Extending the domain of APL

by M. T. Wheatley

This paper explores connectivity mechanisms between APL and other languages and applications available on a modern computer system. The design, implementation, and application of APL facilities such as shared variables, auxiliary processors, external names, file subsystems, and namespaces, as they are implemented in IBM's APL2 product, are discussed and compared.

Due to the persistence and insight of men like Iverson and Falkoff, in APL we are blessed with a language which, after more than 25 years of use, is still elegant, concise, precise, general, usable, and machine-independent.

> The definition of APL is purely abstract: the objects of the language, arrays of numbers and characters, are acted upon by the primitive functions in a manner independent of their representation and independent of any practical interpretation placed upon them. The advantages of such an abstract definition are that it makes the language truly machine independent, and avoids bias in favor of particular application areas.[1]

Despite the importance of machine-independence, a language that is used for computer programming cannot practically exist without access to the computing environment in which it runs. Further, to be useful in a wide variety of applications, such a language must also be able to access many of the other tools, libraries, routines, and subsystems available in that computing environment.

In the last 25 years, APL implementations have grown significantly in their ability to interact with the computing environment, including its associated software tools. This paper reviews the key facilities in APL that provide this function, briefly focusing on their history, objectives, characteristics, benefits, and problems. The discussion is centered around IBM implementations of APL.

Description of facilities

Early APL systems. When APL was first implemented on the IBM System/360* in 1966, it provided two mechanisms that allowed access to the environment: system commands and I-beams. Most APL users are familiar with system commands, since their use has survived and is widespread in current APL implementations. I-beams, on the other hand, are less familiar.

The use of the dyadic I-beam primitive was first introduced in APL\360 to allow execution of IBM System/360 instructions from within an APL program. It was considered an ad hoc facility for the use of system programmers, and was never formally accepted as a primitive or made part of the APL language. Nonetheless, I-beams were very useful and the facility was extended in later APL implementations. Monadic and dyadic definitions provided access to the underlying computing system. The definition of a dyadic I-beam required an integer left argument that specified the subfunction to be per-

formed, and a right argument and result that varied by subfunction. Monadic I-beams, whose right argument specified the subfunction, simply returned a subfunction dependent result.

In APL\360 and APLSV, the use of dyadic I-beams was restricted to privileged users and provided such functions as user and system control and access to memory. The monadic I-beams provided statistics on various aspects of the systems and access to certain key system variables such as time, date, and terminal type. All of the nonprivileged I-beams were replaced by system variables in later APL implementations (see Table 1).

Table 1 Nonprivileged monadic I-beams

I-beam	Description	Replaced By
19	Cumulative keying time	⎕*AI*
20	Time of day	⎕*TS*
21	Compute time since sign on	⎕*AI*
22	Free space in workspace	⎕*WA*
23	Number of users signed on	⎕*UL*
24	Elapsed time since sign on	⎕*AI*
25	Current date	⎕*TS*
26	First value in line counter	⎕*LC*
27	Line counter vector	⎕*LC*
28	Terminal type	⎕*TT*
29	User account number	⎕*AI*

Since the earliest implementations of APL, there have been requests from users for linguistic access to many of the functions provided by system commands. However, it was felt that the useful, usable, and rudimentary syntax of system commands did constitute a language—one that was incompatible with APL and had no constructive potential.[1] Locked functions were therefore provided in APL\360 to allow applications to perform such tasks as setting index origin, or the random seed. These locked functions contained I-beams that performed the actual work. Again, this provided an ad hoc solution to the problem. The long-term solution was implemented with the introduction of system functions and system variables in APLSV.

System functions and variables. In APLSV, two new types of objects, system functions and system variables, were introduced into the APL language. These objects, distinguished by names that start with the character ⎕,[2] are defined in the implementation and are available in every clear workspace. In many senses, they are similar to primitives insofar as they provide specific predefined functions.

