

AAPI

IN PRACTICE

**WHAT YOU NEED
TO KNOW TO
INSTALL AND USE**

**SUCCESSFUL
APL SYSTEMS
AND MAJOR
APPLICATIONS**

EDITED BY

ALLEN J. ROSE
BARBARA A. SCHICK
AND
STAFF OF STSC, INC.

In the early part of the computer era, the cost of the computer itself overshadowed all other costs associated with computing. Over the years, however, engineering advances (in particular the integrated circuit) have steadily chipped away at the cost of computing hardware, and it is no longer the dominant component of computing costs. Labor costs, particularly systems and programming, are the largest component of computing activity today. This situation has encouraged many researchers to augment the features and usage conventions of traditional computing languages to squeeze more efficiency from them.

APL IN PRACTICE deals neither with new approaches nor with facelifts of proven concepts. Rather, it reports the state of APL, a method of interactive computing introduced in the 1960s. APL is one of the most concise, consistent, and powerful programming languages ever devised. The simplification and efficiency offered by its rich and powerful handling of work involving multiple data structures have saved a great deal of time and money for the organizations that have used it. Its proven benefits, however, have been largely obscured in the literature by the many incremental improvements made to more traditional languages.

This book offers some compelling arguments for considering APL for business computing. It also serves as a handbook for those who know APL's advantages, but need help in preparing to use APL for a wide range of applications. **APL IN PRACTICE** offers the collective experience of over fifty proven APL practitioners from virtually every area of practical APL usage.

Three goals dominated the selection and editing of topics in the book:

- To provide general management with sufficient knowledge of APL to cut through the mystique that surrounds the data processing profession.
- To aid data processing managers and working professionals in bridging the gap between their familiar turf and new fields that are easily mastered with APL.
- To broaden the horizons of convinced APL users so that they can better relate to the real problems of general management and data processors.

(Continued on back flap)

(Continued from front flap)

To these ends, the editors have approached the full range of computer-related tasks required of modern business. Included here are innovative applications of APL to applications as diverse as financial planning, marketing management, general ledger, budgeting, manufacturing, and electronic mail. Attention is also given to the special interests of systems management and professional programmers—formatting and reporting, writing maintainable programs, and managing outside computer services.

While intended to encourage adoption of APL solutions to business problems, the depth and diversity of these contributions will also give general management and systems management a clear picture of both the possibilities and limitations of APL for solving their data processing requirements.

ABOUT THE EDITORS

ALLEN J. ROSE is Vice President & Technical Director for STSC, Inc. Prior to assuming his current responsibilities, Mr. Rose was the APL Program Administrator for IBM Corporation. He has also worked as an industrial statistician for Procter and Gamble. Mr. Rose received his B.A. in Psychology from Duke University. He is the co-author of *APL: An Interactive Approach* (Wiley 1976).

BARBARA A. SHICK is currently Manager of Publications for STSC, Inc. Before joining STSC, she was a writer/editor for the Kiplinger Washington Editors. Ms. Schick has a B.A. in English Literature from The Catholic University of America and is currently pursuing her M.B.A. at the University of Maryland.

ALSO OF INTEREST...

A PROGRAMMING LANGUAGE

Kenneth E. Iverson

The book that introduced APL—illustrates the universality and versatility of this concise, powerful programming language, its capacity to compass a complex and detailed topic in a short space, and its utility in theoretical work. Iverson, who invented APL while at Harvard, describes the language, microprogramming, representation of variables, search techniques, metaprograms, sorting, and the logical calculus.

1962

APL

**An Interactive Approach
2nd Ed., Revised Reprinting**

Leonard Gilman and Allen J. Rose

"...this textbook is still top of the line—the finest APL textbook around."

—*Data Processing Digest*

"If you can have only one APL book, there is none better."

—*Computing Reviews*

This critically acclaimed guide to APL features a "hands-on" approach that enables readers to run programs after only two chapters. Covers the language, programming documentation techniques, programming styles, and function definition—using new problems from business and managements—and highlights recent developments in STSC's APL*PLUS System and IBM's APLSV.
1976

**MATERIALS MANAGEMENT SYSTEMS:
A Modular Library**

Robert Goodell Brown

Covers theoretical aspects of system design considerations in forecasting, inventory management, production planning and scheduling, shop floor control, and physical distribution. The book is rapidly gaining acceptance as the leading textbook on the design of computer-based systems for manufacturing and materials management applications.

1977

The letters 'APL' are rendered in a large, bold, three-dimensional font. Each letter is a thick, blocky shape with a slight shadow underneath, giving it a 3D appearance. The letters are white or light gray against the dark background.

WILEY-INTERSCIENCE

a division of JOHN WILEY & SONS
605 Third Avenue, New York, N.Y. 10158
New York ■ Chichester ■ Brisbane ■ Toronto

ISBN 0 471-08275-9

ROSE / SCHICK

PREP

IN PRACTICE

WILEY
INTERSCIENCE



ROSC
SCHICK

THE
JOURNAL
OF
THE
ROYAL
SOCIETY



WILEY-INTERSCIENCE

APL
in Practice

edited by

Allen J. Rose

Barbara A. Schick



APL IN PRACTICE

The Practical APL Conference
Washington, D.C.
9-11 April 1980

STSC, Inc.

APL IN PRACTICE

**What You Need to Know
To Install and Use
Successful APL Systems
And Major Applications**

Edited by
ALLEN J. ROSE
BARBARA A. SCHICK
and
Staff from STSC, Inc.

JOHN WILEY & SONS, INC.
New York, Chichester, Brisbane, Toronto

Copyright © 1980 by STSC, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging in Publication Data

Main entry under title:

APL in practice.

Includes index.

1. APL (Computer program language)
2. Interactive computer systems, I. Rose, Allen J. II. Schick, Barbara A., 1951-
QA76.73.A27A18 001.64'2 80-5351
ISBN 0-471-08275-9

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

APL*PLUS is a service mark and trademark of STSC, Inc., registered in the United States Patent and Trademark Office.

Foreword

In the early part of the computer era, the cost of the computer itself overshadowed all other costs associated with computing. Over the years, however, engineering advances (in particular the integrated circuit) have steadily chipped away at the cost of computing hardware, and it is no longer the dominant component of computing costs.

Labor costs, particularly systems analysis and programming, are the largest component of computing activity today. This situation has encouraged many researchers to augment the features and usage conventions of traditional computing languages to squeeze more efficiency from them.

This book deals neither with new approaches nor with facelifts of proven concepts. Rather, it reports the state of the technology of *APL*, a method of interactive computing introduced in the late 1960s. *APL* is one of the most concise, consistent, and powerful programming languages ever devised. The simplification and efficiency offered by its rich and powerful handling of work involving multiple data structures have saved a great deal of time and money for the organizations that have used it. Its proven benefits, however, have been largely obscured in the literature by the many incremental improvements made to more traditional programming languages.

This book offers some compelling arguments for considering *APL* for business computing. It also serves as a handbook for those who know *APL*'s advantages, but need help in preparing to use *APL* for a wide range of applications. As with any expanding technology, it would be presumptuous for us to claim we have all the answers, although there are some of us who have worked toward making *APL* applicable to the full range of computer-related tasks required of modern business. What we do offer is the collective experience of over fifty proven *APL* practitioners—approximately 250 people-years' worth—drawn from virtually every area of practical *APL* usage.

The papers in this book were prepared as background for presentations given at "The Practical *APL* Conference", which was sponsored by STSC, Inc., and held in Washington, D.C., on 9-11 April 1980. Three goals dominated the selection and editing of the topics covered:

- To provide general management with sufficient knowledge of *APL* to cut through the mystique that surrounds the data processing profession.
- To aid data processing managers and working professionals in bridging the gap between their familiar turf and new fields—such

as financial planning and conceptual information management—that are easily mastered with *APL*.

- To broaden the horizons of convinced *APL* users so that they can better relate to the real problems of general management and data processors.

Although we were always aware of these goals while editing, many of the topics (necessarily) contribute to more than one theme. We suggest that you begin in that part of the book with which you most closely identify, but that you sample the wares of the other parts as well. Most importantly, we hope that this volume will encourage you to apply an *APL* solution to some business problem.

Although the book implies that *APL* can be used for the full range of computer activities, we recognize that successful advances in data processing application come by evolution, rather than revolution. In that spirit we suggest that you start with some small project. Only after success with a variety of applications should you begin addressing the main question: Should *APL* be used for all new application development in your organization?

Many of the presentations in this book contain examples illustrating the use of *APL*. To aid you in distinguishing between user entries and system output, user entries are given in *APL* boldface type.

We are grateful to all the contributing authors who shared their experiences and knowledge in this book. So that you can put each author's contribution in perspective, we have included biographies at the end of each article.

We are also grateful to the following people from STSC, Inc., who assisted in the editing and production of this book: Sarah R. Beirn, Shelly L. Dimmick, Connie L. Kiernan, Karen M. Kromas, Laurie A. Russell, Nancy T. Vernon, and James G. Wheeler, editors; and Donna E. Kromas and Jean Medinger, publications assistants.

Allen J. Rose
Yorktown Heights, New York

Barbara A. Schick
Silver Spring, Maryland

10 April 1980

Contents

Part 1—The Data Processing Viewpoint

- APL* Concepts for Systems Management 1
- Evaluating Telecommunications Networks 14
- Managing an *APL* Installation 21
- An Overview of Reporting and Formatting in *APL* 28
- QUICKPLAN: A Reporting Tool for the Non-Programmer 33
- The *EMMA* Report Generator 39
- When *APL* Is Inappropriate 43
- Managing Outside Computer Services: An Organizational Relationship 49
- Selecting and Managing Outside Computer Services 53
- Converting External Datasets into *APL* Files 56
- A Fully Automated Interface Between Systems in Boston and Bethesda 61
- Making the Inhouse Decision: Some Considerations 67
- Variations in *APL* Flat Major 70
- Travels in VM Land: A Virtual *APL* Primer 76
- Using Shared Variables and Auxiliary Processors in *VS APL* 88
- Practical *VS APL*—FORTRAN Interfacing 98
- Optimization Modeling Systems: An *APL*/MPSX Interface 104
- Real-Life Applications of VM/370 111
- APL* and The Relational Model of Data 114

Part 2—The General Management Viewpoint

- APL* Tutorial for General Management 121
- APL* in the High School Curriculum 130
- A Business School's Approach to Better Business with *APL* 132
- Computer-Assisted Instruction at the Undergraduate Level 135
- Career Growth in an *APL* Environment 138
- The Upjohn Company Customized Financial Planning Model 142
- Financial Planning Applications of *APL* in J. Ray McDermott 147
- Marketing Management Applications 156
- Magazine Distribution Management 161

Computers Ain't Cool	168
Financial Reporting Systems: A Case Study	173
Using <i>APL</i> for Construction Accounting	177
Flexibility in Accounting Systems	185
Manufacturing Applications of <i>APL</i>	190
Managing and Computing	195
What If: The Making of a Vice President of Finance	204
An Evolutionary View of Business Computer Systems	210
Development of the STSC Accounting System	220
<i>APL</i> in the Corporate Service Environment	226
Electronic Mail	232
Business Graphics	241

Part 3—The Core of *APL*

User-to-Application Interface: A Command Processor Approach	249
Data Sharing in Large Application Systems	263
Maintenance Systems	272
Design Considerations of a Financial Planning System	279
QUICKPLAN Design Considerations	285
A Data Management Technique Using a Graph Structure	290
Writing Maintainable <i>APL</i> Programs	306
Making <i>APL</i> Palatable	312
The Use of <i>APL</i> in Applied Econometric Analysis	323
Management Statistics with <i>APL</i>	328
<i>APL</i> and Optimization Modeling	333
The Professional Programmer's Tool Kit	339
STSC's Design and Development Considerations	345
Nested Arrays: The Tool for the Future	350
A Consumer's Guide to Choosing an <i>APL</i> Terminal	356

Index	369
-------	-----

Part 1

The Data Processing Viewpoint

J. Murray Spencer

***APL* Concepts For Systems Management**

“The dogmas of the quiet past are inadequate to the stormy present. The occasion is piled high with difficulty, and we must rise with the occasion. As our case is new, so we must think anew.”

—Lincoln, Second Annual
Message to Congress.

Information processing successes have led information processing clients to expect more and better applications. At the same time, pressures of inflated costs and limited budgets require planners to wisely balance human, software, and hardware resources to meet these expectations.

The price of human effort is steadily rising, while the cost of fast hardware is going down. As fast hardware becomes more available and as its speed increases, improved software is needed to make wise use of the new speed. As “people time” costs go up, it becomes increasingly desirable to require less of it to implement and maintain applications. *APL* serves both of these needs by transferring many of the tedious, error-prone tasks in coding from the programmer to the CPU (central processing unit). *APL* also provides many powerful language and system support features with which to design applications. Not only are the tools more powerful, but because *APL* language features more naturally represent data and calculations, the user’s initial perception of a problem translates more readily and directly into *APL* than into other programming languages.

In fact, programmers often find that knowledge of *APL* notation improves their ability to represent, and consequently to analyze, problems. *APL* accomplishes this improved representation through a variety of data structures and a large, carefully chosen set of primitive functions. The primitives are written in a concise notation that allows complex algorithms to be represented compactly. The *APL* language subsumes detail through array-oriented primitives that reduce the need for explicit control structures. These features reduce the need for housekeeping to a bare minimum and permit the user to state only the essence of the problem. Programmer productivity is improved by working in a higher-level language; *APL* is as much higher than FORTRAN, COBOL, ALGOL, or PL/1 as these are higher than Assembler language.

It is often agreed that a programmer can work effectively with a page of program code at a time. A page of *APL* code is so concise that it is able to represent five to ten times as much algorithm as a page of FORTRAN, COBOL, ALGOL, or PL/1. Therefore, in *APL* a programmer can work with much more of a problem at one time. This reduces the amount of time the programmer spends switching between blocks of code, because there are fewer blocks. Shorter programs also reduce the work of typing the code. This saves time handling code, and reduces the chance of typing errors.

Most *APL* language processors execute code interpretively. Originally this was dictated by the language requirements and the available development resources. The interpretive approach allowed addition of powerful debugging aids as the workspace concept was employed and strengthened. *APL* language processors come with permanent storage facilities for workspaces (as developed by IBM) and permanent storage for data in files (as first developed by STSC, Inc., in 1970). Regardless of the CPU architecture or operating system, *APL* language processors provide a complete working environment for application developers and users. At the heart of *APL* design philosophy for the language and its use is the premise that the programmer and user should be required to know absolutely as little as possible about computer hardware and system software.

APL is a tool for thinking about data and algorithms. How to use *APL* to design algorithms is not the topic of this presentation. Readers who would like to pursue this direction are referred to the paper entitled "Making *APL* Palatable", which appears elsewhere in this book, and to K. E. Iverson's "1979 Turing Lecture", which is to be published in *Communications of the ACM*. The presentation that follows will concentrate on how *APL* accomplishes its wonders via features such as the active workspace, the symbol table, the file system, and the efficiency of the *APL* interpreter.

The Active Workspace: A Dynamic Execution Area

One of the obstacles to executing a conventional assembled or compiled program in most operating systems is the work involved in linking together the assorted routines from various libraries and loading them into an execution area. In *APL* systems, the user is always "in" a working execution area called the *active workspace*. Linking is done automatically among all programs in the active workspace, so all that is required is to copy or load the programs into the workspace. If they are present in the active workspace, they will run. A main program and all of its subroutines are easily saved as a permanent workspace in the user's workspace library with a single command called *)SAVE*.

In addition to programs, the active workspace contains data stored in variables. The user can enter any *APL* statement from his terminal and the statement will be executed immediately in the context of the active workspace. Such an *immediate execution* statement can reference a variable in the active workspace, and it can call any program in the active workspace as a subroutine. In fact, there is no need for a "run" command in *APL* systems, since entering the name of a program in the active workspace causes it to run; it is simply a matter of a valid *APL* statement being entered by the user.

Any *APL* statement that will execute in the active workspace can be a statement in an *APL* program. Thus it is normal to test algorithms one

statement at a time in immediate execution mode before storing them as lines of a program. Immediate execution of statements is a benefit to be expected of an interpretive language processor. It is such an important benefit that in some non-*APL* systems there are separate interpretive language processors solely for testing and debugging (i.e., in addition to the compiler that generates the production version of the tested code). The existence of separate interpreters and compilers for other programming languages raises the question: "How efficient is the *APL* interpreter?" Rather efficient, as will be seen below.

Programs exist to process data, and while some programs get by with small amounts of data, an important measure of a computer system is how much data it can process at one time. In general, the more data that can be referred to at one time by a program, the simpler the program will be. This results from less need for control structures in the program to iterate a solution through repetitions of the fundamental algorithms acting on segments of the data. All data used by *APL* statements is dynamically managed in the active workspace so that the space used by the data can be reused as soon as the data is no longer needed. In a conventionally compiled program, a data array is declared to occupy a certain amount of space. It occupies this space for the entire program execution—even if the array is needed only by a few lines of code constituting one step in the algorithm. Some programmers will actually reuse such an array one or more times elsewhere in their program to save execution space—which does not make the program easier to document and maintain! In *APL*, local variables are automatically erased upon return from a subroutine, and there is an executable erase that can be used (but is rarely needed) right in the subroutine or main program.

The *dynamic storage allocation* in *APL* systems allows an array, or variable, to become larger or smaller at any time. The array is always exactly the size needed by the data being processed. This exact-size quality eliminates the need for accessory variables in programs that keep a count of the number of rows or columns in a table (a considerable coding simplification). It also means that as one array gets smaller, space is immediately available to be used by another array that may be getting larger. Since the language processor does not need to know the size and shape of arrays beforehand, the *APL* programmer is relieved of having to use declaration statements to specify data type or data size and shape. Arrays are allowed to change size, shape, and data type any time a statement reassigns the array variable.

It may be claimed that some documentation is lost by not requiring variable declarations in the program code. However, the rank and type of an array is often quite obvious from its context in an *APL* statement. If it is not, and the programmer believes the code should note these specifications, he is encouraged to add a comment explaining the variable's attributes. Such a comment is likely to be more helpful to the reader than a program statement—like the DIMENSION statement in FORTRAN—that only tells how big the data can become.

The Symbol Table: Signpost to All Identifiers

All *APL* programs, whether main programs or subroutines, are called *functions*. The names of functions and variables are called *identifiers*. All identifiers are cataloged with descriptive and location information in the *symbol table*. The language processor keeps the symbol table completely up to date as each statement executes, because some statements will create new variables and functions as well as change the size of existing variables. The

programmer does not explicitly manipulate the symbol table, although most *APL* statements implicitly reference and modify the symbol table entries. Actually, most *APL* programmers know little or nothing of the existence of the symbol table, and the explanation in this paper is only for those systems-oriented people who are interested in how the language processor works.

Perhaps the most obvious use of the symbol table is in workspace storage management. The location of all items in the workspace is maintained in the symbol table along with information about each item's status. For example, when returning from a subroutine, all the local variables of the subroutine are flagged "to be removed". The actual removal and reorganization of the space waits until an *APL* statement needs to create an array requiring more space than is available in a contiguous block. At that time, items in the workspace that are still in use are moved next to each other and items flagged for removal are discarded. This process is known as "garbage collection". Garbage collection is timed asynchronously. It occurs not when garbage is created, but when the space occupied by the garbage is needed for an array or function.

Other uses of the symbol table support various *APL* language features and language processor efficiencies. For example, the fact that the symbol table is maintained in real time during execution provides late binding of identifiers. Thus, in this program statement:

```
'25A1, 12CF10.2, CF13.2' □FMT (NAMES;NUMS)
```

the identifier *NUMS* could be either a variable or a function. During debugging, *NUMS* might be a workspace variable to test the program until completion of the application module that will manage the application data. When that module is completed, a function *NUMS* could be defined in the workspace, perhaps like this one:

```
▽ NUMBERS ← NUMS
[1] NUMBERS←□FREAD DEPTFILE,CURRENT△DEPARTMENT
▽
```

The format statement above continues to operate without modification, because the language processor does not make a syntactic distinction between a variable and a function that has no argument and that returns an array result. At the moment of execution, it is clear from the context in the workspace what *NUMS* is, because there is always only one current (visible) definition of any identifier. Late binding of identifiers and interpretive execution allow the application implementer to test changes in his application design or coding with a minimum of typing and waiting at the keyboard.

More than one item in a workspace can be named with a particular identifier, but only the most recently created item will be referred to by an *APL* statement. A given identifier could be a global function or variable and it could be local to one or more functions currently executing. Only if an identifier is declared as *local* in a function header statement can there be more than one item with the same name in a workspace. (There is a declaration statement in *APL* after all—only one—and it is the function header statement that declares the function's syntax and its optional local variables.) When a function is called, the local identifiers in the function header statement are marked in the symbol table so that when a value or function is first defined in the function for the local identifier, it does not disturb the previous definition of the identifier, but rather creates a new item referred to by the identifier. An *APL* statement always references the most recent item created for any identifier. If there is no local definition in the currently executing function, the reference is to the most recently defined variable or function with this

identifier. “Most recently defined” implies a scan backward through the chain of function calls to see which, if any, calling function localized (and defined) this identifier. If no calling function localized the identifier, the global definition is used.

Thus *APL* uses “umbrella localization”. That is, an identifier defined as local in a function protects the previously defined items with this identifier, but is available to be read or modified in any functions called by this function—unless one of the called functions localizes this same identifier. When execution “returns” from this function, all the local identifiers are flagged to be erased in the symbol table, and the most recent previous definition of the identifier is again revealed for reference by *APL* statements.

When formal parameters are passed to functions (subroutines), the *APL* language uses “call by value”. That is, no action in the subroutine can modify the array specified as the argument to the calling function, because the arguments of the function are considered local variables inside the function. For example, if

```
[ 9]  I←1
[10]  XG←COMP2 I
[11]  ISQ←I*2
```

calls this function:

```
▽ RESULT←COMP2 NUM
[ 1]  RESULT←NUM
[ 2]  NUM←NUM+1
[ 3]  RESULT←RESULT, NUM×2
▽
```

the calculation on line [11] will use a value of 1 for *I*. Of course, line [2] does change the local variable *NUM*, which uses as its starting value the same data as was in *I* when the function was called.

