dijkstra, E. W.; and Hoare, C. A. R. 1972. Structured Programming. London: Academic Press.

Daly, E. B. 1977. "Management of Software Development." IEEE Trans. on Software Engineering, vol. SE-3, no. 3, pp. 229-242.

Department of Defense. 1978. Requirements for High Order Computer Programming Language "Steelman."

* Dijkstra, E. W. 1968a. "Co-operating Sequential Processes." Programming Languages, ed. F. Genuys. New York: Academic Press, pp. 43-112.

* ————. 1968b. "The Structure of the 'T.H.E.' Multiprogramming System." Comm. ACM, vol. 11, no. 5, pp. 341-346.

————. 1975. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." Comm. ACM, vol. 18, no. 8, pp. 453-457.

————. 1977. A Discipline of Programming. Englewood Cliffs: Prentice-Hall.

Elson, M. 1973. Concepts of Programming Languages. Chicago: Science Research Associates.

Floyd, R. W. 1967. "Assigning Meanings to Programs." Proceed. Am. Math. Soc. Symposia in Applied Mathematics, vol. 19, pp. 19-32.

---

* Recommended reading

* Gerhart, S.; and Yelowitz, L. 1976. "Observations of Fallibility in Applications of Modern Programming Methodologies." IEEE Trans. on Software Engineering, vol. SE-2, no. 3, pp. 195-207.

Goguen, J.; Thatcher, J.; Wagner, E.; and Wright, J. 1975. "Abstract Data Types as Initial Algebras and the Correctness of Data Representations." Proceed. of Conf. on Computer Graphics, Pattern Recognition and Data Structure, pp. 89-93.

Gries, D. 1976. "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs." IEEE Trans. on Software Engineering, vol. SE-2, no. 4, pp. 238-244; Correction (May 1977), p. 262.

Guttag, J. V. 1975. The Specification and Application to Programming of Abstract Data Types. University of Toronto Computer Systems Research Group Technical Report CSRG-59.

---------. 1977. "Abstract Data Types and the Development of Data Structures." Comm. ACM, vol. 20, no. 6, pp. 396-404.

---------. 1980. "Notes on Type Abstraction (Version 2)." IEEE Trans. on Software Engineering, vol. SE-6, no. 1, pp. 13-23.

Guttag, J. V.; and Horowitz, E. 1978. "Abstract Data Types and Software Validation." Comm. ACM, vol. 21, no. 12, pp. 1048-1064.

Habermann, A. N. 1969. "Prevention of System Deadlocks." Comm. ACM, vol. 12, no. 7, pp. 373-377, 385.

* ---------. 1972. "Synchronization of Communicating Processes." Comm. ACM, vol. 15, no. 3, pp. 171-176.

Hall, A. D. 1962. A Methodology For Systems Engineering. Princeton: D. Van Nostrand.

Heitmeyer, C. L.; and Wilson, S. H. 1980. "Military Message Systems: Current Status and Future Directions." IEEE Trans. on Communications, vol. COM-28, no.9, pp. 1645-1654.

* Heninger, K. L. 1980. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application." Trans. on Software Engineering, vol. SE-6, no. 1, pp. 2-13.

Heninger, K. L.; Kallander, J.; Parnas, D. L.; and Shore, J. E. 1978. Software Requirements for the A-7E Aircraft. Naval Research Laboratory Memorandum Report no. 3876.

---

* Recommended reading

Hoare, C. A. R. 1969. "An Axiomatic Basis for Computer Programming." Comm. ACM, vol. 12, no. 10, pp. 576-583.

————. 1974. "Monitors: An Operating System Structuring Concept." Comm. ACM, vol. 17, no. 10, pp. 549-557.

James, V. E. 1975. "Encouraging Use of Reference Documentation." Journal of Systems Management, pp. 32-33.

Jensen, K.; and Wirth, N. 1974. Pascal User Manual and Report. 2nd ed. New York: Springer-Verlag.

Kaiser, C.; Krakowiak, S. 1974. "An Analysis of Some Run-Time Errors in an Operating System." IRIA Rapport de Recherche, no. 49.

Kernighan, B. W.; and Plauger, P. J. 1976. Software Tools. Reading, Mass.: Addison-Wesley.

————. 1978. The Elements of Programming Style. 2nd ed. New York: McGraw-Hill.

* Knuth, D. E. 1974. "Structured Programming With Go To Statements." Computing Surveys, vol. 6, no. 4, pp. 261-301.

Kosy, D. W.; and Farquhar, J. A. 1972. Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s (CCIP-85) -- Technology Trends: Software. Vol. IV of the Air Force Systems Command Development Planning Study Report.

Linden, T. A. 1976. "The Use of Abstract Data Types to Simplify Program Modifications." Proceed. of Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Special Issue, vol. 11, pp. 12-23.

Liskov, B.; and Berzins, V. 1977. "An Appraisal of Program Specifications." Massachusetts Institute of Technology Computation Structures Group Memo 141-1.

Liskov, B.; Snyder, A.; Atkinson, R.; and Schaffert, C. 1977. "Abstraction Mechanisms in CLU." Comm. ACM, vol. 20, no. 8, pp. 564-576.

Liskov, B.; and Zilles, S. 1974. "Programming with Abstract Data Types," SIGPLAN Notices, vol. 9, no. 4, pp. 50-59.

————. 1975. "Specification Techniques for Data Abstractions." IEEE Trans. on Software Engineering, vol. SE-1, no. 1, pp. 7-19.

---

* Recommended reading

Mills, H. D. 1971. Chief Programmer Teams: Principles and Procedures. IBM Federal Systems Division Report no. FSC 71-5108.

