CONTENTS

RESEARCH PAPERS
122-133  Input-Output Systems in High Level Languages, Part II — Pascal and Basic
         R.P. WATKINS

134-136  Reliable Concurrency Control and Crash-recovery in Distributed Data
         Bases
         P.G. REDDY, S. BHALLA and B.E. PRASAD

137-142  A Language for Verifiable Modular Programming
         M.A. MALIK

TUTORIAL PAPERS
143-149  Teleprocessing Monitors and Program Structures
         R. CLARKE

150-154  A Review of Disk Scheduling Policies
         B.R. HOWARTH

CORRIGENDUM
155      A Descriptive Model for the Architectural Design of Information Systems
         J. KAUNITZ

SPECIAL FEATURES
121      Editorial
156      Special Issues of the Journal
157-161  Book Reviews
162-164  Information for Authors
Unique capabilities in a relational data base of a series

The challenge: NonStop™ operation in an on-line environment.

To design a fault tolerant system (no single points of failure) requires a multiple processor environment and thus a distributed system. Programs must be able to run anywhere and to access data anywhere in the system without specific knowledge of physical location. In the event of a failure, system loads must be redistributed dynamically without changing application software. All this whether the application is running in a single multiple processor system or in a multi-node network. Nothing less could assure availability of all resources through an otherwise crippling failure.

Consider the burden facing database management in such an environment.

Users must be able to distribute a database not only across multiple processors, but also across multiple systems in a network.

If the database model is dependent on hard coded pointers within the files, updating all these pointers is a nightmare. The problem is compounded if any one remote location is not available at the time of an update. Without concurrent access, the state of the database is potentially inconsistent.

These pointer problems make both hierarchical and network database models inflexible and difficult to modify. When the requirement to move files and applications among processors and among systems is coupled with the need to maintain the ability to transparently access the database from any point in the network, the problems become staggering.

The solutions are in the relational data base model.

A relational database is a collection of data items represented logically as two dimensional tables. Files use logical fields within records as their only required linking mechanism. Users need not be concerned about details of structure, only about the logical relationships which exist between files. This simplicity and the ease of use inherent to relational data bases has been obscured historically by attempts of vendors to shoehorn or add on relational structure to a conventional operating system. The results are laboratory curiosities, interesting but cumbersome, and they give relational models an undeserved reputation for poor performance.

The Tandem database management system, ENCOMPASS, is the world’s first on-line relational DBMS designed right into the operating system.

It is optimized for high performance and NonStop™ operation. Residing in each processor, ENCOMPASS provides complete independence from concerns about physical location of data. A user or an applications programmer need only know the file name of the data, peripheral device or system to be accessed.

Fully integrated into the operating system, the data base system handles enormous numbers of transactions with speed and efficiency.

Using ENCOMPASS, we can go straight from “Read Record A” to the correct disk without having to utilize the intervening software steps typical of other systems a) DBMS translating the command into appropriate instructions, b) database access methods determining where the data is located and translating the instruction to appropriate disk addresses and c) I/O services, part of the operating system, finally performing the actual retrieval.

Every level of software introduces more overhead and interferes with efficiency. Following the principle “closer to hardware equals greater efficiency” our data base operating system achieves outstanding performance:

In normal use, any random record from a file which fills a 300 M byte disk drive can still be retrieved with an average of one seek, using only a logical key to start the search.

One more reason why the Tandem NonStop™ System is a whole generation ahead.

A whole generation ahead

TANDEM NonStop™ Systems

MANAGEMENT INFORMATION SYSTEMS PTY. LTD.

TANDEM COMPUTERS DISTRIBUTED IN AUSTRALIA BY:

NCAMIS/61 151 South Terrace, Adelaide, S.A. 5000
Telephone: (08) 2121110

22 Atchison Street, St. Leonards, N.S.W. 2065
Telephone: (02) 438 4566

S.G.I.O. Building, Cnr. Turbot & Albert Sts, Brisbane, Queensland 4000
Telephone: (07) 229 3766

3 Bowen Crescent, Melbourne, Vic. 3004
Telephone: (03) 267 4133

3rd Floor, 151 South Terrace, Adelaide, S.A. 5000
Telephone: (08) 2121110
## 1. CUMULATIVE CONTENTS

*Technical Contributions*

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>J. Hext and S. Hirst</td>
<td>The Formal Treatment of State Transition Tables</td>
<td>1-6</td>
</tr>
<tr>
<td>R. Jamieson</td>
<td>Systems Development Data Base Audit — A DP Management Guide</td>
<td>7-11</td>
</tr>
<tr>
<td>M. Schkolnick and P. Sorenson</td>
<td>The Effects of Denormalisation on Database Performance</td>
<td>12-18</td>
</tr>
<tr>
<td>C.E. Roberts, Jr</td>
<td>Variable Order, Variable Stepsize, Componentwise Integration</td>
<td>19-25</td>
</tr>
<tr>
<td>W.H. Wilson, C.F. Burrows and M.M. Sidhom</td>
<td>Design of an Interactive Financial Planning Package</td>
<td>26-31</td>
</tr>
<tr>
<td>P.J. Turner</td>
<td>A Disk Based Observing System for a Radio Telescope</td>
<td>32-34</td>
</tr>
<tr>
<td>B. Dwyer</td>
<td>Cobol, Comments and Correctness</td>
<td>42-47</td>
</tr>
<tr>
<td>R. Clarke</td>
<td>A Background to Program Generators for Commercial Applications</td>
<td>48-55</td>
</tr>
<tr>
<td>J.L. Keedy and I. Richards</td>
<td>A Software Engineering View of Files</td>
<td>56-61</td>
</tr>
<tr>
<td>N. Parkin</td>
<td>The Input Space Model for Software Testing</td>
<td>62-65</td>
</tr>
<tr>
<td>A.M. Lister</td>
<td>Software Science — The Emperor's New Clothes?</td>
<td>66-70</td>
</tr>
<tr>
<td>D.H. Scuse</td>
<td>Data Base Interfaces</td>
<td>71-74</td>
</tr>
<tr>
<td>R.P. Watkins</td>
<td>Input-Output Systems in High Level Languages Part I — Fortran and Algol</td>
<td>77-90</td>
</tr>
<tr>
<td>J. Kaunitz</td>
<td>A Descriptive Model for the Architectural Design of Information Systems (Also Corrigendum p. 155)</td>
<td>91-98</td>
</tr>
<tr>
<td>W.P. Beaumont</td>
<td>A Cobol Project Using Minimal Resources</td>
<td>110-114</td>
</tr>
<tr>
<td>D. Cameron and E. Osborn</td>
<td>Security Aspects of a Magnetic Tape Management System</td>
<td>115-116</td>
</tr>
<tr>
<td>R.P. Watkins</td>
<td>Input-Output Systems in High Level Languages Part II — Pascal and Basic</td>
<td>122-133</td>
</tr>
<tr>
<td>P.G. Reddy, S. Bhalla and B.E. Prasad</td>
<td>Reliable Concurrency Control and Crash-Recovery in Distributed Databases</td>
<td>134-136</td>
</tr>
<tr>
<td>M.A. Malik</td>
<td>A Language for Verifiable Modular Programming</td>
<td>137-142</td>
</tr>
<tr>
<td>R. Clarke</td>
<td>Teleprocessing Monitors and Program Structure</td>
<td>143-149</td>
</tr>
<tr>
<td>B.R. Howarth</td>
<td>A Review of Disk Scheduling Policies</td>
<td>150-154</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Book Reviews</td>
<td>35-40, 118-120, 157-160</td>
</tr>
<tr>
<td></td>
<td>Editorial</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>Letters to the Editor</td>
<td>75-76, 117-118</td>
</tr>
<tr>
<td></td>
<td>Information for Authors</td>
<td>162-164</td>
</tr>
</tbody>
</table>
## 2. TITLE INDEX

*(Titles of books reviewed in the Journal given in italics)*

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advances in Computers</td>
<td>118</td>
</tr>
<tr>
<td>Amorphous Semiconductor Technologies and Devices</td>
<td>120</td>
</tr>
<tr>
<td>Applied Iterative Methods</td>
<td>120</td>
</tr>
<tr>
<td>Architectural Design of Information Systems, A Descriptive Model for</td>
<td>91-98</td>
</tr>
<tr>
<td>Artificial Intelligence, The Handbook of</td>
<td>159</td>
</tr>
<tr>
<td>Build Program Technique: A Practical Approach for the Development of Automatic Software Generation Systems</td>
<td>40</td>
</tr>
<tr>
<td>CAD in Medium Sized and Small Industries</td>
<td>39</td>
</tr>
<tr>
<td>Cobol Project Using Minimal Resources</td>
<td>110-114</td>
</tr>
<tr>
<td>Cobol, Comments and Correctness</td>
<td>42-47</td>
</tr>
<tr>
<td>COBOL (Survival in Business Applications Programming), Beyond</td>
<td>36</td>
</tr>
<tr>
<td>Computation: An Introduction to Computer Science, The Nature of</td>
<td>35</td>
</tr>
<tr>
<td>Computer Vision</td>
<td>158</td>
</tr>
<tr>
<td>Computers and Microprocessors Made Simple</td>
<td>38</td>
</tr>
<tr>
<td>Computers and Social Controversy</td>
<td>118</td>
</tr>
<tr>
<td>Computer-Assisted Analysis and Model Simplification</td>
<td>37</td>
</tr>
<tr>
<td>Computer Networks</td>
<td>37</td>
</tr>
<tr>
<td>Computing in the Twentieth Century, A History of</td>
<td>37</td>
</tr>
<tr>
<td>Data Base Interfaces</td>
<td>71-74</td>
</tr>
<tr>
<td>Denormalisation on Database Performance, The Effects of</td>
<td>12-18</td>
</tr>
<tr>
<td>Digital Systems for Industrial Automation</td>
<td>35</td>
</tr>
<tr>
<td>Discovering Computers</td>
<td>160</td>
</tr>
<tr>
<td>Disk Based Observing System for a Radio Telescope</td>
<td>32-34</td>
</tr>
<tr>
<td>Disk Scheduling Policies, A Review of</td>
<td>150-154</td>
</tr>
<tr>
<td>Distributed Databases, Reliable Concurrency Control and Crash Recovery in</td>
<td>134-136</td>
</tr>
<tr>
<td>Don't (Or How to Care for Your Computer)</td>
<td>159</td>
</tr>
<tr>
<td>Eurographics '80 — Proceedings of the International Conference</td>
<td>36</td>
</tr>
<tr>
<td>Evolutionary Information Systems</td>
<td>157</td>
</tr>
<tr>
<td>Formal Language Theory, Jewels of</td>
<td>158</td>
</tr>
<tr>
<td>Formal Language Theory — Perspectives and Open Problems</td>
<td>36</td>
</tr>
<tr>
<td>Global Talk</td>
<td>39</td>
</tr>
<tr>
<td>Input Space Model for Software Testing, The</td>
<td>62-65</td>
</tr>
<tr>
<td>Input-Output Systems in High Level Languages Part I — Fortran and Algol</td>
<td>77-90</td>
</tr>
<tr>
<td>Input-Output Systems in High Level Languages Part II — Pascal and Basic</td>
<td>122-133</td>
</tr>
<tr>
<td>Interactive Financial Planning Package, Design of an</td>
<td>26-31</td>
</tr>
<tr>
<td>Magnetic Tape Management System, Security Aspects of a</td>
<td>115-116</td>
</tr>
<tr>
<td>Man-Machine Communication in CAD/CAM</td>
<td>157</td>
</tr>
<tr>
<td>Map Data Processing</td>
<td>157</td>
</tr>
<tr>
<td>Microprocessors — From Chips to Systems</td>
<td>37</td>
</tr>
<tr>
<td>Operating System Elements — A User Perspective</td>
<td>157</td>
</tr>
<tr>
<td>Optical Devices and Fibers</td>
<td>159</td>
</tr>
<tr>
<td>Parallel Computers</td>
<td>119</td>
</tr>
<tr>
<td>Pascal Handbook</td>
<td>160</td>
</tr>
<tr>
<td>Pascal, Simple</td>
<td>160</td>
</tr>
<tr>
<td>Pascal, The BYTE book of</td>
<td>38</td>
</tr>
<tr>
<td>Pascal Programs for Scientists and Engineers</td>
<td>38,160</td>
</tr>
<tr>
<td>Performance '81</td>
<td>158</td>
</tr>
<tr>
<td>Personal Computers in Chemistry</td>
<td>39</td>
</tr>
<tr>
<td>Program Generators for Commercial Applications, Background to</td>
<td>48-55</td>
</tr>
<tr>
<td>Programming Languages, A History of</td>
<td>37</td>
</tr>
<tr>
<td>Roundoff Analysis of Matrix Algorithms, Software for</td>
<td>38</td>
</tr>
<tr>
<td>Semiconductor Technologies</td>
<td>120</td>
</tr>
<tr>
<td>Simula 67, Introduction to Software Development Tools</td>
<td>40</td>
</tr>
<tr>
<td>Software Engineering View of Files</td>
<td>39</td>
</tr>
<tr>
<td>Software Reflecteds</td>
<td>56-61</td>
</tr>
<tr>
<td>Software Science — The Emperor’s New Clothes?</td>
<td>160</td>
</tr>
<tr>
<td>Software Soliloquies</td>
<td>66-70</td>
</tr>
<tr>
<td>State Transition Tables, The Formal Treatment of</td>
<td>35</td>
</tr>
<tr>
<td>Structured Computer Vision</td>
<td>1-6</td>
</tr>
<tr>
<td>System Development Systems, A Review of</td>
<td>119</td>
</tr>
<tr>
<td>Teleprocessing Monitors and Program Structure</td>
<td>7-11</td>
</tr>
<tr>
<td>Using Structured Design</td>
<td>143-149</td>
</tr>
<tr>
<td>Variable Order, Variable Stepsize, Componentwise Integration</td>
<td>35</td>
</tr>
<tr>
<td>Verifiable Modular Programming, A Language for</td>
<td>19-25</td>
</tr>
<tr>
<td>Year of the Robot, The</td>
<td>137-142</td>
</tr>
<tr>
<td>&quot;You Just Bought A Personal What?&quot; — A Structured Approach to Creative Programming</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>37</td>
</tr>
</tbody>
</table>
3. CONTRIBUTOR INDEX

Allen, M.W. 119
Anderssen, R.S. 37
Beaumont, W.P. 110-114
Bhalla, S. 134-136
Brent, R.P. 38
Burkitt, J. 159
Burrow, C.F. 26-31
Burston, A.K. 118
Caffyn, R.N. 38
Cameron, D. 115-116
Carrington, D.A. 158
Clark, C.J. 40
Clarke, R. 48-55, 143-149, 160
Dampney, C.N.G. 35
Dwyer, B. 42-47
Fraser, D. 157
Goldschlager, L.M. 158
Green, M.A. 120
Hawryszkiewycz, I.T. 157
Herbison-Evans, D. 157
Hext, J. 1-6, 35, 158
Hirst, S. 1-6
Howarth, B.R. 150-154
Jamieson, R. 7-11
Juliff, P. 36
Kantoritz, J. 91-98, 155
Keedy, J.L. 56-61
Kidman, B.P. 39
Lions, J. 157
Lister, A.M. 66-70
McDougall, A. 160
Malik, M.A. 137-142
Milne, P. 38
Montgomery, A.Y. 39
O'Callaghan, J.F. 36, 119
Osborn, E. 115-116
Parkin, N. 62-65
Pong, L. 99-109
Prasad, B.E. 134-136
Reddy, P.G. 134-136
Reeker, L.H. 36
Richards, I. 56-61
Roberts, C.E. 19-25
Sammut, C. 159
Sammut, R.A. 39, 120, 159
Sand, P. 40
Schlolkopf, M. 12-18
Scuse, D.G. 71-74
Sidbourn, M.M. 26-31
Sorenson, P. 12-18
Tse, T.H. 99-109
Turner, P.J. 32-34
Wallace, C.S. 35
Watkins, R.P. 77-90, 122-133
Watts, R.O. 39
Whale, G. 160
Wilson, W.H. 26-31
Yuen, C.K. 37, 38

4. CR CATEGORIES INDEX

1. GENERAL TOPICS AND EDUCATION
   1.2 History, Biographies  77-90, 122-133
   1.3 Introductory and Survey Articles  42-47
   1.5 Education
      1.52 University Courses and Programs 110-114

2. COMPUTING MILIEU
   2.1 Philosophical and Social Implications
      2.12 The Public and Computers  91-98
   2.4 Administration of Computing Centres
      2.43 Operating Procedures  91-98
      2.44 Equipment Evaluation  91-98

3. APPLICATIONS
   3.5 Management Data Processing
      3.50 General  12-18, 42-47
      3.52 Financial  26-31
      3.59 Miscellaneous  7-11
   3.7 Information Retrieval
      3.70 General  110-114
      3.72 Evaluation of Systems  91-98
      3.73 File Maintenance  91-98, 134-136, 150-154
   3.8 Real-Time Systems  32-34

4. SOFTWARE
   4.0 General  1-6
   4.1 Processors
      4.12 Compilers and Generators  48-55, 62-65, 143-149
   4.19 Miscellaneous  99-109

4.2 Programming Languages  62-65, 137-142
4.20 General  122-133
4.22 Procedure- and Problem-Oriented
   Languages  26-31, 48-55, 77-90, 99-109, 143-149

4.3 Supervisor Systems
   4.31 Basic Monitors  143-149
   4.33 Database  7-11, 12-18, 71-74, 134-136
   4.34 Data Structures  12-18, 36-61
   4.35 Operating Systems  56-61
   4.39 Miscellaneous  115-116

4.4 Utility Programs
   4.41 Input/Output  77-90, 122-133
   4.6 Software Evaluation, Tests and Measurements

5. MATHEMATICS OF COMPUTATION
   5.1 Numerical Analysis
      5.16 Numerical Integration and Differentiation
           19-25
      5.17 Ordinary and Partial Differential
           Equations  19-25
   5.2 Metatheory  66-70
   5.22 Automata  1-6
   5.23 Formal Languages  1-6
   5.24 Analysis of Programs  42-47

6. HARDWARE
   6.3 Components and Circuits
      6.34 Storage Units  150-154
“News Briefs from the Computer World” is a regular feature which covers local and overseas developments in the computer industry including new products, interesting techniques, newsworthy projects and other topical events of interest.

COMPUTER CLASSROOM TECHNOLOGY

Tandy Electronics has released its newest version of the popular network controller for use in classroom configurations of its TRS-80 Model IIs and colour computers. The Network 3 controller connects up to 16 “student stations” to a single “host” computer.

Using the Network 3 system, teachers can upload and download programs to any or all of the student stations, allowing each student to work independently of and yet simultaneously with other stations in the room.

Network 3 system software gives each student station access to the host computer’s disk drives and attached printer, thereby offering cost-effective “sharing” of peripherals and eliminating the need for schools to purchase additional printers, disk drives, etc.

Each student station can save and load programs directly to and from the host, and LLIST and LPRINT to the host printer (printer access requests are “spooled” at the host until the printer is free). Files on the host diskette can be directly manipulated from student stations, allowing automatic student record-keeping.

COLLEGE ESTABLISHES MICROCOMPUTER LABORATORY

The Canberra College of Advanced Education has announced it will establish a microcomputer laboratory.

This follows the purchase of 12 North Star Advantage microcomputers from Anderson Digital Equipment.

The equipment will be used to acquaint students who are enrolled in formal computing units, and those who do not study computing as part of their courses, with appropriate and non-appropriate use of microcomputers. Students using the laboratory will be from a variety of disciplines including education, librarianship, environmental design and accounting.

In the past, the College has had a number of microcomputers in different locations throughout the campus. The new laboratory provides an additional facility. Next year the College will purchase further units and will develop a local area network.

The new equipment will be on stream for first semester in 1983, and will be housed in Building 9, the College’s Instructional Media Centre.

Further information regarding the microcomputer (Continued on page iv)
Your CPT word processor of today can be the first step to your automated office of tomorrow.

Word processing has taken the business world by storm. Offices large and small are discovering this great step forward over ordinary typing.

Unfortunately, that's where most people stop.

They've accepted their new word processors as "fancy typewriters". They don't realize they're on the threshold of an even bigger idea - called office automation.

CPT Corporation of the United States is one of the world's five top word processing companies and has long recognised the potential of office automation. That's largely because it was one of the first companies to make only word processors. That's why all CPT word processors are designed to let you add new powers - a step at a time.

You may wish to start with a CPT entry-level word processor and expand your system later. With CPT, your investment is always protected, because you can upgrade to the more powerful CPT word processors in just minutes, right in your office.

Specialists in your different departments will soon discover the computer power inherent in an advanced word processor. CPT software will let them do complex maths, sort files and make up new lists, as their original documents are being prepared.

Electronic storage is another breakthrough.

For individual workstations, Oce has just announced the CPT Disk Unit series, each of which can store up to 2,600 pages of additional information.

Perhaps the most remarkable step to the automated office of tomorrow is the newly developed CPT Office Dialog Link.

Unlike the present 'office networks' offered by other companies, the Office Dialog Link lets your CPT word processors 'talk' to other brands of equipment.

Up to 984 office machines can be joined by the Office Dialog Link at one time.

All CPT equipment is backed by Oce Reprographics' service with its superbly equipped, streamlined, national service network to ensure that your CPT system of today is the right foundation of your automated office of tomorrow.

CPT word processors are available through Oce offices in Sydney, Melbourne, Canberra, Brisbane, Adelaide and Perth.

START BY SENDING FOR A FREE BOOKLET
This explains that what starts out to be a better way to type is, in reality, a better way to do business when you build on a CPT system.
One of the privileges of the editor of a journal is to write frequent letters and, occasionally, an editorial. Although there seems to be little time for writing anything more, this is not entirely true. In my case, this year I have also been writing programs that help me write letters, and also documentation to help me keep track of both letters and programs. These form part of a system intended to streamline editorial activities. (The scope and variety of the latter may come as a surprise; but that is another story!)

Since I do not enjoy the benefits of regular secretarial help, but do have convenient access to a time-sharing system with facilities for text formatting, starting this project was obvious and easy: just devise a few parameterised standard letters, write a couple of simple command procedures, and I was away. However every new situation is a little different, nearly every new submitted paper presents some new variation or condition, and the number of ”standard letters” always needs to be a little larger (with variations, now well over one hundred, and still growing!). More, more elaborate command procedures are needed, and the handling of large numbers of small data files (one per paper) becomes burdensome. One day the project suddenly ceased to be a small one and became something much larger.

Such a development will come as no surprise to experienced computing professionals. Nor will the need to formalise procedures, and then to streamline them to keep them efficient; and last but not least, the need for comprehensive documentation. Program documentation is a particular interest of mine, and in this case it was important because I wanted the system to be operated by some of my colleagues during my current absence overseas. I estimate that documentation has taken at least 50% of my total effort. Even so, since it is incomplete (it describes only what, but not why, various design decisions were made), its useful life without substantial change should be no more than about 12 months.

What is missing from the documentation for both my system and many others is a description of the basic strategies selected and the reasons why alternatives were not adopted. Such documentation is usually essential to prolong the effective life of the associated software. Many times it can also be usefully shared with others, and should be.