This call by value is considered a very important protection for functions that call each other. However, when the arguments to functions are large arrays requiring thousands of bytes of storage, the CPU time required to set up the local variable arguments—and the very important space taken up by them—can be costly, because often a function will look at but not modify its arguments. If the arguments are not modified, the duplicate copy of the array is unnecessary. However, the symbol table is used to circumvent the inefficiencies of this form of duplication. Without disturbing the *APL* language definition of protecting the arguments to functions, the language processor uses the symbol table information to establish synonyms for arrays. If one array is assigned to another name, as in

```
X2←X1
```

the entries for both *X1* and *X2* in the symbol table will point to the same data stored in the workspace. Thus, *X1* and *X2* are two different names for the same data, or synonyms. This will continue to be the case until either *X1* or *X2* is assigned a new array—even one that differs only minutely from the other—at which time the reassigned identifier will point to its new version of the data. Use of such synonyms reduces CPU time spent on data replication; it also saves space in the workspace and postpones the need to “collect garbage”.

When a function is called, its argument local variables are not given duplicate copies of the data arrays supplied to the function; rather, the argument local variables are treated as synonyms to the data array arguments until the argument local variables are modified by an assignment. The synonym feature—sometimes called data chaining—was not part of early *APL*

implementations. Without it, when coding in a cramped workspace, it was necessary to use a less readable style of function coding in which arguments were not explicitly named in the header. Instead, data to be used by the function was made available through global variables.

The synonym feature provided by having the symbol table maintained in real time gives the protection of call by value with the space and CPU time savings of call by name (when these savings are possible).

Because of the protection of arguments supplied to a function as it is called and the protection of local variables used to control an algorithm, functions can be used recursively with no effort on the programmer's part other than to localize variables to control the flow of processing. Since it is good practice in any case to localize all variables not needed outside the function, *APL* functions are naturally recursive without any extra effort or caution.

Another way in which the symbol table allows increased efficiency is that the language processor passes a pointer to data for the result of a calculation when the calculation happens not to modify the data given to it. Take, for example, the following statement:

```
XCOMP←B/XTABLE×10
```

If *B* happens to contain a scalar (single element) 1, or even if it contains all 1's and no 0's (as it well might, in some cases), the compression function (/) does not create a new data array. That is because with a left argument such as these, compression does not change the data. The symbol table already points to the temporary array result of *XTABLE*×10, so the identifier *XCOMP* is made to point to this same array, which is then no longer considered temporary because there is an identifier referring to it.

APL Interpreter Efficiency: A Contradiction in Terms?

Is it a contradiction to use "efficient" and "interpreter" to describe the same language processor? It need not be. *APL* language processors started out with some powerful strengths and development has continued on them for over ten years. New ones are still being developed. This paper has already looked at several ways in which efficiency of execution has been designed into the language processor.

What further efficiencies should be considered? Efficiency may be considered in terms of the availability and use of particular resources such as CPU time, main memory space, disk memory space, high-speed swapping device space, terminal ports or telecommunications ports, terminals, development programmer time, application user time, and so on. Installations will have varying assortments of resources in short supply, which therefore must be used "efficiently".

Many systems managers are conditioned to think of efficiency first in terms of CPU time used, since this was uniformly expensive when computers first came into use. In recent years, CPU speeds have increased; thus the bottleneck in a system today might more likely be the number of disk accesses per second, or the number of page faults per second. On some systems the high-speed or low-speed input/output capacity may be used up before CPU usage approaches saturation.

As a general technique, interpretation is not always a poor choice. For example, it takes a lot of hardware circuitry to implement every 370 instruction in a CPU. For smaller models of the 370—up to the 370/148—the hardware logic gates do not implement a 370 at all. Rather, they implement a simpler machine capable of emulating (interpreting) a 370. The emulation

code is stored in fast, read-only memory control store. The CPU determines which 370 instruction is to be performed. Then, using a (rather complicated) subroutine of fundamental machine instructions (microcode) from the control store, the CPU emulates the 370 instruction. Considering their performance-per-cost, 370/148s and 4331s have their rightful place in the hardware lineup.

Another example of hardware interpretation that is dear to the pocket-books of its lucky users is the *APL* assist microcode option available on 370/148s and 370/138s. This feature speeds up execution of the *VS APL* language processor by implementing, in extended control store, additional CPU "instructions" much more powerful than 370 instructions and specially designed to replace sections of Assembler code in the *VS APL* language processor. The hardware does not implement these instructions with additional logic circuitry; again, they are subroutines of microcode instructions. This is interpreting at the hardware level, and it causes a 370/148 to considerably outperform a 370/158 when executing *APL* programs.

During program debugging and testing stages, the CPU time saved by not having to compile the language source code is very considerable. For jobs that do not run many times in production mode, the CPU savings alone completely cost justify *APL*.

Perhaps the discussion so far sounds like an apology for a slow language processor that uses a lot of CPU time. The fact is that one can easily find benchmark programs to show that FORTRAN, for example, is faster than *APL*. But it is just as easy to come up with benchmark programs showing that *APL* executes faster than the best code from optimizing compilers. This standoff quickly resolves to the wise saying: "Use only benchmarks that are representative of one's actual workload." Sometimes that is hard to do, since one's actual workload may not yet be coded in *APL*. So the systems manager continues to be wary of claims about interpretive language processors, even after hearing that great effort has been spent to make them use less CPU time.

It is worth looking more carefully at considerations of speed. Usability of the language by humans should be taken into account, because this is a very important speed factor too: "How much people time is needed to design, implement, document, and maintain applications in one language versus another?"

An interpretive language processor is a collection of carefully coded algorithms whose processing efficiency is as high as possible, given the resources used to develop algorithms. Existing *APL* language processors on IBM computers execute subroutines coded entirely in Assembler language for greatest speed. The language supported by the interpreter invokes a sequence of these algorithms. To achieve "processing efficiency", the ratio of time spent executing the carefully coded algorithms to time spent deciding which algorithms to execute should be high. *APL* programs are stored in the workspace in partially translated code strings in which all identifiers (function and variable names) have been resolved into pointers to symbol table entries. Relatively few *APL* symbols and pointers in a code string need be parsed to invoke powerful execution routines. In the following example, only 25 code string elements contain the statement:

```
T1+NAMES[(NAMES[;1ρNV]∧.=NV)/11+ρNAMES;]
P1+ P2 [( P2 [;1ρP3]∧.=P3)/11+ρ P2 ;] (shows pointers)
AB C DE F GHIJ KLMNO PQRSTUW XY (25 elements)
21 2 11 2 1111 21111 2111111 2 11 (31 bytes)
1 2 3 45 6 7 89 (9 functions)
```

Only 31 bytes of code need be parsed to invoke the 9 execution subroutines to evaluate the statement. (Byte count and function count rules are for the XM6-based *APL*PLUS* System; other systems may vary slightly.) What does this statement accomplish?

“Store in table *T1* the complete names from table *NAMES*, which are selected because their leading characters are identical to the characters in *NV*.”

The flexibility in this program statement is noteworthy. The table *NAMES* can have 0, 1, 2, 3, or 10,000 rows (or names). The lookup candidate in *NV* can be expressed with minimum truncation. That is, if “JONES, ROB” uniquely distinguishes an entry, the entire entry “JONES, ROBERT JAMES, JR.” need not be entered in *NV*. Conversely, if *NV* contained only “JO”, *T1* will contain all names beginning with “JO”, whether there are 0, 1, 2, 3, or thousands of them. No rank, type, or size declarations have been made for the variables *T1*, *NAMES*, or *NV*, because none are needed. This same statement will work whether *NAMES* is 1, 2, 3, 20, or hundreds of columns wide.

Efficiency in *APL* is a consequence of the use of powerful primitive functions: larger blocks of processing are meaningfully conveyed with briefer code statements. This produces two important efficiencies. First, the interpreter spends less time parsing statements than in a language with less powerful functions, and relatively more time doing “useful work”—that is, working directly upon data. Second, and perhaps more importantly, the programmer spends less time writing and maintaining the shorter code statements!

Processing efficiency in *APL* execution routines gets a lot of development attention. Often there are alternative execution subroutines for a particular *APL* primitive function. The subroutine actually executed will be chosen based on the amount of data to be processed, whether the data is integer or floating point, and so on. In the previous example, the table lookup (the \wedge = matrix inner product) uses an algorithm especially optimized for character data. Also in that figure, $/\imath$ is recognized as a special composite function and executed by a single subroutine, although the *APL* language defines compression ($/$) completely independently of the index generator (\imath).

The use of alternative execution subroutines to achieve processing speed for the simple arithmetic primitive functions (such as +, −, ×, ÷, \lceil , \lfloor , and \star) goes as far as generating object code tailored to the exact data given with each function call. This object code is discarded after its one use. Generation and execution of object code for such simple arithmetic is much faster than a generalized subroutine execution, except when very little data is given with the function. Where five or fewer data elements are given, the “old-fashioned”, but in this case faster, general subroutine is used.

The *APL* language abounds with powerful primitive functions. The following simple statement:

```
CUMSUMS $\leftarrow$ \TABLE
```

produces cumulative sums for each row of *TABLE* and stores them in *CUMSUMS*, a table with the same number of rows and columns as *TABLE*. As always, storage allocation for the tables—and special cases for empty, or very large, tables—is not the concern of the programmer.

APL interpreters are highly engineered for processing speed. Their ability to surpass the speed of optimized FORTRAN (or other compiled languages) in many instances is based on the fact that a small selection of powerful *APL*

primitives does a job that may require many pages of FORTRAN. The FORTRAN object code produced from these pages of source code has to compete with carefully tuned, hand-coded *APL* execution routines called with a modest overhead of parsing concise statements.

While presently available interpretive language processors are fast enough to be very useful, competition between software suppliers continues to improve the speed of execution. Indeed, *APL* language processors are beginning to appear that generate and save object code for every statement so that subsequent executions are faster than interpretive execution. As these language processors become available for the CPU of the manager's choice, the issue of processing speed will disappear. The use of *APL* will then become even more desirable, given all its other advantages.

Manipulating *APL* Programs: Poof, It's a Program

"Quick, call me a taxi!" "All right, already, you're a taxi."

As this old joke suggests, magical transformations unlock many possibilities for the application implementer. Perhaps the most useful program transformation is a general change of state from executable program (compiled if need be, linked, and loaded in the execution area) to data that a companion program can modify, and back again to executable program. This allows an executing program with self-knowledge to modify its subroutines. Such modifications can be as simple as storage management to free up execution space occupied by infrequently used (large) programs, or as complicated as creating programs to perform contingent, case-dependent code selection and optimization for efficient execution.

A less complicated but frequent use of program-to-data and back-to-program transformations is in *APL* programs that are themselves program development aids and that modify a program while it is in the data state. The program development aids in *APL* systems will continue to grow in versatility without the need for intervention of a systems programmer.

In addition to conveniently formatted program listings with various cross-reference tables (comparable to the listings available to an Assembler programmer—except that they are much shorter!), *APL* application implementers use program development aids of the type exemplified in the Appendix at the end of this paper. These programming aids recognize the syntactic rules of *APL* and can make useful organizational and syntactic changes to a program as well as perform traditional editing chores. There is, of course, a system-provided function editor quite suitable for simple entry and modification of programs.

The *APL* language processor uses a stack to store the control information for the execution in process. An execution may halt with a processing error during debugging. Another execution can be started, and when it is finished, the original execution can be restarted. This allows the use of programs to help analyze and correct problems that occur during debugging. If a program will not halt where it needs to be analyzed, program stops can be set to force a halt. Program traces can be set to cause the language processor to display intermediate results and the flow of processing as each selected statement executes.

Certain data conditions and external events can cause errors that normally halt execution. However, the *APL* application implementer has language and system features permitting him to maintain processing control when an error occurs, treat the error condition if he knows what measures to take, and keep on processing. This "exception handling" allows simplification of the control structures of some sequences of program code, since the program

does not have to test for obscure cases of bad data. Exception handling allows the application designer to further protect the end user from the unexpected events that can occur during processing, including those events caused by the user not following directions!

The File System: Unlimited Storage

The active workspace can be, and often is, permanently saved. While workspaces can contain data, workspaces are saved primarily to save the programs in them. The need for data organization and storage goes beyond having variables in a workspace. In 1971 STSC released its *APL*PLUS* File Subsystem, which has become the one to which other *APL* file systems are compared. The STSC file system is well documented elsewhere, but a brief summary is in order here.

The items stored in a file are entire arrays from a workspace. Every attribute of the array—its rank, shape, type, and all its data—is stored in the file as a single entity, called a component. An array stored in a workspace is stored as a variable with an alphanumeric identifier. The variable name that might be assigned to an array when it is stored in a workspace is not part of the file component. It is identified in the file by its position number in the sequence of components. This is appropriate, because it is often useful to process many components identically with an algorithm that repeats once for each component. This is done by reading the components into the active workspace one at a time and storing them in the same variable on each iteration of the algorithm.

Components of a file can be randomly read or replaced by specifying the component number. A component can be replaced with a completely different array component—different in rank, shape, type, and size. Multiple files can be tied, or “open” at one time. If correctly planned, multiple users can update the same file concurrently. The system provides access tools to control the sequence of updates and prevent one user from modifying the file until another user completes his update.

Files are completely private when created, but after creation, the file owner can allow carefully controlled (with passnumber protection), fine-grained access to users of his choice. For example, the file owner can allow one user to read the file with one password; allow other users to append to the file (but not read it) with another password; and allow certain maintenance users to read, append, and erase the file.

Considerable development effort has gone into designing a file system that cooperates with the multiuser scheduler for maximum system throughput. One technique is to coordinate the swapping of users with the expected completion of file operations. Multiple file operations proceed concurrently.

The file system interface presented to the *APL* programmer is compatible with overall *APL* language design considerations. The file operations are invoked with system functions that behave like *APL* language primitives in syntax and error messages. All file operations can be done under program control.

A Summary Statement: *APL* Is Cost Effective

In the early days of *APL*, it acquired a reputation for being “for mathematicians and scientists only”. This was partly due to the extended

character set and partly to some system support limitations of early implementations (e.g., lack of a file system, lack of an output formatter, and small active workspace size).

APL service companies such as STSC have welcomed use by mathematicians and scientists, but over the years business users have come to account for more and more of the usage (currently 80 to 90 percent for STSC). The very users who should care the most about cost comparisons are the ones who use APL the most. Probably it is because of APL's effectiveness!

Appendix—APL Program Development Aids

This appendix lists some of the program development aids frequently used by users of STSC's APL*PLUS Service. All of these programming aids are APL programs that use the program-to-data transformation ($\square VR$) and the data-to-program transformation ($\square DEF$). The programming aids work on the data that represents the APL program under consideration.

Workspace 11 *TOOLS* contains several functions that search and/or manipulate other functions. Following are brief descriptions for each of the functions in workspace 11 *TOOLS*.

- The function *BRKOUT* modifies a given function to break out embedded assignments into individual statements. For example,

$$H \leftarrow \rho \square FREAD(R+1 \div F), C \leftarrow F[1+O \div 1]$$

would become

$$O \div 1 \diamond C \leftarrow F[1+O] \diamond R+1 \div F \diamond H \leftarrow \rho \square FREAD R, C$$

- The function *FNIDS* searches a given function for identifiers in certain categories (locals, labels, direct assignments, indexed assignments, or \square names). Combinations of identifiers may also be specified (e.g., intersection, union, and complement).
- The function *LOCALIZE* localizes specified functions in a given function.
- The function *ORDLOC* reorders the local identifiers in the header of a given function.
- The function *RELABEL* modifies a given function to use the set of labels *L1*, *L2*, and so on. This function also converts occurrences of *THISL* and *NEXTL* to their corresponding labels. (Also see *RELABEL*.)
- The function *SNUFF* removes comment text from a given function.
- The function *UNPAREN* removes superfluous parentheses from a given function.
- The function *XREF* displays a cross-reference of a given function.
- The function *RELABEL* modifies a given function to use the set of labels *A*, *B*, and so on. This function also converts occurrences of *THISL* and *NEXTL* to their corresponding labels. (Also see *RELABEL*.)

Workspace 11 *FNED* contains the function *FNED*, which edits functions quickly and conveniently, including those whose lines are longer than workspace or terminal widths. *FNED* uses conventions that provide:

- ordinary string searching or syntactic element searching
- single or multiple replacements
- large deletions, moves, copies, and insertions.

Workspace 11 *FNR* contains the functions *FNREPL* and *BY*, which can be copied into the active workspace. These functions are used to modify programs by replacing one character string with another. Their syntax is

```
'function list' FNREPL 'old' BY 'new'
```

where both 'old' and 'new' are character vectors; 'old' must not be empty, but 'new' may be empty; 'function list' can be a character array of any rank (nominally a vector), with the individual function names separated by spaces (nominally), new-line characters, structural significance (rows of a matrix), or any combination of the three.

FNREPL searches all unlocked functions in 'function list' for the syntactic elements represented in 'old'. For each function in which *FNREPL* detects 'old', an indication of how many occurrences is given and 'old' is replaced by 'new' in that function.

Workspace 11 *WSS* contains the functions *WSFIND* and *WSSHOW*, which (when copied into the active workspace) are used to find or show all occurrences of a character string. Their syntax is

```
WSFIND 'characterstring'
WSSHOW 'characterstring'
```

Only unlocked functions are searched. *WSFIND* prints the function names, followed by the line and print position in the line. If the character string appears more than once in the line, an indication is given. *WSSHOW* prints the function name and the number of occurrences of the character string in the function; it then prints the text of each line in which the character string occurs.

Workspace 11 *WSSEARCH* contains the functions *SEFIND* and *SESHOW*, which (when copied into the active workspace) find or show all consecutive occurrences of the syntactic elements represented in the right argument in all unlocked functions in the workspace. Their syntax is

```
SEFIND 'characterstring'
SESHOW 'characterstring'
```

For example, the expression

```
SEFIND 'BCD 234'
```

would find 'BCD 234+5' or 'A+BCD 234', but would ignore 'BCD 2345' and 'ABCD 234'. *SEFIND* prints the function names, followed by the line and print position in the line. If the character string appears more than once in the line, an indication is given. *SESHOW* prints the function name and the number of occurrences of the character string in the function, and then prints the text of each line in which the character string occurs.

Murray Spencer joined STSC in 1970 as branch manager of the company's Washington, D.C., office. He subsequently held the positions of branch manager in San Francisco, APL applications analyst, and manager of product planning

and support. Spencer is currently manager of small computer products for STSC.

Prior to joining STSC, Spencer was a systems programmer for Bell Telephone Laboratories and an EDP product planner for RCA Information Systems. He has a B.S. in applied mathematics from Clemson University.

Fred B. Lear and John W. Myrna

Evaluating Telecommunications Networks

With the availability of packet carriers and specialized common carriers, you, the communications manager, have viable alternatives to meeting your network needs inhouse. "Should you use a packet or specialized carrier rather than doing it yourself? How do you evaluate your options? What, in fact, are your options?"

Over the years, STSC has asked these questions. After considerable technical and financial study, we concluded that the most effective approach for us was to use a mix of packet carrier services with a central inhouse network. We would like to share our analysis, experience, and conclusions with you. Though the conclusions you reach may be different from ours, our analysis and experience should prove useful.

Some Background on STSC

To set the stage we'd like to outline STSC's experience in the communications area. This will help you understand the problem we were addressing when looking at the options.

STSC was founded in 1969 to provide an interactive computer time sharing service based on the *APL* programming language. We decided on a national marketing strategy, which was uncommon at that time. This required a national network from the start. We originally used a simple Time Division Multiplexer network, but we quickly outgrew it. As we increased the trunk speeds to 4800 BPS and expanded our services to cover smaller cities, we became painfully aware of the limitations of the telephone company's network and the quality of available modems and multiplexers.

We eagerly evaluated each new communications offering, hoping that it would hold the solution to our problem. Alas, none solved all of our problems completely, so we were forced to combine offerings that would collectively meet our needs.

Through the years, STSC's communications service has evolved to meet the needs of its customers. The table below shows the growth in this area from 1972 through 1979.

1972	Used Western Union Data Communications Service.
1973	Reviewed and rejected inhouse minicomputer networking.
1974	Began using specialized common carriers—MCI and DATRAN.

	Interfaced to the TYMNET network.
1975	Began using DDS. Became the first user of the minicomputer-based SMART/MUX. Became the first customer of TELENET.
1976	Converted to Infotron multiplexers (240, 180).
1977	Added Infotron smart multiplexers (780).
1978	Installed Comten 3670 front ends, with RPQ code for demuxing (A.M.X.).
1979	Upgraded to Comten 3690 front-end processors.

To this day, we are still searching for the system or technology that will solve the bulk of our problems with one bold stroke.

In 1977, we had reached a pivotal point in our network planning and felt that there were three approaches available to us:

- Develop and extend our inhouse network using intelligent multiplexers.
- Leave the network business entirely and use only a packet carrier.
- Use some combination of the two approaches.

We determined which approach to take based on the answers we found to the following questions:

- What were our objectives?
- What were our options?
- How should we evaluate the options?

Of course, these three questions apply to just about every decision made in business.

STSC's Objectives

Our primary communications objective was to provide an acceptable, cost-effective means of connecting user terminals to our host computers. The importance of understanding the company's objective cannot be overemphasized. To determine the best way to meet our objective, we considered the following questions:

- What type of terminals must we support?
 - low-speed asynchronous?
 - high-speed bisynchronous?
 - polled bisynchronous?
- Where is access required?
 - in major cities?
 - in small, out-of-the-way plant locations?
 - internationally?
- What service level is acceptable?
 - are users generally insensitive to errors?
 - are users very sensitive to errors?
 - are users very sensitive to system availability?

- What is the volume and distribution of usage?
 - a few hours per month from many locations?
 - hundreds of hours per month from a few locations?
- What are the characteristics of usage?
 - short holding times with dial access?
 - hardwired terminals logged on all day?
- What is an acceptable cost?
 - less than long distance?
 - less than using a local minicomputer?

STSC's Options

As mentioned before, our three options were to build and operate an inhouse network, use a commercial Value-Added Network (VAN) such as the packet carriers TELENET and TYMNET, or use some combination of the two.

STSC's Evaluation Process

How did we evaluate the options? We used five basic criteria:

- cost of service
- quality of service
- scope of service
- service sparkle (image)
- the ever popular "other".