When system functions and variables were introduced into the APL language, they were introduced cautiously and only a few were provided. Unfortunately, their introduction was interpreted by some implementers as the long overdue solution to a serious problem—the problem that APL was limited, particularly in its access to system facilities. A number of APL implementers immediately reacted by introducing a large number of new system functions and variables. These functions and variables were introduced without much forethought, with little consistency in syntax or semantics, and with little compatibility between implementations. It was initially believed that system functions and variables were not part of the APL language, so implementers, perhaps installations, and maybe even individual users were free to invent as many as they pleased. System functions and variables, however, are very much a part of the APL language, as is demonstrated (in hindsight) by their inclusion in the APL standard. They now provide one of the more serious impediments to compatibility and portability.

Little thought was given to which functions should be provided as system functions, as primitives, or by means of other mechanisms. Very little guidance on this subject was provided to implementers. Functions such as **format** have been widely implemented both as primitives and system functions. Perhaps they are most appropriately neither; perhaps they should be defined functions. In the rush to provide commonly used, "omnipresent" functions with adequate performance, implementers have clearly gone overboard with system functions and variables. Fortunately, there have been no system operators introduced to date.

Component file systems. The need for file I/O was recognized as a key requirement in APL systems, before the introduction of system functions and variables. Component file systems were developed to fill this need and access to them was provided with locked functions that used the I-beam primitive. These locked functions were replaced with system functions soon after the introduction of those facilities. A typical component file system adds about 20 system functions to the language.

Component file systems provide facilities that allow APL arrays to be stored in and retrieved from external files. They are designed to be fast, straight-

forward, and simple to use in APL applications. They are not primarily designed to provide mechanisms that allow data interchange, via files, with non-APL systems. The file and record formats implemented in component file systems are typically complex and difficult to read or write from other high-level languages.

Shared variables and auxiliary processors. The introduction of shared variables with APLSV was motivated by the same need for file I/O facilities. Lathwell, Falkoff, and others who worked on this problem recognized that a primitive function or system function solution would eventually become unmanageable, particularly if a variety of access methods and file formats were to be supported:

> Most programming languages approach communication and storage problems by defining explicit communication primitives such as READ and WRITE to transfer information. These specialized primitives, used in conjunction with declarative statements and job control languages, result in programs which contain file-handling details irrelevant to the algorithm, and are strongly dependent on host operating systems and file structures. This approach was deemed inappropriate for APL because it conflicted with many of the principles that guide APL design; in particular, it conflicted with the requirement for machine-independent theoretical definitions of primitive functions.[3]
>
> ... there is a high cost associated with the use of primitive functions for communication, as is the rule in most programming languages. This cost takes the form of complications in both syntax and semantics, and follows from the fact that in any language the arguments of a primitive function must be objects in the language. Thus, when functions like READ and WRITE operate on a variety of files, these files must necessarily be included in the language as additional constructs. The situation can become more and more complex, to the point where simple input and output statements are no longer adequate, and auxiliary statements, such as data declarations, must be introduced. These complications then make the language costly to implement, and costly to use.[4]

Further, Lathwell and others working on the problem realized that the requirement was not only for file I/O, but for other types of communication with components of the underlying computing facility. It was decided to implement a solution for the general communication problem, and to use that solution to implement file I/O facilities, among other things. The solution was shared variables, whose use had been originally postulated to describe channel architecture in the APL formal description of the IBM System/360.[5]

With APL2, variables may be shared between APL users on the same computer.

A shared variable differs from a normal APL variable insofar as it is "shared" or owned simultaneously by two "partners" or processes. Each partner can set or use the variable; its value at any given time reflects the last value set by either partner.

A control mechanism is provided to synchronize access to the variable, if such control is desired by the partners. If a shared variable is left uncontrolled, each partner is free to set or use the variable at will. With access control, however, protocols such as "master/slave" or "message passing" can be easily established.

Declaration, control, and management of shared variables is provided with a set of system functions. Variables can be shared between APL users or with other processes, referred to as "auxiliary processors," in the computing environment. Typically, auxiliary processors are programs written in a language other than APL that are designed specifically to share variables with APL applications and to provide specialized functions, such as file I/O, to those applications.