It is not generally true that computer people do not write much: measured in total key strokes per year, many of us have a high output indeed. It is just that short term goals are often allowed to become paramount, and strategic documentation is neglected. This, I submit, is unprofessional. One of the hallmarks of professional activity is the augmentation and refinement of the body of knowledge upon which the profession depends. One of the time-tested ways of refining and disseminating knowledge is via the preparation and publication of serious technical papers. The field of computing has many practitioners recruited almost by accident without formal training. If computing is to become a profession, then its practitioners must act professionally in publishing and recording both their successes and their failures in places where they can be scrutinised publicly. The Australian Computer Journal provides such a place. I plan to complete my documentation. What are you doing?
Input-Output Systems in High Level Languages — Part II Pascal and Basic

R. P. Watkins*

This paper contrasts two languages with different design philosophies and examines why the axiomatically based design of Pascal input-output fails when the pragmatically designed system of Basic succeeds.

Key words and phrases: Input, output, Pascal, Basic, elitism, history of programming languages,
axiomatic file definitions, standardisation.

6. INTRODUCTION

In a previous paper (Watkins, 1982) I examined the developments of I/O in two of the oldest languages, Fortran and Algol. Again, in this paper, an axiomatic and pragmatic language will be discussed and contrasted; but this time they, Basic and Pascal, both started life in the world of teaching and grew up to become (at least to some extent) general purpose languages.

For teachers of introductory programming courses a major stumbling block is to develop I/O proficiency early enough so that students can run programs. Using Fortran leads to high student withdrawal rates and, besides raw speed, "free format" I/O is the obvious change incorporated in WATFOR (Cress, Dirksen and Graham, 1970) when it was developed. Formatted, record-oriented I/O requires too much planning and too many notational hurdles for the beginner. And so both Basic and Pascal contain fundamentally simple (but radically different) I/O systems.

Now, what happens when such languages are taken over by the real world? How well do these I/O systems perform and, when they fail, how are they modified?

7. PASCAL

7.1 The Pascal File Definition

Pascal is about twelve years old and is totally different from Fortran; it is an axiomatic language. The first successful attempts at axiomatic definition were the "activity rules" of Algol 60. These prescribed a nearly consistent definition of program structure and storage allocation, which has formed the basis of Pascal and Ada. In particular, the recursive definition of statements provided a powerful design tool that totally eclipses Fortran's pragmatic single-line statements. (It is interesting to examine the Block IF of Fortran 77 to see how painful and ungainly it is compared with the conditional statement of Algol 60.)

Prior to the development of Pascal, little work of any consequence had been done in the area of file definition and I/O specification. McGee (1969) proposed an abstract definition of file structure (later adopted by SHARE) which bears some resemblance to the Pascal concept. But this work has not formed the basis of a practical I/O system; probably because, unlike Pascal, the definition did not consider activities on files and hence it was a static view unrelated to data processing.

Wirth's contribution was to treat data in an axiomatic manner. He provided an excellent system for data and data structure definition into which he placed formal rules for files and I/O. This was done because he viewed files as a sequence of values of some data type. Stated that way, a file simply sounds like another data structure. Hence, rule one: a file is a data structure. If so, we can (and for consistency must) define a file type and file variables.

Jensen and Wirth (1975) provides a precise, elegant axiomatic definition of file structure and I/O. In order to illustrate the development of Pascal I/O the following definitions are based on Jensen and Wirth (1975) and Wirth (1976) (later we shall examine the changes incorporated in the draft ISO Standard).

Definition 7.1

A file is defined to be a sequence <x[1], x[2], ....., x[m]>, of arbitrary length, of components of the one type. Note that, for typographical reasons, subscripts are represented by the usual programmatic notation.

The specification of file operations requires the following preliminary definitions:

Definition 7.2

(a) < > denotes the empty sequence.
(b) If X = <x[1], x[2], ....., x[m] > and
    Y = <y[1], y[2], ....., y[n]> then concatenation, & is defined by
    X & Y = <x[1], x[2], ....., x[m], y[1], y[2], ....., y[n] >.
    In this paper the concatenation of components is permitted; thus if x[i] is a component then
    X & x[i] = <x[1], x[2], ....., x[m], x[i] >.
    (c) If X = <x[1], x[2], ....., x[m] > then
        first(X) = x[1] and
        rest(X) = <x[2], x[3], ....., x[m] >.
    (d) A file type is defined by
        file of type.
    (e) Let F be a Pascal variable whose type is a file type.
        Then associated with F is a file buffer variable F.t whose type is that of the component type of F. Note that, in Pascal, F may be a component variable and

Copyright © 1982, Australian Computer Society Inc.

General permission to republish, but not for profit, all or part of this material is granted, provided that ACJ's copyright notice is given and that reference is made to the publication, its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society.

*Department of Information Science, University of Tasmania. Manuscript received May 1982.

may be qualified by subscripts or in other ways (as in
(f) The current position of a file F is defined by the two
sequences
X[L] = <x[1], x[2], ..., x[k]>
X[R] = <x[k+1], ..., x[m]>
where X[L] & X[R] = F.

These axioms enable the Pascal I/O system to be pre­
cisely defined as follows:

Definition 7.3
(a) eof(F) \equiv X[R] = <>;
(b) reset(F) \equiv X[L] := < >;
if x[R] = <> then
F#:undefined
else
F#:first(X[R]);
(c) get (F) \equiv if not eof(F) then
X[L] := X[L] & first(X[R]);
X[R] := rest(X[R]);
if X[R] := <> then
F#:undefined
else
F#:first(X[R]);
else
error;
(d) rewrite (F) \equiv F := < >; (thus X[L] := X[R] := < > )
(e) put(F) \equiv if eof(F) then
F#:undefined;
else
error;
(Note that X[R] remains < >).

In addition there are two qualifications:
(i) the behaviour put(F) if X[R] = <> and the last oper­
ation was reset(F) or get(F) is not specified in the
standard.
(ii) The Pascal assignment statement cannot be used with
variables of type file or containing type file compon­
ents.

Although many readers will be aware of some (or all) of the
flaws in these axioms, my argument in the sequel requires
we look at a few problems in detail.

Firstly, it must be remembered that the design of
Pascal I/O was fixed as soon as the decision to treat a file as
a sequence data structure had been taken. Once that axiom
had been decided the rest was inevitable; in particular be­
cause all type definitions are, except for records:

thing of type (a Pascal axiom)

allowing only a single type of component for thing, it was
mandatory that a file be a sequence of components of one
type. Thereafter, the definitions of I/O operations are
obvious derivations. Thus, any concern here is with the
validity of the axiomatic base and the consequent changes
to it.

Let us consider

var X: file of file of T

This definition creates three objects:

X the file
X\#: the file buffer of type file of T
X\#: the buffer for X\# (because it is a file) of type T.

Assuming we wish to create X, it cannot be written by
X\#: := ...; put(X) without firstly moving the contents of
X\# into X\#. As X\# is a file presumably the correct sequence
is

rewrite(X);
repeat
rewrite(X\#);
repeat
X\#: := ...;
put(X\#);
until end of X\# reached;
put(X);
until end of whole file;

To read it we must use

reset(X);
while not eof(X) do
begin
reset(X\#)
while not eof(X\#) do
begin
.... := X\#;
get(X\#);
end;
get(X);
end;

Now consider reset(X) and get(X). These place one file of
T, or subsequence, in X\#; but how is that sequence de­
defined? Because each file of T is a sequence of T values
<t[1], t[2], ..., t[m]>, the concatenation of two such
files is simply the sequence
	<t[1], t[2], ..., t[m], t[1], t[2], ..., t[n]>

There is no eof marker (if there were it would have to be of
type T and, as it would then be indistinguishable from a
data component, it could not be recognised anyway), and,
as the length of any X\# is undefined, reset(X) and get(X)
cannot be described. Thus, because we cannot define what
get(X) transmits to X\# we cannot read this file.

This argument has nothing whatever to do with the
desirability or implementability of the file of file concept.
All that has been done is to show that Definitions 7.1 and
7.2(b) cannot be the basis of a file type which is to be inte­
grated with the other Pascal types and that it can only be
employed if, as has been done, files are hamstrung by ad
hoc restrictions which do not apply to other types; that is,
if the axiomatic base of Pascal is modified by pragmatic
rules.

In fact the foregoing argument is nonsense, even
though the conclusion is correct! (It has been left in this
paper because I believe that much can be learned from
examining incorrect reasoning). The error is that the con­
catenation of two sequences of things is not a sequence of
tings but a sequence of sequences. Thus I should write:

<<<t[1], ..., t[m]>,<t[1], ..., t[n]>>
and now it is clear that sequences are composed of components \( t[i] \) and *delimiters* of some sort. These delimiters are necessary because the *Umwelt* (or universe) of data contains more than just one thing and to distinguish one \( t[i] \) as belonging to a particular sequence rather than another requires such delimiters, be they conceptual or physical; indeed all the definitions use them as a necessity. Thus a *file of file construct* is *permitted* if the delimiters or some equivalent discrimination (such as the NIL pointer value) is input and output as part of the sequence and it is this delimiter which enables \( \text{eof} \) to be determined. The problem with the *Pascal file of file definition* is that its implementation is usually in terms of computers which have no such delimiter capability in memory and hence cannot store such objects. Note that all the other data structures in *Pascal* either are fixed length (delimited by the number of locations) or use the NIL token. It is this lack of delimiters for file components which forms the basis of the following objections.

Consider

\[
\text{type } S = \text{file of } T; \\
\text{var } X: \text{file of } S; \\
\quad Y: S; \\
\text{get } (X); \\
\quad Y:=X1; \\
\]

Now why has the assignment been banned? Some people (Addyman, 1979) argue that such a restriction is unimaginative and file assignments should be permitted. Indeed it has been proposed as a means for defining core-to-core I/O and for separating the file concept from external devices. Thus:

\[
\text{type } X = \text{record} \\
\quad A: \text{integer}; \\
\quad B: \text{file of char}; \\
\quad C: \text{array } [1..10] \text{ of real}; \\
\text{next: } \text{link} \\
\text{end}; \\
\text{link = } \uparrow X; \\
\text{var } P: \text{link}; \\
\text{new } (P); \\
\]

is, in their eyes sensible. I would like to agree with this point of view, but the problem is that Definitions 7.1 to 7.3 *cannot* be used to define these objects because they do *not* recognise the role of the sequence delimiters. Thus, over and above the pragmatic considerations of how to represent such a structure in a traditional computer, the axioms themselves fail to define suitable facilities for such a structure. Note that we cannot *infer* features of a data type which are not defined. Certainly we can implement a defined feature in any way we like, but to implement an undefined feature is to use a different set of axioms!

My third point concerns the supposedly strong typing of files. To illustrate this consider the following two program segments:

\[
(a) \quad \text{type } \text{sex} = \{\text{male, female}\}; \\
\quad \text{out} = \text{file of sex}; \\
\quad \text{var } X: \text{out}; \\
\quad \text{create the file } X
\]

\[
(b) \quad \text{type } \text{sex} = \{\text{female, male}\}; \\
\quad \text{in} = \text{file of sex}; \\
\quad \text{var } X: \text{in}; \\
\quad \text{get the file } X
\]

In practice using the Pascal compilers at the University of Tasmania the second program reverses all sexes! More catastrophic situations are easy to envisage and, without the file axioms defining the storage of type information within the file, they *cannot* be avoided. For example, change type sex above to be pointers. Clearly several more pragmatic restrictions can be discerned which relate to this weakness.

Associated with this typing problem (where there may be a discrepancy between the file and the program's view of the file because the file has an existence independent of the program) is that of files containing mixed types. Consider the *Fortran* statement

\[
\text{WRITE } (3, \text{N,M, } ((A(I,J), I=1,N), J=1,M)}
\]

which generates a *single* record containing two integers and \( N \times M \) reals. How can this object be defined in Pascal? This is an important philosophical question: if a data file exists, is there a need to guarantee that Pascal has the ability to read and write it? The example raises another problem: what is a Pascal "record"? For example:

\[
\text{file of record} \quad \text{case } (X,Y) \text{ of} \\
\qquad X: (N,M: \text{integer}); \\
\qquad Y: (A: \text{array } [1..N, 1..M] \text{ of real}); \\
\quad \text{end}; \\
\text{end;}
\]

is one possible file definition with \( \text{get}(F) \) producing (arbitrarily?) \( F \uparrow .N \) and \( F \uparrow .M, \) or \( F \uparrow .A. \) The fact that two \( \text{get} \) operators are needed suggests either:

\[
(a) \quad \text{get is stream oriented and all the data can be extracted from one record, But this requires } F \uparrow \text{to contain a discriminator } (X \text{ or } Y) \text{ to determine whether two or more values are to be read or written and such a discriminator must be stored on the file!} \\
(b) \quad \text{get is record oriented in which case the process is not the same as Fortran. The discriminator is no longer essential because each record can be constructed to hold sufficient data for either variant.
\]

If the Pascal file axioms defined files with type and sequence discriminators *then* these problems could be overcome. But it does not and hence the definitions are inadequate.

As a final proof of inadequacy consider index sequential files. Implicit in these is the existence of a key (which may not be part of the data) to select a record. Pascal has no such facility and hence its axioms could *never* be applied to such a file.

These foregoing examples illustrate three points about the Pascal I/O axioms:

\[
(a) \quad \text{The definition of sequence makes it impossible to construct file of file forms.} \\
(b) \quad \text{The concept of a variable having a file bound to it creates insurmountable conceptual conflicts.} \\
(c) \quad \text{Components of a single type are inadequate for real file processing. Indeed, any typing of files is largely meaningless.}
\]

The correct conclusion to draw from these points is that
the axiomatic definitions of files are wrong. Instead, the designers adopted to patch the system and thus ruined the axiomatic base.

7.2 Text Files

The foregoing system allows the transfer of data without conversion (generally termed binary I/O). Employing it for the processing of text is very difficult because the user must define a char file and write his own conversion routines to read and analyse character strings; Wirth, in his early works, gives examples of such procedures. In addition to this formatting problem, a file of char has no record structure and hence cannot achieve the line feeds and page throws necessary for printed output. Line feeds could be incorporated by means of

\[
X: \text{file of array } [1..132] \text{ of char}
\]

for example, but then the user's problem in creating formatting procedures are magnified enormously because he now has to store pointers into the array \(X^T\) and include the necessary code to control file reading when the \(X^T\)-information content has been exhausted. The extension to a two dimensional page further complicates matters. Even if these difficulties are overcome there is no guarantee that the code will work because of the total lack of specification of the relationship between the logical file X and the real printer. Thus it would seem that we can either have formatting or line control but not both with a great deal of trouble.

In consequence, Pascal defines a separate, alternative I/O system based on text files which contain line markers. Text files do not fit into the axiomatic base because the line markers are special symbols, and hence such files cannot be defined as file of char as Wirth originally specified them (the elements are not all of the one type). Having created these special files, Wirth then defined special operations, read and write, to act on them which perform translations between character strings and the internal data types as well as actually transferring information to and from files.

In principle, read and write represent intrinsic procedures performing the operations which Wirth specified for text I/O within the basic file system (see for example, Wirth, 1973a). However, because of the desirability of multi-value operations and the need for format control write is a pseudo-procedure with pseudo-parameters such as

\[
\text{exp1 : exp2 : exp3}
\]

where the value of exp1 is output in a field of width exp2 characters and exp3 controls such matters as the position of the decimal point. Because expressions are used, good execution-time control is possible. However, being stream-oriented and lacking any repetitive control some I/O is very tedious.

In contrast there is no format control on input. Although some authors agree with this approach (see the correspondence in Pascal News), a new language which fails to cope with existing data files is in serious jeopardy from the start. Pascal cannot process many Cobol-oriented files, such as the Australian Census data (ABS, 1976), without extensive, user-written routines for data conversion of the type even assembler language programmers should not have to code. We could hardly describe this as contributing to the profession's desire for simple, well structured, high level languages.

The read and write procedures are supplemented by readln, writeln and an eof predicate which allow line control. No definition of pages exists in Wirth's sytem and although a page procedure is specified its operation is, to say the least, vague.

The most interesting feature of text I/O are:

(a) Because it is derived from the file axioms, read(f,ch) is defined to be "ch:=f; get(f);" in order to match the definition of reset(f). This produces the odd appearance of a look-ahead eof:

\[
\begin{align*}
\text{while not eof(f) do} \\
\text{read(f,x)}; \\
\text{end;}
\end{align*}
\]

(b) When reading, say, integer values the separate determination of eof can cause unexpected program termination. Thus, if the above program reads an arbitrary number of lines with an arbitrary number of values per line, the last successful read must leave eof=false and either eol=true or some character in ft. Then the next attempt to read(f,x) will pick up the eof, try to get more data and produce an error.

(c) User-defined scalar constants cannot be read or written.

All three problems are significant and seriously detract from the usability of the text I/O operations.

7.3 The ISO Standard

The foregoing difficulties with the Pascal axiomatic base were recognised by the people concerned with drafting the ISO standard (Addyman, 1979). In particular eof recognition, files of files and text files all faced close examination. However, the latest draft standard (ISO, 1980) vindicates my earlier comments on the enshrinement of language features. For example, the problem with interactive files mentioned in Section 1.4 (Watkins, 1982) was widely canvassed (Addyman, 1979) and the two prominent suggestions were the UCSD Apple solution and a concept of "lazy I/O" where reset didn't bother filling the file buffer variable. Both ugly perversions of the axioms were apparently preferable to throwing away the unworkable basis of the whole I/O system.

ISO (1980) defines files identically to my definitions except that:

(i) files with files in their components are not permitted;
(ii) a file now has a mode (generation or inspection) which defines it to be read-only or write-only; and
(iii) the definitions more precisely specify the actions in cases such as attempting to read a file that is being written.

Thus, by restricting files severely, the problems we have discussed have been circumvented. Note that at one stage (ISO, 1979) the definition of eof was changed to "whether the buffer variable Ft is positioned at the end of the file F". This too-vague definition allows for end-of-file markers or sequence discriminators, and the Pascal axiomatic base can work if it is redefined as:

**Definition 7.4**

(a) A file is a sequence of (i) objects all of the one type and (ii) an end-of-file marker.
Input-Output Systems

(b) \( \text{eof} \equiv \text{first}(X[R]) = <\text{eof marker}> \)

Interestingly enough the inclusion of eof markers enables text files to be defined as

\[
\text{text: file of file of char;}
\]

and eoln to be defined as

\[
\text{eoln (text) }\equiv \text{eof(text)};
\]

From this, alternative definitions for read and write can be constructed which enable the whole of Pascal I/O to be inserted into the axiomatic base. But, as we have seen, the base is riddled with too many problems for this to work. Most importantly, the restrictions listed above, like subscript expressions in Fortran, contradict the whole rationale of an axiomatic system.

The problems with text files have produced a new set of definitions with each succeeding draft of the standard and each version has attempted to resolve the conflict between the axioms and the line marker concept. The current definition (ISO, 1980) defines text to be

\[
\text{file of line sequence}
\]

where a line sequence is a sequence of characters followed by a special component value, the line marker. These line sequences do not in fact exist in Pascal because the draft also stipulates that text behaves as a file of char and thus a get picks up one character. As this contradicts the above definitions, text is in fact

\[
\text{file of symbol}
\]

where symbol is the set of characters and the line marker. Then we have the following situations:

(a) If the line marker is implemented by, say, the EBCDIC CR character get and read will return CR for a char file and return a blank for a text file.

(b) If the line marker is implemented by a special "symbol" then CR will be returned for both types of file.

This is, of course, implementation dependent. However, by producing a definition of text which can be manipulated into the axiomatic base, the draft standard has overcome the fundamental objections raised in Section 7.2 above.

It is now time to go beyond an evaluation of the axioms to consider the overall effectiveness of Pascal I/O. To do so note that, to paraphrase ISO (1980), the language was designed:

(a) to make available a language suitable for teaching programming as a systematic discipline;

(b) to define a language whose implementation may be both reliable and efficient on currently available computers.

The draft then goes on to say that "it has become apparent that Pascal has attributes which go far beyond these original goals. It is now being increasingly used commercially in the writing of both system and application software". Unfortunately the design philosophy adopted for teaching languages is necessarily different from that of production languages and to try to enshrine one within the other is to court disaster.

This can be seen by considering the binding time requirements listed in Section 1 (Watkins, 1982) in the light of the programming activities undertaken by students. Given that Wirth's educational philosophy, as expressed in his books, rests on algorithm design:

(a) Files need not be bound at execution time.

(b) Only sequential access is needed.

(c) Steam-oriented I/O allows execution time binding of lists to be achieved by means of existing statements without having to introduce complex, difficult to comprehend I/O syntax.

(d) From (c) formatting is concerned with single values.

(e) All errors can be ignored (treated as fatal) except eof.

(f) As the primary focus is on algorithmic processes, the system need not cater for any sophisticated file problems (such as random access, complex print layout, record/block size control etc.).

These restrictions, fundamentally identical to those adopted by Basic, the other "teaching" language, are common presumptions of many educators and widely used in practice. Unfortunately they are not viable in the computer industry. Thus the standard proposes an I/O system which, from the outset, is inadequate and, like Algol in the past, may well lead to either the rejection of Pascal or the development of many, non-standard, ad hoc extensions. This can be seen from reading Tiberghien (1981) and the pressures to allow various random access file forms.

7.4 History

I have shown that pragmatically Pascal I/O works quite well, but conceptually it fails. Considering the excellence of the rest of the language it is worth reviewing the history of Pascal to discover how this situation arose.

The earliest available paper (Wirth, 1972a; it must have actually been written in 1970 as it predates Wirth, 1971) presents a Pascal which is hardly recognisable, just as Algol 58 differs from Algol 60. The over-riding impact of the paper is that Wirth is expounding his newly crystallised concept of type. (The paper in its sequel glosses over the concepts of variable and statement structures.) The need for rigorous definitions and reliability are outlined and then precise descriptions of the types and data structures are given, strongly flavoured by mathematical analogies. And here a file is defined by

\[
\text{"type } F = \text{file } m, d \text{ of } T
\]

where \( m \) indicates the mode as either input, output or scratch (that is, variable), \( T \) is the type of the elements of the file, and \( d \) indicates the device . . ., for example, disk or tape" (Wirth, 1972a).

It is of crucial importance that we realised that at this time Wirth's view must have been as follows:

(a) The types and structures are explained in terms of mathematical analogies which include random access (a "mapping") but not sequential access (a "free semigroup"). Thus files are defined to fill the gap, and their purely sequential nature is excused: arrays handle random access "stores" but, being aware that arrays (in those days) resided in main memory, Wirth unconvincingly argues that all I/O devices are sequential. He cannot have the roles of his types overlapping (and hence the later attempts to use arrays for random I/O).

(b) Files are firmly associated with external media through the mode and device specifications. There is no confusion here; files are definitely different from
other types.

(c) Nowhere is the association between type and variable explained. Certainly the idea of a file variable existed (see below) but the implications of this were probably not clearly understood until later.

(d) The variable \( F^t \) is explicitly defined to be the \emph{file buffer} and Pascal is meant to communicate directly with the buffer. Now because assignment can't be used (the file is external) the transfer of data must be by an I/O operation. Wirth, steeped in Algol, naturally goes for strict procedural I/O! But a parameter must be a \emph{variable} (or expression) and so a file variable is proposed. \textit{But he uses the type name for this!} I do not believe that is an error. Rather, Wirth has defined \( F \) to be the name of an \emph{object} (a file on a device) and despite the fact that this name appears in a type definition it is the only sensible name to use (see (c) above). Only later would the need to resolve the (now) apparent conflict have arisen, and the statement of physical characteristics of a specific object within the definition of a \emph{class} of such objects may not have been noticed.