What follows is a discussion of the five criteria and issues STSC considered in 1977. Even today, STSC continually reevaluates its communications needs in terms of these considerations.

1. Cost of Service

The cost of providing service is based on the location of users, the usage profile, and the volume of usage. One way to characterize the *location* of users is high density, low density, and off-net. A high-density location for a packet carrier is usually a major city such as New York. Because of the large number of users sharing common facilities, the cost per user of providing service is less than it is in a low-density location. The packet carriers, therefore, can charge less because of economies of scale. In addition, they may choose to charge less for competitive reasons.

In most systems there are users in locations not serviced by the network. In our case, these are typically small branch offices of client corporations whose major usage is on the network. The relatively low-volume usage from these locations makes some variation of long distance—such as WATS—an acceptable, though relatively expensive, method of support. We were amazed to find that this small percentage of our usage represented 29 percent of our costs.

The *usage profile* has a dramatic effect on the cost of using a packet carrier. The number of characters transmitted per hour and the number of characters per transaction has a noticeable effect on the final bill, since the carrier's charge is based on those factors.

The *volume of usage* affects cost in three ways. First, the cost of the computer's telecommunications front end must be amortized over the total

traffic. As the front end typically has a large, fixed element of cost, the greater the number of usage hours, the lower the cost per hour. Second, packet carriers require a connection to the network—another fixed monthly expense to be amortized. And third, with sufficient volume from a given location the packet carrier may give a price cut and inhouse equipment may be better utilized.

What is a typical breakdown of network costs? The figures in Table 1 are derived from an analysis of providing 20,000 hours of service per month in 90 cities in the United States and Canada. For the analysis, the network was assumed to be 100 percent inhouse or 100 percent packet carrier. For the assumed volume and distribution of usage, a 100 percent packet carrier approach was projected to cost 20 percent less than the equivalent inhouse network. Although the percentages will differ for a different combination of cities and volume, it is still instructive to review costs in this way.

Table 1 — A Cost Comparison

	<u>Inhouse</u>	<u>Packet Carrier</u>
Staff	14%	6%
Front-end Processor	8%	12%
Off-Net Access	29%	3%
Network		
Long Lines and Modems	20%	
Multiplexers	10%	
Local Lines and Data Sets	9%	
Site Rental	4%	
High-Density		34%
Low-Density		42%
Van Connection		3%
Other	6%	
	<u>100%</u>	<u>100%</u>

Looking at Table 1, you will note how large the staff expense is for the inhouse network (14 percent of total expenses). These expenses are much lower for packet carriers, because the carrier provides his own staff and, since the carriers are available in so many cities, there are fewer users off-net.

Economies of scale will decrease costs. As total volume increases in a part of the network, the use of long lines, local lines, datasets, and modems increases. As usage increases, the average cost per usage hour will decrease.

Another cost consideration is that the packet carrier's domestic charges are not dependent on distance. The packet carrier will charge the same amount for one hour of usage in the city where the computer is located as it charges for one hour of usage in a city on the other side of the country. Since long-line rates are distance sensitive, the cost of providing service via an inhouse network increases with increased distance from the host. Thus, a packet carrier may provide the lowest cost for a national network, while an inhouse system may be better for a regional network.

There are additional costs for installation, maintenance, training, software, and equipment. In a dynamic network like STSC's, where we are constantly adding or deleting parts, installation charges can actually exceed the regular monthly charges. In this case, there is an advantage in using a packet carrier.

One additional concern is how to prepare for growth in usage volume. Packet carriers charge for usage as it occurs, so if you must double your usage in a city, you simply do so. With an inhouse network, however, there are new local lines and datasets to be ordered and installed, equipment to be upgraded, and so on.

2. Quality of Service

STSC examines three measures of quality of service: reliability, availability, and response time. *Reliability* is defined as the probability that a user will complete his work without being disconnected by the network. We consider values greater than 97 percent to be acceptable. *Availability* is defined as the probability that a user will be able to connect to the host during scheduled hours. We consider an availability greater than 98 percent to be acceptable. *Response time* is defined as the time the network adds to the user's interaction. We consider values less than one second to be acceptable.

Packet carriers have characteristics that should provide high availability and reliability. They use minicomputers to detect and correct line errors, redundant equipment to minimize outages, and a large number of local-dial lines in each location. Where an inhouse network might be able to provide 90 percent availability, a packet carrier (because of the larger number of lines in each rotor group) could provide 99 percent availability.

However, the response time on a packet carrier tends to be longer than on an inhouse network. This is generally due to the fact that the packet carrier routes users through more intermediate nodes. Another potential problem is that a company's need for increased capacity in a given city may not concur with the packet carrier's schedule for an upgrade, resulting in longer lead times to respond to growth needs. Similarly, mean time to fix problems can become extended, primarily due to the coordination effort necessary between the numerous parties involved.

3. Scope of Service

In comparing VANs such as TELENET and TYMNET, it is important to determine how many of the cities serviced by you are directly served by the VAN. You should also verify that the terminals and features used in your network are supported; for example, in 1977 we desired support for the IBM 3767 and for transmissions of up to 120 characters per second (CPS).

4. Service Sparkle

Our decisions also consider a number of items that may be best described as vendor "sparkle". If poorly handled, these details can be a major irritant to users. For example: "What is the sign-on ritual? How many steps are involved to sign on? How solid is the automatic baud rate detection? If a user makes an error when connecting to the network does he have to hang up and dial again? Does the network support multiple hosts? Does the packet carrier cater to specific needs?" (In our case, one special requirement is the support of *APL* terminals.)

5. Other Considerations

We include a number of other considerations in our decisions. The terms of the contract and its conditions are important as is the availability of technical assistance and the guaranties available on service and price. In addition, there are several broader considerations, such as:

- What are future product directions?
- What economies of scale apply? Today a packet carrier may be ideal, but in a couple of years the same may not be true.
- Reducing the size of inhouse staff may be irreversible.
- Does it make sense to have a networking capability as part of corporate vertical integration?
- Who has control over cost, quality, and innovation?
- Does using a packet carrier affect corporate image?

- Will building and supporting an inhouse network detract from other opportunities, or drain management resources?
- Does using a packet carrier increase vulnerability?
- How flexible will future services be?

STSC's Decision

After much consideration of the technical and financial aspects of communications, STSC decided to use an inhouse network, supplemented by a VAN. Eighty percent of the network load is handled inhouse; the other 20 percent is handled by packet carriers.

You might well ask: "What has happened since this decision was made?" For one thing, the primary packet carrier we chose was slower in meeting our needs than we had anticipated. Consequently, we were unable to move the substantial share of usage to the carrier as planned.

On the other hand, the rapid expansion of the packet carriers (particularly internationally) has been an asset to us, allowing us to substantially reduce our off-net expenses. We also have relied on the packet carriers to provide special services to our customers, such as 33.33 CPS access for IBM 3767 terminals and 120 CPS dial access. Both of these services would have been expensive to add to our network. In addition, TYMNET has served as a backup to our own network. We have relied on the packet carriers to smooth sudden, but temporary, increases in usage. This has saved us the high installation costs of rapidly expanding our own network.

Conclusion

STSC has reaffirmed the value of the packet carriers. They are an integral part of our network. However, with our scale of usage it appears that it will always make sense for us to use a substantial inhouse network.

One last point is that, as a communications manager, you should determine whether you actually have a need for a network. If you have only one application that requires an extensive network, or your network requirements are low, you should consider running the application on a time sharing service.

As has been said before, but is so true, there are no simple solutions—only intelligent choices.

*Fred Lear joined STSC in 1979 as manager of communications. He is in charge of STSC's international telecommunications network, which provides access to the APL*PLUS Service in over 200 cities throughout the world. Prior to joining STSC, he spent eight and one-half years with Boeing Computer Services, where his positions included operations manager (Philadelphia), Washington area communications manager, and supervisor of hardware configurations.*

Lear attended the Institute of Computer Management for one year and also holds certificates from several IBM and COMTEN training programs.

John Myrna joined STSC in 1971 as manager of operations; in this position he organized STSC's Computing Center and nationwide communications network.

He was subsequently promoted to manager of communications in 1973, director of development and design in 1975, director of development in 1977, and to his current position as vice president of development in 1979.

Myrna directs STSC's Operations Group and is a member of its Executive Committee and Technical Management Committee. He is responsible for the production and delivery of computing and telecommunications services and for the development of new applications, products, system features, and technologies.

Myrna holds a B.S.E.E. degree from the New Jersey Institute of Technology and an M.S.E.E. degree from Montana State University.

Michael E. Handelman

Managing an *APL* Installation

In operating and managing an *APL* installation, many complex problems arise. We at STSC, Inc., have addressed these problems and, in this paper, I will present our solutions to you. For purposes of discussion, I have divided the process of developing and operating an *APL* service into four distinct areas: software requirements, hardware requirements, commercial considerations, and staffing requirements.

Software Requirements

Running an *APL* service is similar to operating any other form of computer service; the major difference is the software. Software requirements, which affect all other areas of the operation, are the operating system, the supporting software, and upgrades and other changes.

1. Operating System

An exhaustive analysis was performed by STSC to determine which *APL*-supporting operating system, of all those currently on the market, best fit our particular requirements. IBM's OS/MVT and VM systems proved superior in our ratings. Although VM consumes more resources than other systems investigated, it is an extremely powerful system, and is both reliable and truly dedicated to teleprocessing. OS/MVT, although a batch-oriented system, has been modified by STSC to be a highly reliable and stable operating system. STSC currently operates both OS/MVT-based and VM-based services.

2. Supporting Software

The reliability of any operating system depends on the supporting software. It is necessary for supporting software to be as "bug-free" as possible to ensure satisfactory support of both systems development and pure applications. A stable operating system and an excellent record of vendor support are critical components of bug-free supporting software. STSC has developed a System Support Team (SST) to diagnose problems and determine their origins, whether caused by hardware, the operating system, or STSC. Our team is staffed with dedicated system programmers who recognize the need to keep their knowledge and abilities current with fluid technology and stringent company standards. The benefit is minimal down time and a diminished need to rely on other sources—including the vendor—for problem solution.

3. Upgrades

Upgrades and other *APL* system changes must successfully complete three preparatory stages before being installed on STSC's production system.

The first stage involves the identification of a problem and formulation of a solution or upgrade by our design team. In stage two, our development team installs the potential upgrade on a system identical to our production system. Once the potential upgrade is proven reliable, its modules (code) are passed to SST personnel who perform a technical walk through (TWT) in stage three. When the code has been debugged, stage three results in an upgrade that has survived rigorous testing procedures and standards. An upgrade, having met the challenges of the three-stage preparatory system, is finally incorporated into our production system. It is important to note that the ability to back out of any modification at any stage of the process is designed to be as simple as possible.

Hardware Requirements

Continuing advances in technology, coupled with outstanding price and performance improvements, have caused the hardware area of data processing to change at an astonishing rate. For example, the capabilities of STSC's Amdahl 470V/6 Central Processing Unit (CPU) are astounding when compared with units available just fifteen years ago. Technological advances such as IBM's new 64-bit chip show that for hardware the future is just beginning.

For advocates of *APL*—who are sharing in the resultant interactive language boom—the industry is showing new life and growth. At STSC, we have emphasized the importance of keeping abreast of changes and trends in the industry. Attendance at seminars, the reading of trade journals, and active interface with vendors are encouraged as valuable learning tools. Awareness of hardware requirements for the CPU and peripherals also requires investigation, planning, implementation, and monitoring.

1. Central Processing Unit

In a teleprocessing environment, the CPU—the core of any system's performance—must be constantly monitored to determine if it is overused. When overused, specific bottlenecks must be identified. At STSC we use a hardware monitor designed by TESTDATA and software modified by STSC to provide these results. This enables us to determine the type of upgrade necessary (e.g., more swapping devices rather than more memory or another disk controller to decrease channel busy status occurrences). All this permits the fine tuning of the system to attain optimal use.

2. Peripherals

When discussing hardware requirements we must also mention peripherals. For an *APL* installation, the major form of storage is online storage. This includes swapping devices and file storage units, but excludes communications controllers that are regarded as a separate system in today's world. Tape drives, printers, and card readers are also made available, of course, but their importance is low in a telecommunications system.

Swapping devices—high-speed, fixed-head disks with low storage capability—are used to establish storage areas proportionate to the workspace size. (Workspace size is the actual size of real memory assigned to each user when logged on to *APL*.) Although virtual systems allow for swapping to relatively low-speed storage devices, it is a process that slows down the response time to the end user. Therefore, as the number of users or workspace size increases, high-speed swapping devices become a necessity in keeping pace with user needs. It is the duty of the paging or swapping manager to monitor the performance of the swapping function. Knowing the optimal number of users and the size of real memory on the system, the swapping manager can determine when low-speed swapping is occurring, and when additional high-speed swapping devices would benefit system response time.

File storage of user data, normally kept on an online storage device, must be flexible. Although user data may be stored on tape, cards, or other types of machine-readable media, these methods are entirely unacceptable for normal, interactive use in this era of instant computing. Recognizing the current three-to six-month lead time involved in the acquisition of new hardware disk modules, we must maintain a constant awareness of user storage needs. Only by carefully plotting historical and current usage can proper storage planning be accomplished.

When planning the back-up considerations of file storage, archival and emergency needs must be examined. STSC currently performs nightly incremental back-ups that copy files to a tape, and include any file updates performed that day. A full dump, a tedious process that places all disk files on tape, is run once a week—usually on Friday night. The tape created by the dump is scanned to check for any tape errors or other improper processing. All of the file disk packs are then taken offline and a new set put online. Next, a full restore is run to copy all information from the tapes that were used for the full backup to the new set of disk packs. The results of these procedures are two sets of tapes (one of which is stored offsite in a fireproof vault) and two sets of disk packs.

These backup and restore procedures illustrate the great care that is taken to preserve the integrity of user data. This is all performed as standard procedure for users of our *APL*PLUS* Service.

Commercial Considerations

Commercial considerations are all nontangible items required to support the user community. They include the billing method, scheduling, hardware reliability, and system security.

1. Billing Method

When viewing commercial considerations, it is important to understand the billing method. In addition to charges for online and offline storage, communications needs, and any special services provided, actual CPU and related costs must be calculated. These are presented through Computer Resource Unit (CRU) charges. The CRU is a unit of measure, developed on a base central processor, which is portable to other CPUs by adapting the measure to allow for quantitative differences in machine capabilities. The CRU not only measures the time involved in executing a program, but also recognizes all potential resource usage, thereby simplifying billing through reduction to a simple rate.

2. Scheduling

Scheduling of computer resources—the ability to assign load according to a set of priorities—is another important factor. The most important consideration is the scheduling of external or billable users versus internal or nonbillable users. For example, to make the system load sensitive, user sign-on identifications can be biased so that external users are assigned a high-priority level and internal users a low-priority level.

During light to moderate usage periods, users are unaffected by the priority scheduling. However, during heavy usage, resource availability is weighted in favor of external users. Simply, as resource requests are made they enter a queue and are identified by a timestamp and a user priority level. Adding the priority level to the timestamp determines the access priority. Since the queuing system adheres to first-in-first-out (FIFO) guidelines, an external user who submits a request less than the predetermined number of timestamps after an external user will have first access to available resources.

A hypothetical example of biasing user identifications, using a .0 priority level for external users and a .3 priority level for internal users, is presented in Table 1. STSC uses a similar type of load-sensitive system to maintain excellent response time for our customers.

Table 1 — Biasing User IDs: An Example

Order of Requests	1 (ext.)	2 (int.)	3 (ext.)	4 (ext.)	5 (int.)	6 (int.)	7 (ext.)
Timestamp	1.1	1.3	1.4	1.5	1.7	1.9	2.1
Priority Level	+0	+3	+0	+0	+3	+3	+0
Biased Timestamp	1.1	1.6	1.4	1.5	2.0	2.2	2.1
Biased Order of Access	1	4	2	3	5	7	6

3. Hardware Reliability

In conjunction with providing rapid response time, systems must be available and reliable. One method of assuring both availability and reliability is to maintain redundant hardware. Since economics do not allow 100 percent redundancy, an effective manager must determine the optimum percentage of redundancy. Each system must be rated on importance to the normally operating system and the impact of any resulting operational degradation due to loss of the component.

An example of the process is typified in rating a disk controller versus a single disk unit. Loss of a controller could cause the loss of a whole string of disks—up to 32 drives. When compared to the loss of a single unit, a back-up controller is justified. Further, the primary and back-up controllers could split the 32-unit disk string, each controlling 16 drives. Not only would this afford optimal performance through two-channel switching, but each controller would back up the other allowing improved response time and a back-up disk controller in case of failure of one of the units.

4. System Security

System security is another important consideration in running an *APL* service. In defining and implementing security measures, both physical security and software security must be considered.

The physical security of computer installations is a growing concern, and the marketplace is responding with a variety of security systems. A brief list of considerations regarding the physical security of an installation, and some possible remedies are given below.

- *Machine room access.* Limited access to computer facilities can be attained through the installation of a card key or electronic lock system. These systems can allow entry to a secure waiting area where a guard or receptionist screens individuals before admittance to more highly classified areas. Alarms should, of course, be placed on all doors to and from the facility.
- *Fire protection.* The best method currently available is a Halon fire protection system in conjunction with a cross-zoned detection smoke-protector system. When two detectors in separate cross-zones are activated, an alarm sounds, access doors are automatically closed, and the time-delayed Halon system is enabled. If determined to be a false alarm, the time delay provides for

manually aborting release of the Halon. Detectors for this system should be inside the ceiling and below all raised flooring for maximum effectiveness.

- *Water detection.* Although professed to be waterproof, underfloor cables can present a hazard if subjected to water leakage. Water detectors are inexpensive compared to the cost of potential water damage.
- *Sealed room.* The machine room should be sealed against the entry of outside water leakage and dust. The facility designer must anticipate water leakage from floors above the machine room and should plan for the installation of a drainage system under the raised flooring.

With a significant rise in the incidence of computer crime, physical security alone does not offer sufficient protection of resources. Software security is rapidly becoming a prime concern. The most visible software security measure in an *APL* installation is the sign-on password; users must understand the significance of password protection. Each user should be as possessive of his password as he is of his toothbrush or any other highly personal property. Further protections have been added to the *APL*PLUS* System, as described below:

- Both workspaces and files can be locked with a password.
- Files cannot be addressed by anyone other than the owner, unless the owner overrides the default by specifically giving access to other users.
- By applying privileged levels, the owner of a file can give others permission to read, add, modify, copy, or delete file information (or any combination of these).
- A daily report of possible system security violations is produced. As an added feature of the *APL*PLUS* System, STSC can set a limit on the maximum number of such incidents allowed during a single user's session. If this limit is exceeded, a "burglar" alarm is triggered to notify the *APL*PLUS* System Operator of a possible violation, and pertinent information is displayed.

Physical and software security are critical components of an effective security system, but properly executed manual procedures are equally as important. These manual security procedures provide an effective buffer for the automated physical and software security measures. Operator logs, for example, provide a narrative of daily occurrences at the installation and have proved to be valuable security tools.

Staffing Requirements

Staffing requirements can be divided into three areas: Operations, Communications, and Systems. Figure 1 depicts the organization chart for an *APL* installation, and the sections that follow describe each area in more detail.

1. Operations

Operations at STSC is comprised of all personnel who run the machine room. This excludes Communications staff, but includes the receptionists, clerks, and computer operators who handle day-to-day operations. In an *APL* installation, the operator not only handles normal duties such as mounting tapes, running consoles, operating high-speed printers, and maintaining trouble logs, but also acts as a system user. The operator must both run *APL* user programs and interface with the users, incorporating a knowledge of

overall operations with more detailed knowledge of specific operations. Training in the specific aspects gives the operator confidence in his abilities and gives the user confidence in the overall operation.

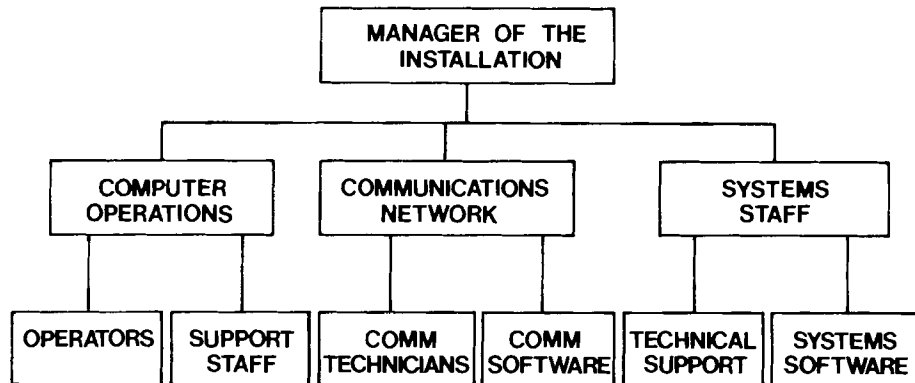


Figure 1—Organization Chart for Operations

Candidates for the position of *APL* operator are carefully evaluated, based on willingness and desire to learn. Although past experience and knowledge of operations, hardware, and operating systems is important, being open to new learning experiences is the quality that will allow growth in the position. At STSC, we view computer operators as entry or junior-level personnel who, with dedication and training, can find rewarding careers in the data processing field.

2. Communications

Communications requirements, which should include all real-time communications support, differ among installations. In a small installation, the Operations and Communications functions can be easily combined. In a larger installation, however, the two should be separate and distinct functions. Responsibilities of Communications personnel should encompass installations, upgrades, network maintenance, and terminal hardware support. Communications personnel should also support communications software which includes development, maintenance, and the interface with both Operations and vendors.

3. Systems

Systems staff at STSC comprises a Technical Support Team (TS) and a Systems Support Team (SST).

TS staff are systems experts who deal with users at all levels. TS is responsible for maintaining system centralization, carrying out special projects and programs, providing customer interface that cannot be handled by Operations, and furnishing all non-*APL* support. TS also provides training and designs tools for all users.

SST staff must possess a high level of technical ability to be fully responsive to the wide range of problems that are encountered. Additionally, the main departmental function—system upgrades—must be allocated according to individual capabilities. Specific areas of an upgrade include thorough testing and debugging, a technical walk through, complete documentation, a weekly upgrade cycle, and fast back-out and recovery procedures. Further,

SST staff provide training and develop tools for the other departments in the Computing Center.