With APL2, variables may be shared between APL users on the same computer, between APL users and an auxiliary processor, or for that matter, between auxiliary processors. Auxiliary processors that exist in the APL user's address space are called "local" processors, and normally share variables only with that APL user. Auxiliary processors may also be implemented as multiservers that exist in separate address spaces and share variables simultaneously

with more than one APL user. Such auxiliary processors are called "global processors," and can provide facilities such as shared file support to a group of APL users.

Experimental facilities have been developed that allow variables to be shared between partners on separate computing facilities that are linked by telecommunication facilities.

Shared variables are handled by a component of the APL system called the "shared variable processor." This component is invoked when either partner attempts to set or use a shared variable. In most APL implementations, the shared variable processor uses an area of memory referred to as "shared memory" to temporarily hold the value of a shared variable until both partners are aware of it. Shared memory is also used to hold control and management information, such as identification, state, and access control for the shared variables and the partners sharing them.

Shared variables were designed to provide a general, asynchronous communication facility.

The initial implementation of shared variables in APLSV supported communication between APL users, and communication with auxiliary processors. One auxiliary processor, TSIO, was provided with the system, and it was expected that installations would write others as required. TSIO provided sequential and direct access to files maintained by the underlying operating system. It was particularly useful for exchanging files with applications written in other languages, but fell short in terms of function and usability when compared with the more special purpose component file systems.

There is no technical reason that a component file system should not be implemented with shared variables and an auxiliary processor. In fact, such implementations eventually emerged. At first, proponents of the component file system refused to consider the use of shared variables. In their defense, it should be pointed out that the use of shared variables was often difficult and complex before general arrays were introduced into the language. Auxiliary processors typically required paired variables and sometimes multiple modes of communication.

Further complicating the issue and polarizing those involved was the fact that many of the auxiliary processors that emerged were inelegant and inherently sequential in their communication protocol. Component file systems, on the other hand, typically presented a more elegant and usable interface.

Finally, it should be remembered that shared variables were designed to provide a general, asynchronous communication facility. It was originally envisaged that they would be used within cover functions to implement a specific communication protocol, or access method interface. Because these cover functions were not "omnipresent" or particularly good performers, however, and because most of the required communication involved simple synchronous protocols (e.g., READ, WRITE), the system function approach remained a more desirable alternative for many users.

When general arrays were introduced into the language, the use of shared variables and the implementation of auxiliary processors became considerably simpler. The command and data could be packaged together in a single WRITE request, and the return code and data could be packaged together for READ. Paired shared variables, with all of their associated complications, were no longer required.

Name association and external functions. Thus far, we have dealt mainly with issues involving file I/O. Since the emergence of APL there has been an additional requirement voiced by users for facilities that allow non-APL programs to be called from APL and to exchange data with APL. Over the years there have been a number of attempts to provide such facilities, typically with specialized auxiliary processors. While these auxiliary processors provided at least some of the needed function, their use never became widespread, probably for the following reasons:

- The auxiliary processors were difficult and cumbersome to use. Their use depended on shared variables for passage of control and data. Typi-

cally multiple variables had to be shared, and typically the interface was complex.
- The shared variable interface used was inherently asynchronous, while the primary requirement was for a synchronous interface to subroutines written in languages other than APL.
- Passing argument data was difficult. The shared variable processor sometimes imposed limits on the size of data that could be passed to a subroutine. Further, subroutines in other languages often required multiple heterogeneous arguments that were difficult to package and send across the shared variable interface.
- It was difficult to access routines that were not specifically designed to interface to APL. Existing libraries of subroutines required argument data types not supported by APL or specialized interface conventions.

General arrays presented a practical solution to some of these problems. They allow parameter passing on subroutine calls with a syntax amazingly similar to that commonly used in other languages, as shown in the following example.

APL:

```
A←10 20 30
B←'ABCDE'
C←1.2 1.3
PROCESS (A B C)
```

FORTRAN:

```
INTEGER*4 A(3)/10 20 30/
CHAR*5 B/'ABCDE'/
REAL*8 C(2)/1.2,1.3/
CALL PROCESS(A,B,C)
```

When this was recognized, it became clear that subroutines written in other languages could be treated syntactically as locked APL functions. To complete the design of this facility, "associated processors" were invented and the system function □*NA* was introduced to declare a name to be external to APL.