(e) Only one procedure \( \text{next}(F) \) is defined which inputs or outputs depending on the mode of \( F \) (Wirth overlooks what happens with \textit{scratch}).

Like all people who have "good" ideas, Wirth initially leaves many things vague, over-emphasises justifications, and does not want to abandon or compromise the beauty and simplicity of his original conception. So any future refinement is just that; a clarification rather than a major modification.

This takes us to the second paper (Wirth, 1971; written before October 1970). Now the idea of type has crystallised and the concept of a variable is explicit. A data type defines the set of values which may be assumed by a variable and a variable is defined by

\[ \text{var} \ X : \text{type} \]

In this context it is almost possible to visualise the reasoning followed by Wirth:

(a) The axiomatic definitions of type and variables are excellent and are to be kept pure and simple at all costs.

(b) The idea of a file type is natural and desirable. But no other type requires mode and device information ("type \( X = \) real main-memory" is ridiculous!) so files shouldn't have it. Also why can't a file be on tape one day and disk the next? So scrub those bits.

(c) Then \( \text{next} \) no longer works, so we need two procedures \( \text{get} \) and \( \text{put} \).

(d) These procedures have a variable as a parameter so we had better have

\[ \text{type} \ X = \text{file of} \ T; \]
\[ \text{var} \ F : X; \]
\[ \text{put}(F); \]

for example. Now the whole definition of file fits into the axioms, so all we need is a semantic interpretation.

The semantic definition proved to be harder. A mode indicator was kept (so that you couldn't update files) but it was hidden. The file definition remained unaltered except that \( F^t \) became a pointer operation and buffers ceased to exist. In this context the file variable \( F \) is treated as a pointer and file operations move this pointer. Then \( F^t \) references the accessible component directly. Thus \( \text{put}(F) \) merely advances the pointer! The data is "written" on the file by means of

\[ F^t := \ldots \]

This is consistent with the other types, \( F^t \) being used to access a file component just as an index is used to access an array element. (Note that pointers are not mentioned in any context in the first paper; they didn't exist at that time.)

What has happened is that Wirth has suppressed reality (that files exist externally and there are buffers) in order to integrate files into the axioms used for all other types. The procedures \( \text{get, put} \) and \( \text{reset} \) are merely pointer manipulators and their necessity is justified and acceptable.

At this stage the battle had been both won and lost. Elegance, simplicity and uniformity have been achieved, but someone suddenly realised that

\[ \text{file of file of} \]
\[ X := Y \]

where both are files, can be generated by Pascal's syntax. The abstraction has created a thing called a file which is \emph{not} the same thing that programmers call a file.

The main defect of this system is the lack of legible I/O both in terms of line structure and formatting. Although Wirth (1973a) can show how to "home-brew" formatting it was clear that this wasn't good enough. Also there was still confusion about whether \( F^t \) was a buffer or not.

In 1972 (Wirth, 1972b and 1972c) a "clarification" of the I/O was made and some new operations defined:

(a) \( F^t \) reverts to being a buffer and \text{put and} \text{get} "append" or "pick off" elements. However, \text{reset} is defined in terms of \( F^t \) being a pointer! Wirth says reset "implies a subsequent \emph{get} operation" but the axiomatic definition and examples are consistent with pointer operations. Perhaps this approach was necessitated by the need for compatibility with the existing compiler (very likely) but the ensuing problems with terminal I/O make it an unhappy choice.

(b) Text files existed and \text{read} and \text{write} were defined for type char only. At this time they were pure procedures and end of line was an explicit character denoted \textit{eol} (a consequence of the CDC system). Wirth (1972b) doesn't define text files and presumably they were pragmatic extensions yet to be integrated into the axioms.

In actual fact Jensen (1972) had already incorporated a "standard procedure WRITE" to print integer, real, boolean and alpha (one word long character strings) under format control; but the first mention of it by Wirth does not occur until December 1972. This reluctance presumably stems from the fact that WRITE has become a pseudo-procedure and the loss of purity would have been undesirable. Thus Wirth (1973a) only hints at multiple parameters and non-character data transmission.

The final development, besides some short-lived things like buffer size specification, came when the eol character had to be dropped because it simply didn't exist.
Input-Output Systems

on some machines. Then Wirth (1973b) proposed putln and getln as well as READLN and WRITELN.

The foregoing was written without reference to Wirth's own assessment (Wirth, 1975), where he presents further criticisms of files coupled with some of the arguments and influences (notably the CDC 6000 system) which affected his decisions. His remarks there reinforce my argument that the attempt to insert files into the axiomatic base has failed because of the need to surround file definitions with a number of arbitrary restrictions. The oscillation of the concept of F7 between a pragmatic buffer and an axiomatically consistent pointer, and the alternating inclusion and omission of file mode are clear indicators of this.

But the rejection of the underlying design principles, of great significance to language designers, seems to have little relevance to the programmer whose primary concern is whether the I/O system works. The usability of Pascal I/O can be criticised at two levels: convenience and restrictions. In terms of convenience there is no doubt that it is a simple, easily taught language and it therefore satisfies one of its main aims. However, like other stream-oriented I/O systems based on single-value data transfer, it is tedious to use and the omission of formatted input is a major fault.

The restrictions are far more serious. Firstly the lack of file attachment facilities creates binding time problems. As an example, Pascal on an Apple II does not run under an operating system with file attachment facilities. Thus to read a named disk file the syntax of reset had to be changed to include a file name (Apple, 1979), for example:

```pascal
RESET (items, #5:DATA. TEXT)
```

Pascal presumes an operating system capability. Also there is no way to indicate the post-execution status of files (to be scratched or saved). Of course no program requiring dynamic file attachment can be written in Pascal.

Secondly only sequential files are defined. Here the axiomatic base intrudes by prohibiting random access files. Wirth (1976) attempts to allow such files, but his ill-considered argument is based on the unimplementable file of file construct, a restrictive view of random access and undefined ideas of a "possibly faster skipping mechanism". The text has a feeling of embarrassment about it (strengthened by the fact that he is still trying to avoid text I/O of numbers in 1976!). A seek procedure (Apple, 1979) could be defined in principle as:

```pascal
procedure seek (id,n);
begin
  reset(id);
  i:=1;
  while i <> n do begin
    get(id);
    i := i + 1;
  end;
end;
```

although it would not, in fact, be implemented in this manner. (Axioms provide a descriptive mechanism which explains the behaviour of the system but not necessarily its implementation. They must be consistent and comprehensive but need not be realistic.)

Cichelli (1980) argues that such extensions to file processing which contravene the "sequential" requirement are unnecessary. He proposes the use of arrays for random access files with such arrays appearing as program parameters. This solution, effectively Fortran 77 relative files, is unacceptable because:

(a) There would then be three types of I/O (sequential, text and random) processed by three separate sets of I/O statement.

(b) Formatting facilities could not be included and hence legible random access files are impossible.

(c) No facilities are available to define the post-execution status of such arrays.

(d) The concept of indexed and hash-key files is not implementable.

Thirdly, the omission of error control (also apparent with out of range values) means that Pascal is not particularly robust, especially when used interactively or for real-time processing.

It may be argued that these deficiencies are not important. In that case let us accept Pascal as an excellent teaching language and stop trying to use it commercially.

8. BASIC

8.1 The Influence of Design Criteria

Before examining Basic it is first necessary to consider the ways in which design criteria influence the structure of languages; and this includes not only questions of orientation (such as scientific versus commercial) but more significantly the way in which the designer perceives programs and programmers and hence his conceptual approach to language construction.

The first language I shall view in this light is Pascal because it provides the clearest illustration of the points to be made. This language was designed to teach programming as a systematic discipline. The importance of this statement can be seen most forcefully in the context of the general principles of Piagetian Learning Theory (Brainerd, 1978). In this theory it is proposed that human intellectual ability develops through a number of well-defined cognitive stages varying from concrete sensorimotor activities through to highly developed formal operations. The exact details of these stages are not at issue here; what is important is:

(a) The fact that hypothetical, formal operations are the highest level of attainment and are not achieved by many of the population;

(b) The level of achievement of one person is discipline dependent (Collins and Biggs, 1979) and an individual may be a formal thinker in, say, mathematics and a concrete thinker (at a lower level of cognitive development) in English Literature simultaneously;

(c) The level of achievement may vary with time; in particular the level of ability may drop in some circumstances such as stress (Collis and Biggs, 1979); and

(d) Academic experts in a discipline are (usually) functioning at the formal operations level within that discipline.

Wirth, without doubt, would be considered an abstract thinker of the highest calibre within the discipline of programming language design, as is evidenced by his writings. And it is in this light that we must view the stated design criterion.

To teach programming as a systematic discipline necessitates that it is approached as an abstract, formal theory and in consequence requires cognitive processing at the formal operations level. For example, both recursive
algorithms and axiomatic data structures, as they appear in Pascal, are presented as parts of a coherent system of deductions from well-argued hypotheses; and the subsequent programs are viewed as expressions of predicate calculus relationships and invariants. It follows that an understanding of Pascal (as opposed to an ability to use it) requires the learner to have the capacity to manipulate formal operations fluidly. There is, however, much anecdotal evidence to suggest that although many people achieve some competence with Pascal, only relatively few can master the language and gain a full appreciation of its axioms as a systematic discipline.

In consequence, Pascal must be viewed as elitist in the sense that its design is oriented towards those possessing very high levels of cognitive ability in the area of formal languages. And thus the frequent occurrences of misuse by students reflects the fact that these people are functioning at less structured, more concrete cognitive levels with an inevitable lack of comprehension of the virtues of the language being espoused by the teacher.

This same elitist viewpoint (where the word indicates a requirement of ability which can only be expected within a small percentage of the population) is also apparent in Algol 60. That many people failed to understand the language's statement structure (illustrated by the peculiar adaptation of it in PL/1) can be directly attributed to this aspect of its design. Such elitism is not necessarily bad. It can be recognised that the development of formal operational ability is desirable, and Pascal and Algol provide excellent tools through which this intellectual advancement can be achieved. Thus in the context of university education a strong preference for these languages is to be expected; but outside that arena this premise of ability becomes very dubious.

In contrast, Basic stresses an anti-elitist view, most clearly represented by the statement: "... there are more people in the world than there are programmers" (Kurtz, 1978). That is, Basic was conceived as a language for anyone, as devoid of dependencies on formal knowledge as possible, and requiring merely average ability to use successfully. This criterion, based on the belief that most (if not all) people need some knowledge of computing, parallels that of the teacher who aspires to enhance the abilities of all his charges knowing full well that many of them fall far short of our ideal picture of a student; whereas the academic can and does discard those who do not reach the level of excellence that university courses require.

An essential corollary is that such a language must have an existence independent of formal theory and the presence of recursive structures (both data and algorithmic), for example, cannot be tolerated.

Hence Basic was designed to be "friendly" (Kurtz, 1978) and in doing so it had to come down to the level of ability of the students. In contrast, Pascal was, perhaps subconsciously, designed to be remote and superior, the goal at the summit to be achieved by those (few) who were able. Indeed Pascal is addressed to the "reasonable man" (Herbert, 1977). Another example is the penchant for formal theorists to force constructs (such as I/O) into inadequate frameworks rather than fall back on pragmatic but far more effective solutions as illustrated by the failure of the file axioms of Pascal and their consequent problems.

Thus an examination of Basic I/O cannot be conducted on the same criteria that were employed for Pascal. The two languages, brought together by their common educational goal, are so fundamentally different in conception that they must be treated separately until such time as the fundamental intellectual attitude becomes the focus of attention.

8.2 The Development of Basic

Given an anti-elitist philosophy, the design of a language for programming courses must follow closely the guidelines listed by Kurtz (1978). In particular the system must be "friendly, easy-to-learn and use and not require students to go out of their way", and it must be useable by all with a minimum of difficulty to learn about computing. This approach must be seen in context of the period 1956 to 1964 when academic interest lay firmly centred on the mathematical sciences (a consequence of the origins of computer science within engineering and physics departments).

To produce a language which satisfies such criteria the Dartmouth team made some fundamental decisions affecting I/O:

(a) In the main the student programming exercises would be necessarily short, simple problems representative of real, numerical algorithms. In such an environment the central thrust of teaching would be to develop an understanding of algorithms and I/O is an essential but minor aspect of this. Thus it is sufficient to have a single source of input and a single legible output medium.

(b) The recognition and fluid handling of technical distinctions, such as between real and integer or between compile and execute, requires a quite detailed understanding of the organisation of computer systems. Basic attempted to remove or hide these distinctions with great success. For example, its integration of system editor tasks was so complete that most students did not, and did not need to, recognise that LET is Basic and NEW is system. Indeed this confusion was such that some system commands became enshrined in the standard (ANSI, 1980) as part of the language! It is interesting to compare this with Apple Pascal with which students also experience a confusion between the system levels and Pascal but, in contrast, this confusion creates some serious learning problems.
The attitudes stated in (a) and (b) above led naturally to the complete absence of files. The very mention of files requires the explanation of inter-file relationships, file binding, directories and the recognition that programs as well as data are files. To avoid the inevitable learning problems, input data was integrated with the program and output produced, apparently directly, on a printer. The input was an unstructured stream of numbers (the only data type) and the output was tabular with values printed in fixed zones and a text printing facility for headings.

From these decisions came an I/O system which can be learned in a few minutes. Unlike students learning Fortran (or even Pascal) these people can write and run programs shortly after commencing a course.

Over the succeeding seven years of development the primary function of education was maintained and this led to a loose form of de facto standardisation. At no stage was it felt necessary to retain language features, but any change had to preserve the fundamental simplicity of the language. Thus various alternative forms were created and one example of this dynamic approach is: "Less clear was what substring notation should be used. We therefore chose to use string functions, since they could be changed more easily than new syntax" (Kurtz, 1978, my emphasis). In this sense constructs were chosen, in part, with a view to change and the present existence of many distinct versions of Basic stems from this refusal to formally standardise. But all these variants share in common the retention of simple, easily taught features. Note that this philosophy stems from an initial and long-held belief that Basic was not a production language and that no corpus of programs should be regarded as a reason for retention of constructs. This contrasts sharply with IBM's attitude to Fortran.

The need to retain simplistic operations led to:

(a) The ability to replace one simple operation with another equally simple operation if necessary; and

(b) The restriction of language extensions to the addition of optional clauses (or variants) in existing statements or the creation of new, separate statement forms.

For example, the original PRINT statement still exists but is now complemented by variants which allow field packing, formatting and file access. This condition on modification meant, necessarily, that Basic I/O could not develop along coherent, axiomatic lines and the changes had to be essentially ad hoc.

The easiest way to see this process of development is to note the chronological development of Basic I/O during the six versions produced at Dartmouth College (Kurtz, 1978):

(a) Version 1 (1964): There were no files and only one data type (numeric). Input was by READ and DATA statements, both within the program text, and output was in zoned tabular form with the ability to print text.

(b) Version 2 (1964): An alternative list item separator (the semi-colon) was added to allow output to be packed to the nearest multiple of three spaces.

(c) Version 3 (1966): A RESTORE statement was added to allow DATA input to be reprocessed and an INPUT statement permitted the program to accept data from the terminal (the only I/O device) during execution. Matrix I/O statements and matrix operations were introduced. At this time GE developed its own version of Basic and introduced files, clearly because that company was concerned with real users as opposed to teaching.

(d) Version 4 (1967): String variables and data were added without changing the existing language by adding "$" suffixes to variables. Information in DATA statements was separated into independent numeric and string data blocks with separate RESTORE commands. The PRINT semi-colon separator was changed to produce contiguous output.

(e) Version 5 (1969): Files are introduced for the first time. These will be discussed later, but for the moment note that sequential files were considered to be "terminal format" files maintaining uniformity with the student's perception of Basic.

(f) Version 6 (1971): Some modifications were made to the file system and for the first time a formatting facility (through a print image within the PRINT statement) was provided.

There is, in this list, several examples of arbitrary change illustrating the rejection of standardisation (as in the PRINT semi-colon delimiter) and of modification constrained by the educational objectives (the way in which strings are introduced).

8.3 Basic Files

From our point of view, Basic does not become interesting until files are introduced, the earlier forms of I/O offering little other than an example of simple, ad hoc design. But while examining Basic's perception of files it is necessary that we are aware of two fundamental advantages of this language over others such as Fortran and Pascal; that is, the existence of a character string data type and of an integrated operating system.

Initially files were associated with an internal file identifier through a FILES statement. This merely took the external file identification and associated a number with it according to its position in the statement. Thus (Burroughs, 1975 and Honeywell, 1971):

```
FILES HEFFA, LUMP
```

enabled the file with external identification HEFFA to be accessed through the internal designation #1 and LUMP through #2. Note that the external names are not strings but identifiers and thus the association is bound before execution. However the internal designation can be viewed as a member of the number data type and thus the execution time file selection is identical to Fortran's:

```
PRINT #1, ...
```

where the variable I has some appropriate value.

This declarative file binding, introduced in version 5 (1970) was thrown out in 1971 when version 6 was produced. Instead a FILE statement achieved execution-time file binding by recognising the fact that, in the operating system being used, the external file identifiers were character strings. Thus

```
FILE #3, string expression
```

performed the same function but far more elegantly. The use of strings would have been pointless if string variables and expressions had not existed to enable the creation of

---

external file names. But with these facilities execution time binding became trivial:

```
PRINT "input file name";
INPUT A$;
FILE #1, A$
```

gives the crux of the generalised file listing problem which is such a pain for so many other languages. From this starting point it is easy to develop effective (if unesthetic) I/O operations to bind all file attributes during execution.

The underlying file structure was based on the requirements of the operating system and two distinct I/O packages were established.

(a) **Terminal format files.** These files employed INPUT and PRINT statements and were strictly sequential files containing images of terminal data. End-of-file could be detected and both INPUT and PRINT could be applied to a file (thus these files could be updated).

(b) **Random access files.** These used READ and WRITE statements and were binary files. Initially such files could contain either numeric or string data but not both, and were recognised in FILE statements by suffix % and $ characters respectively. The current position (LOC), file length (LOF) could be determined and the file could be repositioned anywhere by means of RESET. (It is interesting to note the use of pseudo-parameters with LOC and LOF as illustrated by Kurtz' example of detecting the end of a random file:

```
IF LOC (#1) = LOF (#1) THEN . . .
```

It must be remembered that Basic was a system, not just a language, and hence the file directory and naming mechanisms were created in conjunction with the I/O statements. This unified approach (totally absent in other common languages), whereby editing operations form part of the syntax, enabled far simpler and more definitive file specification.

The dichotomy between string and numeric binary files is based on the physical representation of these files. Honeywell (1971) and Burroughs (1975) indicate that random access files are composed of binary word images with each word being individually accessed. Thus a file might contain 720 words and RESET #1 30 would position the file at the 30th word. To store strings a control word was inserted followed by the characters packed into words. Clearly indiscriminate use of RESET on a file containing mixed data could be catastrophic and both Honeywell and Burroughs take great pains to point out the problems and to define the physical representation.

This difficulty vanishes if mixed data is banned and RESET (Bull and Freeman, 1971) is defined in terms of items of data. Then a fixed block of words can be used to hold strings and a more sensible definition of the file pointer results. In consequence, as with terminal format files, the user may be oblivious of the physical file structure and treat it as a sequence of indexed items.

In practice, considerable variations occurred between different implementations. GE (Honeywell, 1971) used all four I/O statements for terminal format files; INPUT/PRINT handling data without line numbers and with space separators, and READ/WRITE processing line numbers and comma separators. Then they use # for terminal format and : for binary file designators. Honeywell (1971), Control Data (1978) and Burroughs (1975) also eliminated the single-type binary files and allowed mixed strings and numbers while retaining single word addressing. These systems are all reminiscent of the Fortran I DRUM operations. Digital Equipment (PDP, 1975) only allow terminal-format files and implement the single type binary files as virtual arrays on disk. Thus, instead of

```
FILE : 1, "TEST"
WRITE : 1, A,B,C,D
```

this system would use, roughly,

```
DIM #1, A(4)
A(1) = A
A(2) = B
A(3) = C
A(4) = D
```

where #1 is a file designator for the virtual array A. Basic-Plus (PDP, 1975) also has a third I/O system, akin to Cobol I/O, discussion of which will be deferred until later.

It would appear that the moral of these early developments is that Basic has never been viewed as a vehicle for communication, but rather as an in-house system. Hence each manufacturer has modified the language and adopted I/O processes to match existing operating systems rather than modify the operating systems to match the language. Or else, Basic, as in the case of Burroughs, has never been integrated into the system at all and remains a separate, largely unlinked sub-system. It is only in the (ill-defined) area of terminal format files that any standardisation of communication is feasible.

The new proposed standard (ANSI, 1980) goes a long way towards producing a coherent file definition from these early, simplistic ideas. Like Fortran it refuses to consider physical file representation and states that "... the term file ... [refers] to the logical interaction of external data with a Basic program..." Subject to this condition:

(a) Four access modes are provided: Sequential (with multi-value records), stream (sequential with single value records), relative and keyed.

(b) Three data representations are specified: Display-format (character data), internal-format (numeric and string data) and native-format.

This system contains two curiosities, native format and stream access, which need explanation.

Native format files are a consequence of the "isolation" of the Basic system (a feature I have already hinted at). The underlying philosophy that Basic is a complete system and not just a language has created, in the past, the inability to guarantee that its files, display and internal format, can be read or created by non-Basic systems. Thus in order to allow "for the exchange of data among different language processors" (ANSI, 1980) the implementation defined native format is introduced. These files can be read and written by Basic but their features are left to the individual operating systems.

Stream access, on the other hand, is a pure fudge! Basic has never been sure whether it is record-oriented like Fortran or stream oriented like Algol. Originally READ was stream oriented and PRINT record oriented, and then a
new option (a list separator after the last list item) was added to indicate stream processing. This peculiar admixture has led to a standard which is fundamentally record-oriented. To try and recapture the simplicity of stream I/O, any file can be opened as a stream file! (I think the standard fails to distinguish access mode and file format clearly and omits to detail some combinations). That is, the "actual" organisation may be ignored and the file treated as single-value records and processed as a stream-oriented file. Here again we see the ambiguity which arises from failure to specify the relationship between logical and "physical" files. This one permitted case immediately raises the question: what happens if a sequential file is opened as relative or keyed? Utter disaster?

The choice of random access methods is interesting because it has considerable appeal. The Fortran 77 relative file is clearly derivative from earlier, pragmatic disk-file access methods and is closely related to the Fortran I DRUM statements; and the NU-Algo1 system (see Watkins, 1982) of key records in sequential files is, to say the least, ungainly. In contrast, the Basic standard has presented two abstracted file access mechanisms which are clear, simple and complementary.

One treats a file as an array of records and allows any element to be processed. The other regards each record as being named and access is strictly by name only. Note that:
(a) If a record exists in a sequential file it is always presumed to have a defined value (see ISO, 1980).
(b) Records always exist in relative files but need not have a defined value.
(c) If keyed files can be accessed by name (the key) then its records always have a defined value but the record need not exist.

Clearly these three file types differ in some fundamental manner.

The existence of this flexible, if ad hoc, I/O system is a consequence of the string data type, just as the introduction of good I/O control in Fortran 77 had to follow the new character data type in that language. ANSI (1980) provides execution time binding of file names, access mode, format and record structure when a file is opened; allows these attributes and the file status to be interrogated; and provides control over file positioning, end-of-file detection and, in the case of relative files, missing data detection for deleted records.