Conclusion

The most important consideration in managing an *APL* installation is service to the user. Quality service can only be provided if the installation is reliable and the organization is qualified. Thus, there must be adequate coverage of the Operations, Communications, and Systems areas of the organization. Additionally, security measures must be constantly monitored to ensure the privacy of users' files and programs. To foresee and prepare for future needs, a planning analyst should—under the auspices of the manager—direct full-time effort to capacity planning and software monitoring.

The single most important, yet most overlooked, influence on service is the people. A manager must maintain a responsive team of qualified professionals backed with a strong training program, especially for entry-level staff. The effective installation manager recognizes the direct correlation between the quality of support staff and the quality of service provided by the installation.

Michael Handelman joined STSC in 1979 as manager of operations for the STSC Computing Center. Prior to that he had several years' experience at the George Washington University Computer Center, where from 1976 until 1979 he was the supervisor of computer operations.

Handelman holds B.B.A. and M.B.A. degrees from George Washington University.

Janet H. Faltz

An Overview of Reporting And Formatting in *APL*

Report formatting is the process by which raw data is transformed into a pleasing and readable format. Ideally, the data is presented so that its full import is obvious to the reader and key information can be extracted easily from the report. This may involve manipulating the data in some way, such as performing calculations on it or changing the order in which it is presented, or it may require adorning the data with explanatory material that will assist the reader in understanding its content.

Computers *should* be ideal tools for aiding in the process of reporting. Data—the reason for the existence of a report in the first place—can be manipulated by computers in large quantities, at amazingly fast speeds, and with virtually no probability of error. However, the embellishment of the data is often a nontrivial task.

In the *APL* environment, character data is presented exactly as it is defined, allowing for the effects of terminal printing width. Numeric data is displayed according to a set of fairly complex conventions that may differ from one *APL* implementation to another, and may be subject to change. Even this is an improvement upon other computer language environments, where the process of removing data from the heart of the machine and displaying it elsewhere can be painful. The skillful *APL* programmer can manipulate *APL* data-display conventions to his advantage, but the less sophisticated user must either have *APL* formatting tools available or be content with system conventions.

APL formatting tools should assist the user in many ways. For example, they should provide the ability to:

- Mix text with numeric data, such as in report titles, row names, column headings, and footnotes.
- Display numeric data in a format different from its internal representation; for example, round decimal data and display it as integer data.
- Display decorative text such as currency markers, percent signs, and commas.
- Handle several data arrays concurrently.
- Control the precision of numeric display.
- Support patterned data formats.

The format primitive ($\text{\textcircled{v}}$), sometimes called “thorn”, is familiar to many users of *APL* implementations. When used monadically, format produces a

character array whose visual appearance is identical to the original data. Thus, for character data, monadic format produces no changes; for numeric data, it is subject to system conventions.

A←3 5ρ'BLUE RED GREEN' ⋄ A

BLUE
RED
GREEN

▼A

BLUE
RED
GREEN

⊖10

1 2 3 4 5 6 7 8 9 10

ρ⊖10

10

▼⊖10

1 2 3 4 5 6 7 8 9 10

ρ▼⊖10

20

When used dyadically, format also produces character output, but allows control over the precision and spacing applied to the data. Pairs of control numbers are the left arguments to dyadic format, where each pair applies to one scalar, one element of a vector, or one column of a matrix right argument. These control numbers specify the width of the resultant formatted field and the data type and precision of the presentation.

TABLE

1.05	4.55	2.8	3.5
1.4	0.35	4.2	4.2
5.6	2.45	3.15	5.25

4 0 ▼ TABLE

Integer format.

1	5	3	4
1	0	4	4
6	2	3	5

4 1 6 2 6 3 8 4 ▼ TABLE

Fixed-point format.

1.0	4.55	2.800	3.5000
1.4	.35	4.200	4.2000
5.6	2.45	3.150	5.2500

3 ▼ TABLE

Default width.

1.050	4.550	2.800	3.500
1.400	.350	4.200	4.200
5.600	2.450	3.150	5.250

By producing character output, the format primitive provides the first step in report generation. Only one array may be passed to format at a time. Furthermore, if decorative text or titles are desired, they must be forcibly inserted into the character data result.

SALES

1401	300.2	416.3	299.5	317	245.5	247.5
1765	247.5	299.6	300.2	416.3	299.5	317
1900	416.5	444	506.6	509	511.1	499.6
2316	267.5	397.5	305.4	399.6	399.6	417.5

4 0 8 2 8 2 8 2 8 2 8 2 8 2 ▼ SALES

1401	300.20	416.30	299.50	317.00	245.50	247.50
1765	247.50	299.60	300.20	416.30	299.50	317.00
1900	416.50	444.00	506.60	509.00	511.10	499.60
2316	267.40	397.50	305.40	399.60	399.60	417.50

```

ρ 4 0 8 2 8 2 8 2 8 2 8 2 8 2 8 2 ▾ SALES
4 52
FINAL←(52↑'
FINAL←FINAL,[1] 52↑' REP    JAN    FEB    MAR    APR    MAY    JUN'
FINAL←FINAL,[1] 4 0 8 2 8 2 8 2 8 2 8 2 8 2 8 2 ▾ SALES
□TCNL ◇ FINAL

```

SIX MONTHS SALES DATA

REP	JAN	FEB	MAR	APR	MAY	JUN
1401	300.20	416.30	299.50	317.00	245.50	247.50
1765	247.50	299.60	300.20	416.30	299.50	317.00
1900	416.50	444.00	506.60	509.00	511.10	499.60
2316	267.50	397.50	305.40	399.60	399.60	417.50

On other APL systems, formatting capabilities such as alpha (α) go a few steps further than the format primitive by supporting patterned data display and some decorative text.

On STSC's APL*PLUS System, the function $\square FMT$ provides all the facilities listed above as desired features of a formatting tool. $\square FMT$ allows concurrent formatting of many arrays; allows character arrays to be formatted at the same time as numeric arrays; provides patterned data display; allows automatic handling of report titles, column headings, and row names; supports decorative text; and supports absolute and relative tabulation. $\square FMT$ is a dyadic system function whose representation is

$result \leftarrow 'formatstring' \square FMT (data1;data2;. . .;data_n)$

where *'formatstring'* contains the instructions that control the display of the data arrays and determine where special features will be invoked in the display. Each part of the format string (excluding tabs and blank spaces) applies to one scalar right argument, one element of a vector argument, or one column of a matrix argument specified in the data list. For example,

```

COLORS← 3 5ρ 'RED GREENBLUE '
COLORS
RED
GREEN
BLUE

QUANT←5
NUMBERS←3 2ρ 1.25 6.3 1.4 7.45 0.65 3.95
NUMBERS
1.25      6.3
1.4       7.45
0.65      3.95

OUTPUT←'5A1,X4,I1,X2,F3.1,X2,F4.2' □FMT (COLORS;QUANT;NUMBERS)
OUTPUT
RED      5  1.3  6.30
GREEN    1.4  7.45
BLUE     0.6  3.95

ρ OUTPUT
3 21

```

A complete description of $\square FMT$ is beyond the scope of this paper. The reader is referred to the publication entitled *Formatting in the APL*PLUS System* (STSC, 1977) for more information. Several examples showing key features of $\square FMT$ follow.

Pattern Editing:

```

PHONES← 3 1ρ3016578220 9194932478 9144286910
PHONES
3016578220
9194932478
9144286910

'G←(999) 999-9999▷' □FMT PHONES
(301) 657-8220
(919) 493-2478
(914) 428-6910
    
```

Parentheses around Negative Numbers:

```

MAT
-54.48      47.67      74.91      -68.1      -44.265
10.215     54.48      34.05     -57.885     71.505
61.29      23.835     -6.81      57.885     -27.24
51.075     61.29      78.315     30.645     20.43

'M<(▷N<)>Q< ▷F9.3' □FMT MAT
(54.480)  47.670    74.910   (68.100)  (44.265)
10.215    54.480    34.050   (57.885)  71.505
61.290    23.835    (6.810)  57.885   (27.240)
51.075    61.290    78.315   30.645   20.430
    
```

Floating Dollar Signs:

```

'M<($▷N<)>P<$▷ Q< ▷F9.2,X2' □FMT MAT
($54.48)  $47.67    $174.91   ($68.10)   ($44.26)
$10.22    $54.48    $34.05    ($57.88)   $71.51
$61.29    $23.84    ($6.81)   $57.89     ($27.24)
$51.08    $61.29    $78.32    $3.65      $20.43
    
```

Check Protection:

```

MONEY
100.45 1.53 17.99 4055.75

'R←*▷CLP←$▷F10.2' □FMT MONEY
$100.45***
$1.53*****
$17.99****
$4,055.75*
    
```

Accounting Notation:

```

'M< ▷N< CR▷Q< DR▷F10.2,X2' □FMT MAT
54.48 CR    47.67 DR    174.91 DR    68.10 CR    44.26 CR
10.22 DR    54.48 DR    34.05 DR    57.88 CR    71.51 DR
61.29 DR    23.84 DR    6.81 CR     57.89 DR    27.24 CR
51.08 DR    61.29 DR    78.32 DR     3.65 DR     20.43 DR
    
```

Workspace 1 *FORMAT* on STSC's *APL*PLUS* System contains functions that allow you to place titles and row and column names on a report. In the following example, we will format with ease the report obtained from a previous example using the functions *CENTER* and *COLNAMES* from workspace 1 *FORMAT*.

```

SALES
1401 300.2 416.3 299.5 317 245.5 247.5
1765 247.5 299.6 300.2 416.3 299.5 317
1900 416.5 444 506.6 509 511.1 499.6
2316 267.5 397.5 305.4 399.6 399.6 417.5
    
```

```

▽ R←FORM DATA;FS
[1] FS←'I4,6(X2,F6.2) '
[2] R←FS CENTER 'SIX MONTHS SALES DATA' ◇ R←R,[1] ' '
[3] R←R,[1] FS COLNAMES '◦REP◦JAN◦FEB◦MAR◦APR◦MAY◦JUN'
[4] R←R,[1] FS □FMT DATA
▽

```

□TCNL ◇ FORM SALES

SIX MONTHS SALES DATA

REP	JAN	FEB	MAR	APR	MAY	JUN
1401	300.20	416.30	299.50	317.00	245.50	247.50
1765	247.50	299.60	300.20	416.30	299.50	317.00
1900	416.50	444.00	506.60	509.00	511.10	499.60
2316	267.50	397.50	305.40	399.60	399.60	417.50

The system function □FMT is an extremely powerful formatting tool for APL technicians. Its use requires familiarity with APL data arrays, APL syntax, and □FMT commands and capabilities.

There is also a great need for formatting tools for non-programmers. These tools should allow the businessman to focus on the key elements of the desired report: the data and the verbal information. The businessman should not have to be concerned with the technical aspects of the supporting system. Two such facilities provided by STSC—QUICKPLAN™, The Quick Planning and Reporting System, and the EMMA Report Generator—are discussed in papers that appear elsewhere in this book: "QUICKPLAN: A Reporting Tool for the Non-Programmer" and "The EMMA Report Generator".

Janet Faltz started with STSC in 1974 as a marketing representative and is currently branch manager of STSC's Southeast Branch located in Chapel Hill, North Carolina. Before joining STSC she worked as an applications programmer and publications editor at the University of North Carolina Computation Center and as a management information systems analyst for Continental Can Company.

Faltz has a B.A. in mathematics from Douglass College and an M.A. in educational technology from Columbia University.

David L. Hopkins

QUICKPLAN: A Reporting Tool for the Non-Programmer

Reporting. What is it? Who does it? Most importantly, how does QUICKPLAN™, STSC's Quick Planning and Reporting System, meet the requirements of a reporting system? These are the questions I propose to answer.

Reporting is the process of organizing and presenting data or information in a useful form, with one or more purposes in mind. Reporting is done by virtually everyone, from elementary school students reporting "current events" to large corporations reporting to their shareholders. Reports are as varied as their creators and users, and can contain any combination of text, pictures, graphs, plots, and tables of numbers.

In business and government, a simple report can be generated by an analyst working with pen and paper and, perhaps, a calculator. The report so generated will probably be typed to appear more uniform and legible. Or, the analyst will take a set of specifications for a report, with a request for computer resources, to his company's data processing center.

Both of these procedures, however, have obvious drawbacks. The typist may make mistakes, and the process of correcting them is often time consuming and annoying. Requesting reports from the data processing center may be time consuming as well, as there will typically be many other demands placed on the center. Furthermore, adjustments or changes to the report specifications are more difficult to accomplish. In both cases, turnaround time may not be as quick as the report user would like.

What the businessman needs is often more than either of the above methods can provide. He needs the ability to produce reports using data that may be entered specifically for a particular report, or that may be retrieved from an existing data file. In either case, substantial data calculations or manipulations may be required to complete a report.

In other words, the businessman needs access to all sorts of data, and he needs to be able to work with that data. He wants the capability to present the data in many different, and sometimes unexpected, formats. And, most importantly, he wants the resulting reports available in a timely manner. A system that meets these requirements is more than a reporting mechanism—it is a tool with which the businessman increases his productivity and the accuracy of his decisions.

If this is all within reach by using high-speed computers (and it is), then why doesn't everyone with access to a computer perform his own report generation? The answer to this question is not the lack of native intelligence

on the part of the average user; the answer is, however, related to intelligence. That intelligence is the method of accomplishing the task.

In a batch computer environment, the user may find it necessary to learn such programming languages as COBOL, FORTRAN, or PL/1. In addition, he has to contend with the detailed mechanics of how information is entered into the computer (on punched cards, for example). He is also subject to the turnaround time. Even in a time sharing environment, languages such as *APL* prove to be too complex for the businessman to use directly. Too often, a businessman finds it necessary to take the additional time necessary to learn and use commands and symbols that bear little relation to the finished report.

Let's look at a typical, though simplified, business report (see Figure 1). Though reports have widely varying formats, this report has a common format. It presents numeric data in rows across the page and columns down the page. The rows and columns have labels, and the report has titles at the top and comments at the bottom.

CAPITAL COMPARISON FOR MIDWEST BANKS			
BANK	CAPITAL TO ASSETS	CAPITAL (\$ MILLION)	DEBT TO CAPITAL %
BIG BANK	4.4	501.0	25.0
FRED'S BANK *	4.6	1,903.0	22.4
BANCO DEL ORO BANK	5.1	40.0	15.0
WHY BANK *	5.2	54.0	24.2
FAST BANK	5.3	217.0	25.4
* AS OF MARCH 31, 1979.			

Figure 1—A Typical Report

Ideally, the user who wishes to create a report like that shown in Figure 1 should be concerned only with describing the key components and specifying the order in which those components should appear. This ideal situation is rarely the case, though, as most businessmen do not have the proper tools. QUICKPLAN was developed to provide the tools. Its English style commands, simplifying assumptions, and full database interaction make it a natural tool for reporting.

To see firsthand how QUICKPLAN meets the requirements of a useful reporting tool, we will use the system to create the report shown in Figure 1. After signing on to STSC's *APL*PLUS* System, we access the QUICKPLAN System and the tools it provides:

```
)LOAD 333 QUICKPLAN
SAVED . . .
```

Next, we create a filing area called *QUICK* for the Report Generating System (RGS) and its data and programs.

```
GPCREATE
G/P SYSTEM NAME? QUICK
9999999 QUICK CREATED.
```

Now we're ready to create the RGS, which we'll call *BANK*. The RGS is used to store information for report titles, headings, line names, line numbers, and data.

BUILDRGS

RGS NAME: **BANK**
 HOW MANY COLUMNS?
:
 3
 BANK CREATED.

With these steps completed, we are ready to enter the specific information for our report. It is important to note that the above steps are necessary only when setting up a report for the first time. Many reports can be generated using the same "file" (*QUICK*) and the same RGS (*BANK*).

Next, let's enter the character information for our report (titles, column headings, and line names). For each item, we must specify a number (used for later reference), the justification (left, right, or centered), and, of course, a name. Line names also offer two additional options for the data that will appear in each line—formats and scale factors—but we will not use these options.

ENTERTITLES *First we enter the titles.*

RGS NAME: **BANK**
 DEFAULT: LJUST, CENTER, RJUST **C**
 SEQUENTIAL? **NO**
 ENTER TITLE NUMBER
:
 12
 12: |**CAPITAL COMPARISON FOR MIDWEST BANKS**
 ENTER TITLE NUMBER
:
 END
 MORE? **NO**
 TITLE NAMES STORED.

ENTERHEADINGS *Next we enter the headings.*

RGS NAME: **BANK**
 ITEMS TO BE ENTERED: NAME, FORMAT, SCALE: **NAME**
 DEFAULT - LJUST, CENTER, RJUST: **R**
 USE <, n, OR > TO OVERRIDE DEFAULT. USE < FOR NEW LINE
 SEQUENTIAL? **NO**
 ENTER HEADING NUMBER
:
 0
 0 NAME: |**C**BANK<-----
 ENTER HEADING NUMBER
:
 1
 1 NAME: |**CAPITAL TO ASSETS**<-----
 ENTER HEADING NUMBER
:
 2
 2 NAME: |**CAPITAL**<(\$MILLION)<-----
 ENTER HEADING NUMBER
:
 3
 3 NAME: |**DEBT TO CAPITAL**< % / % <-----
 ENTER HEADING NUMBER
:
 END
 MORE? **NO**
 HEADINGS STORED

ENTERLINES *And, finally, we enter the line names.*

RGS NAME: **BANK**
 ITEMS TO BE ENTERED: NAME, FORMAT, SCALE: **NAME**
 SEQUENTIAL? **NO**
 ENTER LINE NUMBER
:
 10
 10 NAME: |**BIG BANK**

```

ENTER LINE NUMBER
□:
    12
12 NAME: | FRED'S BANK*
ENTER LINE NUMBER
□:
    20
20 NAME: | BANCO DEL ORO BANK
ENTER LINE NUMBER
□:
    24
24 NAME: | WHY BANK*
ENTER LINE NUMBER
□:
    31
31 NAME: | FAST BANK
ENTER LINE NUMBER
□:
    END
MORE? NO
LINE NAMES STORED.

```

Now we can try a "first cut" at our report. All we need to do is enter a simple program that specifies the order in which the components should be printed:

```

VREPORT
[1] FIELDS 26 18 18 18  AFIELD WIDTH FOR PRINTING COLUMNS
[2] TITLES 12 0 0      APRINT TITLES 12 0 0 (0 PRODUCES A BLANK LINE)
[3] HEADINGS 0 1 2 3  AOVER THE COLUMNS, PRINT HEADINGS 0 1 2 3
[4] LINES 0 THRU 99   APRINT VALID LINES IN THE RANGE 0 TO 99
[5] V

```

We run our report program at this point to check the format of the report (the resulting "report" is shown in Figure 2):

REPORT

CAPITAL COMPARISON FOR MIDWEST BANKS

<u>BANK</u>	<u>CAPITAL TO ASSETS</u>	<u>CAPITAL</u> <u>(\$MILLION)</u>	<u>DEBT TO CAPITAL</u> <u>%</u>
BIG BANK	0	0	0
FRED'S BANK*	0	0	0
BANCO DEL ORO BANK	0	0	0
WHY BANK*	0	0	0
FAST BANK	0	0	0

Figure 2—Checking the Report Format

If the report format is correct, we can begin to enter the data. A simple data input program like the one given below shows us what data must be entered.

```

VINPUT
[1] GETSYSPGM 'PENTERDATA'
[2] 1 2 3 PENTERDATA 0 THRU 99
[3] A ENTER DATA IN COLS. 1 2 3 FOR LINES 0 THRU 99
[4] V

```

We run the input program and enter the appropriate data:

```

INPUT
10: BIG BANK = 0 0 0
□:
    4.4 501 25

```

```

12: FRED'S BANK* = 0 0 0
□:
    4.6 1903 22.4
20: BANCO DEL ORO BANK = 0 0 0
□:
    5.1 401 15
24: WHY BANK* = 0 0 0
□:
    5.2 540 24.2
31: FAST BANK = 0 0 0
□:
    5.3 217 25.4

```

We modify our report program slightly to add cosmetic additions such as spacing and comments.

```

▽REPORT1
[1] 'ALIGN PAPER'◊PAUSE  a STOP TO LET USER ALIGN PAPER
[2] FIELDS 26 18 18 18
[3] FORMAT '1'  a SHOW ONE DECIMAL PLACE
[4] TITLES 12 0 0
[5] HEADINGS 0 1 2 3
[6] LINES S,(0 THRU 99),S,S  a 'S' GIVES BLANK LINE
[7] COMMENT'* AS OF MARCH 31, 1979.'
[8] ▽

```

Finally, we run the modified report program, and we have a finished report as shown in Figure 3.

REPORT1

CAPITAL COMPARISON FOR MIDWEST BANKS

<u>BANK</u>	<u>CAPITAL TO ASSETS</u>	<u>CAPITAL</u> <u>(\$MILLION)</u>	<u>DEBT TO CAPITAL</u> <u>°/°</u>
BIG BANK	4.4	501.0	25.0
FRED'S BANK*	4.6	1,903.0	22.4
BANCO DEL ORO BANK	5.1	401.0	15.0
WHY BANK*	5.2	540.0	24.2
FAST BANK	5.3	217.0	25.4

* AS OF MARCH 31, 1979.

Figure 3—The Finished QUICKPLAN Report

QUICKPLAN can do much more than produce reports, since it contains its own database manager called the GET/PUT facility. Data is stored with PUT commands and retrieved with GET commands. The user need not be concerned with the structure of files; he addresses all data items by names that he has chosen. GET/PUT databases can be shared, and many people can simultaneously put data into a database or retrieve data from it.

To expand our example, let's assume that one QUICKPLAN GET/PUT database contains data about banks. It might contain all the operating data on every bank in the United States, or in a specific state. Once this database exists, the user can select data that meets any given criteria. In our example the report was for midwest banks, but we could as easily have selected data for another region or for banks with greater than a specified capital level. If data were stored in the database by year, we would have yet another dimension in our database. The user could then produce reports for specified time periods.

Having selected any subset of the stored data, the user can perform calculations and produce reports, or put the calculated data back in the

database for later access. The possibilities for manipulating and reporting data become endless. Better yet, these possibilities are all within the reach of QUICKPLAN and its database system.

Conclusion

Stepping back from the mechanics of QUICKPLAN, let's repeat the necessary elements of a complete reporting system. We can then decide whether QUICKPLAN meets these requirements.