---------. 1972. Mathematical Foundations for Structured Programming. IBM Federal Systems Division Report no. FSC 72-6012, pp. 225-238.

---------. 1975. "How to Write Correct Programs and Know It." Proceed. 1975 Conf. on Reliable Software, IEEE Cat. no. 75CH0940-7CSR, pp. 363-370.

---------. 1975. "New Math of Computer Programming." Comm. ACM, vol. 18, no. 1, pp. 43-48.

MIL-STD-1679. 1978. Weapon System Development.

Navy Manpower and Material Analysis Center, Pacific. 1978a. Navy Manpower Planning System (NAMPS) Software Development Guidebook. NAVMMACPAC Document no. GB-01, rev. 0.

---------. 1978b. Interim Navy Manpower Planning System (NAMPS): Functional Description. NAVMMACPAC Document no. FD-01.

* Parker, A.; Heninger, K.; Parnas, D.; and Shore, J. 1980. Abstract Interface Specifications for the A-7 Device Interface Modules. Naval Research Laboratory Memorandum Report no. 4385.

Parnas, D. L. 1971. "Information Distribution Aspects of Design Methodology." Proceed. of IFIP Congress 71, pp. 339-344.

---------. 1972a. "A Technique for Software Module Specification with Examples." Comm. ACM, vol. 15, no. 5, pp. 330-336.

* ---------. 1972b. "On the Criteria To Be Used in Decomposing Systems into Modules." Comm. ACM, vol. 15, no. 12, pp. 1053-1058.

---------. 1974. "On a 'Buzzword': Hierarchical Structure." Proceed. of IFIP Congress 74, pp. 336-339.

---------. 1975a. "The Influence of Software Structure on Reliability." Proceed. of the 1975 International Conf. on Reliable Software, pp. 358-362.

---------. 1975b. "On the Solution to the Cigarette Smoker's Problem (Without Conditional Statements)." Comm. ACM, vol. 18, no. 3, pp. 181-183.

* ---------. 1976a. "On the Design and Development of Program Families." IEEE Trans. on Software Engineering, vol. SE-2, no. 1, pp. 1-9.

---

* Recommended reading

———. 1976b. Some Hypotheses about the "Uses" Hierarchy for Operating Systems. Technical Report. Darmstadt, W. Germany: Technische Hochschule Darmstadt.

* ———. 1977a. Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems. Naval Research Laboratory Report no. 8047.

———. 1977b. "The Use of Precise Specifications in the Development of Software." Proceed. of the IFIP 1977, pp. 861-867.

* ———. 1979. "Designing Software for Ease of Extension and Contraction." IEEE Trans. on Software Engineering, vol. SE-5, no. 2, pp. 128-137.

Parnas, D. L.; Bartussek, W.; Handzel, G.; and Wuerges, H. 1976. Using Predicate Transformers to Verify the Effects of "Real" Programs. University of North Carolina Report no. TR-76-101.

Parnas, D. L.; and Handzel, G. 1975. More on Specification Techniques for Software Modules. Fachbereich Informatik, Technische Hochschule Darmstadt.

Parnas, D. L.; Shore, J. E.; and Elliot, W. D. 1975. On the Need for Fewer Restrictions in Changing Compile-Time Environments. Naval Research Laboratory Report no. 7847.

Parnas, D. L.; Shore, J. E.; and Weiss, D. M. 1976. "Abstract Data Types Defined as Classes of Variables." Proceed. of Conf. on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Special Issue, vol. 11, pp. 149-154. Also Naval Research Laboratory Report no. 7998.

* Parnas, D. L.; and Wuerges, H. 1976. "Response to Undesired Events in Software Systems." Proceed. of Second International Conf. on Software Engineering, pp. 437-446.

Patil, S. 1971. Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes. Proj. MAC, Computational Structures Group Memo 57.

Randell, B.; Lee, P. A.; and Treleaven, P. C. 1978. "Reliability Issues in Computer System Design." Computing Surveys, vol. 10, no. 2, pp. 123-165.

Reed, D. P.; and Kanodia, R. K. 1979. "Synchronization with Eventcounts and Sequences." Comm. ACM, vol. 22, no. 2, pp. 115-123.

Reference Manual for the Ada Programming Language, July, 1980.

---

* Recommended reading

Satterthwaite, E. 1972. "Debugging Tools for High-Level Languages." Software -- Practice and Experience, vol. 2, no. 3, pp. 197-217.

Shaw, A. C. 1974. The Logical Design of Operating Systems. Englewood Cliffs: Prentice-Hall.

Tucker, A. E. 1975. "The Correlation of Computer Programming with Test Effort." System Development Corp. TM-221900, pp. 1-36.

* Turski, W. M. 1978. Computer Programming Methodology. London: Heyden.

* Weinberg, G. M. 1971. The Psychology of Computer Programming. New York: Van Nostrand.

Wirth, N. 1977a. "MODULA: A Language for Modular Multiprogramming." Software -- Practice and Experience, vol. 7, no. 1, pp. 3-35.

Wirth, N. 1977b. "The Use of MODULA." Software -- Practice and Experience, vol. 7, no. 1, pp. 37-65.

---------. 1977c. "Design and Implementation of MODULA." Software -- Practice and Experience, vol. 7, no. 1, pp. 67-84.

---------. 1977d. "Towards a Discipline of Real-Time Programming." Comm. ACM, vol. 20, no. 8, pp. 577-583.

Wolverton, R. W. 1974. "The Cost of Developing Large-Scale Software." IEEE Trans. on Computers, vol. C-23, no. 6, pp. 615-636.

* Recommended reading


DA
FILM