9. CONCLUSIONS

This examination of Basic and Pascal has raised a number of points:
(a) File types. In Watkins (1982) only sequential files were considered, except for fleeting reference to Fortran 77 relative files. The problems with Pascal and the existence of three major file types (sequential, relative and keyed) in Basic have drawn attention to the need to describe several, possibly distinct file organisations and to provide suitable operations on them. Basic is the only language of the four discussed which satisfactorily recognises the process of "record selection" and its features prompt the question: are there any other useful file organisations not covered by Basic? (a question that will be addressed in a later paper).
(b) Standardisation. Like Fortran, Pascal exhibits all the symptoms of premature standardisation. In contrast Basic shows some of the disadvantages of no standardisation. Or does it? In many ways the early flexibility has allowed Basic to experiment and grow, and thus the proposed standard (sixteen years after the initial development) reflects an effective, embracing system which would be hard to improve on. It might have been better if (being idealistic) the language had commenced with a coherent view of the eventual file organisations, but the refusal to be influenced by the existing corpus of programs is a second best (and far more realistic) approach leading to an I/O design which easily outstrips Pascal.

(c) Statement Form. Again, as with Algol 60 and Fortran, the initial statement syntax decisions have permeated throughout language changes. Thus Pascal has tended towards pseudo-procedures whereas Basic has been able to incorporate modifications in a much more rational manner because of its recognition of the separate sub-tasks in I/O activities.

(d) Intellectual Direction. The contrast between these two languages makes one wonder whether it is possible to develop an I/O system (and hence a whole language) which meets the needs of both philosophies. Pascal is intellectually superior but Basic is better! Can the two be integrated?

REFERENCES

ADDYMAN, A.M. (1979): Comments on a Draft Description of Pascal, University of Manchester, UK (AD Micrographics, Elgin AFB FL52542).
BURROUGHS (1975): BS6700 Basic Language Information Manual, Form 5000383-002, Burroughs Corp., USA.
HONEYWELL (1971): GECOS III Time-Sharing Basic, Form BR36 (rev.0), Honeywell Information Systems, USA.


WIRTH, N. (1972c): The Programming Language Pascal (Revised Report), typed manuscript photocopy. (Undated, but presumably early 1972.)


BIOGRAPHICAL NOTE

Richard Watkins is Senior Lecturer in the Department of Information Science at the University of Tasmania. He received a PhD degree in Computer Science from Melbourne University in 1972 and spent three years at RMIT before moving to Tasmania.
Reliable Concurrency Control and Crash-recovery in Distributed Databases

P. G. Reddy*, S. Bhalla* and B. E. Prasad*

A centralised concurrency control procedure of the type proposed by Kung (Kung and Robinson, 1979) is discussed for distributed database systems. The essential point is that validation (detection whether a conflict happened) and final write of the modified data must be performed in a critical section. In the distributed environment a final write is only allowed after we can be sure that the final write is possible on all sites involved. A centralised certifier based algorithm is proposed for validation and central-broadcast of the final write. The necessary communication precludes the implementation of any other kind of critical section.

We have assumed that a copy of the database is available at each node in the network. Though fully replicated databases have limited use (i.e. in military applications etc.), the assumption has been made for expository convenience. In case the database is not fully replicated the same algorithm can be used with minor modification (see section 2.1). Instead of sending NS-Readset to the central node, one will have to send a set of NS-Readset values, i.e. one for each access that is made at various sites.

It is also shown that at no extra communication cost the certifier can be made crash-resistant. In the last section, we discuss the efficiency of the proposed algorithm.

2. THE CENTRALISED CERTIFIER

In the proposed scheme based on the centralised certifier, problems (pertaining to synchronisation of certifier effort at various sites) associated with the previously proposed schemes of Badal (Badal, 1979) and Casanova (Casanova, 1979) — also pointed out by Bernstein (Bernstein and Goodman, 1980) — have been removed.

The working of the centralised certifier is based on its receiving the following information from the rest of the nodes in the system:

- An update request, with values that need to be broadcast to the other nodes (the write-set).
- The read-set of the update transaction i.e. the set of values that were read by it for performing the update computation at the requesting node.
- Information pertaining to the number of previous update broadcasts (by the certifier) that have been implemented at the requesting site.

The certifier need only check if any of the items in the read-set of the requested transaction have since been changed by the later broadcasts. If so, then the request is referred back to the requesting node; otherwise, the certifier allows it a sequence number, notes down items updated


Copyright © 1982, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that ACI’s copyright notice is given and that reference is made to the publication, its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society.
Concurrency Control in DDMS

by it and broadcasts it for update at other sites in the system. A few underlying assumptions follow.
1. The participating nodes have only two states, up, when functioning correctly; down, otherwise.
2. The network is assumed to be perfectly reliable: if Site A sends a message to another Site B, B is guaranteed to receive the message without error. In addition, we assume between any pair of sites, the network delivers messages in the order they were sent.

The proposed set-up can support any number of copies of the same data with equal ease. We further assume (for expository convenience) that each transaction which arrives at a node finds all the items it needs at the local site.

The two components of the proposed scheme, i.e. processing at the central node and processing at the requesting sites, are discussed separately. The centralised certifier is created at the central node.

2.1 Processing at the Requesting Site

The nodes in the network give priority to updating their own copy of data as soon as they receive an update. The Sequence Number of the last update made at the site is the node status (NS) of the corresponding site. As soon as a transaction arrives at the site, its update values are computed instantaneously and the request is sent to the central node along with the NS of the site. This forms the bulk of the processing at the requesting site. If the transaction is found to be in conflict with others, it is referred back to the same node for a restart. The requesting node assigns a Local Identification Number (LIN) to each transaction it sends to the certifier, containing a serial number plus the site identification number. Figure 1 contains information about the packet of information that is sent to the central node.

<table>
<thead>
<tr>
<th>LIN</th>
<th>NS ——— Read-set</th>
</tr>
</thead>
</table>

Figure 1.

2.2 Processing at the Central Node

On receiving its first update request, the certifier creates an entry in the Item Activity Register (IAR), in which items updated by the first arriving transaction are noted against a sequence number SN (SN equal to 1 initially); next SN = previous SN+1). The update values are then broadcast to all the nodes. The SN of the last broadcast transaction becomes the NS of the central node. When the next arriving update request is checked, if the NS of the requesting site equals the NS of the central node, the same procedure stated above is adopted; otherwise for all the later broadcasts by the central-node the read-set of the requested transaction is checked for a conflict. In cases of no-conflict the corresponding entry is created in the IAR and the update is broadcast. In cases where there exists a conflict, one further check is made as follows:

- For all cases where the read-set is in conflict with some previous ‘Write-Set’, if the write-set of the requested transaction is in no conflict with any of these ‘Write-sets’ the update can still be broadcast (see serialisability [Papadimitriou, 1979 and Bernstein and Goodman, 1980]).
- Otherwise, the transaction is referred back to the requesting site for a restart.

In order to curb the size of the IAR, the central node, on arriving at a conclusion that no node in the network has NS below S (say), may delete all entries pertaining to SN < S from its IAR. This is termed the Bottom Sequence Number (BSN), Figure 2 gives the details of IAR.

3. CRASH RECOVERY

It is assumed that all nodes in the system maintain a recovery IAR on the basis of broadcasts by the central-node. The recovery IAR at a site lags behind the IAR at the central node by those broadcasts which have not yet been received at the site. Also, the central node is assumed to broadcast the BSN frequently, along with the regular broadcasts.

All the broadcasts from the central node are assumed to be atomic. That is, the message is either broadcast fully or not broadcast at all. As soon as a node in the system is able to detect the central node failure, it announces its decision to become the central node. It also broadcasts its NS. All the nodes in the system which have their NS equal to or less than the above value send ‘OK’ messages. All the other nodes send their LS to the central node. On the basis of the above information, the IAR and the local copy at the new central node are fully recovered. All such nodes in the set-up which had some request pending at the central node retransmit their requests to the new central node. For such a request, the certifier also checks the LIN for a match with any of the entries it checks from the IAR. This is to avoid duplication.

If there are N nodes in the system at the time of the central node failure, the new central node broadcasts N-2 failure messages. It receives N-2 return messages. Hence, the recovery of the central node merely requires 2(N-2) messages, which is not a significant cost in view of what most concurrency control techniques require to perform a single update.

The failure of any node other than the central node does not affect processing of transactions at any other neighbouring site. Such is not the case with any other existing concurrency control algorithms. In case of such a failure, the IAR at the central node starts growing in size because the certifier at the central node receives no information about the NS of the crashed site. If the crashed site recovers soon enough, it can resume processing as normal. The central node, after a certain period of time and after receiving confirmation of its failure from a neighbouring
site, may simply ignore such a site until it recovers fully.
For which, a routine recovery procedure may be applied.

4. ALGORITHM EFFICIENCY
For such requirements, where concurrency control is
not a bottleneck, the proposed algorithm is most suitable,
being simple and easy to implement. For the general case,
the three major cost considerations are (Bernstein and
Goodman, 1980) as follows:
1. Communication cost (in terms of number of mes­sages).
2. Local processing overhead.
3. Cost of restarts (roll-backs).
Communication cost is an important consideration
under optimistic circumstances (i.e. conflicts are rare),
whereas cost of restarts is a significant criterion under
pessimistic circumstances.
The number of messages required to perform an
update in the proposed scheme is N (one request to the
certifier, N-1 broadcasts) for a N node network. This
compares favourably with all the methods proposed so far.
The local processing overhead is approximated by the
amount of information that a site needs to maintain con­sistency (Bernstein and Goodman, 1980). The proposed
algorithm is easy to implement and has no significant
information maintenance or local processing cost. Each
site maintains an IAR and information about its NS.
Further, processing at a site is independent of the other
sites — a failure of a site does not affect processing of
transactions at the other sites.
In the technique proposed by Badal (Badal, 1979)
a roll-back means undoing information logged earlier,
regarding the read accesses made by the transaction.
Other roll-back oriented algorithms are similar. In the
proposed scheme a roll-back incurs no such expenditure.
It simply involves computing the update values using
newly updated read-set and sending it for certification.
In addition, the centralised control technique can
help us afford priority response for a particular trans­
anction. For example, as soon as a transaction is able to
identify its read-set completely, it can send a priority
request to the central node. The certifier will
create a dummy entry at the central node IAR and will
not allow any incoming request to write on any item
listed in the above entry until a certain time period has
elapsed. Until such a time, the transaction can compute
its update values and send the request to the certifier.

5. SUMMARY
We have considered the implementation of the con­
currency control technique proposed by Kung (Kung and
Robinson, 1979), for distributed databases. A centralised
certifier has been designed in the proposed work. Contrary
to popular belief, it has been shown that the central node
can be made crash-resistant without adding significant cost
to processing in a centralised concurrency control tech­
nique, for distributed databases. The proposed setup is also
able to reap advantages of centralised control, as is some­
times desirable (Rothnie, Bernstein, Fox, Goodman,
Hammer, Landers, Reeve, Shipman and Wong, 1980) for
priority response or control.

REFERENCES
BADAL, D.Z. (1979), "Correctness of Concurrency Control and
Implementation in Distributed Databases", Proc. COMPSAC
Algorithms for Concurrency Control in Distributed Data­
bases Systems", Report TR-FEB-1980, Harvard University,
CASANOVA, M.A. (1979), "The Concurrency Control Problems for
sity, 1979.
KUNG, H.T., and ROBINSON, J.T. (1979), "On Optimistic Meth­ods
for Concurrency Control", Report CMU-CS-79-149.
PAPADIMITRIOU, C.H. (1979), "Serialisability of Concurrent
631-53.
ROTHNIE, J.R., BERNSTEIN, P.A., FOX, S.A., GOODMAN,
tem for Distributed Databases", ACM Trans. on Data Base

BIOGRAPHICAL NOTE
Dr P.G. Reddy has a PhD (1967) from IIT, Delhi and a
DIC (1970) from London. He is at present a professor (on
deputation) at Computer Science Department, Osmania
University, Hyderabad, India. He has been the project co­
ordinator of the ICL-2960 project and also incharge-oper­
as at IIT Delhi. He has been a visiting research fellow at
Imperial College, London (1969-71), Computer Science
Department, Stanford University and at MIT during 1980.
His current research interests include database systems,
simulation studies, artificial intelligence, decision support
systems and distributed computing. Dr Reddy is a mem­
ber of the British Computer Society and is also a member
of several Government bodies: Electronic Corporation of
India, University Grants Commission and Regional Com­
puting Centres. He is currently involved with planning of the
Regional Computing Centre at Hyderabad (India) and the
Institute of Computer Science to be set-up at Hyderabad.

S. Bhalja is currently working towards a PhD degree
at Indian Institute of Technology, Delhi. His research inter­
ests include database management systems and computer
communication networks. He did his BTech in Civil Engi­
neering in 1978 and Postgraduate Diploma in Computer
Science in 1979, both from IIT Delhi.

B.E. Prasad is a lecturer at Computer Centre, IIT
Delhi. He received his MSc in Mathematics from Andhra
University in 1976 and PhD in Computer Science from
Indian Institute of Technology, Delhi in 1981. His research
interests include computer data security, distributed data­
bases and hospital information systems.
A Language for Verifiable Modular Programming

M. A. Malik*

Using a constructive approach in program design, large programs are developed as a hierarchy of program modules. This paper describes a simple language suitable for the design of well-structured program modules. The language presented here was developed as part of a program verification system developed at the Queen's University of Belfast and consequently its design was greatly influenced by the program verification techniques used by that system. The paper does not include annotation relevant to verification or verification stages. The language is based on PASCAL but omits some unstructured and most structured types, user-defined types and global variables. It includes a modified version of SIMULA-67 class as a model data representation feature.

Key words and phrases: Modular programming, verification, data representation.
CR category: 4.2.

1. INTRODUCTION
While existing program verification systems have been used to verify many interesting and challenging examples, the range of programs amenable to program proving techniques is still very limited and formal proofs of correctness of large systems seem nowhere in sight. A number of reasons make it impossible to prove correctness of large systems; one basic reason being that most programming languages, as well as common programming practices, lack concern for formal verification of programs — partly because high-level languages incorporate sophisticated features.

There is a growing realisation that the theory of structured program design can significantly contribute to the development of reliable software. Well-structured and properly designed programs are more reliable and amenable to program proving techniques than conventional improperly structured programs. The general method of developing well-structured programs consists of successive refinements of the general (abstract) description of an algorithm. Each refinement in the description of a task is accompanied by a refinement of the description of the data through which the subtasks communicate.

Dahl and Hoare, in a Monograph on Hierarchical Program Structures (Dahl, Dijkstra and Hoare, 1972) show that any concept useful for understanding or constructing a computing process includes a data structure as well as one or more associated operations. They conclude that the only efficient way to deal with complicated systems is to develop such systems as a hierarchy of concepts. A system is described in terms of lower-level concepts. The lowest level corresponds to the concepts provided by the programming language used to implement the system; the high-level concepts correspond, in one form or another, to the system components.

Thus it is desirable that programming languages facilitate simple and efficient decomposition of programs. Many languages provide insufficiently sophisticated decomposition tools. Generally, block or procedure structures are used but these allow neither efficient information distribution nor the establishment of secure relationships between global data and the operations on that data defined by the procedure. A more powerful tool in this respect is the 'class' structure of SIMULA (Dahl, Myhrhaug, Nyggard 1970) which provides a mechanism for program construction that implicitly associates data and its operators. A program may use a number of classes, a class instance being an 'object' consisting of a data structure, operators on that data and code for initialisation of the data structure. Whereas the variables of an ALGOL or PASCAL procedure block cease to exist when the block is exited, the data attributes of a class object in SIMULA outlive the block in which the class object was created.

A data structuring facility such as the class would appear to have great significance in structured programming — the binding between data and operators created mentally by the programmer can be explicitly represented in his program thereby improving the structure (and hence the reliability) of the program and providing greater modularisation of large programs. In addition, proof techniques have been applied to the class structure (Hoare 1972).

2. THE LANGUAGE
The language is intended to be simple in structure, to include features which assist in writing well-structured programs and most importantly to facilitate proofs of their correctness. We describe the language by mentioning below modifications and extensions to the languages PASCAL (Wirth 1971) and SIMULA (Dahl et al. 1970) with which the reader is assumed to be familiar. The language does not include Label declarations, Constant definitions or Type definitions.

2.1 Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple-type</td>
<td>integer</td>
</tr>
<tr>
<td>array-type</td>
<td>class-type</td>
</tr>
</tbody>
</table>

Copyright © 1982, Australian Computer Society Inc.
General permission to republish, but not for profit, all or part of this material is granted, provided that ACJ's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society.

*Department of Mathematics, King Abdulaziz University, PO Box 9028, Jeddah, Saudi Arabia. Manuscript received March 1981, revised June 1982.

array-type ::= array [index-type {,index-type}] of component-type
index-type ::= integer-expression..integer-expression
component-type ::= type
class-type ::= class-identifier actual-parameter-list

Thus, only a restricted class of Pascal data types is permitted. Type definitions, enumerated types and records, sets, files and pointers are omitted to enhance the verifiability of the resulting program modules. This was done to avoid the complexity of proofs of non-trivial programs using these types. In contrast with Pascal, dynamic arrays are permitted.

Example: array [n..m*n+n] of real
where m and n are integers.

A class type specifies a data structure defined in the manner of a class. Each instance of a class consists of data, operations on that data and code for initialisation of the data. The operations are defined as local procedures and functions. Class definitions can appear in any block; they precede all other declarations local to a block and are described in Section 2.4.

2.2 Statements
All Pascal statements with the exception of with- and for-statements are allowed. In a case-statement, case labels are required to be integer constants due to the omission of enumerated types. As stated earlier, the language was designed as part of a program verification system (Malik 1976) and the omission of the for-statement and the restriction on the case labels are, among many others, a result of the limitations imposed by that system.

2.3 Procedures, Functions and Parameters
The language permits only two kinds of parameters like Pascal: constant (or call-by-value) parameters and variable (or call-by-reference) parameters. Moreover functions must not have variable parameters.

The language prohibits use of global variables except in procedures and functions local to a class definition. Although global variables are found to be a convenient feature in a programming language (e.g., Algol, Pascal and derivatives), it is now realised that these features make programs difficult to understand and result in ill-structured programs (Wulf and Shaw 1973). Moreover, global variables obstruct program proving. Whenever a procedure or function is called, the 'effects' of its execution are transmitted to the place-of-call. If access to global variables is permitted, the description of such effects may include the identifier of a global variable. If this description is interpreted at a point-of-call where the same identifier refers to a local variable, the identifier reference is ambiguous.

This restriction adds significantly to the modularity of procedures and functions since parameters are the only means of communication and each procedure and function can be treated as a complete and well-defined textual unit. However procedure and function attributes of a class need access to the variable attributes of the class. This exception is justified by ensuring that such variable attributes are qualified appropriately when referenced remotely.

In a procedure call, all actual parameters which may be modified by the procedure (i.e. those corresponding to variable parameters) must be non-subscripted variables, distinct from each other. This restriction is necessary to avoid erratic procedure calls, to guarantee their detection at compile-time, and to utilise Hoare's rule of substitution (Hoare 1971) for generation of lemmas by the program verification system (Malik 1976).

2.3.1 Standard Procedures and Functions
The following standard procedures and functions are included:
Procedures: read, write.
Functions: abs, sqr, sin, cos, exp, ln, sqrt, arctan, odd, trunc, int, chr.

Two new pre-defined functions, namely 'lolimit' and 'hilimit' are also available. If A is a two-dimensional array, then

lolimit (A,1) denotes the lower-bound of the first index of A, and
hilimit (A,2) denotes the upper-bound of the second index of A.

2.4 Classes
A class is identical to a procedure except that:
(i) Whereas local identifiers of a procedure block cease to exist when the block is exited, those of a class block (i.e. the attributes) outlive the class block.
(ii) A procedure is activated by a procedure statement, whereas a class instance in this language is created and bound to an identifier on translating a variable declaration which includes the class as a type. Such a call of classes results in the creation of appropriate objects as described in the next section.
(iii) A class may have value parameters only.

2.4.1 Class Definitions
In a class definition, local variables constitute the data representation being defined as a class, procedure and function declarations define operations on the data representation, and the statement part initialises it. (Thus the statement part is alternatively called the initialisation part.) All variables, procedures and functions local to a class are called its attributes. The name of any attribute of a class can be prefixed by an asterisk in its declaration to indicate that it is accessible from outside the class definition by remote identification (Section 2.4.2). Since a procedure attribute of a class defines an 'updating' of the class by modification of its variable attributes, a procedure attribute is restricted to have only value parameters.

class-definition ::= class-heading block
class-heading ::= class class-identifier
formal-parameter-part
block ::= class-definition-part
variable-declaration-part
procedure-and-function-declaration-part
statement-part
class-definition-part ::= {class-definition;}

As an example, consider a class definition (Figure 1) which represents a sequence of integers. It is deliberately simplified and ignores checks on errors, such as appending an item to an already full sequence and getting an item from an empty sequence.
class cyclicbuffer (max:integer);
  var *count, firstp:integer;
  table:array [0..max-1] of integer;
procedure *append(i:integer);
  begin firstp:= (firstp+count)mod max;
      count:= count+1
  end; /*appends an item at the end of the sequence*/
function *get: integer;
  begin get := table [firstp]; count:= count-1;
      firstp:= (firstp+1)mod max
  end; /*returns, and also removes the first item*/
function *firstvalue: integer;
  begin firstvalue := table [firstp] end;
/*returns but does not remove the first item*/
  begin firstp := 0; count := 0 end; /*initialisation*/
end; /*cyclicbuffer*/

Figure 1. The class definition cyclicbuffer.

An object of type 'cyclicbuffer' can hold up to 'Max' items which are stored in the array 'Table', used as a circular buffer. Whenever an item is stored in Table [Max-1], the next item is automatically sent to Table [0]. The number of items in the object is denoted by 'Count' and 'Firstp' gives the index of the first item at any instant. Three operations are defined on objects of type 'Cyclicbuffer'; they are appending an item at the end of the sequence, fetching and removing the first item, and fetching the value of the first item.

Max, Count, Firstp, Table, Append, Get and Firstvalue are attributes of Cyclicbuffer.

A class type appearing in a variable declaration generates an object for each of the variables being declared; each object is effectively an activation record of the corresponding class definition after execution of its statement part. Subsequent to the above class definition, a variable declaration like

C1,C2 : Cyclicbuffer (20);

makes available to the user two objects named C1 and C2. Each of the objects can hold up to 20 integers, is initialised to an empty sequence and is ready for any of the permissible operations - Append, Get and First-value. C1 and C2 are called variants of type cyclicbuffer.

2.4.2 Accessing Attributes by Remote Identification
Those attributes of a class variable that are preceded by an asterisk in their declaration can be accessed by remote identification using an attribute-value or an attribute procedure call. A remote identification consists of a class variable, followed by a dot, followed by an attribute.

(a) ATTRIBUTE-VALUE

attribute-value ::= class-variable, target-attribute
      target-attribute ::= variable | function-designator

An attribute-value appears in an expression and either accesses a variable attribute or activates a function attribute of the class variable preceding dot.