- The reporting system must be clear, concise, and unambiguous.
- It must contain all the necessary commands, including the selection criteria, to facilitate interaction between the user and the databases in which relevant data is stored.
- The user should not be asked to deal directly with the underlying programming language (in our case, *APL*). That is, all error messages and data manipulations should be handled by the user-oriented language of the reporting system.
- The system should be column or line oriented, or both, and should provide headings for columns and lines.
- Numbers should be presented, by default, with standard business notation (using dollar signs, commas, parentheses, and decimal points).
- The system should prevent the user from doing harm to a database.
- The system should allow the user (and the database manager) to change anything he has done—easily and quickly.
- Finally, the reporting system should be compatible with other systems written in the same language so that the systems can be easily linked.

When these requirements are met, the reporting system will best meet the needs of the businessman. It will also be a useful tool for programmers.

QUICKPLAN does, in fact, meet these requirements. Furthermore, one can easily learn to use QUICKPLAN. In one day, the average person can master QUICKPLAN's reporting capabilities. Little additional time is required to learn the database management capabilities.

Let's return to the original questions. It is fairly obvious what reporting is. With QUICKPLAN, the answer to "Who does it?" is "Anyone with a need for reports, a few minutes, and access to a terminal". Does that sound too simple? If so, that's because QUICKPLAN makes business reporting so simple that most users can master the system in a day.

Dave Hopkins earned his B.S. in computing and information science at Trinity University in Texas, where he worked part-time for two years at the university's computing center. After receiving his M.B.A. from Southern Methodist University, Hopkins joined STSC in 1978 as an applications consultant.

Hopkins has developed many customized QUICKPLAN reporting systems for STSC customers, particularly for major oil and energy producing companies located in Houston.

Robert R. DeCloss

The *EMMA* Report Generator

In physics we learn that an “erg” is a unit of energy or work. The ERG System (*EMMA*™ Report Generator System) allows a user to define and obtain numerous reports easily and quickly, making his work more productive, and saving him the time and effort spent otherwise collecting that data.

ERG is a system designed for non-programmers who, with a surprisingly small “vocabulary”, can generate virtually unlimited reports in formats they specify. For that reason, ERG is particularly useful to management. Management’s reporting requirements vary almost daily. Since ERG requires no programming, an executive can define reports and have them in minutes.

The ERG System operates from an *EMMA* file. *EMMA* (Extended Management Macros in *APL*) is a proprietary collection of programs developed by STSC to manipulate, select, replace, and compare data. In contrast to *EMMA*, ERG has only a few user programs, which I will explain a bit later. First, I would like to share some of the history of ERG’s development.

ERG was the result of a great design process; we spent *hours* considering alternatives and options. Here is how it all began:

One of our clients needed many different kinds of reports for its management. They had been using PERT*PLUS, STSC’s Interactive Project Management System, but they had come to realize that the system was solving only parts of their problem at an expensive price. One Friday afternoon, they gave us a list of the capabilities they wanted. For example, they wanted to be able to print different columns and have data paged for easy separation and distribution to different departments. They also wanted the ability to total columns of data based on a major category and to subtotal based on a subcategory.

From this “wish list” we had a better idea of what the client’s needs were. When we went to see the client on Monday, we proved that we had not only satisfied their needs, but had also added the capability for the user to format his own reports.

After the client registered mild shock at the speed with which we had solved their problem, we asked if our design was acceptable. It was; we were off and running; and ERG was born!

I suppose I could be accused of heresy, but I believe a majority of products and packages are developed in this way—at least a majority of useful products. *APL* is the only language I’m aware of that will let you accomplish what I’ve just described. With *APL* you can have results within a couple of hours of

receiving ideas from a client. The client reviews the initial design, and usually thinks of additional requirements. After input and suggestions from both sides, you go back and add some “features” (a technical term for “bells and whistles”). Soon you’ve met the client’s requirements, and probably given him much more!

The best way to substantiate my claims of the value and simplicity of ERG is to describe its characteristics.

The current version of ERG (others are under development) works from a single *EMMA* file. It can be installed quite easily by anyone who understands the basic concepts and nomenclature of the system.

That brings me to an extremely valuable overall design consideration in any product or package I develop: ease of use. This may be an overused phrase, but I *do* go to great lengths to avoid computer jargon and to implement ideas that fit the customer’s business, *not* the computer business. But, I digress.

To return to ERG, five of the main user programs are

- *REPORT*—A conversational program that asks what fields to print, how to sort the data, what fields to total, where to put page breaks, and what field to break on within a page.
- *SUMMARY*—A program similar to *REPORT*. The only difference is that the data is summarized and no detail information is printed. Break totals and page totals are printed on the report.
- *PRINT*—A program that prints a previously created report. The report specified can be printed at a terminal or can be submitted for printing on a remote, high-speed printer.
- *DIRECTORY*—A program that displays all report files currently existing.
- *ERASE*—A program that erases a report file.

To store the report information, the user fills out a worksheet and enters the information into the system. Editing features allow the user to set or change formats or column headings very easily. Once satisfied with the headings, formats, tables, and names, the user saves the information. A variety of reports are now ready for production, waiting only for a request from the user.

In generating reports, a user must become familiar with the following eight concepts.

- print fields
- sort fields
- page break field
- page total fields
- break field
- break total fields
- selection criteria
- report name.

Most of these concepts are relatively straightforward and become second nature quickly.

The *print fields* specify the columns of data the user wishes to display in the report. The *sort fields* indicate how the printed fields should be sorted, in major to minor order. The sort fields *do not* have to be included in the print fields.

The *page break field* is used to force a new page every time the specified column of data, after being sorted, changes. The *page total fields* allow the user to specify which columns of data are to be totaled before each page break. (If no page breaks are specified, the user is prompted for grand total fields rather than page total fields.)

The *break field* allows a subtotal within a page break field. Thus, a user can get branch totals within each cost center or subtask totals within each task. The *break total fields* allow a user to select which columns of printed fields are to be subtotaled. Page total fields can be different from break total fields.

Now, for the only slightly complex part of the whole system—selection criteria. The *selection criteria* specify what data is to be printed, using abbreviations of English words such as: *FROM*, *BETW*, *EQ*, *GT* (greater than), *AND*, and *OR*.

Parentheses can be used to alter a definition or form complex statements. For example:

```
SELECTION CRITERIA: (SAL BETW 1000 2000) AND EARN GT 100000
```

For experienced *APL* programmers, the same selection criteria can be specified using raw *APL*:

```
SELECTION CRITERIA: (SAL>1000)^(SAL<2000)^EARN>100000
```

Let's consider an example. Suppose we want to generate a report that prints the cost center, branch number, employee number and name, employee salary, and revenue generated by each employee. We want to sort it by cost center and by branch within cost center, and we want only cost centers less than 400. We are only interested in the top producers, so we want only those who have generated year-to-date revenues in excess of \$130,000.

Here is all we do:

```

)LOAD 99999999 ERGDEMO
SAVED . . .
REPORT
PRINT FIELDS: HELP
VALID FIELD NAMES AND NUMBERS ARE LISTED BELOW:
(NOTE: WHEN USING NAMES YOU MUST USE A COMMA AS A SEPARATOR).
CC=1 , BR=2 , EMP=3 , NAME=4 , SAL=5 , DATE=6 , EARN=7 , STATE=8

PRINT FIELDS: CC,BR,EMP,NAME,SAL,EARN
SORT FIELDS: CC,BR
PAGE BREAK FIELD: SKIP
GRAND TOTAL FIELDS: SAL,EARN
BREAK FIELD: CC
BREAK TOTAL FIELDS: SAL, EARN
SELECTION CRITERIA: (CC LT 400) AND (SAL FROM 1000 3000) AND EARN GT 130000
REPORT NAME: MYREPORT
REPORT TITLE: TOP PERFORMERS FISCAL YEAR 1980
FIRST SUBTITLE: (SPACE,RETURN)
DONE

```

After entering the report specifications in this manner, you can run the program *PRINT* and produce the report, as shown in Figure 1. Using slightly different report specifications, and other *ERG* options not demonstrated here, you can produce many variations of this basic report.

I can't stress enough how easy it is to use *ERG*. The user does not have to learn a programming language, nor does he have to figure out what a work file is or even how to create one. All the user has to know is what data should appear on the report, and in what order. This gives each user more time to review reports containing exactly the information he wants to see—no more and no less. Each report is customized so that the user sees only the columns

and headings pertinent to him. Managers don't get detailed reports crowded with data that doesn't interest them, or worse, annoys them.

OFFICE PRODUCTS SERVICE COMPANY
TOP PERFORMERS FISCAL YEAR 1980

PAGE 1; 2/19/80

<i>COST CENTER</i>	<i>BRANCH NUMBER</i>	<i>EMPLOYEE NUMBER</i>	<i>EMPLOYEE NAME</i>	<i>EMPLOYEE SALARY</i>	<i>DOLLARS EARNED</i>
311	1	1137	FLANIGAN, JOAN	2,900	136,500
311	3	1069	RYAN, KAREN E.	1,600	133,500
311	3	1087	ESKINAZI, KEVIN	1,600	147,500
				6,100	417,500
332	4	1107	GURGOLD, JOHN	2,800	144,000
				2,800	144,000
341	1	1115	DAAR, ARLENE	2,700	138,500
341	1	1007	CARTER, CLIF	1,000	137,500
341	1	1122	CHANDLER, JAK	2,700	138,000
341	1	1108	WEAVER, JEFF S.	2,500	143,500
341	3	1028	KARPF, ELEANOR T.	1,300	133,000
341	3	1098	KRANISH, RON S.	1,200	132,000
				11,400	822,500
				20,300	1,384,000

Figure 1—A Sample ERG Report

The combination of ERG and APL provides managers with a powerful tool for producing reports that meet their requirements, even if those requirements change daily. Furthermore, valuable information is provided on time and at a very reasonable price. What better way to work? That is the key to ERG—making work better and easier.

Bob DeCloss joined STSC in 1973 as a programmer. He took a leave of absence in 1975 to become treasurer of the Irwin Trading Company and Irwin Management Company, but later in 1975 rejoined STSC in the APL Development Department. Since 1978 he has been the branch manager of STSC's Denver office.

DeCloss co-authored with Roy A. Sykes, Jr., a paper for the APL 75 Conference in Pisa, Italy, titled "EMMA : Extended Management Macros in APL" (APL75 Conference Proceedings, ACM, 1975). In 1977 he wrote the EMMA Reference Manual (STSC, 1978). He has designed and implemented several systems dealing with report generation, database management, and construction accounting.

DeCloss has an M.A. in mathematics from Claremont Graduate School.

Richard W. Butterworth

When *APL* Is Inappropriate

The use of *APL*, or any high-level programming language, can be inappropriate for a given application, particularly when the use of a different language offers a lower *overall* cost to achieve the objectives of the user and the application. From this somewhat over-simplified beginning, we will discuss several issues that help determine when *APL* (or any high-level language) is appropriate. These issues frequently become the deciding factors in choosing a programming language.

Of course, there are few, if any, hard and fast rules in a field whose technology is changing so rapidly. The following discussion and examples will provide some principles that can be examined when resolving the issue of whether *APL* is appropriate.

Program Readability

The primary issue in language choice is *program readability*. For purposes of this discussion, program readability refers to the inherent difficulty and associated costs of reading a program, determining precisely what the programmer intended the program to do, and executing the program. This issue includes not only "people" readability (a person learning the input, process, and output characteristics), but also "machine" readability (the initial interpretation of the program into machine-executable code and subsequent executions of the program). As might be expected, high-level languages favor people readability, while low-level languages favor machine readability.

The issue of readability fits naturally into the concept of a program's total life-cycle cost. A program's life-cycle cost can be viewed as the sum of its people readability costs and its machine readability costs. The former might be measured in manhours, and the latter in machine cycles. These two costs are determined by the amount of "reading" the program is subject to over its life and the associated costs. Hence, a program that is read primarily by machines, such as a system utility, will have life-cycle costs dominated by machine readability costs. Conversely, a program read primarily by people, such as a program for quantitative support of management decisions, will have life-cycle costs dominated by the people readability costs. Proper language selection, then, entails knowing the program's intended use and environment and selecting the language that minimizes the composite costs.

Foretelling total people readability costs is a subjective process. People readability costs, for example, include not only the original programmer's time, but also the time of subsequent persons (programmers and users) who

need to understand the program from a conceptual, technical, or operational viewpoint. These costs are usually underestimated.

People readability costs also contribute heavily to a program's development cycle time, since during the development phase a program is read almost exclusively by people. In a linguistic sense, as with high-level spoken languages, a high-level programming language greatly assists the communication of ideas among designers and programmers, thereby reducing the time it takes to understand the problem. This asset, and the ease of translating the algorithmic concepts into executable programming statements, shortens development time significantly. This suggests that some programs requiring short development cycles may be infeasible unless undertaken in a high-level language.

Machine readability costs are equally difficult to estimate. The primary factors are the cost of machine resources and the execution cost of the program. Though CPU power is becoming increasingly cheap, additional machine cycles cannot always be purchased. For example, many products today contain dedicated microcomputers or minicomputers; the application environment is precisely defined and the CPU power restricted. Another example sometimes occurs with the Federal Government, which occasionally chooses low-level languages for their ability to conserve CPU resources made scarce not by financial or architectural restrictions, but rather by an extremely slow procurement process. Thus, machine readability, like people readability, may be constrained by a variety of cost parameters. These parameters, which include opportunity costs as well as actual costs incurred, usually have highly subjective attributes.

Readability and the Performance Issue

The life-cycle costs of computer applications are then a combination of the reading costs incurred by people and machines. To minimize expected total cost, a compromise is usually sought between costs of software development and maintenance, and costs of hardware capacity and sophistication. *APL*, with its natural ability to manipulate data arrays and its user-oriented implementations, offers distinct benefits in reducing software development costs. Predictably, *APL* is most effective in applications such as modeling systems and decision support systems, where software development costs are highest. In these systems, primary design criteria are ease of use, ease of program adaptability, a short development cycle, and frequently, a customized approach.

Today, however, a majority of computer applications are characterized by repetitive processes. Programs that are changed infrequently and that do not require interactive processing are less able to take advantage of *APL*'s assets. Moreover, on most implementations, the interactive interpreter makes it difficult to move these processes out of the very busy (and, consequently, the most expensive) prime operating time.

The current trend toward distributed processing provides another example where programs are duplicated to run at many locations or on many machines at the same location. This reduces software cost per unit hardware cost, and creates leverage for low-level languages. Take, for example, the system supporting a large retailer's point-of-sale terminals. Hardware costs are amplified many times by the number of terminals involved, but software, once written, is likely to remain very stable. It is clear from this discussion that the language issue cannot be decided outside the application's context.

In current machine architecture, which requires that a programmer's code be reduced to common machine code before execution, the performance

requirements of a program can become a major issue in language selection. Some programs are read primarily by machines; programmers read the code only for development and maintenance and the end users do not read it at all. In these cases, machine performance is a main concern, and a low-level language is clearly preferred. Examples of this are utilities such as file sorts, random number generators, and internal checks and balances.

A different example involves a scientific simulation project. The operation and maintenance of emergency diesel generators at nuclear power plants were simulated to study the effect of the maintenance and test plan on the generators' reliability and availability. Though extensive documentation of the program was not required, people readability was an issue, since the model being simulated was not completely specified and was likely to undergo change.

These factors would appear to indicate that *APL* would be a good programming language for the project. However, performance became the deciding issue, as the precision of the results depended heavily on multiple replications over long horizons. A few short programs had to be executed hundreds of thousands of times to evaluate each scenario. Consequently, the associated machine costs became a limiting factor to the system's use. FORTRAN was finally chosen as the programming language for the project; the random number generator was provided by the host compiler, probably in Assembler language. The complexity of the model precluded any low-level language approach, due to the people readability issue.

A secondary issue in the simulation project was machine portability (i.e., the ability to move and maintain an application on two or more computer systems). The portability requirement, while essentially a machine readability issue, is usually resolved by using a high-level language. In this case, *APL* was not available on the alternate system, but FORTRAN was. The FORTRAN simulation program, when moved to the alternate installation, ran and duplicated earlier results with a change to only one line of source code.

Another type of problem not generally handled in *APL* is the linear programming (LP) problem. Linear programs are comprised of "matrix generators" that develop application data in a canonical LP format. The LP problem, an optimization of a linear function of independent variables subject to linear constraints, is then solved using the simplex algorithm, or some variation of the simplex technique. Finally, the solution is translated back into the terms of the problem and output reports are produced.

Although LP problems are characterized by matrix data structures, only relatively small problems (those having 50 or fewer equations) seem suited to an *APL* approach due to the nonlinear increase in iterative computations. Even when an LP problem is small, other factors must also be present to suggest an *APL* solution (e.g., a fluid problem definition requiring constant revision).

Readability and the Intelligence Issue

As the prerequisite intelligence of a program increases, so does the need for increased people readability. (The word "intelligence" in this paper refers to a program's ability to handle complex sets of logical rules and to deal gracefully with unanticipated input.) Complex ideas are difficult to communicate with the limited "vocabulary" of low-level languages and are nearly impossible to grasp by reading a program written in a low-level language. Just as high program intelligence generally supports the use of a high-level language, low program intelligence generally favors use of a low-level language (i.e., the non-*APL* solution).

An example of a low-intelligence task that can become expensive when undertaken in *APL* is large-volume record processing to update and maintain a database. Large personnel systems, for example, usually maintain a record on each individual, perhaps in a sequential dataset consisting of many tapes. The database must be updated weekly, or even daily, and update reports must be produced. The procedures for performing such updates are relatively simple; consequently, the matrix manipulation capabilities of *APL* would be largely under utilized. The machine expense associated with constant interpretation of the code to simply process the next record creates a situation in which the system's overall cost is dominated by the machine readability of the program.

It is worth noting, however, that non-*APL* programs can build *APL* databases that can subsequently serve many information needs. These databases generally condense entity data into frequency of occurrence data that summarizes the activity. Such databases can become excellent sources of top-down management information, supporting the increasingly popular decision support systems (DSS).

In our experience, two personnel systems have been implemented in this manner. One tracks 30,000 persons and the other tracks over 500,000 individuals. In both systems, *APL* was judged inappropriate for the database development because of the large number of record manipulations required. However, management models that used the data *were* developed in *APL* specifically to obtain ease of model development and flexibility. This illustrates that large systems can easily contain applications appropriate for both non-*APL* and *APL* programs.

In fact, it is typical in large application systems for *APL* to be appropriate for some programming tasks, but not for others. One such case is a flight-routing system called OPARS (Optimum Path Air-Routing System). OPARS provides flight plans on a production basis for a subset of the Navy's flight community. Flight plans are requested a few hours before flight time by naval weather personnel or flight personnel, but seldom by computer personnel. To request a flight plan, the user responds to a few questions in an interactive terminal session. The result of the session is a request file, which is forwarded to a batch input queue. The flight plan is printed at the user's terminal five to ten minutes later, and can be revised if necessary before takeoff.

The flight plan consists of an optimum routing from point of departure to point of arrival. The criterion for optimality is minimum fuel consumption, which may be subject to user constraints such as mandatory fly-overs or fly-arounds, or use or nonuse of FAA jet routes. Wind and temperature forecasts from real-time databases are used to develop a dynamic network to which a shortest path branch-and-bound algorithm is applied. The final result is a formatted flight plan showing the suggested routing, expected fuel consumption, forecasted winds enroute, and a checkpoint schedule.

Because of the machine performance issue, *APL* was inappropriate for the production version of this shortest path network optimization program. The nature of the network algorithm, which sequentially examines arcs for potential inclusion in the shortest path, precluded the use of "matrix-type" calculations, and suggested a "looping" design instead. However, *APL* was used during the design effort to test the network and algorithm design concepts by developing a prototype program for the optimization. The *APL* program contributed to a proof of concept, but was relatively inefficient for repetitive execution of the algorithm in a large-scale production environment.

APL was also found to be inappropriate for implementing the main flight-routing program, which begins with the request input file and terminates with a formatted flight plan file. In spite of the complexity of the program, the

performance issue was overriding. The number of flight plans to be prepared daily could not be forecasted; however, as system activity increased, the number was expected to grow from between 10 and 20 plans to between 100 and 500 plans. Given the turnaround requirement of 10 minutes, the application required a language that kept machine performance high and machine readability costs low. The languages selected were FORTRAN (for the flight routing) and Assembler (for the input/output operations).

The interactive request generator, though developed in FORTRAN because of institutional constraints, was a suitable candidate for *APL* implementation. This interface had to be interactive and “user friendly”, had to handle sparse amounts of data, and had to have a fair amount of intelligence to determine whether requests were well formed and complete. Requirements for this program were expected to change as system capabilities were added or temporarily suspended—another factor favoring the use of *APL*.

Conclusion

The readability theme of this presentation focuses on the reading and interpretation costs of programs. Readability costs are accumulated by machines, in machine cycles, and by people, in manhours. The key to resolving the language issue is to look at the relative costs of having the program read by people and by machines. If the life-cycle program costs will be dominated by machine costs, *APL* may not be the best language choice, in spite of its more productive use of people’s time.

Language choices are still likely to be somewhat subjective, however, as factors affecting hardware costs and personnel costs continually change. Examples of factors directly influencing the choice of *APL* are the decreasing costs of hardware, the continual improvements to the *APL* interpreter, and the possibilities of bringing native hardware operations structurally closer to *APL* primitive operations. In the foreseeable future, however, many situations will arise where compelling cases can be made for low-level language approaches. The best solutions, *APL* or not, will capitalize on the assets of both high- and low-level languages, in composite solutions.

Some points discussed in the examples are summarized as follows:

- General support utilities, such as file sorts, are good candidates for non-*APL* implementation.
- Simple tasks that require little intelligence and emphasize high-volume processing (e.g., sequential record updates) are generally not recommended for *APL* implementation.
- Real-time performance requirements of complex tasks may not permit a responsive solution with *APL*. These applications tend to become “expensive” to implement because of a somewhat constrained language choice.
- Portability, which usually favors the high-level language approach, can work against *APL* until such time as *APL* is implemented on a wider range of machines.
- Hybrid solutions offer numerous benefits. While *APL* has many attractive features, there are valid and compelling reasons to select other languages for certain system segments to complement the segments written in *APL*.