□*NA* is used to declare the name of a variable, function, or operator to be external to APL and to be associated with a specified processor. When that name is subsequently encountered during execution of an APL expression, control is passed to the associated processor to perform the computation required to reference or specify the variable, or to execute the function or operator with the arguments and operands provided. On completion of this synchronous call to the associated processor, execution of the APL expression continues with any results returned.

The processing to be performed on an external name when control is passed to its associated processor is not defined in the APL language. An APL system may provide many associated processors to deliver different sorts of function to the APL users. When this facility was initially introduced in APL2 Version 1, Release 2, two associated processors were supplied to provide support for calls to routines written in FORTRAN, assembler, and REXX. Since that time, users have used these processors to call a wide variety of routines and languages including PL/I, COBOL, C, and Pascal.

The problem of argument coercion to the data types expected by the external routines in languages like FORTRAN was solved by providing facilities in the associated processor to allow descriptive information to be associated with any of the called routines. This information, among other things, provides descriptions of the expected arguments and their data types for an external function. When the function is called, it is used to determine if the expected arguments have been provided, and if the data types of those arguments need to be transformed to data types expected by the external function. A similar process is used to transform results from the external function to data types acceptable to APL.

One of the real advantages of this solution to the requirement for calls to non-APL routines is that these external routines look just like APL locked functions. Thus it is possible to write an application entirely in APL and then replace portions of it with routines written in other languages; or it is possible to design a heterogeneous application without doing damage to the syntax of the APL portions of that application.

In APL2 Version 1, Release 3, the facilities supporting external functions were extended to allow external functions called from APL to issue calls back to APL. Using these facilities, non-APL routines can request execution of APL functions or operators, or can reference or specify APL variables. This extension could be particularly useful for external operators whose operands might be APL functions, or for an APL compiler that might choose to compile parts of an application but use APL primitives for other parts.

Recently, an enhancement to APL2 Release 3 was made to allow non-APL application programs to invoke APL and issue calls to it. Using these facilities, applications written in a wide variety of languages can conveniently and simply execute APL functions, passing arguments to them and receiving results from them. Using the same facilities, the non-APL application can also reference or specify APL variables, or pass control to the APL interactive environment.

Namespaces represent an important advance in APL systems.

APL namespaces. When external functions and associated processors were designed, the interface was structured such that calls to routines written in APL could be accommodated. In particular, ambivalent functions and operators were not excluded in the interface syntax.

After considerable discussion and experimentation, it was decided to use this facility to address the problems of name scope isolation and shared code for APL applications.[6]

With an extended interface provided in APL2 Release 3, it is possible to declare an APL variable, function, or operator to be external to the workspace and to exist in another "namespace." A namespace differs from an APL workspace in two ways. First, it is formatted to allow it to be handled by the operating system facilities used to load programs, rather than in the normal format of a saved workspace. Second, it is accessed in a read-only mode; the results of computations are never actually stored in a namespace, but rather in the user's active workspace from which the namespace was accessed.

Like the active workspace, each namespace defines a name scope. A name scope is simply a set of names of variables, functions, and operators and the values and definitions associated with them. Users are able to declare names to exist in a namespace, in much the same way that external function names are declared with ⎕*NA*. When the name of an external APL function, operator, or variable is encountered during the execution of an APL expression, the system locates the namespace in which it exists and switches to the name scope of that namespace in order to process that name.

For an external APL function, this means that arguments to the function are provided from the caller's name scope, but names referred to in the body of the function come from the namespace's name scope. For an external APL variable, it means that the value comes from the namespace name scope when the variable is referenced, and is set in the name scope of the namespace when the variable is specified.

Since namespaces are accessed on a read-only basis, they may be shared between users. New or modified values or function definitions in a namespace name scope are actually saved in the user's active workspace. Thus, if more than one user accesses the same namespace, the system behaves as if each has its own private instance of it. Further, the state of the namespace, if modified as a result of execution, is maintained and can be saved and reloaded along with the workspace with which it is associated.