Examples: C1.count C2.get

It may be noted that C1.table [5] is invalid since Table is not prefixed with an asterisk in its declaration (Figure 1).

(b) ATTRIBUTE PROCEDURE CALL

attribute-procedure call ::= class-variable.
      procedure-statement

Example: C1.append(i), where i is an integer.

2.4.3 Attribute Procedures and Functions

Procedures and functions which are attributes of a class differ significantly from those which are not in respect of life-time, accessibility and purpose. These differences as well as considerations of program modularity and program proving necessitate the following 'exceptions' in the treatment of attribute procedures and functions:

1. They must not have variable parameters.
2. They are allowed access to the variable attributes of the corresponding class.
3. Attribute functions without parameters are permitted.
4. They may not be called in the initialisation part of the corresponding class, i.e., their role is to amend variable attributes after initialisation.
5. Within a class definition, call of a procedure or function which 'modifies' the class object is prohibited.

3. EXAMPLES

3.1 Smallintset

This example is due to Hoare (Hoare 1972) and involves the unification of a data structure and its associated operations into a single structure.

It is required to store and manipulate several small sets of integers, it being already known that the maximum size of such a set is, say, 20. Initially a set is empty. Various simple operations are defined on a set e.g. insertion or removal of specified integers and a membership test.

class smallintset;
  var *size: integer;
  a:array [1..20] of integer;
procedure *insert (i: integer);
  var j: integer;
  begin j := 1;
       while j <= size do
          if a [j] = i then goto 1 else j := j+1;
       if size = 20 then text ('error-too many',
              'members')
        else begin size := size+1; a [size] := i end;
  1: end; /*insert*/

procedure *remove (i: integer);
  var j: integer;
  begin j := 1;
       while j <= size do
          if a [j] != i then j := j+1
        else begin a [j] := a [size];
            size := size-1; goto 1 end;
  1: end; /*remove*/

function *has (i: integer) : boolean;
  var j: integer;
  begin has := false; j := 1;
       while j <= size do
          if a [j] = i then begin has := true; goto 1
                end else j := j+1;
  1: end; /*has*/
  begin size := 0 end; /*smallintset*/
A class smallintset is thus defined. Local to this definition are declared an array A to hold the integers in a set and an integer size to denote the number of members in the set. The operations of insertion, removal and membership testing, which are peculiar to the class of sets smallintset, are defined as local procedures/function of the class. The initialisation appears as the statement part of the class.

To make such a set available within a program, a class variable is declared:

```plaintext
var s1, s2: smallintset;
```

The action of such a declaration is to allocate storage for the data structure defined by the local variable declarations within smallintset and to execute the statement part initialising the class variable, i.e., size := 0.

Within the program, the actions of inserting and removing members of a set are achieved by prefixing the relevant procedure call with the name of the appropriate set e.g.

```plaintext
s1.insert(7)
if s1.has(i) then s2.insert(i).
```

Similarly size of any set may be ascertained, e.g.

```plaintext
k := s1.size.
```

A fully annotated version of the class Smallintset appears in (Malik 1980); lemmas generated for this annotated version and their proofs are given in (Malik 1976).

### 3.2 Frequency Histogram

This example is similar to an example of Dahl (Dahl et al. 1972). The real axis between the values "lower" and "upper" is partitioned into N equal intervals. A frequency histogram of a set of observations may be represented as a sequence of integers $T_0, \ldots, T_{N+1}$, where

- $T_0$ is the number of observations less than 'lower',
- $T_i$ (i=1, ..., N) is the number of observations in the i-th interval, and
- $T_{N+1}$ is the number of observations greater than 'upper'.

The observations are tabulated using an array Table [0...N+1].

Two standard operations are carried out by local procedures Tabulate, which updates the array Table according to the value of an observation, and output which prints out the information contained in the histogram. The initialisation of the histogram requires the zeroing of Table and Total, the current number of observations.

Since a histogram requires different numbers of intervals, the number of intervals and the lower and upper limits are parameters. The implementation of class variables with simple (unstructured) type parameters presents no problems (the introduction of parameters which are class variables is discussed in (Elder 1975)), since run-time storage for them is allocated within the class variable's workspace exactly as for local variables. The class Histogram may thus be defined.

```plaintext
class histogram(lower, upper: real; n: integer);
var partitions: array [0..n] of real;
*table: array [0..n+1] of integer;
step: real;
total, i: integer;
```

```plaintext
procedure *tabulate(y: real);
var i: integer;
begin
  if y <= lower then i := 0 else if y > upper then i := n+1
  else i := (y-lower)/step;
  table[i] := table[i]+1; total := total+1
end; /* tabulate */
```

```plaintext
procedure *output;
var j: integer;
begins
  if odd (j) then write (' ', table[(j-1)/2])
  else write (partitions[j/2], ' ', table[j]);
  write (eol);
end; /* output */
```

In the above example, the only variable whose value may be accessed from outside the class is the array Table and, since none of its elements may be assigned new values except by the procedure Tabulate a high degree of modularisation and security has been achieved.

Histograms may be declared with appropriate actual parameters e.g.

```plaintext
var ages : histogram(0, 100, .10);
weights : histogram(0, 20, .10);
```

and new observations recorded in the relevant histogram, e.g.

```plaintext
weights.tabulate(X) If X represents a weight,
ages.tabulate(A) If A represents an age.
```

### 4. IMPLEMENTATION OF CLASS

The effect of declaration of a class variable is to allocate, within the run-time storage area (workspace) of the procedure in which it is declared, contiguous area of storage to hold the following information:

(a) Control information associated with execution of the initialisation of the class variable.

(b) Storage for local variables of the class.

The object code initialising the class variable is then executed. By use of non-contiguous storage the storage used by (a) can then be reclaimed.

Access to variable attributes, e.g. s1.size, is simply by offset within the workspace of the class variable. Attribute procedure calls cause a workspace for the procedure to be
allocated on top of the run-time stack and this workspace is textually (statically) linked to the workspace of its qualifying class variable to enable reference to other attributes of that class variable, or a display technique may be employed. Upon exit from a procedure, its workspace (which will be on top of the dynamic run-time stack) is automatically deallocated and this will of course result in deallocation of the workspace of any local class variables. Thus garbage collection, as required in the Simula heap implementation, is avoided.

An object program normally consists of a series of object code sequences, each of which corresponds to a procedure or class definition statement part, in the source program. However, simple classes with small statement parts, such as smallintset, may be implemented very efficiently by macro(textual)-substitution.

5. CONCLUSIONS

Many languages reported in the recent past incorporate modularity constructs based on the Simula class (Dahl et al. 1970). Most of them are claimed to be designed for specific areas (e.g. Modula [Wirth 1977], Pascal-Plus [Welsh and Bustard 1979] and Concurrent Pascal [Hansen 1975] for multiprogramming, Ada [Barnas 1980] for embedded computer systems) with mostly no, and in some cases limited, concern for verification of the resulting software. Since the language described in this paper is designed with a different goal in mind, it differs considerably from these languages. For example, it provides neither multiprogramming facilities nor generic procedures/structures. Its class structure, although derived from Simula class like Module in Modula, Package in Ada and Envelope in Pascal-Plus, differs from them in respect of simplicity, structure, creation of class instances and access mechanism.

The class structure described here, although restrictive as compared with some of the existing data representation mechanisms, offers a reasonably safe and simple method of defining data representations whose correctness can be attempted using a proof method formulated by Hoare (Hoare 1972). This class structure has important advantages over existing methods of representation in Pascal:

1. The construction and proof of programs can be broken into modules.
2. The package-writer can rely upon compile-time scope checks to prevent misuse of his package (as a result of his ability to place access restrictions on variables and procedures).
3. The use of side-effects is well-disciplined.
4. The qualification mechanism for accessing attributes of a class considerably reduces the problems associated with name parameters, as they become less necessary in the case of classes. Instead of a class variable appearing as a name parameter in a procedure parameter list, it appears instead as the qualifier for the procedure call.

Comparing this class structure with that of Simula, two immediate advantages appear:

1. Class variables are accessed via identifiers (their names) rather than via references, so problems associated with pointers are removed.
2. Restricting the life-time of a class variable to that of the procedure in which it is declared means that its storage is de-allocated at the same time as that of its enclosing procedure and garbage collection is avoided.

Among other languages with improved structuring facilities, a module in Modula is comparable to our class structure but Modula lacks the facility to declare several instances of the same class. In Pascal-Plus, several instances of an envelope can be declared in a block but each instance envelope any instances declared after it, and then the body of the block itself — a feature suitable only for multiprogramming.

The language presented in this paper was developed as part of a program verification system. It was conceived as a language with simple structure, limited complexity and size, and possessing features that support well-structured program modules amenable to existing program verification techniques. These considerations lead to the omission of, or restricted use of, many common as well as advanced features of the current high-level languages.

In order to enhance verifiability of the resulting program modules and to enforce modularity effectively, some other restrictions become necessary. Reference to non-local variables is almost entirely excluded from the language; the only exception is that the function and procedure attributes of a class are allowed access to the variable attributes of the class. To avoid functions with side-effects, functions cannot take variable parameters. Since a class defines a new data representation always (i.e. it does not modify any existing object), its declaration can use only value parameters. Also by definition, an attribute procedure implements an operation on the class itself (i.e. it only updates the class object), therefore variable parameters are prohibited in attribute procedures.

However, label declarations, constant definitions and for-statements can be added to the language without jeopardising the verifiability considerations stated earlier.

Experience with some examples possessing characteristics of real life programs (Malik 1976) provides an indication that the language described is suitable for designing a variety of program modules. A proof of conditional correctness of such modules is then possible using a program verification system, after the modules are appropriately annotated using a specification language (Malik 1980) in a manner described in (Malik and Clint). It is hoped that this language will contribute to the design of reliable, well-structured and verifiable program modules.

ACKNOWLEDGEMENTS

The author is grateful to Tony Hoare for inspiration, to Jim Welsh, John Elder and Maurice Clint for many critical comments, and to the referees for suggesting significant improvements in the earlier versions of this paper.

REFERENCES


BIOGRAPHICAL NOTE
Dr. Malik has a Master's degree in Mathematics from Lahore (Pakistan) and he obtained his PhD in Computer Science from Queen's University of Belfast (UK). He has taught Mathematics and Computer Science in Pakistan and is presently teaching at King Abdulaziz University, Jeddah. His current research interest areas include programming languages, specification and verification of programs and data structures.
Teleprocessing Monitors and Program Structure

Roger Clarke*

This article discusses teleprocessing monitors, their impact on program-structure, and the development of such programs using the DELTA program-generator package.

Key words and phrases: DELTA, online programming, program-generators, program-structure, teleprocessing monitors, transaction processing.

CR categories: 4.12, 4.22, 4.31.

SCOPE OF THE ARTICLE

A previous article dealt with the preparation of 'self-contained' online programs using the program-generator package DELTA (Clarke, 1982b). These were defined as dialogue programs running under an operating system specifically designed for online or mixed online/batch operation. In such an environment terminal-communication is performed using inbuilt commands such as the ANSI COBOL SEND and RECEIVE verbs or extensions to the DISPLAY and ACCEPT verbs; or by CALLs to one or more special subprograms which perform the physical data transfer and then return control to the calling program in the normal manner.

This article deals with online programs of another type, those which run as subprograms to a so-called TP (Teleprocessing) Monitor or DC (Data Communications) Monitor. Commercial products of this kind include IBM CICS and the DC part of IMS/DC, the CINCOM product for IBM and IBM-compatible systems ENVIRON-1, Univac's TIP/CMS, Honeywell's TDS and Burroughs GEMCOS. The short form 'TPM' will be used in this article, and the term 'TP Program' will refer to a program running under a TPM.

The article will discuss the methods used to generate TP Programs. In order to do this however some features of TPMs must be first discussed. As the author has found few references which discuss TPMs in a suitable and supplier neutral manner, this preliminary discussion takes up a considerable part of the article.

MAIN MEMORY UTILISATION

A large installation is interacting with hundreds of active terminals. Each requires space in memory for the program it is communicating with, hence potentially vast amounts of main memory are demanded. Effective management of main memory is discussed here in isolation from the many other factors traded off by a multi-tasking operating system.

A first measure to save space is to enable a program to service more than one terminal. This is referred to as 'multi-threading', since each task 'threads' its way through the maze independently from, and unaffected by, the others. This is only possible if the program is in 'reentrant' form, i.e. if all data that may vary is stored in a 'variant' or 'data' segment. The procedural segment need exist only once in main memory and is referred to as 'shared code'; a data segment must exist for each active program. A COBOL analogue to a reentrant program is a program in which no variables are defined in the Working-Storage Section (only in the Linkage Section), and Commands which result in the generation of working areas are avoided — CALL, PERFORM, VARYING and ALTER being the main candidates (PDV, 1979).

Another contribution is made by a virtual-storage paging arrangement in which procedural and data segments not currently in use may be unloaded onto secondary (drum/disc) storage and reloaded when next required. This involves operating system overhead, but due to the enormous speed of processors and main memory (of the order of $10^{-6}$ seconds) compared with secondary storage ($2 \times 10^{-2}$) and especially terminal operators ($1$ to $10^2$ seconds — human real-time), considerable net savings can be achieved. DP practitioners are recommended to Keedy (1980) as a reference on virtual memory.

Yet another step is to reduce both the size of individual programs and the amount of processing squandered in administering paged-out segments. This is achieved by requiring programs to 'die' immediately after communicating with the Operating System, rather than merely being suspended pending the arrival of the next input. It can be argued that even batch programs should be organised in this way, although with most batch input coming from secondary storage the gains are likely to be far less than in the case of online programs whose input comes from a relatively very slow human being.

Against this potential gain must be balanced the size of the additional OS subsystem (the TPM) necessary to administer these very short-lived programs, plus the additional main and secondary memory management to enable programs to pass common data to their successor or to themselves in their forthcoming reincarnation. The net effect can be (but not necessarily is) a considerable gain. This is particularly so in the case of the originally batch oriented OS which have had terminal-handling and virtual/paging facilities progressively tacked on. Such (predominantly 'mainframe') OS are consequently somewhat less...
efficient in managing main memory, and the benefits shown by ephemeral programs appears correspondingly greater.

A primary factor stimulating the development of TPMs was the conservation of scarce main memory resources but a range of other factors were involved.

OTHER FACTORS LEADING TO TP-MONITORS

Centralised Terminal-Handling

The varying physical characteristics of terminals can be catered for by a central subsystem, enabling application programs to deal with a standardised 'logical' interface. This is really an argument for a Terminal Control Program (TCP) and is equally applicable to environments in which online programs are self-contained.

Centralised Formal Editing of Input Data

'Automatic' checking of the appropriateness of data in each field can be performed by the TPM (or for that matter by the TCP).

Optimised Data Stream Transmission

It can be important to keep line traffic to a minimum, especially in highly dispersed networks. A significant contribution can be made by a routine (be it in the OS, the TCP or in the TPM), which compares the desired screen image with that currently displayed and transmits the minimum data needed to effect the change. (Reducing terminal-to-processor data-flow is of course less easily achieved, requiring intelligence and storage in the terminal itself.)

Supply of Preprocessed Data Streams to the Application Program

The simplest approach is to provide the program with the current contents of the screen, irrespective of what was received in the most recent transmission. An improved service might be to supply, in addition to the data stream, tables showing which fields were changed, which remain unkeyed, which contain data inappropriate to the field definition, etc. This can be provided equally well in a TCP as in a TPM, since it is closely bound to the physical characteristics of the device.

Centralised Control over Control Flow

Software must decide which program is required to service the input from the terminal. This function can be built into the TPM. It can be performed equally well by a (relatively tiny) table-driven menu-handler embedded in the OS; or by a user-written master program which administers the menu displays and arranges for the chosen program to be run immediately after its own demise (commonly referred to as 'Call-Next-Program'). Each application program running in such an environment must admittedly comply with the requirement that it pass control back to the master program on completion, but as will be seen, a similar discipline is required with a TPM.

From the above it will be clear that these factors, important as they are, justify the creation of a Terminal Control Program to administer the link between application program and terminal, but not the conception, realisation and implementation of the altogether more complex TP-Monitor. The real reason for TP-Monitors must be sought elsewhere.

 TRANSACTION-ORIENTED PROGRAMMING

It can be argued that the 'natural' way to study organisations, and to 'objectively' document their present and intended functions, is to identify 'work steps' or organisational transactions. If so (and the issue remains unresolved) then the 'natural' form that programs should take is also transaction-oriented. The argument isn't just one of structural elegance: much software development activity is presently invested in the translation of design information from one form into another, hence great savings could be made if all stages of the production line were to acknowledge the same methodological framework.

In such an environment we would then redefine the role of information-processing services as the recording and processing of organisational transactions, each transaction being triggered by an organisational 'event'. This brings commercial processing much closer in concept to process control or 'real-time' processing.

Batch Processing is then seen as an alternative means for handling large-volume or low-priority work. High-volume, long lead-time tasks need to be performed 'asynchronously' with respect to the terminal, such that the end-user can himself schedule them, but without blocking his own terminal. There is then no reason why batch (terminal-asynchronous) jobs cannot run in a transaction-oriented manner, under the same Monitor as online (terminal-synchronous) tasks.

This is very convenient when an application comprises sub-functions that are to be performed in either and both online and batch modes. One example is a sub-function like the look-up of article price and article discount (often dependent on a range of attributes of customer, article and order), which may be needed in the online program 'Urgent Quotations' and in the batch program 'Low-Priority Invoices'. Another is recovery forward from a checkpoint, when a perhaps large number of events that were originally handled synchronously are to be reprocessed in asynchronous mode.

DEFINITION OF A TPM

Much of the literature on the subject comprises the reference material of the various suppliers of commercial products. These are (quite justifiably) biased towards the specifics of their own marketing strategy, host Operating System(s), etc. See however Mills 1972, Davenport 1974, KDCS 1978, PDV 1979 and Datapro 1979a and 1979b.

A working definition is suggested as follows:

A Teleprocessing Monitor is a subsystem of an Operating System, which administers the logical and perhaps also the physical link between each terminal and the program(s) invoked to perform tasks initiated by that terminal.

The term 'logical link' refers to communication with an idealised terminal, independent of the physical characteristics of whatever terminal is physically on the other end of the wire. The physical interfacing tasks therefore include code conversion, synchronisation of physical data flows and line-control/protocol-handling to the extent that this is not performed by a communications network monitor. As indicated in Figure 1, these functions may be embedded in the TPM or delegated to a Terminal Control module.
FUNCTIONS OF A TP-MONITOR

These are grouped according to whether or not they are assessed by the author as being essential to the above definition.

Essential Functions
Administration of I/O-Data
- Data streams from terminals.
- Data streams to terminals.

Administration of Control Flow
- Selection of the next program.
- Passing control to that program.
- Receiving control back from that program.

Administration of Working Data
- Making data available to programs.
- Receiving data from programs.

Additional Functions
Physical terminal-handling
Embedding TCP within TPM.

Interface with Permanent Data
Special commands or calls may provide the application program with an improved and/or standardised interface with data files (sequential, direct, indexed, multi-indexed, tape, etc) or database. If the DBMS software is integrated with the TPM, the resulting conglomerate is referred to as a DB/DC Monitor. Some TPMs go so far as to preclude the use of standard file-handling commands.

Starting of Asynchronous (Batch) Jobs

Screen-Definition Facilities
Utilities may be associated with the TPM to support the specification and assembly of program-independent masks. The term 'Forms Processor' is also used. At one level the preparation of the physical layout may be supported ('screen painting' or 'forms editing'); at another the specification of mask and field attributes.

Logfiles, Checkpoints, Restart/Recovery

Accounting

News Broadcasts and Terminal/Terminal Communications

Security
- e.g. User-Authorisation Checking.

Testing Facilities

A further function that can be performed by a TPM relates to control-passing between the application-programs.

PASSING OF CONTROL WITHIN A TP-MONITOR APPLICATION

The simplest possible arrangement is that the TP-Monitor decides on the basis of some field in the input data stream which program is to be called. That program passes control to the TPM when it terminates, together with a data stream for transmission back to the terminal. This data stream may contain the name of the program which is to be invoked when the operator next transmits.

If the field is unprotected then the operator will be able to override the default next-program call. It may be necessary to key the whole program number in, or, more conveniently a transaction-identification, function-key, or selection-number or mnemonic which is converted into the program name by table-lookup within the TPM. In this way the operator is able to remain oblivious to program initiation and termination.

The whole of the processing may in principle be placed in one large program. More realistically it may be sub-divided into various subprograms (see Figure 2a). With
complex processing main-memory requirements may still become too large, as the control program will remain in memory at the same time as the invoked processing-code.

As an alternative to the CALL mechanism, the TPM can provide the means for transferring control, as shown in Figure 2b. A chain of control is passed along, with the 'terminal' programs having contact with the screen, the processing programs only with other programs, the TPM and the permanent data. The advantages are that some main memory overhead is avoided and that only one type of data-passing is involved i.e. 'messages' to and from the TPM. A major disadvantage is that the application's structure is scattered throughout the entire application.

A further alternative has been provided by some TPM suppliers who support a 'pseudo-conversational' mode. This enables the program to be written in a self-contained form (see Clarke 1982b) with control decisions embedded within the program. The program then commences with a decision as to where within the processing it should commence (in COBOL terminology a GO TO DEPENDING; there is arguably no equivalent construct recognised by structured programming theory). The program increases in size, but this is of little consequence under an Operating System which supports virtual memory.

STRUCTURE OF TP PROGRAMS

The effects of a TP-environment upon program structure are considerable. In Figure 3a is shown a simple 'logical program' as it might be designed using Jackson's Program Design Method (Jackson, 1981). It could be implemented in that form as a self-contained program (see Clarke 1982b).

In Figure 3b the same logical program is shown as it might appear under a TP-Monitor. The first difference is that there are several 'physical programs' required to implement the single 'logical program'.

A more critical difference is that each (physical) program commences with the receipt of a message and concludes with the sending of a message. A way to describe the relationship between the two structures in Figures 3a and 3b, is to say that the first has been 'inverted' with respect to its driving file to produce the second.

Each 'TP-Program' runs as a subprogram to the TPM. Data may be transferred between TPM and TP-Program via the LINKAGE SECTION and/or via additional facilities depending on the particular TP Monitor. This article focuses on control-structure rather than message-passing.
A single 'logical program' may do more than merely, say, display a record on the screen. It may also, depending (commonly) upon an Operation Code, create, amend or delete a data record. There would be then a Selection Construct within the Processing Body, and the flow of control between the various 'physical programs' making up the 'logical program' quickly becomes tortuous.

There are three ways to handle the decision-making:

— embed it within the TP-Monitor (requiring a language powerful enough to express the various possibilities). Figure 3c illustrates this;
— embed it within a control program which remains memory-resident, invoking the chosen functions as subprograms (Figure 2a);
— perform the decision-making within a control program which passes control via the TPM to the chosen program (Figure 2b).

Where all decision-making and routing for a single logical program is performed by a control program, a means is required for identifying the current context in an accurate and efficient manner. State Transition Tables are an appropriate device (see for example Juliff 1980, Peterieit 1980, Hext 1982). This article will not deal in any detail with this approach.

Two classes of TP Program can be identified. Those which exchange messages not only with other programs but also with the terminal are referred to here as Terminal-Handling Programs. Those which communicate exclusively with other programs perform a strictly processing function and are referred to here as Data-Processing Programs. With a little care there is no reason why the latter should not be able to be invoked alike by terminal-synchronous (online) and terminal-asynchronous (batch) tasks. The use of TP-Monitors to control batch processing is not further discussed in this article.