Richard Butterworth, technical director of the Advanced Analytical Applications Division of SEI, is experienced in operations research, with specific interests and applications including military manpower analysis, energy systems reliability, statistical time-series forecasting, and interactive decision support systems. At SEI he led the development of DELIS, the Navy's Executive Level Information System, and OPARS, a global Navy flight-routing system.

Prior to joining SEI, Butterworth was associate professor of operations research at the Naval Postgraduate School, where he developed a new course in interactive computing. He holds a Ph.D. in operations research from the University of California at Berkeley.

Thomas A. Gull

Managing Outside Computer Services: An Organizational Relationship

The Basic Relationship

If you are a time sharing coordinator or an important user of an outside computer service, your organization holds you responsible for meeting certain objectives. These objectives may be quite specific, or they may be very general. In either case, you are responsible for using resources to meet those objectives, and your management will examine the difference between the costs of resources used and the value derived from those resources.

The resources allocated to you may include budgeted funds, personnel, supplies, and time. Availability of these resources may give you the option of buying various services from another organization. When you use an outside computer service, you are going to build a business relationship between two organizations, and that relationship must be carefully managed.

In real life, of course, the "relationship" between two organizations really can be thought of as some function of all the personal relationships between members of the organizations. These personal relationships, however, work well when there is general agreement on why the people interact on a continuing basis. This agreement occurs when goals have been set by the leaders of each organization, the goals are supported by members of both organizations, and everyone can see that the personal interactions help meet those goals. In effect, this is a "business relationship" regardless of the goal of either organization.

Simply stated, your organization is going to get services and pay for them in some way. You, personally, are not receiving or paying for services; by the same logic, no one person in the vendor organization is serving you and being paid for it. Your understanding of the distinctions between organizational relationships and personal relationships has a huge effect on the quality of the service you will receive.

Identifying the Common Ground

Whether a vendor is a profit or non-profit organization, goals will have been defined for the entire organization. When you first choose an outside computer service, spend some time ferreting out those corporate goals. For a vendor shooting for monetary profit, the goals may be well publicized and easy to understand. A non-profit vendor may survive with vague goals, though some research or service groups have very clear objectives. You do not know if you can get service from a vendor until you know what they want in trade for that service. As harsh as it may sound, your goodwill by itself may not keep you on

as a client of a service having difficulty meeting its goals. In other words, getting services free sets up an unbalanced relationship in which you will have little or no influence when circumstances change. You will get what you pay for.

When you know what motivates the vendor to give you service, you can examine your available resources to see if you can afford the service you need. Most vendors will prefer to deal with organizations that can contribute to meeting goals at some significant level. For example, getting service from a giant firm can be a problem if you would be its smallest client. You should be realistic about what you expect to spend on computer services, since this will give your vendor an honest picture of what resources to set aside for your use. Your estimates of what you will need will be used by the vendor to manage the allocation of his resources; bad estimates may contribute heavily to bad management in either organization, perhaps increasing your costs or even preventing you from meeting your objectives.

Common ground for any business relationship has four main components:

1. What services does your organization need?
2. What can you pay for those services?
3. Can the vendor provide those services?
4. Are you giving the vendor enough incentive to provide those services?

This approach implies that you will receive the most useful service when you and the vendor can both (1) meet your respective goals, and (2) avoid using resources unnecessarily. If you have identified the common ground correctly, you should be able to predict what level of service you will receive from any vendor.

Communications with the Vendor

Your vendor will assign one or more persons to work with you and your organization. Your basic communications with the vendor organization will go through this assigned person, so you must be able to deal effectively with this representative of the vendor. You will receive the best service when both of you take the time to identify the "common ground", and this process requires honesty and skill. If you have trouble dealing with the assigned person, ask for another representative.

As you work with your representative, be aware of what organizational goals he has been given. In a company like STSC, for example, the marketing representative has been hired to increase revenue at a reasonable expense. That is the focus around which decisions will be made. However, the manner in which goals are met will vary widely from vendor to vendor. For example, one vendor may sell only machine time and provide no other services. Another vendor may provide machine time, consulting, educational programs, and so forth.

The overall quality of a computer service is actually more dependent on the manner in which the service is delivered than on the hard economics of the vendor's goals. To have a complete picture of what service a vendor will deliver to you, you should consider not only the vendor's goals, but also his business philosophy and his level of involvement with the customer.

Some vendors concentrate on high business volume mixed with low personal service. Their favored customers will probably not interact much with the marketing representatives. Other vendors may concentrate on medium level volume coupled with high personal service. In the former case,

vendors will not have many highly trained support personnel available, since personalized service is discouraged. In the latter case, there will be support personnel spread throughout the organization.

The presence or absence of support personnel and a listing of the resources available to a marketing representative, and, therefore, to a customer, can outline a company's basic business philosophy very clearly; these factors are a direct communication to you describing how a vendor intends to meet his goals. If his intentions don't match your needs, look for another vendor.

When you work with a marketing representative for an *APL*-based service company, there is a subtle trap both you and the marketing representative can fall into. In some cases, particularly when using *APL*, your representative may have enough technical skill to solve many of your problems directly. This may give you the impression that the representative can, and should, personally solve all your technical problems. In practice, it is better to hold the representative responsible for obtaining the resources you need, without necessarily being the resource himself. This distinction in attitude helps to ensure that the business relationship between your two organizations is not overdependent on the skill and goodwill of one or two individuals.

Overdependence on one person seems to be fairly common in the *APL* environment. The productivity of each *APL* analyst is very high compared to, say, a FORTRAN or COBOL analyst, and many projects are completed from start to finish by only one person. If you, as the user, deal only with that person, you are encouraging bad habits in your vendor that may ultimately hurt both organizations.

For example, if you have a vendor build a general ledger system in *APL*, it may take only one person to do the job. During development of the system, communications about concerns will go to that person. When the system is completed, a crucial moment occurs. If you continue to call the developer when something needs changing or fixing, none of the other vendor personnel will gain experience with the project. The net effect is that you will get quick service only as long as the developer remains available; should he move on, your vendor may lose the ability to serve you easily on that project.

The average analyst using *APL* works so quickly that customers are usually convinced that the analyst assigned to their project is uniquely competent, and so the customer prefers to be served by that analyst. Since many customers may feel this way, the cumulative effect upon the analyst can be devastating. Actually, a professional analyst works in such a way that another analyst could easily deal with many of the questions asked about a project. It is part of the analyst's job to complete projects with advice from other analysts, with the expectation that others will modify or support the project in the future. If there is such interaction, the vendor organization is able to give you good service even if a particular analyst is on vacation or has switched jobs.

For example, the Washington, D.C., marketing branch of STSC handles customer needs by assigning a different person each day to screen incoming phone calls. Problems taking up to roughly 20 minutes to solve are handled directly by the "hotline" person. More complex problems are referred to the appropriate marketing representative or to the manager of the branch technical resources. These complex problems can then be worked into a flexible schedule involving the entire branch. If one analyst is on vacation, a customer will be readily and effectively served by another STSC analyst, and problems of overdependence on one person disappear.

Conclusion

You can get the computer services you need from an outside vendor, but you will need to analyze such a relationship carefully. The crucial distinction suggested in this paper involves the difference between organizational and personal relationships. If your job involves meeting organizational objectives, then your relationship with an outside vendor should focus at that level. Some ways of working person-to-person seem effective but actually will not help you accomplish your job. Your vendor should be aware of this fact, and the organizational structure of the vendor ought to reflect an ongoing concern with what makes good business sense. If you lay the ground rules for a business relationship, personal relationships between members of the two organizations are likely to be effective and comfortable.

Dealing with an outside computer service effectively involves using an organizational approach and having the willingness to communicate honestly with your vendor's representatives. Ideally, those representatives should approach you in the same manner.

Tom Gull joined STSC in 1974 and has held positions as an applications consultant, marketing representative, and account manager. In his current position as an applications consultant manager he is responsible for managing the technical resources of the Washington, D.C., branch office, including the scheduling and planning of technical consulting activities. Gull has also helped develop and implement new business methods and applications and acts as a liaison between marketing and the Design and Development Department. Gull graduated from Cornell University in 1974 with a B.A. in sociology.

Frank Vogt

Selecting and Managing Outside Computer Services

Selecting and managing outside computer resources effectively is important to business and government users of computer applications. It is an interesting and complex task that requires juggling a diverse assortment of components including hardware, software, maintenance, technical support, communications, training, and documentation.

Proper management must cover the entire cycle of outside services; that is, the *selection*, *utilization*, and *termination* processes. *Selection* is the process of evaluating, comparing, and contracting with the vendors to service one's applications. *Utilization* is the day-to-day procurement of the services selected. *Termination* is the end of such utilization—normally due to the end of an application (in its present form), a transition to an inhouse system, or a transition to another outside vendor.

Note the phrase “day-to-day procurement of services”. It is important to bear in mind that such services are generally ordered “by the drink” and that the source of supply can vary throughout the life of an application—that is, one can always “go to another bar”.

This paper will highlight some critical aspects of choosing and managing outside computer services. The points presented are intended to stimulate and help organize the thinking of the person undertaking this task. The presentation will be grounded largely in government terminology and philosophy; this is due to my background as a government teleprocessing user and my current involvement in implementing existing government procedures for competitive acquisition of such resources.

Government awareness and policy have evolved considerably in this area during the past few years. Much has been learned, often at painful expense. It is hoped that this paper will help readers, from both the public and the private sector, to avoid going through the same trials.

We will address the following topics:

- connect/communications
- storage
- processing
- software
- performance
- account administration role.

The management of *connect/communications* is critical to many applications. The marketplace offers a variety of coverage, pricing plans, and types of service.

The selection of baud rates to be used is a basic decision; charges by suppliers will vary by baud rate, by the volume of data transmitted, or both. Terminal availability will be a strong factor in this decision.

One approach to data transmission is to employ intelligent terminals in an application. Using intelligent terminals means that data can be entered locally at leisure, corrected later, and finally transmitted to the remote service in a very efficient fashion.

Some applications may justify a distributive network or more sophisticated equipment to collect and perform a portion of the data manipulation locally. Some may justify a user-implemented network to supplement or optimize the use of a supplier's network.

A decision must be made concerning bulk terminals; cost tradeoffs must be considered when determining whether to lease or share lines and whether to rent or purchase equipment.

Approximately 20 to 25 percent of teleprocessing charges fall in the area of *connect/communications*.

Storage has been judged by many to be the most misused and abused aspect of remote computing services. It is available in many different forms. Immediate Access Storage (IAS) can be purchased by the character—either at one pricing plan or by the track, sector, or pack at rates that differ considerably.

Offline storage may be the better choice for part of an application; the data can be kept on magnetic tapes or on the client's own mountable disk pack. In either case, large savings will be realized. It is crucial for the client to determine how long he can wait to access his various files and what guarantees the vendor will make regarding mount and read time.

Old versions of programs on data files should be policed regularly; meticulous archiving practices can enormously benefit budget and operations. "Junk files" should be moved to a very low cost medium for later cleanup.

Often the operating system or database management system will allocate and control storage. This can result in some surprisingly large storage charges and thus should be monitored carefully.

Storage charges constitute 20 to 30 percent of charges to the average user.

Processing costs typically are responsible for about half of the dollars spent by users of outside computing services. Understanding prime and nonprime pricing structures can lead to a noticeable decrease in costs as well as an increase in throughput. For example, west coast users might get better performance at reduced costs from an east coast mainframe by running large jobs at the end of the work day.

Proper use of batch processing and established priorities is essential. Discipline is also essential to this aspect of an application.

The benchmark programs employed in the process of selecting outside services should be carefully chosen to test the capability, timing, and pricing of the system appropriate to the client's needs. Without a benchmark, processing costs will be essentially unknown.

Choice of *software* is of major importance. The remote service vendor is usually the best source of software support pertinent to his system; the user should make his needs clear to the vendor. Client needs and vendor support must be blended carefully.

A recent General Accounting Office estimate quoted \$8.00 per line for software development in languages such as FORTRAN and COBOL. With a language as powerful as *APL*, the number of program lines required is greatly reduced, but the per line costs may be higher. At these rates, it is crucial to use skilled personnel to get the most from your software development expenditures.

After programs are in place, it is important to create a system for the modifications that will be required as needs change. Assistance from remote service vendors varies in price, availability, and quality. This service, and the availability of vendor-written software, are deciding factors in selecting or using a remote service. Prewritten application packages can result in dramatic savings. But much of the prewritten software will entail an extra charge forcing a rent, create, or buy decision.

The *performance* of a remote system should be looked at very carefully, both when it is selected and throughout its usage. The hardware and operations will be managed by the vendor—not the client's own staff. Run times should be measured and monitored, as should job costs. A variance resulting in budget or mission problems must be dealt with. It is often a good idea to get post-selection benchmark runs to police performance and justify possible billing adjustments.

Any remote service usage, regardless of size, requires *account administration*. This duty could be full time or simply a portion of a particular user's responsibilities.

The account administrator should control user identifications to permit proper access; work with management to estimate and justify expenditures; verify billings by benchmark runs as well as by actual usage; monitor storage and access patterns to optimize the use of available pricing plans; monitor priority usage and assign priorities based on management goals; track and project usage for budgeting purposes; and work with the remote service vendor.

A "single-point interface" with the vendor need not be a rule, but it will enhance the relationship with the vendor, reduce redundant queries, and enable trends to be recognized and remedied.

Proper management of outside computer resources is not a trivial task. Basic common sense will rule, but familiarity with the services provided and a comprehensive approach are required.

Frank Vogt is currently an independent telecommunications consultant and a general partner in TSP Training Associates. He previously worked as a technical advisor for the General Services Administration (GSA). In this capacity, he acted as a negotiator on many contracts for GSA's teleprocessing services program. He assisted in developing a structure for evaluating time sharing service vendors for the government's procurement services.

Vogt has a B.A. in mathematics from the College of Steubenville and has done graduate work in mathematics at Denver University. He served as an officer in the Air Force and previously worked as a mathematician with the Navy, as an operations research analyst for the Army, and as a data system manager for the state of Ohio.

Clif Kranish

Converting External Datasets Into *APL* Files

APL files on STSC's *APL*PLUS* Time Sharing System are structured to work efficiently with the powerful primitive functions of the *APL* language. Consequently, it is often useful to convert data files produced on other computer systems or by programs written in other languages into *APL* files so that the data can be accessed and manipulated by *APL* programs. This paper describes methods used at STSC to perform these conversions.

What Is an *APL* File System?

On early *APL* systems, such as *APL*\360, the only way to store data was as a variable in the *APL* workspace. While this was satisfactory at first, it was soon found to be too restrictive, since all data for an application had to fit into a single workspace. In fact, the most common criticism of the *APL* language was that "*APL* couldn't handle large amounts of data". It didn't seem to matter how much the workspace size was increased; as long as there was an upper limit, *APL* programmers quickly reached it.

Another problem with early *APL* systems was that there was no way for *APL* to get at data produced by programs in other languages, even if these programs were running on the same computer as the *APL* system. The often heard complaint was that "*APL* couldn't talk to the rest of the world". The only way to make this data accessible to *APL* programs was to type it in at a terminal.

Other shortcomings of early *APL* systems were their inability to store and retrieve data under program control and their cumbersome procedures for sharing databases among several users.

To address these problems, most commercial vendors of *APL* developed file subsystems to go along with their *APL* systems. These file subsystems allowed data to be stored outside the workspace and made managing large amounts of data much easier. Although each single data object was still limited by the size of a workspace, there was no real limit on the total amount of data that could be stored.

In 1970, STSC offered one of the first *APL* file subsystems, called the *APL*PLUS* File Subsystem. Soon after, other vendors such as Burroughs and Digital Equipment Corporation (DEC), offered files with their *APL* systems. Although the storage and access methods are different, all of these systems store data outside the workspace.

One important advantage of storing data outside the *APL* workspace is that programs written in other languages can access the data. In effect, the

APL file systems opened a “window” to the rest of the world. Data could be interchanged between *APL* programs and other languages.

What Is an *APL* File?

An *APL* file is made up of components containing *APL* data that can be stored as variables in a workspace. Thus an *APL* file can contain components of different rank, shape, or data type. Data is stored outside the *APL* workspace, and the amount of data is not subject to workspace size constraints. System functions are used to transfer data between files and workspaces.

Most *APL* users aren't concerned with the internal representation of data or the use of different data types (except for the distinction between character and numeric data). Conversion from one numeric data type to another is done automatically.

However, the internal representation of data *does* affect the amount of storage required for an *APL* data item. It often affects the amount of computer resources required to perform some arithmetic operations on the data. The data representation is also important when considering a conversion from some external medium to an *APL* file.

All data stored on a computer is represented internally as a sequence of binary (0 or 1) values or *bits*. The meaning of each bit sequence depends on the type of data it represents. Most systems based on *APL*\360, including the *APL*PLUS* System, support four different data types. The four *APL* data types and the number of bits required for a single value are

- Boolean—1 bit
- Character—8 bits
- Integer—32 bits
- Real—64 bits.

Why Is Conversion Necessary?

External files, called *datasets*, are structured differently from *APL* files. These datasets are described in terms of logical *records*. For example, for a dataset on 80-column punched cards, each card is a logical record. For a dataset on magnetic tape the records may be grouped into *blocks* for efficient storage and processing.

To use *APL* programs to process data on external datasets, the datasets must first be converted into *APL* files. For external datasets consisting entirely of data in one of the four *APL* data types, conversion can be accomplished without interpretation using a simple batch program. The program reads the data on the cards or tape and appends the data to an *APL* file.

Sometimes, however, it is necessary to convert external datasets that contain more than one *APL* data type, or that contain data types that don't exist in *APL* (e.g., packed decimal). To convert these datasets, more sophisticated techniques are required. STSC offers one such technique, known as the File Conversion Generator (FCGEN).

How the File Conversion Generator Works

Data in an external dataset is described in terms of *fields* and *records*. Each field generally contains one type of information. These fields often have different data representations depending on the type of data that is stored. For example, one field in a personnel file may contain social security numbers, another field may contain names.

A record contains the information for a single entity in the file. For example, one record in a personnel file may contain the social security number, name, and other information for one individual. A programmer using COBOL or other high-level languages includes, as part of the program that processes the data, a description of each field in the input record.

To convert an external dataset into an *APL* file, the user must specify how the various fields and records are to be handled. This could be done using a program written in a language other than *APL*, but not without great difficulty. The special nature of *APL*PLUS* System files makes it difficult to deal with *APL* data representations in languages other than *APL*.

In addition, the *APL*PLUS* System provides special security checks not available with the OS/MVT operating system under which it runs. Consequently, it is advantageous to write the initial conversion program in *APL*.

But what about the COBOL programmer who is unfamiliar with *APL*? It seems unfair to ask him to learn *APL* so that he can write a conversion program.

This is where FCGEN proves most useful. FCGEN is a package of *APL* subroutines that are used to write an *APL* conversion program. The unique characteristic of FCGEN is that its subroutines accept data descriptions in notation closely resembling that commonly used in COBOL for describing record formats. Other *APL* programs check the *APL* conversion program for syntax and consistent specifications and generate a COBOL program, which actually performs the file conversion.

Advantages to this approach are twofold:

- The notation used in the *APL* program is familiar to both COBOL and *APL* programmers.
- Indirect generation of the COBOL conversion program allows the user to take advantage of the security checks provided on the *APL*PLUS* System.

In writing the *APL* conversion program, the user must specify the following information.

- *The fields in the input record to be converted.* Each field in the input record is listed. If the field is to be converted, the *FIELD* statement is used. For fields that are to be skipped, the *FILLER* statement is used.
- *The records to be converted.* Normally each record from the input dataset is read and converted. To select only certain records, an *OSFILE* statement is used. Records can be selected by specifying the first or last record to be read or by specifying that every *n*th record is to be read.
- *The records to select, by field values.* It is often useful to select only certain records by the values of certain fields. For example, if the value of an "amount" field is zero, that record could be ignored by using a *DISCARD* statement.
- *The data type and length of each field.* Each field of the input record can contain character data or numeric data in decimal or other representations. Each field description must include the data type and length in standard COBOL notation.
- *The translation of character data.* Character data on external media is generally not in the *APL* character representation, but in EBCDIC or ASCII. A standard translate table has been defined for characters with a direct correspondence. For example, the digits 0

to 9 and the uppercase letters A to Z are translated to their APL equivalents. For special graphics, "reasonable" translations are defined. Extensions or changes to the default translate table can be made using the *TRANSLATE* statement.

- *The distribution of data in a file.* When converted to an APL file, data is stored in file components as numeric or character matrices with values from successive records occupying successive rows of a matrix. Character and numeric values must, of course, occupy separate components. However, it may be useful to store a field in a component of its own if it is to be accessed frequently, or it may be useful to store related fields in the same component. The distribution of fields into APL file components is specified by using *TARGET* statements.
- *The file structure.* The data can be directed to several different files or to consecutive components of a single file.
- *The blocking factor.* The maximum number of input records to be stored in a single component is specified with the *BLOCKING* option. A large blocking factor will be more efficient in that it will require fewer file accesses, but the resulting components will require more workspace area when manipulating the data. If the file is ordered by a certain field, it may be useful to start a new component when the value of that field changes. This is done with the *NEWBLOCK* option.
- *The name of the summary file.* As part of the conversion process some error checking is performed. Defects such as incorrect characters, translation errors, nonnumeric values in numeric fields, numeric overflow, and magnetic tape errors are reported in a summary file.

Sample Conversion Program

The following sample input file contains six records. Arrows indicate the field breaks.

ALBANI	5000	ODENSE	1859
CARLSBERG	2900	HELLERUP	1847
CERES	8000	AARHUS	1856
FAXE	4640	FAKSE	1901
THOR	8000	AARHUS	1910
TUBORG	2900	HELLERUP	1847
↑	↑	↑	↑

The records of the sample input file are processed with the following FCGEN program.

```

▽ DKBREW
[1]  APLFILES BLOCKING 5
[2]  '12345 SUMMARY' GETS SUMMARY
[3]  '12345 DATA' GETS CHARDATA
[4]  '12345 DATA' GETS NUMDATA
[5]  F1:FIELD 'X(10)' ▯ BYTES 1-10 NAME
[6]  F2:FIELD 'S9(4)' ▯ BYTES 11-14 CODE
[7]  F3:FIELD 'X(10)' ▯ BYTES 15-24 LOCATION
[8]  F4:FIELD 'S9999' ▯ BYTES 25-28 EST
[9]  FILLER 'XX' ▯ LOGICAL RECORD LENGTH IS 30
▽

```

The program creates three files: a summary file to report the outcome of the conversion and errors, if any; a file to collect character data; and a file to collect numeric data. All four fields of all six records are converted.