Namespaces represent an important advance in APL systems:

- They provide a simple, convenient, and powerful way to segment applications and to deal with the problems of "name pollution" common in large applications.
- They allow dynamic access to segments of an application without)*LOAD* or)*COPY* commands.
- They provide a mechanism where application programs can be shared by multiple simultaneous users; this is particularly important for large popular APL application packages.

Comparison of facilities

As previously described, there are three major facilities provided in the APL language that allow access to things outside the APL workspace: system functions and variables, shared variables, and name association.

Had all three of these facilities been proposed for incorporation into the APL language at the same

time, all three probably would have been accepted. Clearly, there are advantages and useful applications for each of the facilities. It should also be clear that there is a substantial amount of overlap in the applications for which each facility has been used. Many applications could be implemented with any one of the facilities, and the specific choice that was made in many cases reflected the state of APL implementations at the time, rather than any particular reason that one facility was better for an application than another.

System functions and variables offer the advantage that they are "omnipresent," and create no name conflicts with application-defined names. A unique function or variable, however, is required for each distinct operation. Unless restrictions are placed on implementers, this will inevitably lead to a large and unmanageable number of system functions and variables, and conflicting names between implementations. The APL standard defines about 20 system functions and variables; APL2 defines 41; another popular implementation defines over 120.

Some system functions and variables are clearly part of the language and are required for execution of most applications. ⎕*IO*, ⎕*CT*, and ⎕*NC* are certainly in this class. Further, it is appropriate that they be implemented as system functions and variables rather than primitives, because they have to do with the implementation of APL as a programming language, rather than as a machine-independent language. Other functions like ⎕*SVO* or ⎕*NA* must be implemented as system functions if they are to provide access to facilities that in turn provide extra-linguistic function.

It is not clear, however, that system functions and variables like ⎕*DL*, ⎕*ARBOUT*, ⎕*AI*, and ⎕*UL* should be part of the language. None of these is required for proper operation of the primitive functions and each could easily be implemented as an external function or with shared variables.

There are no explicit rules or guidelines to tell implementers whether a facility should be implemented as a system function, a primitive, or an external function. There is some consensus that primitive functions should deal only with abstract objects (arrays of numbers and characters), while management of the APL environment or interface to things outside the APL environment should be provided with nonprimitive functions. All of the system functions defined in the APL standard or APL2 have to do with APL as a computer programming language, and thus are appropriate nonprimitives. There are, however, a number of primitive functions like ⍋, ⍒, *?*, and ⌹ which might better be implemented as something other than primitives.

The distinction between the shared variable and name association facilities is a little clearer. Shared variables implement a general-purpose, asynchronous communication facility between cooperating but independent processes. Name association, on the other hand, allows the processing associated with function call and variable reference or specification to be handled in a synchronous manner by an external processor and in a name scope other than the user's active workspace.

There is some consensus that primitive functions should deal only with abstract objects.

Because system functions and shared variables predated the implementation of name association, these earlier facilities were sometimes used to implement function that is more appropriately handled by name association. File I/O is a good example. There is a need for access to many different file subsystems from APL, which often require the use of different syntax and arguments and whose use may be desirable in one application but not in another. Typically, the access to file subsystems is most conveniently implemented with synchronous subfunction calls, rather than with the more complicated shared variable interface. Because of the diverse requirements for functions to handle these interfaces and because of the number of functions required for full support of an access method, it makes most sense to implement these functions as external functions rather than system functions. One final advantage of the external function approach is that it is possible in some cases to change access methods by merely changing the name association of the external functions.

Another class of functions that are more appropriately provided as external functions include *?*, ⌹,

dyadic ⍕, and ⎕FMT. Each of these functions implements one of a set of acceptable solutions. For example, ? generates random numbers with a flat distribution. While this is acceptable in many applications, there are certainly lots of other applications where other distributions would be more appropriate. Where functions exhibit this characteristic, they should be provided as defined or external functions rather than primitives or system functions.