A general structure is suggested in Figure 4, sufficient to cater for most eventualities. The possible processing functions (not all of which need necessarily be relevant to any given program) are enveloped by functions catering for the communications from and to the TPM. The actual implementation of these communication functions depends on the particular software environment.

On the basis of this general structure a set of macros will be discussed which provides close support to the programming phase. First a brief introduction to program generators is in order.

PROGRAM GENERATORS IN GENERAL

An assembler converts a source file directly into executable form. A compiler deals with a source file differently organised and sequenced from the object code it is to create. A program-generator differs from them in the following ways:

— the source-code is function- rather than procedure-oriented;
— its output may be a high-level language, for input to a compiler. Early versions of compilers often used such a two-step technique by outputing an assembler program. In the case of program generators, however, this is not necessarily a temporary measure, as it caters for multiple incompatible target compilers;
— as with the more modern assemblers and some compilable languages, it commonly includes a macro-language and processor, such that the source-language is user-extensible.

Few satisfactory products are marketed. Most are specific to particular machines, e.g. MANTIS and UFO under particular IBM operating systems, NoCode from General Automation, LINC from Burroughs NZ, and the cutely-named 'The Last One', a UK product generating Basic in CP/M and UNIX environments. Philip's PET/MAESTRO development-machine incorporates generator functions. Clarke (1982a) provides an introduction to the topic.

A powerful generator package with which the author is familiar is independent of both its host software environment and its target environments. DELTA is a Swiss product, marketed since 1976 in German-speaking areas and since 1980 also in the UK and Australia, with some 150 installations to date.

ONLINE PROGRAMS USING DELTA

The primary objectives of the development phase (quick and cheap development, a clean product, portability and low maintenance and enhancement costs and lead-times) can be readily supported by DELTA together with some customised macros.

Several developments in Germany and Switzerland, notably at Systema GmbH, Mannheim (Clemens 1981, but see also Ahrens 1981 and Thurner 1981) have used DELTA in a context of control programs using State Transition Tables. The structure suggested in Figure 4 requires extension to serve the purpose of such control programs, but because of its relative simplicity will be used below to illustrate the use of the DELTA tool-kit.
In addition to the facilities provided by the product itself, a set of macros is required, to generate from a short list of parameters the appropriate program shell, the internal decision-making structure, and the communications with both the TPM and the permanent-data handling environment. The actual processing can be coded in COBOL or PL/1, or where portability is important, exclusively in invocations of DELTA macros.

The invocation of the program structure can be nested within the generation of the basic program shell:

```
. PROG-progname, AUTHOR=xxxx, DATE-WRITTEN=xx/xx/xx, TYPE=(DPP, COMMON=NO, FORMAL=NO, XEDIT-FILE=NO),
   MASK=xxxx, MSG=(xxxx,xxxx,xxxx,...)
```

The Keyword-Parameter 'TYPE' controls the class of program to be generated (Terminal Handling will be in this case excluded), and particular sub-functions can be selected out (or if preferred, selected in), with a default list applicable. The list shown above would exclude the sub-functions 'Fetch Common Data', 'Formal Field-Editing' and 'Cross-Editing Against Reference-Files'. The Keyword-Parameters MASK and MSG define those screen-related data areas and TPM-communication records that are to be invoked from the Data Dictionary.

The other sub-functions are made available, and one or more 'Locations' defined in each, to enable the processing code to be inserted. These Locations are named EDIT-FLD, XEDIT-FLDS and PROCESS. The code would be inserted in the following form:

```
.SL=EDIT-FLD
.ADD TESTRANGE, CUST-DISCOUNT, (0, 6), 417
.ADD TESTLIST, CUST-SLSZONE, (1,4,5,6), 418
.SL=XEDIT-FLDS
.ADD TESTEXCL, (CUST-DISCOUNT NOT = ZERO), (CUST-SLSZONE = 6), 419
.SL=PROCESS
.ADD MOVE, CUST, xxxx, (NAME, ADDRESS1, ADDRESS2, POSTCVD)
.ADD LASTUPDAT, CUST
.ADD PUT, NEW, CUST
.ADD PREPLOG, CUST
.ADD PUT, APPEND, LOG
.ADD PREPMASK, (MSG = 'CUSTOMER CREATED')
```

The processing code is generated by the minor macros invoked here. In each case the interfaces to error handling and exception-processing functions are generated automatically.

In practice even deeply-nested sub-functions can contain low level selection- and iteration-constructs in addition to the sequential processing of the above example. Of the several DELTA tools available for this task the 'pseudo-code' interpreter SPP is the most suitable.

The program end is signified, together with the field-name which contains the Next-Program name:

```
.PROG-END SELNPROG
```

The above approach is generalised and simplified. Nonetheless the experience of Systema GmbH with a more powerful model is relevant: during 1980/81 typical program-modules required 25-40 lines of specification at the design stage and 150-250 lines of coding. The (COBOL) code generated was of the order of 1500-2500 lines.

**OVERVIEW OF THE MACRO STRUCTURE**

The design and construction effort to provide such a macro-set is not small. It is important to identify the different levels of abstraction, and to recognise those nested functions which may vary, if only subtly, between one program and another. These should be implemented in separate macros if flexibility is to be maintained.

Three broad levels of abstraction are useful (although the classes are clearly not disjoint):
- the programmer interface;
- nested macros to provide program flow control and the logical interfaces;
- deeply-nested macros to handle the physical interfaces.

The list below illustrates what is included in each of these levels. Low-level macros can themselves invoke further macros.

<table>
<thead>
<tr>
<th>Programmer Interface</th>
<th>Logical Level</th>
<th>Physical Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROG</td>
<td>THP/DPP</td>
<td>RC-TAB-ANSI</td>
</tr>
<tr>
<td>TESTRANGE</td>
<td>ERRMSG</td>
<td>RC-TAB-SPEC</td>
</tr>
<tr>
<td>TESTLIST</td>
<td>ERRMSG</td>
<td>PHERMSG</td>
</tr>
<tr>
<td>TESTEXCL</td>
<td>ERRMSG</td>
<td>PHERMSG</td>
</tr>
<tr>
<td>LASTUPDAT</td>
<td>MOVE</td>
<td>MOVE</td>
</tr>
<tr>
<td>PUT</td>
<td>MOVED</td>
<td>MOVED</td>
</tr>
<tr>
<td>PREPLOG</td>
<td>MOVE</td>
<td>MOVE</td>
</tr>
<tr>
<td>PREPMASK</td>
<td>MOVE</td>
<td>MOVE</td>
</tr>
<tr>
<td>LR-CUST</td>
<td>LR-ROUTINES</td>
<td>FD-CUST-ISRFILE</td>
</tr>
<tr>
<td>PROGEND</td>
<td>SELNPORG</td>
<td>PR-CUST-1</td>
</tr>
<tr>
<td>SELNPORG</td>
<td></td>
<td>PR-CUST-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PHSELPREG</td>
</tr>
</tbody>
</table>

**CONCLUSION**

Transaction-oriented on-line programs exhibit a variety of forms, depending on the particular TP-Monitor in use, and the using organisation's experience and philosophy.

Design and development of such programs using an advanced program generator such as DELTA enables standardisation of methods of working, savings in development and maintenance, improved planning and control, and portability of product.

**ACKNOWLEDGEMENT**

Die Hilfe von verschiedenen Kollegen im deutschsprachlichen Raum wird dankend anerkannt, insbesondere Hrn Karl-Heinz Clemens und Hrn Dr Rainer Petereit von der Firma Systema GmbH, Mannheim.

REFERENCES


BIBLIOGRAPHICAL NOTE

Details were published in the May 1982 issue of this Journal. Since then the author has returned to Australia. He is acting as a consultant, primarily in the field of commercial software development. He is associated with Software Solutions Pty Ltd, for whom he manages the program generator package DELTA.
A Review of Disk Scheduling Policies

B. R. Howarth*

INTRODUCTION

To obtain good performance from a computer system, it is necessary to investigate the performance of all sections of the system. Ultimately, the subsystems cannot be considered only in isolation, but individual study of the subsystems is essential to intelligent analysis of the system as a whole.

This review discusses ways of improving performance of disk systems, where “performance” is measured as transfers per unit time. Smaller timesharing systems in particular may be disk-bound, that is, they may show a combination of poor response times, low central processor utilisation, and high disk activity. We are concerned with measures applicable to a single disk drive.

This review is intended to help those responsible for system operations in assessing the possibilities open to them for improving disk performance. The next section gives a brief outline of disk organisation. The value of data placement and selection of appropriate record size are then discussed. An analysis of some of the possible scheduling policies is next presented. This analysis is supplemented and extended by consideration of the results of a simulation. Finally, we consider the practical utility of the approaches discussed.

REVIEW

Physical Disk Organisation

The principles of operation of moving head disk systems are well described elsewhere, e.g., Hamacher et al. (1978). Here, it will suffice to characterise the device in terms of Figure 1. The disk pack is a set of platters with magnetically active surfaces on a common drive spindle, revolving typically at speeds of 2400 rpm. The set of read-write heads “fly” in close proximity to these surfaces, and are connected to a common drive mechanism which can move them radially to any one of a number of fixed positions. This assembly, including motors, etc., makes up a disk drive.

With this layout, the information storage capacity of the disk pack can be divided into physical entities. A track is the data that can be accessed by one head at one radius on a disk surface. A cylinder is a set of tracks that can be

specialised by the formula $t = a + bc$, where $a$ and $b$ are constants, and $c$ is the number of cylinders crossed.

The values of $a$ and $b$ in the above expression, and the accuracy of the linearity approximation, vary greatly among drives. It is generally true, however, that $a$ is considerably larger than $b$. As an example, in the system simulated in this paper, $a$ is 9 and $b$ is 0.3, to give a seek time in milliseconds (see Table 1). This is due to the times required for acceleration, deceleration, and positioning. As a result, the seek time for one cylinder is not much less than the seek time for several cylinders. In our example, the seek time for one cylinder (9.3 msec) is only doubled for a seek of 32 cylinders.

Many installations have several disk drives, sometimes called spindles. Different manufacturers use different ways of connecting disk drives to the computer. For our purposes, it is sufficient to identify the elements controller and

Figure 1. Principal components of a moving-head disk drive.

In many computer systems, overall throughput is limited by the number of transfers per second between the disks and main memory. This article reviews techniques proposed in the literature for improving performance of a single disk drive. The methods of selecting the next request to be serviced from those waiting (i.e., scheduling policies) are the principal area studied. Results of a simulation confirm the analyses. The review shows that while scheduling policies more complex than FIFO offer improvements in average service time, these are not very great under normal operating regimes of most computers. Better improvement can often be gained by careful placement of data on the disk.

Key words and phrases: performance evaluation, disk scheduling, FIFO, SSTF, simulation.

CR categories: 3.73, 4.6, 6.34 [B.4.4, D.4.3].

*Lecturer, School of Computing Sciences, NSW Institute of Technology. Manuscript received January, 1982.
The transfer rate can be very high (3 Mbytes per second in modern drives). However, the preliminaries mentioned above generally take considerably longer than the actual transfer, so that the average transfer rate is much lower than this.

Directories have many uses: not least, they allow implementation of security measures. Directories need concern us no further here, except to realise that their use may add considerably to the number of disk requests made in a system. However, the preliminaries mentioned above generally take considerably longer than the actual transfer, so that the average transfer rate is much lower than this.

**Simulation Runs**

Each run consisted of an initial "settling period" of 200 requests, followed by statistics gathering over about 500 requests. Runs were made for nominal request rates of 5, 10, 15, 20, 25, 30, 35, 40 requests per second. Because of queue size limitations, the highest request rates were not achieved.

**Logical Organisation of Data**

Only in the simplest systems is data accessed directly from known physical locations on disk by application programs. In most cases, the data on disk is organised into logical entities called files. The records belonging to a file may be contiguous, or distributed more or less randomly across a disk. The operating system converts file references from the application programs into physical addresses on a disk.

Information about files such as the file owner, file name, starting address is kept on disk in special files called directories. There may be one directory on a disk, or there may be many directories arranged in a tree structure. Directories have many uses: not least, they allow implementation of security measures. Directories need concern us no further here, except to realise that their use may add considerably to the number of disk requests made in a system. A request to open a file and read its first record may involve access to several directories as well as to the file itself, before the record can be transferred to main storage.

**Improving Transfer Capacity**

Once a record is accessed, the rate of transfer can be very high (3 Mbytes per second in modern drives). However, the preliminaries mentioned above generally take considerably longer than the actual transfer, so that the average transfer rate is much lower than this.

**Disk Scheduling Policies**

In the sections that follow, we describe several of the techniques that have been proposed to improve average disk transfer rate. Most of these concentrate on reduction of average seek distance, and therefore of seek time.

**Data Placement on Disk**

Where it can be applied, a most effective way of reducing average seek distance is to place the most frequently used records physically close to each other and close to the middle cylinders on the disk. These will usually include directories, as well as heavily used data files.

The reduction of mean seek distance by use of this technique can be very great indeed. Hofri (1980) quotes an earlier reference (Lynch, 1972), where it was reported that in one system, successive disk requests were for the same cylinder two thirds of the time.

This approach is not always possible. A large, heavily-used system may not get time to do the regular re-organisations required to maximise the benefit of this approach. In a time-sharing system, the set of most-used files at any time will depend on which users are logged in, and these will vary with time in a largely unpredictable fashion.

There is one simple application of this principle, however, that may be possible in many systems: the placement of all directories in the middle of the disk, i.e., on cylinder 100 of a 200-cylinder disk. Many systems by default place the master directories on cylinder zero, so that the directory reference required to open or close a file could result in a seek right across the disk. If the directories are in the middle of the disk, average seek distances should be reduced.

**Optimisation of Record Size**

One way to improve apparent disk performance is to reduce the number of transfers required. Clearly, if all of a sequential file is to be accessed, doubling the disk block size (as opposed to the logical record size) will halve the number of disk transfers required for that file.

It is obvious that disk block sizes cannot usefully be increased without limit. Larger amounts of main storage are required for buffering larger records, and these buffers must always be resident during use. Also, large block sizes are not always efficient: if only a small amount of data is required then most of the resources used to transfer a large record will be wasted.

In consequence, there will be some optimum block size for a given system. If block sizes can be changed (not all disk systems allow this), and if the criteria for optimality can be defined, then an analysis to find the most suitable block size, and use of that block size, should result in some improvement in overall disk performance.

**Scheduling Policies**

When one disk transfer is completed, and more than one request is waiting for service, one of those waiting must be selected as the next to be serviced. Several selection (or scheduling) policies have been described in the literature; some of these are discussed and compared in this section.

Added insight into the effects of the policies is obtained from simulation results. The system modelled is an early configuration of the NSWIT School of Computing Science's PRIME 550 (the machine has since been expanded). Simulation parameters are given in Table 1.

The scheduling policies are discussed here mainly in terms of their effect on mean seek distance. To provide a

---

Table 1. Simulation Parameters

<table>
<thead>
<tr>
<th>Disk Drive Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Cylinders: 200</td>
</tr>
<tr>
<td>Seek time: 0 for no seek; 9 + .3n msec for n cylinders crossed</td>
</tr>
<tr>
<td>Rotational speed: 2400 rpm</td>
</tr>
<tr>
<td>Record transfer time: 1/16th of revolution time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Request Sequence Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cylinder numbers: Uniformly distributed. Assigned by use of a dedicated random number generator. The same sequence is used for each simulation run.</td>
</tr>
<tr>
<td>Maximum size of request queue: 50 entries.</td>
</tr>
<tr>
<td>Request inter-arrival times: exponentially distributed at low rates, changing towards replacement of removed queue item at high rates.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulation Runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each run consisted of an initial &quot;settling period&quot; of 200 requests, followed by statistics gathering over about 500 requests.</td>
</tr>
<tr>
<td>Runs were made for nominal request rates of 5, 10, 15, 20, 25, 30, 35, 40 requests per second. Because of queue size limitations, the highest request rates were not achieved.</td>
</tr>
</tbody>
</table>

---

The controller is the hardware element that instructs the individual drives as to the head movements, etc, required. The channel is the communication medium for transmission of data, including instructions as to what is to be read or written as well as the data itself. In general, there will be multiple controllers and channels in a large system. A disk subsystem may share a channel with other peripherals; in larger systems, a disk subsystem may be connected to two channels.

For a particular record to be written to or read from main storage, some or all of the following steps are required. The channel must first be seized to transmit a seek command to the requisite drive. A seek must be performed, if necessary. The channel must be seized again, to allow the transfer to take place. The disk must then rotate until the desired record approaches the read-write head; this is called rotational latency. The transfer can then take place.

The controller is the hardware element that instructs the individual drives as to the head movements, etc, required. The channel is the communication medium for transmission of data, including instructions as to what is to be read or written as well as the data itself. In general, there will be multiple controllers and channels in a large system. A disk subsystem may share a channel with other peripherals; in larger systems, a disk subsystem may be connected to two channels.

For a particular record to be written to or read from main storage, some or all of the following steps are required. The channel must first be seized to transmit a seek command to the requisite drive. A seek must be performed, if necessary. The channel must be seized again, to allow the transfer to take place. The disk must then rotate until the desired record approaches the read-write head; this is called rotational latency. The transfer can then take place.

Logical Organisation of Data

Only in the simplest systems is data accessed directly from known physical locations on disk by application programs. In most cases, the data on disk is organised into logical entities called files. The records belonging to a file may be contiguous, or distributed more or less randomly across a disk. The operating system converts file references from the application programs into physical addresses on a disk.

Information about files such as the file owner, file name, starting address is kept on disk in special files called directories. There may be one directory on a disk, or there may be many directories arranged in a tree structure. Directories have many uses: not least, they allow implementation of security measures. Directories need concern us no further here, except to realise that their use may add considerably to the number of disk requests made in a system. A request to open a file and read its first record may involve access to several directories as well as to the file itself, before the record can be transferred to main storage.

**Improving Transfer Capacity**

Once a record is accessed, the rate of transfer can be very high (3 Mbytes per second in modern drives). However, the preliminaries mentioned above generally take considerably longer than the actual transfer, so that the average transfer rate is much lower than this.

In the sections that follow, we describe several of the techniques that have been proposed to improve average disk transfer rate. Most of these concentrate on reduction of average seek distance, and therefore of seek time.

**Data Placement on Disk**

Where it can be applied, a most effective way of reducing average seek distance is to place the most frequently used records physically close to each other and close to the middle cylinders on the disk. These will usually include directories, as well as heavily used data files.

The reduction of mean seek distance by use of this technique can be very great indeed. Hofri (1980) quotes an earlier reference (Lynch, 1972), where it was reported that in one system, successive disk requests were for the same cylinder two thirds of the time.

This approach is not always possible. A large, heavily-used system may not get time to do the regular re-organisations required to maximise the benefit of this approach. In a time-sharing system, the set of most-used files at any time will depend on which users are logged in, and these will vary with time in a largely unpredictable fashion.

There is one simple application of this principle, however, that may be possible in many systems: the placement of all directories in the middle of the disk, i.e., on cylinder 100 of a 200-cylinder disk. Many systems by default place the master directories on cylinder zero, so that the directory reference required to open or close a file could result in a seek right across the disk. If the directories are in the middle of the disk, average seek distances should be reduced.

**Optimisation of Record Size**

One way to improve apparent disk performance is to reduce the number of transfers required. Clearly, if all of a sequential file is to be accessed, doubling the disk block size (as opposed to the logical record size) will halve the number of disk transfers required for that file.

It is obvious that disk block sizes cannot usefully be increased without limit. Larger amounts of main storage are required for buffering larger records, and these buffers must always be resident during use. Also, large block sizes are not always efficient: if only a small amount of data is required then most of the resources used to transfer a large record will be wasted.

In consequence, there will be some optimum block size for a given system. If block sizes can be changed (not all disk systems allow this), and if the criteria for optimality can be defined, then an analysis to find the most suitable block size, and use of that block size, should result in some improvement in overall disk performance.

**Scheduling Policies**

When one disk transfer is completed, and more than one request is waiting for service, one of those waiting must be selected as the next to be serviced. Several selection (or scheduling) policies have been described in the literature; some of these are discussed and compared in this section.

Added insight into the effects of the policies is obtained from simulation results. The system modelled is an early configuration of the NSWIT School of Computing Science's PRIME 550 (the machine has since been expanded). Simulation parameters are given in Table 1.

The scheduling policies are discussed here mainly in terms of their effect on mean seek distance. To provide a
uniform basis of comparison, and to duplicate the conditions of the simulation, the requested records are assumed to be distributed randomly, with each record on the disk having equal probability of selection, i.e., a uniform distribution of requests. This is a "worst case", since any concentration of requests due to the techniques mentioned above will reduce mean seek distances. The uniform distribution is also the easiest to model mathematically.

All the results below assume that delay is introduced (i.e., queues are formed) only at a disk. If several devices contend for a channel, queues may form for the channel. Likewise, if several drives are attached to one controller, queues may form at the controller. Consideration of the effects of these additional contentions is outside the scope of this paper.

**FIFO**

The simplest scheduling policy is first-in, first-out (FIFO, also called first-come-first-served, or FCFS). The requests are maintained as a queue, or linked list. New requests are added to the tail of the queue, and the request selected for service is that at the head of the queue.

Wilhelm (1976) showed that for uniformly distributed requests, the mean seek distance is \( \text{MAXCYL}/3 \), where MAXCYL is the number of cylinders on the disk, i.e., the largest seek that can be requested.

FIFO appears to be the most commonly used scheduling policy.

**SSTF**

In the "shortest-seek-time-first" policy (SSTF), all waiting requests are examined when service of a request has been completed. The request that requires the shortest seek (or no seek at all) is taken from the request pool and serviced next. Obviously, if only one request at most is ever waiting, i.e., at low request rates, the policy gives results indistinguishable from FIFO.

The mean seek distance for SSTF and the following policy, SCAN, are difficult to determine analytically. It is not difficult to show that if the queue length of waiting requests could be fixed at \( N \), then the mean seek distance would be \( \text{MAXCYL}/(N+2) \). However, queue length varies randomly, as a result of short-term mismatches between arrival rate and service time, so that this result can only be an approximation. It does reflect the relationship between mean seek distance and queue length that could be expected, since a larger pool of requests is more likely to produce a favourable selection. Queue length and service rate are thus mutually interdependent. From our simulation, average seek distance is plotted against average queue length in Figure 2, as is the function \( \text{MAXCYL}/(N+2) \).

It can be seen that the simulation results for both SSTF and SCAN follow the approximate curve remarkably closely for intermediate queue lengths. For small queue lengths, FIFO behaviour is approached, in agreement with our earlier statement. At large queue lengths, the simulation results fall below the curve slightly. Part of the reason for this is that successive services are sometimes for the same cylinder.

SSTF has the potential disadvantage that if all but one of a burst of requests are concentrated on a few cylinders, then the one that requests a "distant" cylinder will be the last one to be serviced. The logical extension of this is that if the disk is continuously busy, i.e., fully utilised, some requests may never be serviced.

**SCAN**

The SCAN policy was developed from SSTF in light of the problems raised in the last paragraph. In this policy, the disk heads are kept moving in one direction by selecting for service the request that is closest in that direction. When no further requests remain for service in that direction, the direction of travel is reversed. Thus, any request that enters the queue will be serviced within two sweeps (one in each direction) of the disk.