Conclusion

FCGEN allows programmers to control the way in which external datasets are converted into *APL* files. Many different data representations can be converted into *APL* data using FCGEN. In addition, both *APL* and COBOL programmers can easily write the conversion programs using simple FCGEN notation.

Notes

1. C. Kranish, STSC Working Memorandum No. 127, *FCGEN—File Conversion Generator*, (STSC, 1978).
2. L. Gilman and A. J. Rose, *APL: An Interactive Approach*, (Wiley, 1976).

Clif Kranish joined STSC as an operator/programmer and is currently a senior programmer in the company's Technical Support Group. He participated in the design and development of STSC's Source Level Transfer System, which provides a means for transferring a workspace from one APL system to another. He also participated in the implementation of STSC's File Conversion Generator (FCGEN) and authored Working Memorandum No. 127, FCGEN—File Conversion Generator (STSC, 1978).

Kranish currently provides support for programs written in PL/1, COBOL, and 370 Assembler, as well as APL. He has taught introductory APL courses for STSC personnel and customers and currently teaches introductory programming at George Washington University.

Kranish has a B.S. in systems and information science from Syracuse University and is currently pursuing a master's degree in computer science at George Washington University.

**John A. Estep, Richard C. Geden,
Jack S. Reynolds, and Howard M. Sternlieb**

A Fully Automated Interface between Systems In Boston and Bethesda

This paper describes the evolution and implementation of a fully automated interface that transfers data from the Gillette Safety Razor Division in Boston, Massachusetts, to STSC in Bethesda, Maryland. The interface runs weekly and requires no involvement from a user at a time sharing terminal.

The Safety Razor Division (SRD) at Gillette has implemented a manufacturing resource planning system that runs on STSC's *APL*PLUS* Time Sharing Service and that is based on STSC's Comprehensive Manufacturing Control System, CMCS™ (see notes 1 and 2). Before this implementation, some of the inventory balance information required by CMCS was already being maintained by a Purchasing and Material Reporting System running on Gillette's inhouse IBM System/370, Model 158 computer. Gillette and STSC personnel developed—in stages—an interface that now automatically transfers status of open purchase orders and raw material inventory from Gillette's system to STSC's system.

Interface Specifications

When the interface specifications were developed two years ago, the CMCS installation was in its early stages. Contributing to the specifications were STSC representatives, SRD's Manufacturing Systems and Production and Material Control Departments, and Gillette's Management Information Systems (MIS) Department. An interface of this kind had not been previously attempted at Gillette, and its development had to be coordinated with the CMCS installation. Consequently, it was important to involve both users and systems personnel.

The group defined the following steps (similar in concept to manufacturing process routings) through which each purchasing transaction was to progress:

1. Issue purchase requisition.
2. Issue purchase order.
3. Receive material.
4. Inspect material.
5. Dispose of rejected material and post accepted material to stock.

The users selected a pilot group of parts, monitoring their progress step by step using the CMCS worksheet facility and manually posting information from the inhouse system to CMCS. This simulation identified conceptually the tasks that the interface would be required to perform. It also helped the users

develop their own procedures, giving them confidence in their ability to use the system to accomplish their tasks. The group determined that the information should be passed weekly from Gillette's system to CMCS.

Gillette's MIS and Manufacturing Systems Departments then developed two programs—one for each end—incorporating the lessons learned from the simulation. On Gillette's end, the program created two files containing purchase order progress and material status transactions. On STSC's end, the program validated the transactions and used them to update the CMCS database. Activity transactions (rather than a snapshot of status) were used in the initial implementation because the information could be captured easily and required only minor changes to Gillette's existing systems. This approach did require, however, some duplication of the Purchasing and Material Reporting System's logic at STSC's end. With *APL*, this task was accomplished with little difficulty.

Transmission

With the programs in place, the only task remaining was to actually move the data. At first the data was written to tape and physically carried to STSC. However, the resulting two-day turnaround did not meet Gillette's timing requirements. To save time, Howard Sternlieb of SRD's Manufacturing Systems Department introduced a Hewlett Packard 7260 table-top card reader. The information was punched on cards, read into the card reader (which was connected to an ordinary time sharing terminal), and transmitted to STSC over the telephone lines at 300 baud and later at 1200 baud. This cut down the cycle time considerably—from two days to as little as two hours. Unfortunately, the card system was susceptible to line noise and card jamming, so it was necessary for someone to "babysit" the terminal. Sometimes the job took as long as half a day.

The next improvement came with the elimination of the card reader and the direct transfer of data from a disk file on Gillette's system to a disk file at STSC. This was accomplished using two facilities:

- STSC's High-Speed Data Terminal Service (HSDTS), which moves data at high speed (2400 or 4800 baud) in either direction between STSC and a HASP terminal (see note 3).
- The HASP Remote Workstation (HRWS) program. (HRWS, which was originally the IBM program HASP.RMI360, was extensively modified by the University of Iowa and has been further modified by STSC).

HRWS is run as an ordinary batch job on Gillette's system, but it copies the two files from disk to the computer's communications interface, which the operator has connected (by dialing) to STSC. HRWS makes Gillette's computer appear to be a HASP terminal to STSC's system. HRWS and HSDTS reduced turnaround time for the transmission to approximately 12 minutes.

Complete Automation

Once the transmission time was reduced to an acceptable level, we turned our attention to STSC's interface program, which verifies the data received and updates the CMCS database. The program was interactive, requiring a user to sign on, run it, and answer a lengthy series of questions. This process was inconvenient, since it meant that the one person responsible had to sign on every Monday by 7:30 AM (the file transmission took place on Saturday) to complete the database update before other users arrived. Since the data was coming from a file—not from the operator—the answers to the questions were invariably the same every week. This was a good opportunity for automation

using STSC's Deferred Execution System, a facility which allows the automatic scheduling of production jobs (see notes 4 and 5).

The goal for this part of the interface was to have the update performed automatically each week. The job had to be completed by 7:30 AM every Monday, and it had to be performed without user intervention. By running the job as a "deferred task", we could not only free the user from the drudgery of entering repetitive data, but we could also take advantage of the substantial discount available to deferred jobs with batch priority.

Before automation of the CMCS update, the following procedure was performed on a weekly basis:

1. Gillette's computer operations staff initiates HRWS, which transmits the data.
2. HASP updates the *APL* files.
3. A Gillette user runs the interface program, which updates the database and prints the reports.
4. The Gillette user "cleans up" and prepares for the next transmission.

To complete file updates and generate reports by 7:30 AM on Monday, we were having the data transmitted (step 1) sometime over the weekend. This still left steps 3 and 4 in the above procedure as obvious candidates for automation.

To automate these steps it was necessary to examine the procedure more carefully and make some changes. First, it was necessary to insert a new step between steps 2 and 3. This new step was to verify the completion of the transmission and the creation of the *APL* files (i.e., to verify the successful completion of steps 1 and 2). In fact, this verification was already being performed. It was, however, a simple check at the beginning of the interface (step 3). By automating the data transmission, we had introduced some uncertainty into the procedure; that is, we could not know exactly when, over the weekend, the transmission would be performed. Consequently, the verification step became a more important part of the procedure.

The second change in the procedure was to separate (in step 3) the generation of the reports from the printing of the reports. We decided not to automate the final printing of the reports; instead this task was left for the end user to complete on Monday morning.

With these changes implemented, the procedure looked as follows:

1. Gillette's computer operations staff initiates HRWS, which transmits the data.
2. HASP updates the *APL* files.
3. A deferred task verifies the successful completion of steps 1 and 2.
4. The deferred task runs the interface program, which updates the database and generates the reports.
5. The deferred task "cleans up" and prepares for the next transmission.
6. A Gillette user prints the reports.

Now, with the additional automation of steps 3 through 5, all file updates and report generations are performed before Monday morning. The only exception to this is on rare occasions when the data transmission fails over the weekend, or when there is some system problem that prevents steps 3 through 5 from being run as scheduled.

The automation of steps 3 through 5 in this procedure was accomplished using STSC's Deferred Execution System. This system allows users to request

that *APL* programs be run at some future time. A key feature of the system is that submission and monitoring of the deferred jobs is accomplished with *APL* programs. This makes it possible for one deferred job to submit additional deferred execution requests. In this way, a job that is to run on a regular basis (weekly, in the case of our interface) can perpetuate itself indefinitely into the future without any involvement from the end user.

A secondary goal of the automation was to take advantage of the lower CRU (Computer Resource Unit) rates offered for deferred overnight processing. This presented a problem because the lower rates apply only if the job is run with a batch priority class. This means that the actual scheduling of the job is left up to the Deferred Execution System. Uncertainty regarding when the deferred task would run, together with uncertainty regarding when the data would be transmitted, presented the only complication in automating the interface.

We resolved our problem by scheduling two overnight jobs each weekend. One job runs on Saturday night and performs the interface if, in fact, the data was transmitted on Saturday. The other job runs on Sunday night and does nothing if the interface was successfully completed on Saturday. If, however, the data was not transmitted until Sunday, the Sunday night job performs the interface.

At first glance, it might seem easier to have just one job on Sunday perform the interface. With this approach, you would not have to worry about when the data was actually transmitted. Indeed, this approach was considered and rejected because of the possibility of a system problem occurring on Sunday that would prohibit the interface from successfully running. The "two-job" approach was selected because it gives the system two nights to complete the interface, and therefore increases the probability of its being successfully completed by Monday morning.

Once the two-job approach was selected, the only task that remained was establishing conventions for each job so that it could communicate its progress to other jobs and to the end user on Monday morning. With these conventions established, the weekly procedure was quickly and easily automated.

The Future

The last step in the automation of Gillette's interface is to convert from processing purchase transactions to transferring a snapshot of the entire file. The transaction approach currently being used has two principal disadvantages: (1) the logic is complex and expensive to maintain, and (2) recovery from failures is hampered by the lack of a clean restart point. Snapshot logic is simple and recovery from failures is accomplished by merely rerunning the job.

The transaction approach can be more suitable than the snapshot approach when the volume of data involved is significantly less using the transaction approach. For Gillette, the amount of data is the same using either approach. The switch to the snapshot approach did, however, require writing a new interface program at each end, but the logic has proved so simple that the task was accomplished in less than three days. At this writing, final testing is in progress.

Conclusion

Interface development is a combined systems and user effort. Specifications must be clear and represent a consensus; results must be closely monitored.

During each stage of development at Gillette, the interface continued to provide vital inventory status information to CMCS so that planning could proceed. Gillette's interface has evolved in the direction of maximizing the use of computer system capabilities and reducing clerical involvement. At the same time, each improvement in the interface has reduced both the time to transfer the information and the cost of operating the interface.

An important side benefit of the interface effort at Gillette is the relationships established between Gillette's users and systems personnel and STSC's representatives. These relationships not only aided the interface effort, but have also proved invaluable in other work undertaken by these groups.

Notes

1. R. G. Brown, *Materials Management Systems: A Modular Library*, (Wiley, 1977).
2. *Comprehensive Manufacturing Control System User's Guide*, (STSC, 1978).
3. J. J. Prats, Working Memorandum No. 104, *High-Speed Data Terminal User's Guide*, (STSC, 1978).
4. J. G. Wheeler, *Deferred Execution Reference Manual*, (STSC, 1979).
5. J. G. Wheeler, *Deferred Execution User's Guide*, (STSC, 1979).

John Estep joined STSC in 1977 and is currently a materials management consultant, responsible for the sale, customization, installation, and support of systems for finished goods management, production and capacity planning, and production control. Prior to coming to STSC, he worked in operations research and systems design for the Talon Division of Textron and for the Connecticut General and Massachusetts Mutual life insurance companies.

Estep holds a B.S. in mathematics from Allegheny College and an M.S. in electrical and computer engineering from the University of Massachusetts, where he is currently a candidate for a Ph.D.

Dick Geden has worked with the Gillette Company for over 11 years. He previously managed the Blade Dispenser Loading Department and Machine Shop Planning Department, and was the project manager for MRP (Material Requirements Planning) development. Currently he is manager of manufacturing systems.

Geden holds a B.S.B.A. degree from Boston College and an M.B.A. from Babson College. He is a certified practitioner of APICS, the American Production and Inventory Control Society.

Jack Reynolds is currently an applications consultant manager in STSC's Boston office. Before joining STSC, he was with IBM where he learned APL and developed expertise in a variety of database design and data storage and retrieval techniques. Recently he completed installation of a portfolio management package for an insurance company, and he is currently directing development of an inventory cost accounting system for a national manufacturing firm.

Reynolds holds a B.A. in mathematics from Dartmouth College.

Howard Sternlieb is currently a technical coordinator in the manufacturing systems department of Gillette Company's Safety Razor Division. He has responsibility for all technical aspects of computing, including programming, hardware selection, user support, and inhouse education. Sternlieb joined Gillette in 1974. He previously worked as manager of sales support with Wang Laboratories, and as a senior materials analyst at Honeywell.

Sternlieb earned his B.S.B.A. from Northeastern University and his M.B.A. in computer science from Boston College. He is on the staff of the Computer Information Systems Department at Bentley College.

Robert E. Cook

Making the Inhouse Decision: Some Considerations

As the popularity of *APL* grows, more *APL* users are looking beyond the traditional *APL* time sharing vendors for alternative ways of getting the computer power they need. Because of the dramatic decline in the per function cost of computer hardware over the past ten years, large users question the variable costs of commercial time sharing. They search for ways to lower and bound costs for *APL* and other time sharing usage.

To objectively evaluate the alternatives to commercial *APL* time sharing, more than hardware costs must be examined. A full understanding of the time sharing vendor's environment must be reached, as well as a reasonable understanding of the hardware and system software environments. Making a decision to move a series of *APL* systems from commercial time sharing to an inhouse processing environment, without detailed investigation of these various environments, is a disservice not only to the *APL* user, but also to the general management of the user's enterprise.

The scope of this discussion does not permit a detailed investigation of the relative merits and drawbacks of commercial *APL* time sharing versus an inhouse processing environment. Rather, a "road map" to and "background briefing" for this kind of investigation will be provided from the perspective of the *APL* time sharing vendor. A brief discussion of the vendor environment, the hardware environment, and the software environment will be provided, together with observations concerning the trends in technology and pricing that should be considered when evaluating the long-term implications of alternative means of delivering *APL* computer power.

When an analyst evaluates the environment of an *APL* time sharing vendor, he usually assumes that the vendor is fully aware of technological trends and will act in his own best interests to ensure the continued viability of his particular approach to business. The continued trend toward reduced hardware cost per computing function, when viewed on a relative CPU cycle-cost basis, makes the inhouse time sharing environment quite attractive. However, the time sharing vendor provides much more than a CPU service; in fact, CPU power is a relatively minor portion (less than 10 percent for STSC) of the cost. Although most commercial *APL* vendors find it convenient to charge customers in terms of CPU (or "CRU") usage, services such as "free" customer training and "free" telephone customer assistance are bundled into the seemingly simple charges. The analyst must consider the impact of removing these "free" services from the user environment.

When pricing an inhouse *APL* service, the analyst must also consider the disruption caused by removal of ancillary software services; ancillary software

is seldom provided in a readily installed form by the hardware vendor. Prime examples of this are the administrative or "housekeeping" software that provide services such as file backups, accounting and billing, sorting and merging, and high-volume printing. These functions are of critical importance to only a small subset of the total user population and, therefore, might be easily overlooked in an evaluation.

Additionally, *APL* language enhancements and proprietary application software—the most important assets of an *APL* time sharing vendor—are often woven into many of the user's own application programs. This transparent, proprietary software is usually designed to enhance the already high productivity of *APL* programmers and, for that reason, is of great value to users. The cost of losing this proprietary software must also be factored into the ultimate investment decision.

The hardware environment, of the costs to be discussed, is probably the least complex. Hardware prices will continue to plunge. Not only will prices for CPU power drop, but relative prices for all kinds of storage will also drop. Again, one must assume that the hardware vendors will act in their own best interests and protect the viability of their enterprises.

If hardware prices are to continue their downward trend, and if the costs of hardware manufacturers are to remain relatively constant, where will the manufacturers get their profit margin? At least some of the margin will come from economies of scale based on increased sales volume. Before long, however, competition for market share will drive high-volume hardware prices down, again squeezing the profit margins of the hardware vendor.

One method available to assist the hardware vendor in addressing this quandary is a substantial increase in the use of microcode or firmware to implement application and system software. The use of proprietary microcode both improves system performance and allows the hardware vendor to establish a proprietary edge that permits reestablishment of high-margin pricing.

A careful investigation of the true long-term costs of an inhouse hardware/software alternative to commercial *APL* time sharing must include a recognition of the trends established by the hardware vendors toward "unbundled", or individual, pricing of all proprietary system and application software. This unbundling implies a dramatic increase in the price of software vended by the manufacturer. A *Computerworld* article (Lundell, "Software for IBM 4300 May Cost More than Hardware", *Computerworld*, 30 April 1979) documented that software charges from IBM could exceed hardware charges for an IBM 4300-series computer over the life of the system. This trend, and the costs associated with it, must be recognized by the time sharing user community in investment analyses.

The single most important consideration in evaluating alternative means of delivering *APL* computing power is software—both system software and *APL*-related software. Proprietary application software may also have relevance to this decision.

With regard to software, response time is a major consideration. It is impossible, or at least unrealistic, to plan on maintaining a "happy" *APL* user community with terminal response time of more than two seconds for a trivial terminal command. This is an extremely important consideration, since users will reject an otherwise robust and well-rounded *APL* system if the terminal response time is unacceptably high.

The competition of an inhouse *APL* system is usually the commercial time sharing system that preceded it; users measure the new system accordingly. At STSC, for example, a 0.5-second response time is the standard at which the acceptability of terminal response is measured. A two-second response time—a

400 percent degradation from STSC's standard—appears to be a reasonable measure of the tolerance of the *APL* user community in an inhouse *APL* environment.

A primary question, then, must be "What operating system or system control program (SCP) is best suited for delivering acceptable response time in an interactive terminal environment?" After a full calendar year of testing, STSC concluded that IBM's VM/370 SCP is the best available choice. VM/370 (or its unbundled successor) is a superior interactive time sharing system with terminal response time in the two-second range. Moreover, it is adequate for an ancillary, low-volume batch workload. IBM OS/VS2 (MVS) was also carefully evaluated and then rejected as an *APL* processor. In the opinion of STSC, MVS is a superior system for batch processing and an adequate system for an ancillary, low-volume interactive time sharing workload (if five-second terminal response time is acceptable to the user community).

It is important that a fully supported SCP be used rather than a heavily modified, unsupported operating system such as DOS. The trends toward firmware and microcode options mentioned earlier necessitate a system approach that will allow the user to take advantage of at least some of the relative economies offered by the hardware vendors' microcode, without impacting the system's ability to process the *APL* workload.

In summary, weighing the relative benefits of an inhouse decision requires thorough investigation, especially of the less obvious aspects of providing *APL* processing facilities. Too often, decisions are based on insufficient data and fail to recognize the true costs associated with them. Complete, in-depth analysis, careful planning, and superior plan execution are critical to the successful conversion of the user community to the alternative system.

Bob Cook joined STSC in 1977 as director of corporate planning and has been vice president of market development since April 1978. His current responsibilities include project management for STSC's inhouse APL systems marketed for use on IBM-compatible hardware. Cook previously held management positions with Basic Four Corporation, Boeing Computer Services, and U.S. Time Sharing, Inc.

Cook earned a B.S. in mathematics from Indiana University of Pennsylvania and an M.S. in business administration from George Washington University.

Michael F. C. Crick

Variations in A^{pl} Flat Major

This paper is a personal survey of the interesting features found in *APL* systems produced by IBM and other mainframe manufacturers such as Control Data Corporation (CDC) and Burroughs. It is intended to give the audience a general view of what is going on outside the cozy environment of STSC's *APL*PLUS* System.

Overview

Most users work with one or perhaps two different *APL* implementations and, thus, rarely have the opportunity to see a large variety of *APL* systems. As an independent consultant, I am in the position of working with one version of *APL* one week and a different version the next. I would like to share with you some of my personal observations.

In the short space allotted it is clearly not practical to provide a detailed comparison of all available versions of *APL*—nor would such a comparison be very interesting. What I have attempted to do here is discuss a selected set of *APL* systems and to present only those details that to me seemed interesting and memorable. I shall thus discuss in turn *APLSV*, *VS APL*, *APLUM* (CDC), *APL/700* (Burroughs), *APLSF* (DEC), and Harris *APL* all in relation to STSC's *APL*PLUS* System (see note 1).

Is IBM Drowning in Its Own Alphabet Soup?

IBM has two major *APL* systems at this time: *APLSV* (*APL* Shared Variables) and *VS APL* (Virtual Systems *APL*). *APLSV* is a descendant of the famous XM-6 version from which the STSC implementations of the *APL*PLUS* System are derived. *VS APL* is IBM's official Program Product. It is much cleaner internally since it was written from scratch, whereas *APLSV* evolved from a series of earlier implementations. *VS APL* relies on its host system to provide many services such as swapping and terminal support, whereas *APLSV* provides its own. The new version of the *APL*PLUS* System running under VM is an extension of *VS APL*. The *APL* on the 5110 and 5120 is a direct crib of *APLSV*.

Current IBM implementations are mainly notable for what they do not have. There is no support for the diamond statement separator (◇), for error trapping, or for anything corresponding to Automatic Control of Execution (ACE), a proprietary product of STSC that provides the system facilities necessary to run production programs automatically without a user signed on

at a terminal. Additionally, the file systems offered by IBM are very hard to use and have significantly fewer capabilities than those offered by STSC.

At present, IBM is suffering a severe case of schizophrenia about *APL*. The idea evolved early in the development of *APL* that it was a “scientific language”—a fact reflected in STSC’s original name, “Scientific Time Sharing Corporation”. Since *APL* was designed as an extension of mathematics, that assumption was not unreasonable. Yet, everyone who has contact with the real commercial world knows that *APL* has in practice triumphed as a commercial language. IBM’s persistence of the vision of *APL* as a scientific or engineering language only shows how far IBM is out of touch with reality.