Choice of the correct facility. From the foregoing, it should be clear that the choice of a "correct" facility for the implementation of a specific function is not simple. There are no clear-cut guidelines, and many new proposals fall into grey areas. Nonetheless, there are some principles that should be kept in mind when choosing a facility to implement specific function:

- APL is designed to be an abstract language whose definition is machine-independent and need not be associated with a computer system in any way. Primitives in the language should adhere to these principles.
- Primitives in the language should be useful across a wide variety of applications and a wide variety of users. Further, they should be general and usable in conjunction with other primitives to provide rich function.
- Function should not be implemented as primitive where only one of a set of commonly acceptable solutions is implemented. Random number generation is an example of such a function. It is useful only if the particular mathematical algorithm used is appropriate to the user's problem.
- System functions and variables are part of the language. Users should be able to depend on their availability across implementations. Use of a system function or variable should not inhibit the portability of an APL application.
- There is no such thing as a primitive variable. Thus, variables such as ⎕IO or ⎕CT, which are implicit arguments to primitive functions, are appropriately implemented as system variables.
- Functions that are needed to declare the machine-dependent characteristics of an APL object (such as "shared variable" or "external function") are appropriately implemented as system functions.
- Functions required to manage the contents of a workspace, such as ⎕NC, ⎕NL, ⎕CR, and ⎕FX, are appropriately implemented as system functions. Care should be exercised in this area, however, since other commonly accepted system functions such as ⎕TF can be easily defined based upon ⍕, ⎕CR, and ⎕FX. Redundant function should be avoided.
- The availability of external functions and variables makes it possible to implement a great deal of commonly used function with acceptable performance characteristics. In a large number of cases, external functions and variables are a more appropriate implementation vehicle than system functions and variables.
- External functions use a synchronous interface to facilities outside APL that can be thought of as a subroutine call. Shared variables, on the other hand, provide an asynchronous communication channel and are more appropriately used where this asynchronous characteristic is important.

Improvements and extensions

Given the opportunity to do it all again, there are certainly some things that would be done differently. In a perfect world, implementers would be more clairvoyant and would easily choose between primitives, system functions, external functions, and shared variables. Unfortunately, given the broad base of existing users and their investment in APL application code, it will be difficult to make any radical changes in the short term. Existing facilities will have to continue to be supported, probably for a considerable length of time. We can hope, however, that as new function is implemented, appropriate facilities will be used, and that the benefits inherent in the use of that new function will quickly attract users.

With regard to the facilities themselves, however, a number of improvements and extensions can be envisaged:

- While the use of system functions and variables to implement new function should be avoided in many cases, the usability of system variables could be improved with a simple extension. If pass-through localization[7] was provided for system variables, certain operations, which are cumbersome now, could be made much simpler. For example, with pass-through localization a function could easily capture its caller's ⎕IO before setting its own:

```
    ∇Z←L F R;⎕IO;IO
[1]  IO←⎕IO ⍝ GET CALLER'S ⎕IO
[2]  ⎕IO←0  ⍝ BUT USE ⎕IO←0
     .....
```

- It is sometimes possible to make simple changes to auxiliary processors that result in substantial performance or usability improvements. For example, APL2's AP 111 has been extended recently to support matrix output. It could also easily be extended to support matrix input.

- Variables in APL namespaces are currently copied into the user's workspace before they are used. It was just simpler to implement the system that way. An obvious extension would allow external variables to be used without first having to make a copy of them. With such an extension, namespaces could be used as data spaces housing large, shared, in-memory tables of data.

- Shared variable processor facilities could be extended to allow communication between physical machines. Such an extension might be particularly useful between APL applications running in a client/server relationship, for example, between workstation and host-based applications.

- Similarly, associated processors could be developed to generate remote procedure calls to cause external functions to execute on a different physical machine. Again, such an extension would be particularly useful to a workstation APL implementation where the power and facilities of a host machine might be highly attractive. Such an extension would allow true distributed processing without any change to the language or to many existing applications.