As with SSTF, mean seek distance is a function of queue length, and at low request rates, SCAN becomes indistinguishable from FIFO. One could expect that mean seek distances might be longer with SCAN than for SSTF.

In fact, as Figure 2 shows, the mean seek distance has almost identical dependence on queue length for SSTF and SCAN. Teory and Pinkerton (1972) point out that under SSTF at high loads, the heads will sweep the disk surface in an irregular fashion, because the density of requests will be higher "in front of" the heads than "behind" them. Figure 3 shows a trace of head movement using SSTF policy for the system simulated here, in which this behav-
Disk Scheduling Policies

The coefficient of variation of seek distance vs queue length for SSTF and SCAN policies.

Figure 4. Coefficient of variation of seek distance vs queue length for SSTF and SCAN policies.

Disk Scheduling Policies

In any event, regardless of the potential utility of the more complex service policies, the possibility of improving disk layout should be considered. This is likely to give better benefits than use of SCAN or SSTF.

Estimation of Benefits

If a system administrator wishes to consider installing one of the more complex policies, estimation of likely benefits is highly desirable. A technique for making a rough estimate of benefits is outlined here.

The first step involved must be the determination of mean queue lengths for disk requests. If measurements of these are not available, they must be estimated. In attempting to do this, it should be noted that in the majority of cases, executing programs will only have one disk request outstanding at a time. This will always be true for page faults, and will generally be true for input and output. Thus, it will be unusual for the total number of disk requests waiting for service to be larger than the degree of multiprogramming allowed by the operating system. The mean number of outstanding requests will presumably be somewhat less than this.

The effect of multiple spindles must also be considered. If requests are uniformly distributed among spindles, and if a separate queue is maintained for each spindle (obviously desirable on performance grounds), then mean queue length will be approximately equal to the number of outstanding requests divided by the number of spindles. This ideal distribution will not in general be achieved, but we can expect the individual queues in a multi-spindle system to be relatively short.

With either estimated or measured numbers of disk requests outstanding, and properties of the disk drives available, a rough quantitative comparison of service times can be made as follows:

a) Compute FIFO service as the time for a seek of MAXCYL/3 cylinders, plus latency of one-half of revolution time, plus read/write time.

b) If measured queue lengths are not available, assume mean queue length is given by the degree of multiprogramming divided by the number of spindles. Degree of multiprogramming is a parameter that can be obtained from the operating system or its description. Use measured or estimated queue lengths to estimate mean seek distance from Figure 2, then add seek, latency and read-write times for the disk under consideration, to estimate mean service time for SCAN or SSTF.

The ratio of these two figures gives an indication of the maximum increase in disk transfer rate that can be expected from switching to the more complex policies. If the system is disk-bound, this same ratio will indicate the maximum increase in overall capacity that can be expected.

The values obtained will be an upper bound on the improvement possible for two main reasons. First, we have assumed disk requests to be uniformly distributed across cylinders and spindles. This will not usually be the case in actual systems. Second, any improvement in service rate will tend to decrease queue lengths. If, therefore, this analysis shows only a small improvement is possible from changing to one of the more complex policies, it may well not be worthwhile making the change.

CONCLUSION

The major options available for improving average transfer rates between disks and main storage have been explored. Of these, the most effective action will generally be adjustment of information layout on the disk, if this is possible.

In the majority of cases, the operating system will keep disk queues short. Thus, scheduling policies such as SSTF and SCAN, which rely on selection from a large number of requests to reduce seek distance are unlikely to yield great improvements in performance. However, these policies will work to "smooth out" heavy disk loads, because of the way service time is reduced at high loads.

REFERENCES


BIографical note

Bruce R. Howarth received a BSc degree from the University of NSW in 1964, and a PhD from the University of California, Berkeley, in 1970, both in Chemical Engineering. Since then, he has held various positions in the computing industry, and is currently a lecturer in the School of Computing Science of the NSW Institute of Technology. His main field of interest is computer performance evaluation.
CORRIGENDUM


The figure labelled Figure 2 of the above paper was, in fact, a reduction of Figure 3 on page 97 of the paper. The correct Figure 2 appears below. We apologise to the author and the readers for this error.

Figure 2. Centralised Database with intelligent terminals.
SPECIAL ISSUE

ON PROGRAMMING LANGUAGES

The Australian Computer Journal will publish a special issue on "Programming Languages" in February, 1983. Research papers, tutorial articles and industry case studies on all aspects of the subject will be included.

Professor J.B. Hext,
ACJ Guest Editor,
School of Mathematics and Physics,
Macquarie University,
North Ryde, NSW 2113

SPECIAL ISSUE ON

SOCIAL CONSEQUENCES OF COMPUTING TECHNOLOGY

The Australian Computer Journal will publish a special issue on "Social Consequences of Computing Technology" in November 1983. Research papers, tutorial articles and industry case studies on all aspects of the subject will be welcome, and both full papers and short communications will be considered.

Prospective authors should write as soon as possible to:

Ashley W. Goldsworthy,
P.O. Box 554,
Fortitude Valley, Qld. 4006

to notify him of their intention to submit material for the issue and provide a brief summary of their intended contribution.

In order to allow adequate time for refereeing and editorial review, complete manuscripts will be required no later than 15 June 1983.

Papers should be prepared in accordance with the guidelines published in the November 1982 issue of the Journal. Authors are requested to pay particular regard to the Journal's preferred style for references.

A new wave of books on operating systems seems recently to have started: this is one of them. Many of the better known books in the field were published in the 1970's in the aftermath of the COSINE Committee's report that advocated that this subject be presented and taught as a set of general principles rather than in terms of one or a few intensive case studies. The origins of this book go back to that time so that it is not surprising that it too has followed the general principles approach and thus approaches many of its predecessors. However the longer gestation period has resulted in a more mature and considered treatment of many topics and allowed a wider set of examples to be drawn from practical systems (IBM OS . . ., MVS, Univac EXEC n, TENEX, KRONOS, Burroughs MCP, and even PRIMOS and UNIX). Further, the author has attempted to build on the strengths he perceived in the earlier books, and this is acknowledged in several places.

This book differs from earlier ones in several ways, but it is no giant step. We are treated to no less than two pages on the infelicities of multiprogramming with fixed memory partitions as inflicted by IBM on its System/360 customers about 15 years ago, but are denied an inattention to problems rather than solving them. The author claims particular emphasis on resource management, but there is no mention of the very successful Cambridge "share scheduling" scheme. The difficulties in providing an adequate definition of the term "process" are not solved here either. Process synchronisation is treated quite thoroughly in terms of P and V operations on semaphores but Hoare's monitors receive only one page, and "path expressions", nothing at all.

The book has some obvious strengths: treatment of the standard topics is balanced, but by no means exhaustive; the index is extensive; and, to one engaged in teaching about operating systems, the sets of exercises at the ends of chapters are timely, useful and very welcome. The examples chosen from actual systems seem to be well done. There are pitfalls here - most of these systems offer a lifetime's study in themselves - but what is the reader to infer about the whole when he encounters the occasional inaccuracy in the description of a familiar system (for example, the number of process control blocks for UNIX files is incorrectly given as seven, not eleven, on p. 181)? There are some occasional phrases that should have seen the editor's blue pencil (e.g. "If a job just waked up has . . ." on p. 56), but generally the writing is clear, and the overall production is up to the publisher's usual high standards.

Had this book appeared three years ago or earlier, it would have been outstandingly successful. But already changes in technology and practice leave it seeming a little dated. On the safe assumption that the book for 1982 will not appear before 1982, the author can safely recommend Dr Calingaert's book as a useful addition to any library and for serious consideration for adoption as a textbook.

J. LIONS
University of New South Wales


This book is a collection of papers from the Proceedings of the IFIP TC 8 Working Conference on Evolutionary Information Systems held in Budapest, Hungary, on 1-3 September, 1981. It is made up of four review papers together with 11 contributed papers and includes some of the conference discussion.

As a summary the papers tend to be aimed at exploring the problems of evolution in information systems rather than providing any concrete or pragmatic tools for their solution. Indeed the notion of evolution is itself found to be ill-defined and the first review paper by Frank Land sets out to define it. Three possible meanings are identified:

- evolution of new systems as characterised by various life cycles,
- evolution of system requirements over time, and
- evolution of technology.

A finer classification is then developed by Land within these three classes.

The other three review papers concentrate on the tools and techniques for systems development. These are primarily survey papers and emphasise European approaches to systems design.

Book Reviews

The book is a collection of eighteen papers on this theme. The authors are experts in their diverse fields; for example, Azriel Rosenfeld from computer vision, Raymond Boyle from cartography and electrical engineering, others from mathematics, computer science and geology.

The style is informal. For example, George Nagy begins with 'Because I have been devoting lost moments to this problem (what is a good data structure for 2D points) for more than five years with great enjoyment but very little progress, I welcomed the opportunity to share it with the distinguished and amiable participants of the Institute...' But such a style is quite suitable for an initial understanding of the subject — all too often the actual methods by which theories are derived are lost in a formal, analytical writing. In summary, for anyone interested in the subject, but who is not yet an expert in all areas (1), the book provides a good, but superficial discussion of the current state of the art.

Donald Fraser


The Year of the Robot is not about robots, but rather about simple feedback mechanisms. The main theme of the book is that human intellectual qualities result simply from the possession of feedback rather than stemming from complex brain circuitry. It is argued that even trivial machines with feedback exhibit the human qualities of "broadmindedness, even temper/fairness, agility/responsiveness, vision/less self-doubt and resoluteness/reliability".

For example, by calling the bandwidth of an amplifier its "breadth of mind", the author deduces that feedback to an amplifier makes it more broadminded because feedback can increase the bandwidth. Similar vague arguments establish the other human qualities of an amplifier.

The author finds further examples of these human traits in complex systems with feedback such as democratic societies. He claims that increasing feedback will improve the broadmindedness of our society, and perhaps he has a point there. But I wouldn't recommend your purchasing the book.

L.M. Goldschlager, University of Sydney


The Year of the Robot is not about robots, but rather about simple feedback mechanisms. The main theme of the book is that human intellectual qualities result simply from the possession of feedback rather than stemming from complex brain circuitry. It is argued that even trivial machines with feedback exhibit the human qualities of "broadmindedness, even temper/fairness, agility/responsiveness, vision/less self-doubt and resoluteness/reliability".

For example, by calling the bandwidth of an amplifier its "breadth of mind", the author deduces that feedback to an amplifier makes it more broadminded because feedback can increase the bandwidth. Similar vague arguments establish the other human qualities of an amplifier.

The author finds further examples of these human traits in complex systems with feedback such as democratic societies. He claims that increasing feedback will improve the broadmindedness of our society, and perhaps he has a point there. But I wouldn't recommend your purchasing the book.

L.M. Goldschlager, University of Sydney


Jewels are objects of beauty. The ones presented in this book are taken from the field of formal language theory and, although their basic themes are familiar to computer scientists, their appeal is mainly mathematical. Here is a small example that occurs as one of the exercises:

Let $L$ be any language whatsoever over the alphabet $\{a\}$. Prove that $L^*$ is regular.

It is simply stated and yet it is surprising — after all, $L^*$ may be non-regular. And therein lies the beauty. But do not ask for practical utility as well: these are rubies, not diamonds.

The author presents his showpieces with great care and precision. He also provides a challenging range of exercises and a good set of up-to-date references. Even so, mathematicians will find that author could have given neater proofs and a more enlightening presentation.

In summary, for anyone interested in the subject, but who is not yet an expert in all areas, the book provides a good, but superficial discussion of the current state of the art.

Donal Fraser


The contents of this book have been reprinted from the journal Artificial Intelligence, Vol. 17, August 1981.

The papers fall loosely into four categories: reviews, the retrieval of shape from a single view, the processing of single line drawings, dynamic processes.

There are two survey papers. Mike Brady (the editor) has a good general historical introduction and review of current trends in the Preface. L.S. Davis and A. Rosenfeld give a survey of cooperating processes in low-level vision. These are iterative techniques for local comparisons in the process of labelling picture parts.

In the second category there is a paper by A.P. Witkin "Recovering Surface Shape and Orientation from Textures", the method is based on the assumption that the contribution of projection to an image texture is systematic. The method is applied to planar surfaces and, by simple extension, to curved surfaces. R.J. Woodham's paper "Analysing Images of Curved Surfaces" introduces the Hessian matrix as a viewer centred representation of surface curvature. This is applied to delineate shape at object boundaries and over sections of smoothly curved surfaces. The technique "photometric stereo" is also introduced which varies the illumination between successive images while holding the single viewing direction constant. K. Ikeuchi and B.C.P. Horn have a paper "Numerical Shape from Shading and Occluding Boundaries" which proposes an iterative technique for computing surface orientations from shading information. H.K. Nishihara discusses the type of information that the vision processes must make explicit in an "Intensity, Visible Surface, and Volumetric Representations". In "Recovery of the Three-Dimensional Shape of an Object from a Single View", T. Kanade identifies some assumptions about the world. These are shown to be necessary in recovering the shape of a box and a chair. In "Psychophysical and Computational Studies towards a Theory of Human Stereopsis" J.W. Mayhew and J.P. Frisby give local and global combination rules which they suggest will prove to be a sufficient basis for a powerful stereo algorithm.

The third category contains papers which are substantially concerned with line drawings. K.A. Stevens' paper "The Visual Interpretation of Surface Contours" introduces an approach towards understanding how surface contours may be used as information about surface shape. In "Interpreting Line Drawings as Three-Dimensional Surfaces" H.G. Barrow and J.M. Tenenbaum propose a model for interpretation based on constraints on local surface orientation along extremal and discontinuity boundaries. Then they give solutions to two problems: recovering the three dimensional configuration of a space curve and interpolating smooth surfaces from orientation constraints along extremal boundaries. J.M. Blum's paper "Inferring Surfaces from Images" discusses the generation of scene descriptions of surfaces from image boundaries. In "Symbolic Reasoning Among 3-D Models and 2-D Images", R.A. Brooks describes the philosophy and implementation of ACRONYM, a model-based vision system. This system tries to interpret images by locating instances of modelled objects. S.W. Draper's paper "The Use of Gradient and Dual Space in Line Drawing Interpretation" is an in-depth review of the application of gradient space and dual space in programs that interpret line drawings.

In the last category B.K.P. Horn and B.G. Schunck's paper "Determining Optical Flow" gives a method for computing optical flow from a sequence of images. In "A Theory for Spatio-Temporal Aperture Illusion in Vision" M. Lindeberg and T. Masekaran propose a theory which shows how elements can be grouped into
The fibres as long-distance transmission media in England a little over two or three years to all facets of optical communications. It is a collection of twenty-six papers which are largely reviews of work published in the recent specialist literature in Japan and elsewhere. As such, it is not intended as an introductory text for the non-specialist. It is, however, a very useful collection for those with some interest in the subject as it brings together well-written review papers on a range of important topics—devices, components, fibres and systems.

The papers on devices discuss topics such as the manufacture and reliability of long-wavelength lasers, effects of optical feedback, spectral measurements, ultra-short pulse generation, avalanche photodiodes, wavelength demultiplexing photodiodes, integration of laser diodes with electronic devices and optical modulators. This section is the largest in the book, reflecting both the amount of work done in Japan and the expertise of the editor.

The papers on components deal with connector technology, micro-optic circuitry, a waveguide modulator, and an optical circulator.

The “fibre” section contains papers on various aspects of the VAD method, submarine cable design, dispersion in single-mode fibres, single-polarisation fibres and backscattering measurements.

Finally, the “systems” section discusses trunk transmission systems, single-mode technology, analog video transmission and optical communications for electric power companies.

Given the aim of the book—to make available a summary of work carried out in Japan—it is helpful to have about 65 per cent of the very large number of references given are to Japanese work. However, if the book is to be regarded as a description of the state of the art in optical communications, one minor criticism is that the selection of topics gives a slightly slanted view of worldwide progress in the past couple of years. For example, there is very little mention of measurement techniques and the papers related to fibre manufacture concentrate exclusively on the vapour-phase axial deposition method (VAD) developed by FIR labs in Japan—there are no papers on advances in fibre manufacture by the modified chemical vapour deposition method (MCVD) which is much more widely used throughout the world.

Finally it should be pointed out that, like many conference proceedings and collections of technical papers, this volume is quite expensive (over $100). Nevertheless, it should find a useful place in research libraries with an interest in the latest developments in optical communications.

R.A. Sammut,
University of New South Wales


The Handbook of Artificial Intelligence is a three volume work, intended to be an encyclopaedia of the major developments in AI over the past 25 years. It contains over 200 articles; each one describes a key concept or an important programming technique or an outstanding AI program. The articles are grouped into chapters which correspond to the major subdivisions of AI.

Volume 1 contains five chapters: The first chapter gives a general introduction to AI, discussing the goals and history of the field. Chapter 2 discusses various search techniques used in computer problem-solving. Chapter 3 is concerned with knowledge representations models used in current programs. Chapters 4 and 5 describe AI research in understanding natural language, both written and spoken.

The Handbook was conceived as a self-help encyclopedia to which the aspiring practitioner could turn for explanations of fundamental ideas, descriptions of methods, and discussions of well-known programs as case studies. This is a particularly worthwhile goal since most AI research, especially in the United States, is only published within a limited group of people. The book serves as a good introduction to their efforts and provides references for further reading.

Of the articles reviewed in Volume 1 are generally well written and self-contained. However, they were not intended to be “text book” descriptions. There are no exercises and although no previous knowledge is assumed, the reader should be familiar with computers. The size (three volumes) and price also prohibit the Handbook from being used as a textbook. However, it is an excellent reference book and would be a worthwhile addition to a computer science library. Lecturers and graduate students in AI should also find it very useful. The bibliographies provided with each article are particularly good.


159
The only criticism I have of the Handbook is that it reflects the closed nature of the group which edited it. The major effort was due to the Stanford Artificial Intelligence Laboratory, one of the "big four" institutions in AI. Little of the work done in AI outside Stanford, SRI International, Carnegie-Mellon and MIT is reported. Had this been done the book probably would have grown to several more volumes, so perhaps limiting the scope was not such a bad idea! 

C. Sammut, University of New South Wales


Discovering Computers has been written to provide material for a computer literacy course for high school students. The book consists of two independent sections. The first six chapters are concerned with "the computer program" and Pascal programs; the last two chapters are devoted to procedures and functions. The last four chapters present short articles and readings on the computer industry, uses of computers, social implications and the future. The book uses an open layout with many diagrams, photographs and cartoons. The first section includes chapter summaries. Exercises, clearly marked, are interspersed with the text. 

The authors claim that the book provides a foundation for further study of computer science. Unfortunately, the limited coverage of the language is more likely to instil programming habits (in particular, the unimaginative use of data structures) which would have to be undone should the student attempt formal study of the highly competitive field. Introductory Pascal texts, Simple Pascal is unlikely to be noticed.


Most text books on numerical methods present algorithms in the form of FORTRAN subprograms or flowcharts. Pascal Programs is not primarily intended as a text, but the fifty-odd algorithms presented are complemented by concise discussion of the theory behind them, as well as consideration of the problems which may arise due to round-off and precision limitations. 


The text abounds with examples, though too many are of simple type and not met until the seventh of ten chapters). The reader, a restricted subset of Pascal is presented in a rather curious order (assuming no prior knowledge of computers on the part of the language has widened. The three books here reviewed exemplify this diversity.

The authors claim that the book provides a foundation for further study of computer science. Unfortunately, the limited coverage of the language is more likely to instil programming habits (in particular, the unimaginative use of data structures) which would have to be undone should the student attempt formal study of the highly competitive field. Introductory Pascal texts, Simple Pascal is unlikely to be noticed.

Most text books on numerical methods present algorithms in the form of FORTRAN subprograms or flowcharts. Pascal Programs is not primarily intended as a text, but the fifty-odd algorithms presented are complemented by concise discussion of the theory behind them, as well as consideration of the problems which may arise due to round-off and precision limitations. Topics include statistical analysis, matrix operations and simultaneous equations, curve fitting, sorting, and numerical integration. A chapter on evaluating the floating-point accuracy of a Pascal implementation and an appendix on the syntax of Pascal are also included.

The programs were developed using a CP/M implementation of the language, but use few non-standard features. A compromise is evident between the utilisation of some aspects of Pascal and the ease with which the programs could be translated to a lower-level language. Data structure more elaborate than those avoided, for example, and a non-reursive Quicksort is included along with the recursive algorithm. The absence of algorithms operating on complex quantities would appear to be a deficiency in a volume concerned with engineering applications. It can be argued that Pascal, without a complex data type or the ability to declare functions returning structured types, is not the best choice in this area.

Pascal Programs will be of some benefit as quick reference to numerical algorithms for non-specialists in numerical methods.

The Pascal Handbook would be a much slimmer volume were it not for the proliferation of non-standard versions of the language. The book documents both standard and non-standard features of four dialects of Pascal — Hewlett-Packard's Pascal 1000 for the HP1000, OMSI Pascal-1 (PDP-11), Pascal-Z (280 microcomputers), and UCSD Pascal — as well as Jensen and Wirth's original definition implemented on the CDC 6000 series machines. The proposed ISO standard is used as the reference.

Each of the 182 entries refers to a reserved word or operator, a predefined identifier, or a concept (such as "scope" or "statement"). Subsections within each entry detail syntax (using syntax diagrams where appropriate), description, implementation-dependent features, and examples. Each page is clearly marked with the entry name, making consultation simple.

Although the bulk of the entries refer to non-standard procedures and functions, the book is also a useful reference for standard Pascal. Its ease of use is a far cry from the inconvenient index lookup necessary with many reference manuals.

In summary, The Pascal Handbook is a boon to users concerned with program portability; Pascal Programs is a worthwhile investment for occasional dabblers in the field of numerical methods; while Simple Pascal is sadly outdated.

Anne McDougall, Monash University


The solution that Baber sees to collapsing programs involves more, and more serious, computing science education. He presents a test for programming practitioners (10 pages of questions and 60 of answers) that would rate as a good stiff standard for entry to Fellowship of the Australian Computer Society. While I confess that my disagreement with the author stems in part from embarrassment at not performing all that well on the test, it's not hard to justify my feelings.

Although standards certainly need improving, Baber has failed to notice the popularisation of computing and the emergence of two classes of practitioner: the computer-literature coder and the heavyweight professional. Much as he might decry the construction of buildings or programs by inadequately trained people, he can't single-handedly overturn the processes of technological development. The fact is that all new technologies go through an early phase in which progress is made by mysterious experts, but that subsequently ways are found to overlay the intricacies and enable quite moderately educated people to harness the new power. One reason for his error is that (despite his pleas for more careful and precise methodology) he commits a methodological blunder by confusing the evolution of scientific disciplines with that of technologies (p. 73).

Strangely I recommend this book more to practitioners than to computing scientists. The more academically inclined are already converted, and may be already so respectable that they'll cavil at the looseness of an anecdotal and scenario-based argument. But the practising professional will find the ideas accessible and stimulating. There are a few bonuses too: an academic sets out to specify generalised printer-handling (p. 160) and numeric-field validation (p. 164); and I doubt that I'm the last remaining practitioner to appreciate clarification of recursion (p. 144) and automata (p. 138).