The illusion of schizophrenia is further fostered by the fact that IBM has, at this writing, both two *APLs* and two groups working on *APL*—the research group at Watson Research Center in Yorktown Heights, New York, and the development group in San Jose, California.

IBM is developing an interesting research version of *APL* at Yorktown Heights that supports operators, non-simple and homogenous arrays, and many other major new features (see note 2). What the development group is doing is not known. Whether IBM can get its act together and produce a single new *APL* that reflects the reality of user requirements remains to be seen.

Contrast at Control Data

The first versions of *APL* distributed by CDC were total disasters. An unofficial version written by Jim Burrill and Clark Weidmann at the University of Massachusetts (*APLUM*) moved in to fill the void. CDC now recognizes *APLUM* as the official CDC *APL* and has sole rights to distribute the product. The University of Massachusetts has complete control of development.

CDC *APL* operates under a handicap—Control Data machines are scientific machines and not commercial machines. CDC machines are poor at manipulating bit and character data, and their operating system is not designed for major file-sharing applications.

We all know the sort of person who is “handicapped” by being very short (or very tall), or by being from a foreign country. A person with such a handicap usually responds by trying harder and being more adaptable. That is how I would characterize *APLUM*. The implementers, as outsiders, were always insecure. They *had* to do a better job despite the limitations of the hardware and software they had.

The result is an *APL* that is everywhere characterized by what I think of as “good” design. Their innovations are always extremely clean and logical, and they have managed to avoid perpetuating many of the strange features of *APL* that have been supported over the years almost as acts of faith.

For example, why are certain *APL* operations such as `)COPY` only permitted as manual operations? In the days of Automatic Control of Execution, this makes no sense. Some vendors have bypassed this limitation by allowing the execute primitive (`⍎`) to operate on system commands or by sharing a variable with the input stack. *APLUM* has done it right. It uses `⍋LOAD`, `⍋COPY` and the whole implementation is “clean”. Other *APLs* should copy this approach.

Why does every major *APL* lack a decent context editor? Context editors are usually to be found as functions or perhaps as part of the host system. *APLUM* has integrated the context editor into the standard *APL* editor. Again the implementation is simple and clean. Another feature other *APLs* should copy.

Why do most *APLs* force you to preallocate space for symbols and the stack, not to mention the shared variable processor or the user workspace?

APLUM has one pool that is allocated automatically as needs dictate. If the pool runs out, the system asks the host for a bigger swap area in which to run itself—all performed automatically with no user intervention. This is a design standard that other *APL*s should emulate where possible.

APLUM has an extremely simple form of error trapping and its batch support is reasonably good. Its main defects are the curious and idiosyncratic file system and the lack of support for the diamond statement separator. It is very fast for floating-point operations, but can take forever to perform such character operations as concatenating two character arrays.

Overall, the group at the University of Massachusetts has done a superb job. Despite the unsuitability of the host machine and the operating system, and despite their lack of clout as a group external to CDC, they have produced an excellent *APL*. Their current direction is toward developing an *APL* compiler to let *APL* compete with FORTRAN and perhaps ultimately be accepted by engineers.

Burroughs and *APL/700*

Burroughs *APL* is another non-IBM *APL* that is worth considering in some detail. It is a “liberal”, user-friendly *APL* that owes much of its character to Jim Ryan. It has numerous minor but useful extensions—many of which deserve to become a permanent part of the language.

For example, *APL/700* supports set operations. These are perhaps more useful than one might suspect. The most common use is to eliminate duplicates from a set of numbers thus:

```
(10) ∪ ARRAY
```

It would seem logical to define a monadic form of union (unique?) to eliminate the need for the 10 on the left.

APL/700 has introduced the use of assignment as an operator, taking any scalar dyadic function as its left argument. For example:

```
I←+1           (Meaning I←I+1.)
```

This feature comes into its own when the variable being incremented has a large and complex subscript. Curiously, concatenate is not allowed with this construction—one might have thought that would be the most useful case.

APL/700 has extended transpose (to turn vectors into column matrices) and reshape (to operate on empty vectors). The axis operator has also been extended to operate on scalar dyadic functions thus:

```
(2 3ρ12)+[1] 10 20 30
11 22 33
14 25 36
```

This eliminates the need for a lot of wasteful reshaping and should be used more widely.

APL/700 offers reasonable support for ASCII terminals using a visually pleasing character substitution approach. On an ASCII terminal one can enter:

```
X<IS>3 4<RHO><IOTA>12
```

There are also a number of useful extensions to tracing and editing that are not described easily, but are very useful. The file system uses special symbols such as \boxplus and \boxminus . There are some new goodies like “pop”, “map”, and compress. The file system is generally like that of STSC’s *APL*PLUS* System,

differing only in detail. For example, files are accessed by name rather than by tie number, and security is handled by the host rather than by access matrices. *APL/700* does not support the diamond statement separator and generally shows up poorly when benchmarked. This is partly compensated for by the fact that Burroughs machines can support a large number of central processors on one system.

In 1977 Jim Ryan went to work at Data Resources, Inc., but he is now back at Burroughs working on SYBIL—a new language derived from *APL* which uses words instead of special characters. The new language will have an extended notion of workspaces known as namespaces. Meanwhile, *APL/700* is fully supported by a separate group at Burroughs.

Diversity at DEC

If *APL/700* bears the stamp of Jim Ryan, DEC's *APLSF* bears the stamp of Alan Perlis. As developed by the group at Carnegie-Mellon University in the early seventies, DEC *APL* was ahead of its time. Now that the rest of the world has caught up, DEC *APL* has been forced to do a certain amount of backtracking to become consistent with everyone else. There are, for example, three format functions in the language—the official IBM “thumbtack” (⍒), a version of $\square FMT$ (using the symbol $\$$), and the pioneering monadic encode (⍒) that will now probably be phased out.

For the same reasons, execute may be performed by ϵ , \perp , or ⍒ . Not only may one execute system functions, but one may also execute a character matrix. This brings *APL* closer to LISP, where data and functions are all the same thing.

My favorite extension is the omega function, which performs a “where” operation as shown below:

$$\omega B \leftrightarrow B/\text{⍒}B$$

$$\begin{array}{cccccccc} & \omega & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 3 & 4 & 6 & & & & \end{array}$$

This is particularly useful when B is a complex expression. I gather that this primitive is being phased out to allow for future inclusion of alpha and omega as defined by Kenneth Iverson.

APLSF has an elaborate file system that uses special symbols, system functions, and system variables—powerful but messy. There is no support for the diamond statement separator. As a whole, DEC *APL* illuminates the pitfalls one faces for being too liberal. It must have been infuriating as well as gratifying to those at Carnegie-Mellon to see IBM use many of their ideas in slightly modified form.

Another DEC innovation is to provide two levels of *APL* at different prices. The inexpensive beginners version lacks certain enhancements, notably the file system. A major disadvantage of *APL* compared with other major languages is that it comes in only one size. Nobody selling shoes or houses of only one size would stay in business, but in the case of *APL* the general policy has always been all or nothing. The market for a small, carefully chosen subset of *APL* has not been properly served.

We must commend DEC for its pioneering efforts to expand the language. Just as the first platoon out of the trenches suffers the heaviest casualties, DEC has had to pay the penalty for being first.

Doing It Straight at Harris

If *APLSF* is liberal, the new Harris *APL* is conservative. One is reminded of the eager new member of the club whose dress is always a little too correct

and who can be relied upon never to raise an eyebrow. Such an attitude probably befits a newcomer—only the old guard can break new ground and get away with it.

The new Harris *APL* is quite impressive. Its file system is very close to that of STSC's *APL*PLUS* System, as is its support for error trapping. It supports batch processing, and shared variable support is imminent.

The only real differences are the use of `□ST` and `□TR` for stop and trace, and a variant definition for the diamond symbol. On Harris *APL*, if three statements are placed on a line, the next line gets the line number plus three rather than the next sequential number, as on the *APL*PLUS* System. Thus, diamond can be considered as an instruction to the function display module rather than as a piece of punctuation. One cannot write:

```
→NEXTL IF B>5 ◇ 'B IS TOO SMALL' ◇ →E5
```

This, and the fact that Harris *APL* does not support `□FMT`, would appear to make conversion from STSC's *APL*PLUS* System to Harris *APL* infeasible for major applications, despite the correspondence of most other features. Conversion from Harris *APL* to STSC's system, on the other hand, is likely to be particularly easy.

Harris *APL* is fast—impressively so according to their benchmarks. One should be aware that this speed is achieved in part by doing all numeric operations in floating point, with 39 bits (11 decimal places) of precision. This makes Harris *APL* unsuitable for financial applications requiring precision to the penny for large dollar amounts.

...And What About the *APL*PLUS* System?

Since this presentation was prepared at the request of STSC, I have assumed, possibly incorrectly, that all readers are familiar with the *APL*PLUS* System. The discussion has been largely in terms of how other *APL* systems compare to the *APL*PLUS* System. This is because the *APL*PLUS* System is a recognized leader in the *APL* community; what STSC does, others copy.

There are many new developments in *APL*, and STSC is a major force in initiating them. Since I write this paper without being privy to what STSC is going to unveil at its April 1980 conference, to talk at length about the unique features of the *APL*PLUS* System would be to talk about those features that others have not yet copied. Much work is going on in areas like relations, support for systems on different hardware, systems software, and generalized arrays. Other presentations in this book (e.g., "Nested Arrays: The Tool for the Future") address these subjects in detail.

Notes

1. The latest manuals for the *APL* systems discussed are properly obtained by contacting the local sales office of the company involved.
2. J. A. Brown, "Evaluating Extensions to *APL*", *APL79 Conference Proceedings* and *APL Quote Quad*, Vol. 9, No. 4, June 1979.

Michael Crick is an independent software consultant and financial advisor in Seattle, Washington. His involvement with APL began while he was employed by IBM, where he received an outstanding contribution award for his efforts in

the development of APL (CMS). Crick was also instrumental in the development of MAINSTREAM-APL at Boeing Computer Services and was branch manager of the Seattle office of I. P. Sharp Associates.

Crick holds both a B.Sc. and M.Sc. from the University of London.

Robert L. McGhee and James G. Wheeler

Travels in VM Land: A Virtual *APL* Primer

Virtual *adj.* Existing or resulting in essence or effect though not in actual fact, form, or name.

—*American Heritage Dictionary.*

Many of our readers will already know that VM stands for “Virtual Machine”, but far fewer will have a clear grasp of just what the term means to them as *APL* users. Simply introducing the idea of a machine may worry some *APL* users. STSC's *APL*PLUS* System running under OS/MVT has traditionally isolated the user from the details of *real* machines, for *APL* naturally tends to make the computer on which it is executing invisible to the user. A programmer can work successfully on the *APL*PLUS* System without thinking about computers at all; instead, he can imagine that he has a magic terminal that executes *APL*, and he can let things go at that.

There are many positive things to say about this isolation, in particular the way it lets the programmer keep his thoughts on the conceptual, problem-solving plateau instead of worrying about physical hardware or the internals of software. But this isolation also tends to limit the types of solutions that the programmer can choose. Three of the biggest limitations are these:

1. *APL* programs cannot access data that is used by programs written in other programming languages. Traditional *APL* provides no means for an *APL* program to use the same data as a program written, say, in FORTRAN.
2. This isolation from other languages also prevents *APL* users from enjoying the use of software packages that are not written in *APL* (and, all chauvinism aside, some of the best software around is written in other languages).
3. A less awesome limitation, but still an irksome one, is that *APL* traditionally has a fixed workspace size. This severely limits the size of the data objects that can be used by an *APL* program and often requires that applications be divided into multiple workspaces.

Using STSC's *APL*PLUS* VM System, an *APL* programmer can conquer all of these limitations and still enjoy the full problem-solving power of *APL*. VM is, however, unfamiliar territory to most *APL* users and the bookcase full of manuals on the subject may discourage the uninitiated, no matter how much they may want to exploit the new capabilities VM offers.

The purpose of this paper is to present the basic concepts involved in using VM. Because of the size of the subject, we will not try to present even the minimum knowledge needed to be a successful VM user. Instead, we will try to convey the basic ideas needed for a user to feel at home learning about VM and understanding it intuitively. We will present three different views of VM:

1. *The virtual machine perspective.* This is the “hardware” point of view. Since VM simulates a private computing center for each user, this perspective is essential to understanding the capabilities of the virtual machine.
2. *The software perspective.* Working in VM means moving into and out of the *APL* environment. This perspective is important in understanding the relationships among the various software environments of the VM system.
3. *The APL user’s perspective.* Our interest in VM is due primarily to the way it complements and extends the power of *APL*. We will use a case study of transferring an *APL* workspace to VM to show just how valuable these new capabilities can be.

The Virtual Machine Perspective

The central idea behind VM, indeed the one that gives it its name, is the *virtual machine*. VM/370 simulates a complete, private computing center for each time sharing user. Simply by signing on to VM (a process known as “logging on”), you can in effect obtain the exclusive use of hundreds of thousands of dollars worth of computer equipment. Of course, it’s almost all virtual hardware, which means that it will not perform as briskly in real time as actual equipment. But since *APL* users are quite accustomed to doing without real-time capability, no significant sacrifice is involved. In fact, considering that one real computer is giving each of its many users the illusion of having a personal computer, the performance is really quite good.

Each user’s virtual machine is configured at log-on to a predetermined initial configuration. A typical VM user at STSC might be given the following virtual hardware at log-on:

- An IBM 370-series mainframe (CPU) with 512 kilobytes of main storage.
- An “operator’s console”, which is simply the terminal that the user is using.
- A virtual line printer (known to most *APL* users as a “high-speed printer”).
- A virtual card reader (*APL*ers who gasp at this idea should catch their breaths before reading the next item).
- A virtual card punch (how else would we produce virtual card decks for the virtual card reader?)
- Three virtual disk drives, otherwise known as Direct-Access Storage Devices, or DASDs.

This virtual hardware can be put to a number of good uses, giving the user the following capabilities and more:

- Printing files by *spooling* them to the virtual line printer, in a manner similar to that provided by the Fileprint Facility on STSC’s OS/MVT system. The files are printed on real paper by a real high-speed printer shortly after the spooling operation is “closed”.
- Transmitting files to other virtual machines (i.e., other VM users) by punching virtual card decks for them. In reality, the “cards”

are records in system spool files that magically appear in the “input hopper” of the other user’s virtual card reader.

- Reading virtual card decks from the virtual card reader, completing the transfer of files between virtual machines. Virtual card decks are not the only means by which data can be transferred from one virtual machine to another, but they are useful for sending a file to another user when that user is not signed on.
- Reading from and writing to files residing on virtual disk drives. These disks may belong to the user’s virtual machine or may be shared by other users.
- Disconnecting the console (i.e., the terminal), leaving the virtual machine running; this provides a facility comparable to the Detached Execution Facility available on STSC’s OS/MVT system.

In VM jargon, a virtual disk drive is referred to as a *minidisk*. This term reinforces the concept that minidisks are in fact portions of real disks dedicated to specific users. The term “virtual disk” is discouraged because, unlike virtual main storage, the amount of permanent storage available on minidisks cannot be altered by the user.

The three disks typically available to the user at log-on are

- The user’s private minidisk, called the A disk.
- A system minidisk, called the S disk, containing system software.
- Another system minidisk, called the Y disk, containing system and application software.

The A disk contains the user’s private files and the user is permitted to both read from and write to this disk. The A disk is analogous to an *APL* user’s private library. In fact, the user’s private saved workspaces are presently stored on this disk.

The S disk and Y disk are analogous to public libraries in an *APL* system. Both are read-only disks in that the user can read files from the disk but not modify the files. All virtual machines have access to these disks. The S and Y disks contain such things as the *APL* interpreter, the FORTRAN compiler, and applications like SCRIPT (a text formatter developed by the Department of Computer Services at the University of Waterloo), BMDP (Biomedical Computer Programs, P-Series), and SPSS™ (Statistical Package for the Social Sciences).

Each minidisk contains a number of *files*. Unlike *APL*, where files store only *APL* data values, these files are of many different types and are used for widely varying purposes. Some files contain data, and a wide variety of different data formats are possible. Other files contain compiled programs that can be invoked by using the file name as a command. Still other files, called EXECs, are programs made up of sequences of VM system commands.

The user’s virtual hardware configuration can be selectively modified as needs for resources change. For example, the typical virtual machine that we have discussed is illustrated in Figure 1. The size of the user’s *APL* workspace is directly related to the amount of main storage in his virtual machine. If this user finds he does not have a big enough workspace, he can increase the amount of main storage to, say, two megabytes. If he needs a large amount of temporary file space, he can create a new virtual disk drive (this file space is lost at sign-off). If he needs a tape drive to read from or write to a tape, he can have a *real* tape drive attached to his virtual machine. After performing these actions, the same virtual machine would have the configuration shown in Figure 2.

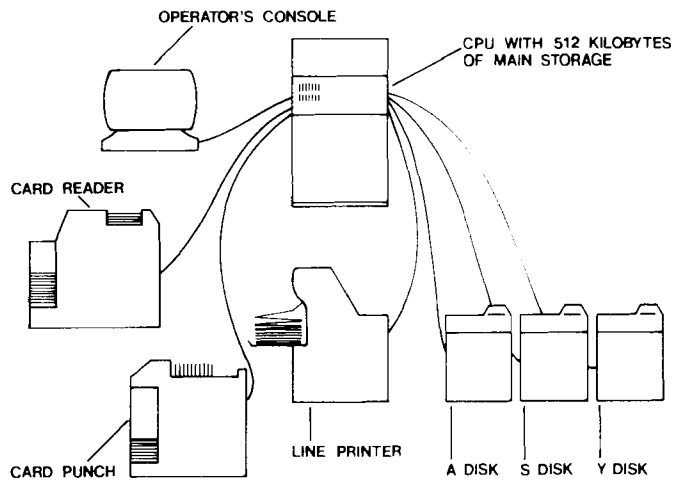


Figure 1—Typical Default Virtual Machine

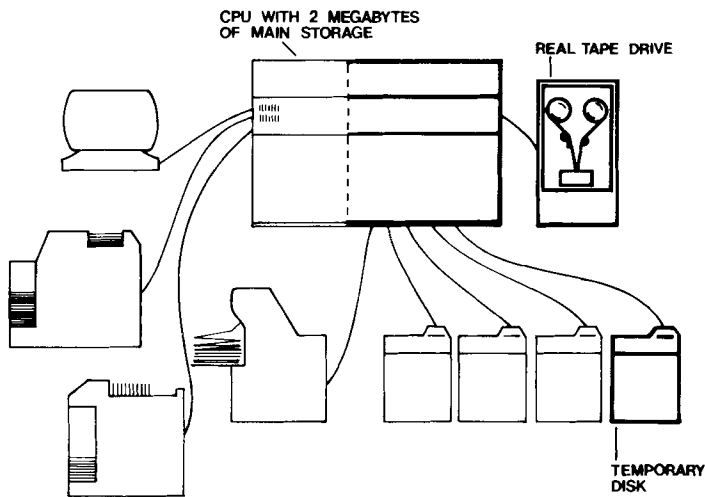


Figure 2—Reconfigured Virtual Machine

Software Perspective

The VM/370 operating system is divided into two main components—the Control Program (CP) and the Conversational Monitor System (CMS). CP's role is management of the real hardware; system resources are distributed among users so that each has the illusion of controlling a full-scale, private computer. CMS is an operating system designed to give the time sharing user a friendly and yet versatile way of working in the virtual machine created by CP. While it is possible to run different operating systems under CP (including STSC's

OS/MVT system), CMS has been built specifically to run efficiently in a virtual machine and to serve a single user at a single terminal.

CMS runs under the control of CP, providing interactive use of various applications and language processors, notably the *APL*PLUS* System interpreter. CMS processes commands entered from the terminal, including CMS and CP commands. By using the proper tools, it is also possible to execute some CP and CMS commands from within the *APL* environment.

This hierarchy of environments and the various routes between them can be very confusing to the new VM user, particularly one who has previously been accustomed to using *APL* on a system that provides only *APL* computing. To help the reader develop an intuitive sense of the hierarchy and relationships between the multiple environments, we'll use a spatial and architectural model.

In this model, we'll imagine that a virtual machine is equivalent to a house. The real computer can be considered a small community of many users, each living in a private house (see Figure 3). The control program CP can then be thought of as the main street in this community. Each user enters his private house from the street and departs it by the same route.

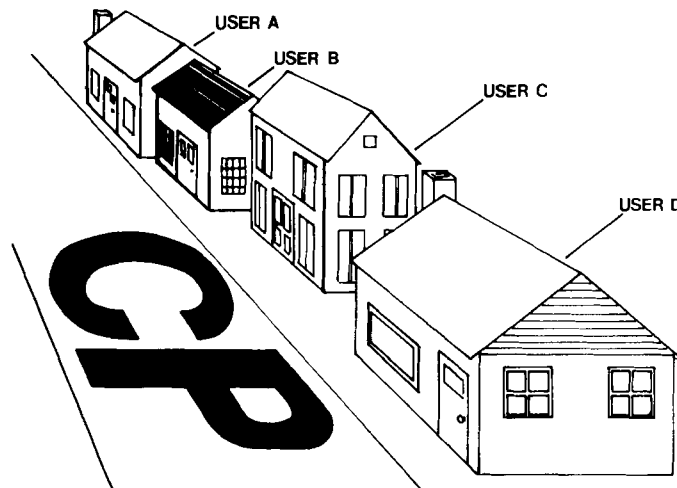


Figure 3—Main Street in VM Land

Logging on to the VM system is accomplished by communicating with CP. The CP command LOGON is roughly equivalent to a request to “build me a house”. Each user has a set of master blueprints stored in a CP directory, and CP constructs the user’s default house according to these blueprints. As can be seen from Figure 3, different users can have houses of different sizes and types. If a user wants to change the size of his house, he must “move out” temporarily while the old house is torn down and CP builds a new one.

The walls of the houses provide ample isolation and privacy for each user, and each user’s computing is done entirely within the boundaries of his own house. Facilities exist for appropriate interaction between houses (virtual machines); these will be described a little later. Communication and transfer of data between houses requires mutual cooperation, however. One user cannot invade someone else’s house or impinge on his privacy without invitation.