- The introduction of external functions and associated processors into APL represents an important advance, allowing hybrid applications to be constructed from a variety of tools or languages. The facilities provided with APL2 are nonetheless relatively rudimentary at the present time and could be extended and simplified to make the construction, testing, and maintenance of such hybrid applications considerably simpler.

- As described in this paper, facilities to perform input/output (e.g., file I/O, screen I/O, etc.) have been implemented in a variety of ways including locked functions, system functions, shared variables, and external functions. All of these implementations introduce a degree of complexity to the APL user who simply wants to treat data as data, irrespective of the source or destination. The introduction of large workspaces in APL2 demonstrated that when all data used by an application could be maintained in APL variables in the workspace, the complexity of the application was often reduced substantially. The technology provided with associated processors, if extended in a few areas, could provide a mechanism that would allow data on files, or for that matter, data on the user's screen to be treated by the application as if the data were resident in variables in the user's workspace. Indeed, limited forms of this approach have been implemented in some systems with shared variables or system variables used to access external data. The use of external variables and associated processors offers an opportunity for generality and power not afforded by earlier approaches.

These examples of improvements and extensions range from suggestions that would make the facilities in today's APL implementations more usable and more valuable, to extensions that open up new opportunities for APL applications and for the exploitation of system facilities from an APL environment.

Conclusion

APL was originally conceived as a mathematical notation used to express ideas and algorithms. When it was later found to be a useful computer programming language, it became evident that its domain had to be expanded to provide connectivity to systems and facilities outside the APL workspace.

The mechanisms that provide connectivity between APL and other facilities in the computing environment have evolved over more than 20 years. There is no evidence to suggest that this evolution is complete. In fact, it seems to have been accelerating recently. In the first 20 years, we made many mistakes by rushing to use existing interfaces to solve all problems, often without a good understanding of the interfaces and without attempting to determine whether completely new types of interfaces need to be developed. The unfortunate part of this story is that users have made substantial investments in application code that is often difficult and costly to migrate to new and better facilities as they emerge.

The wise APL application developer develops an application as a set of building blocks that can be replaced as better technology becomes available.

* Trademark or registered trademark of International Business Machines Corporation.

Cited references and notes

1. A. D. Falkoff and K. E. Iverson, "The Design of APL," *IBM Journal of Research and Development* **17**, No. 4, 324–334 (1973).
2. Formally, the names of system functions and variables may begin with either ⎕ or ⍞; however, to date no ⍞ names, other than ⍞ itself, have been introduced.
3. R. H. Lathwell, "System Formulation and APL Shared Variables," *IBM Journal of Research and Development* **17**, No. 4, 353–359 (1973).
4. A. D. Falkoff, *Some Implications of Shared Variables*, Technical Report 02.688, IBM San Jose, CA (June 1975).
5. A. D. Falkoff, K. E. Iverson, E. H. Sussenguth, "A Formal Description of System/360," *IBM Systems Journal* **3**, Nos. 2 and 3, 198–261 (1964).
6. It should be noted that this choice is implemented by a particular associated processor provided with APL2. Other associated processors could be implemented to offer other choices to the user.
7. With pass-through localization, a local variable retains its global value until specified.

Accepted for publication July 24, 1991.

Michael T. Wheatley *IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141.*

Mr. Wheatley is currently a Senior Technical Staff Member in the language products development organization in the IBM Santa Teresa Laboratory. He has been involved with APL marketing, support, and development within IBM for over 20 years. From 1979 to 1989 he worked with James Brown on the design and implementation of APL2. As part of that effort, Mr. Wheatley led the design and implementation teams for the shared variable processor, auxiliary processor, associated processor, external function, and namespace components of APL2. Mr. Wheatley graduated with a B.S. in mathematics from the University of Montreal in 1966. He holds two patents, one patent on file, and one published invention disclosure, all of which are APL-related. He is a recipient of an IBM Outstanding Innovation Award for his work in the design and implementation of APL2 namespaces. Mr. Wheatley is currently the cross language architect in the Santa Teresa Laboratory with lead technical responsibility for IBM language products.

Reprint Order No. G321-5445.