Roger Clarke,
Xamax Consultancy Pty Ltd, Sydney
Information for Authors

The Australian Computer Journal invites original contributions relating to the design, understanding and application of digital computers. The range of material invited includes research papers, tutorial and review papers, industry case studies, short communications and letters to the editor.

Compliance with generally accepted practices for the style and organisation of scientific papers is expected. Each paper must be in English and must be certified by the author(s) that it is his/her or their own original work, that it has not been copyrighted, published or submitted for publication elsewhere and that, if the work described has been sponsored, the paper has been cleared for publication by the sponsoring organisation. The author(s) are asked to agree, if the paper is accepted for publication, to assign copyright in the paper to the Australian Computer Society Incorporated in accordance with normal practice.

All papers submitted will be evaluated for accuracy, originality and relevance by at least two referees or members of the editorial committee, and revisions may be requested. Every effort is made to notify authors promptly whether their papers will be accepted for publication and the expected date thereof.

Four to six printed pages is the preferred size for full papers, including all diagrams and tables. (One printed page equals about 950 words of text.) Longer papers may be accepted if space permits. Short communications (one or two printed pages) describing, for example, novel applications, short algorithms or supporting data, such detail should be included in the subject matter should be prepared in the same style as for the main paper, which should characterise the contents of the paper in as short a span as possible, should appear at the head of the title page. (The title, or a shortened version of it, plus a page number, should appear on the top line of each subsequent sheet of the text.)

Below the title of the paper should appear the author's name (initials and surname only — no titles or degrees should be shown). If there is more than one author, the principal author should be listed first. For each author, a sufficient postal address (possibly including the name of an employer), suitable for printing as a footnote, should be supplied.

The abstract on the title page should be informative and summarise the paper as concisely as possible in about 100 words. Since the abstract may be republished separately, it should contain no footnotes or references to other published works. Finally, the title page should include standard content indicators: lists of Key Words and Phrases, and "Computing Reviews" categories. A substantial revision of these categories was published in Computing Reviews, 23, No. 1, January 1982. The new classification is to be used by this Journal from the beginning of 1983.

A list of the new categories is printed below.

The text should be clearly written and follow the English conventions for spelling and punctuation. Text prepared using a word processor should not be hyphenated or right-justified. Subheadings and paragraphs should be clearly shown. Major and minor sections may be numbered at the author's discretion.

Footnotes should not be used unless absolutely necessary (they are not used for reference citations). If a footnote is unavoidable, then it should be included in the text of the manuscript immediately below the line to which it refers and should be separated from the rest of the text by full width lines immediately above and below the note.

Tables should be numbered consecutively. Each table should have a separate heading at the top. Column headings should be brief and clearly shown. Preferably tables will be constructed to fit neatly into a single column (8.3 cm x 22.9 cm) or exceptionally, across two columns (width 17.8 cm). Large tables should be typed on separate sheets from the text. More than one table may be typed on a single sheet, but tables should not be split between sheets. The proper location for each table should be noted in the margin of the text.

Camera-ready copies for each figure, diagram or flowchart should be supplied. These should be drawn in Indian ink on stiff white paper or Bristol board. Drawings should be about twice the expected finished size (single column width is 8.3 cm, and double, 17.8 cm). Legends and labels should be clearly drawn, positioned suitably and large enough to remain legible after reduction. Each drawing should be on a separate sheet, packed flat and have on the reverse side, the author's name and the figure number and caption. For photographs, please submit glossy prints. Avoid using clips or writing heavily on the reverse side when recording author name, figure number and caption. Full-colour photographs may be submitted in appropriate cases (prior consultation with the Editor is advised). Programs that illustrate the text or that convey important algorithms may be published. It will usually simplify matters considerably if camera-ready listings of such programs can be submitted.

Unusual mathematical symbols, subscripts and mathematical formulae can be difficult to set in type. Please choose symbols carefully, and avoid excessive use of subscripts, plan formulae to fit within the standard column width and use alternative forms that are easier to set in type, e.g. $\epsilon x$ rather than $\epsilon^{-x}$. Avoid using the letters O and I in situations where they may be confused with the digits 0 and 1. If papers require the presentation of extensive mathematical treatments or supporting data, such detail should be included in one of the appendices at the end of the paper. Appendices should be numbered if there are more than one, and have suitable headings. Subject matter should be prepared in the same style as for the main text.

Careful adherence to the Journal's style for citation and quotation of references is required. This style is the so-called Harvard style and has several advantages. Reference citations should be made as in the following examples: "It was shown by Curtis and Osborne (1966) that . . ." or "It has been shown elsewhere (Cohen and Nguyen-Dinh, 1966) that . . ." or "It may be shown (e.g. see Knuth, 1973a) that . . .". Note that the citation is composed of the author's surname and the year of publication. Where this is ambiguous, different works by the same author(s) in the same year are distinguished by adding a single lower case letter to the year. Only one level of parenthetical is used. The author's name appears outside the parentheses if the reference is direct, or inside if the reference is indirect. In the latter case, the name and year are separated by a comma. If there are two authors, both names are used as part of the reference. If there are three or more authors, then all author surnames should appear in the initial citation but subsequent citations may be abbreviated by replacing the second and later author names by the phrase "et al.". Thus, for example, a second citation of the book by "Aho, Hopcroft and Ullman (1974)" may be made as "Aho et al. (1974)". In this style the citations are
not intrusive, they are often enough for the reader to recognise the work without further effort and they may be added or deleted easily during drafting without disturbing any pre-arranged numbering scheme.

A section at the end of the paper headed "References" should quote full details for all references cited in the text (and no others). These must be sufficient to allow an ordinary reader to locate the reference without undue difficulty, and they should be arranged so that the textual citations can be located readily, i.e. the reference list should be ordered alphabetically by author name and then year. To this end, the publication year (together with an alphabetical letter suffix) is listed immediately after the author's name and before the title of the work and the remaining details. The following items illustrate the style of a reference list:


Authors are responsible for ensuring the accuracy of all details of all references quoted. Guidelines to be observed during the preparation of the reference list are: author names are capitalised (for prominence) with initials following the surname; the year of publication is enclosed in parentheses and followed by a colon; for references to books and monographs, the year of publication is followed by the title (underlined), a page reference (if appropriate), the name of the publisher and the place of publication; for references to journal articles, the year of publication is followed by the title (underlined), the name of the journal (underlined), the volume number, the issue within the volume, and the page number range (introduced by the symbol "pp."); references to unpublished works should be avoided as far as possible; private communications, if they must be quoted at all, should be designated as such within the text, e.g. "F.G. Smith (private communication)", and not included in the reference list.

Journal titles appearing in citations should be abbreviated in accordance with accepted standards. (See ISO 4, Documentation-International code for the abbreviation of titles and periodicals and ISO 833, Documentation - International list of periodical title word abbreviations. A relevant summary of these can be found in Computing Reviews, 22, pp. 418-427, September, 1981.) Titles consisting of a single word are never abbreviated, and a leading The is usually omitted. For longer titles, the following abbreviations should be used: ACM (Association for Computing Machinery), Abstr. (Abstracts), Aust. (Australian), Bull. (Bulletin), Comput. (Computers, Computing), Des. (Design), Eng. (Engineering), J. (Journal), Math. (Mathematics), Proc. (Proceedings), Rev. (Review), Sci. (Science(s)), Scientist), Softw. (Software), Surv. (Surveys), Syst. (Systems), Trans. (Transactions). Thus, sample abbreviations are Austral. Comput. J., Datamation and IEEE Trans. Softw. Eng.

Biographical Note. Each author of a paper is asked to supply a short biographical note (about one hundred words) for publication with the paper.

Submissions. All papers and correspondence should be addressed to The Editor, Australian Computer Journal, Department of Computer Science, University of New South Wales, Kensington, NSW 2033, Australia.

Proofs. Papers that are accepted for publication will be typeset and shortly before publication, a proof copy will be sent for proof-reading and correction to the author, or to the first author if there is more than one, or to his nominee. (Overseas authors are particularly requested to nominate a representative in Australia who can perform this task.) The symbols used for correcting proofs should be those defined in the Style Manual published by the Commonwealth Government Printer, Canberra, pp. 76-79. Authors should note that proof reading is intended to correct errors introduced by the editing and typesetting processes. Extensive changes or additions to an article introduced during proof-reading may delay publication and be charged to the author. Corrected proofs must be returned promptly or publication cannot be guaranteed.

Reprints. Fifty reprints of articles will be supplied free of charge. Information for ordering additional reprints will be provided by the printer with the proof copy.

NEW COMPUTING REVIEWS CLASSIFICATION SCHEME
(Top two levels)

A. GENERAL LITERATURE
A.0 GENERAL
A.1 INTRODUCTORY AND SURVEY
A.2 REFERENCE (e.g., dictionaries, encyclopaedias, glossaries)
A.m MISCELLANEOUS

B. HARDWARE
B.0 GENERAL
B.1 CONTROL STRUCTURES AND MICROPROGRAMMING (D.3.2)
B.2 ARITHMETIC AND LOGIC STRUCTURES
B.3 MEMORY STRUCTURES
B.4 INPUT/OUTPUT AND DATA COMMUNICATIONS
B.5 REGISTER-TRANSFER-LEVEL IMPLEMENTATION
B.6 LOGIC DESIGN
B.7 INTEGRATED CIRCUITS
B.m MISCELLANEOUS

C. COMPUTER SYSTEMS ORGANISATION
C.0 GENERAL
C.1 PROCESSOR ARCHITECTURES
C.2 COMPUTER-COMMUNICATION NETWORKS
C.3 SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS
C.4 PERFORMANCE OF SYSTEMS
C.m MISCELLANEOUS

D. SOFTWARE
D.0 GENERAL
D.1 PROGRAMMING TECHNIQUES (E)
D.2 SOFTWARE ENGINEERING (K.6.3)
D.3 PROGRAMMING LANGUAGES
D.4 OPERATING SYSTEMS (C)
D.m MISCELLANEOUS

E. DATA
E.0 GENERAL
E.1 DATA STRUCTURES
E.2 DATA STORAGE REPRESENTATIONS
E.3 DATA CRYPTOGRAPHY
E.4 CODING AND INFORMATION THEORY (H.1.1)
E.m MISCELLANEOUS

F. THEORY OF COMPUTATION
F.0 GENERAL
F.1 COMPUTATION BY ABSTRACT DEVICES
F.2 ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY (B.6-7, F.1.3)
F.3 LOGICS AND MEANINGS OF PROGRAMS
F.4 MATHEMATICAL LOGIC AND FORMAL LANGUAGES
F.m MISCELLANEOUS
Information for Authors

G. MATHEMATICS OF COMPUTING
  G.0 GENERAL
  G.1 NUMERICAL ANALYSIS
  G.2 DISCRETE MATHEMATICS
  G.3 PROBABILITY AND STATISTICS
  G.4 MATHEMATICAL SOFTWARE
  G.m MISCELLANEOUS

H. INFORMATION SYSTEMS
  H.0 GENERAL
  H.1 MODELS AND PRINCIPLES
  H.2 DATABASE MANAGEMENT
  H.3 INFORMATION STORAGE AND RETRIEVAL
  H.4 INFORMATION SYSTEMS APPLICATIONS
  H.m MISCELLANEOUS

I. COMPUTING METHODOLOGIES
  I.0 GENERAL
  I.1 ALGEBRAIC MANIPULATION
  I.2 ARTIFICIAL INTELLIGENCE
  I.3 COMPUTER GRAPHICS
  I.4 IMAGE PROCESSING
  I.5 PATTERN RECOGNITION
  I.6 SIMULATION AND MODELING (G.3)
  I.7 TEXT PROCESSING (H.4)
  I.m MISCELLANEOUS

J. COMPUTER APPLICATIONS
  J.0 GENERAL
  J.1 ADMINISTRATIVE DATA PROCESSING
  J.2 PHYSICAL SCIENCES AND ENGINEERING
  J.3 LIFE AND MEDICAL SCIENCES
  J.4 SOCIAL AND BEHAVIOURAL SCIENCES
  J.5 ARTS AND HUMANITIES
  J.6 COMPUTER-AIDED ENGINEERING
  J.7 COMPUTERS IN OTHER SYSTEMS
  J.m MISCELLANEOUS

K. COMPUTING MILIEUX
  K.0 GENERAL
  K.1 THE COMPUTER INDUSTRY
  K.2 HISTORY OF COMPUTING
  K.3 COMPUTERS AND EDUCATION
  K.4 COMPUTERS AND SOCIETY
  K.5 LEGAL ASPECTS OF COMPUTING
  K.6 MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS
  K.7 THE COMPUTING PROFESSION
  K.8 PERSONAL COMPUTING
  K.m MISCELLANEOUS
Because Miller Graphic Controls is part of a world-wide group of companies, we have access to the very latest technology. And we can supply any plotter media you need.

From Apple to Zeta.
From Mylar film to tracing paper, and 100% Rag Bond.
In every size you could need.
And at manufacturer prices.

We also have pens. And Inks.
Blank Rolls & Custom Printed.
All from the Single Source.

What's more, our range is always expanding. Miller Graphic Controls is The Single Source in Australia for all industrial, technical and medical graphic media including Facsimile Paper and Electrostatic Paper.

Miller Graphic Controls Pty. Ltd.
VIC. P.O. Box 199, Clifton Hill, 3068.
Phone (03) 481 2022. Telex: AA 36386.
NSW. P.O. Box 162, Rydalmere, 2116.
Phone (02) 684 4522. Telex: AA 25827.

Ofices and agents throughout Australia, New Zealand and South-East Asia.
laboratory may be obtained by contacting Mr Dennis Millar, Director, CCAE Computer Centre (062) 52-2010.

LOCKABLE FLOPPY TRAYS

Twinlock has introduced a range of lockable plastic trays for the storage of both 5¼ and 8” diskettes, or floppy disks.

There are four trays in the range, with 45 and 90 diskette capacity for both sizes of floppy disk. Produced in high impact ABS Plastic with a smoked perspex lid, the trays are lockable and are supplied complete with divider plates and self-adhesive title strips for indexing.

Prices range from $34.72 for a storage tray for 45 5¼” floppy disks, and from $46.96 for 45 8” floppy disk capacity tray.

For more information please contact Mr T.G. Johnson on (03) 584-4000.

JOINT GOVERNMENT/INDUSTRY TRAINING PROGRAM PRODUCES FIRST COMPUTER GRADUATES

A computer programmer training program strongly backed by Commonwealth and NSW Governments in association with the Australian Computer Society has been hailed as an outstanding success.

This comment was made by the NSW Director of the Commonwealth Department of Employment and Industrial Relations, Mr Ralph Clark, at a ceremony this week to mark the graduation of the initial group of 30 trainees.

Since the computer industry training program was launched two years ago over 120 young people have been inducted, receiving both formal instruction as well as on-the-job experience.

The program was initiated by the Australian Computer Society through the NSW Department of Industrial Relations with training being conducted by the NSW Department of Technical and Further Education. Funding has been provided by the Commonwealth under the Skills in Demand Program administered by the Department of Employment and Industrial Relations.

Mr Clark said the success of the program was largely attributable to the ability of the society, and the two levels of government, to work closely together to ensure that an adequate level of skilled labour is maintained.

“The Commonwealth Government has once again indicated its commitment to this type of training approach in the 1982 budget by allocating $7 million to skills in demand training for the 1982/83 financial year,” Mr Clark said.

“This represents almost treble the previous financial year’s expenditure.”

“I think the 1982 allocation of $250,000 — the same amount provided over the previous two-year period — shows the government’s concern that assistance be given to solve this industry’s skills problems and to help up to 100 young people find interesting and challenging careers during 1982.”

Mr Clark said he was confident that companies within the computer industry would recognise the long-term importance of the program and continue to support this training initiative.

ICCC-84 — SYDNEY, AUSTRALIA

The organisers of the 7th International Conference on Computer Communication (ICCC-84) — to be held in Sydney, New South Wales, Australia, 30 October-2 November, 1984 — announce the official theme now adopted: ‘The New World of the Information Society’.

For the over 800 delegates expected from all over the world and from Australia, the program will embrace three main topics: social and human factors, service and business aspects, and systems technology.

The Invitation leaflet gives details; and the Call for Papers leaflet outlines the specific requirements to be met for would-be contributors’ papers.

Organisers of ICCC-84 are The Overseas Telecommunications Commission (Australia) and Telecom Australia. Conference Governor is Dr. Carl Hammer, Vice-President, International Council for Computer Communications; Conference Chairman is Mr J.H. Curtis CB, previously Managing Director of Telecom Australia, and a Commissioner of OTC; and Conference Director is Mr J.R.R. Cook, Assistant General Manager (Personnel and Administration), OTC.

The Conference organisation consists of a distinguished Patrons Committee; and a Conference Organising Committee, chaired by the Conference Chairman. There are functional committees actively at work covering the program, finance and registrations, local arrangements and publicity and publications. The organisation also embraces Regional Liaison Committees already in being in central Europe, France, Japan, New Zealand, the United Kingdom and USA.

Copies of the Invitation and Call for Papers leaflets referred to above can be obtained from The Conference Secretary, ICCC-84, GPO Box 2367, Sydney, NSW, 2001 Australia.

Pinpoint power problems . . .

fast!

Portable Model 3600 Powerline Disturbance Monitor can track three AC phases and one DC Channel simultaneously, for fluctuations likely to cause problems in sensitive equipment. Push button keyboard for programming parameters . . LED display . . Large thermal print out tape . . RS232 interface . . all mounted in shock protected briefcase.
Now a Hewlett-Packard calculator specifically designed for programmers.

For a long while, ironically, computer analysts and programmers had to use the computer they were writing programs for, to write the programs themselves.

It was not always possible, not always convenient, and never very cost efficient.

Now, fortunately, Hewlett-Packard has come to the rescue, with the HP-16C.

Because the HP-16C is programmable, it can handle a wide variety of computer science problems, including format conversions, bit extraction and simulation of selected microprocessor instructions.

But its specialised design becomes apparent when it's switched to 'integer' mode for bit calculations.

Here, numbers can easily be entered in and converted to hexadecimal, decimal, octal or binary bases.

And all this in a handheld calculator. With 18 different bit-manipulation functions, 4 boolean operators and floating point calculator mode to boot.

The new HP-16C from Hewlett-Packard. It's a programmable specifically designed for programmers.

To see the new HP-16C calculator, call Hewlett-Packard Personal Computation Group for dealer locations.

Sydney: 887 1611; Melbourne: 890 6351; Adelaide: 272 5911; Perth: 383 2188; Brisbane: 369 8022; Canberra: 804 244.

HEWLETT PACKARD
The Australian Computer Journal is an official publication of the Australian Computer Society Incorporated.

OFFICE BEARERS: President: A.W. Goldsworthy, Vice-President: B.M. Hannelly, M.G. Jackson; Immediate Past President: G.E. Waste; National Treasurer: R.A. Hamilton; Chief Executive Officer: R. Shelley, PO Box N26, Grosvenor Street, Sydney, 2000, telephone (02) 267-5725.


SUBSCRIPTIONS: The annual subscription is $20.00. All subscriptions to the Journal are payable in advance and should be sent (in Australian currency) to the Australian Computer Society Inc., PO Box N26, Grosvenor Street, Sydney, 2000. A subscription form may be found below.

PRICE TO NON-MEMBERS: There are now four issues per annum. The price of individual copies of back issues still available is $2.00. Some are already out of print. Issues for the current year are available at $5.00 each. All of these may be obtained from the National Secretariat, PO Box N26, Grosvenor Street, Sydney, 2000. No trade discounts are given, and agents should recover their own handling charges.

MEMBERS: The current issue of the Journal is supplied to personal members and to Corresponding Institutions. A member joining part-way through a calendar year is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. Back numbers are supplied to members while supplies last, for a charge of $2.00 per copy. To ensure receipt of all issues, members should advise the Branch Honorary Secretary concerned, or the National Secretariat, promptly, of any change of address.

MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries. Canberra: PO Box 446, Canberra City, ACT, 2601. NSW: Science House, 35-43 Clarence St, Sydney, NSW, 2000. Qld: Box 1484, GPO, Brisbane, Qld, 4001. SA: Box 2423, GPO, Adelaide, SA, 5001. WA: Box F320, GPO, Perth, WA, 6001. Vic: PO Box 98, East Melbourne, Vic, 3002. Tas: PO Box 216, Sandy Bay, Tas, 7005.

CONTRIBUTIONS: All material for publication should be sent to: Associate Professor J. Lions, Editor, Australian Computer Journal, Department of Computer Science, University of New South Wales, Kensington, NSW 2033. Prospective authors may wish to consult manuscript preparation guidelines published in the November 1982 issue. The paragraphs below briefly summarise the essential details.

Types of Material: Four regular categories of material are published: Papers, Short Communications, Letters to the Editor and Book Reviews. Generally speaking, a paper will discuss significant new results of computing research and development, or provide a comprehensive summary of existing computing knowledge with the aim of broadening the outlook of Journal readers, or describe important computing experience or insight. Short Communications are concise discussions of computing research or application. A letter to the Editor will briefly comment on material previously appearing in the Journal or discuss a computing topic of current interest. Descriptions of new software packages are also published to facilitate free distribution.

Refereeing: Papers and Short Communications are accepted if recommended by anonymous referees. Letters are published at the discretion of the Editor, and Book Reviews are written at the Editor’s invitation upon receipt of review copies of published books. All accepted contributions may be subject to minor modifications to ensure uniformity of style. Referees may suggest major revisions to be performed by the author.

Proofs and Reprints: Page proofs of Papers and Short Communications are sent to the authors for correction prior to publication. Fifty copies of reprints will be supplied to authors without charge. Reprints of individual papers may be purchased from Associated Business Publications, PO Box 440, Broadway, NSW, 2007. Microfilm reprints are available from University Microfilms International, Ann Arbor/London.

Format: Papers, Short Communications and Book Reviews should be typewritten in double spacing on A4 size paper, with 2.5cm margins on all four sides. The original, plus two clear bond-paper copies, should be submitted. References should be cited in standard Journal form, and generally diagrams should be ink-drawn on tracing paper or board with stencil or Letraset lettering. Papers and Short Communications should have a brief Abstract, Key word list and CR categories on the leading page, with authors’ affiliations as a footnote. The authors of an accepted paper will be asked to supply a brief biographical note for publication with the paper.

This Journal is Abstracted or Reviewed by the following services:

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Bibliography and Subject Index of Current Computing Literature.</td>
</tr>
<tr>
<td>AMS</td>
<td>Mathematical Reviews.</td>
</tr>
<tr>
<td>INSPEC</td>
<td>Computer and Control Abstracts.</td>
</tr>
<tr>
<td>INSPEC</td>
<td>Electrical and Electronic Abstracts.</td>
</tr>
<tr>
<td>ISI</td>
<td>Current Contents/CompuMath</td>
</tr>
<tr>
<td>ISI</td>
<td>CompuMath Citation Index.</td>
</tr>
<tr>
<td>SPRINGER-VERLAG</td>
<td>Zentralblatt fur Mathematik und ihre Grenzgebiete.</td>
</tr>
</tbody>
</table>

Copyright © 1982. Australian Computer Society Inc.

Production Management: Associated Business Publications, Room 104, 3 Small Street, Ultimo, NSW 2007 (PO Box 440, Broadway, NSW 2007). Tel: 212-2780, 212-2780.

All advertising enquiries should be referred to the above address.

Printed by: Publicity Press (NSW) Pty Ltd, 66 O’Riordan Street, Alexandria, NSW 2015.