JULY/AUGUST 1981
Volume 9/Number 4

I.P. Sharp
newsletter

ENHANCEMENTS IN

NEW RELEASE OF SHARP APL

Paul Berry, Palo Alto

Twice a year I.P. Sharp Associates brings out a new release of SHARP APL to its time-sharing customers and to the installations that run SHARP APL in-house. While such things are hard to quantify, it seems fair to say that the June '81 release contains more significant new features than any since files were first added to APL\360 in 1970.

(see over)

MAJOR NETWORK CHANGE

Roger Moore, Toronto

During May a major change was made in network software: the method used to route data between a terminal and an APL T-task has been altered. The change will allow a larger and a more complex network. Complex interconnection provides more protection for the user against *LINES DOWN*, with more than one route from a terminal to APL. At the same time, protection against network errors caused by very badly delayed packets has been incorporated into the system.

The network topology has already been modified to take advantage of the new software. Links have been installed between Seattle and Vancouver and between San Francisco and Newport Beach. These new links provide extra paths to APL for Californian and western Canadian nodes. The chance of getting a continuing *LINES DOWN* condition is reduced by the new software and the new links.

(see page T1)

IN THIS ISSUE

ENHANCEMENTS IN THE NEW RELEASE

The New Features

- **Enclosed arrays** Each element of an array may itself be an array in "enclosed" form.

- **New functions**

$<\omega$	*Enclose*	Encode array ω as a scalar
$>\omega$	*Disclose*	Inverse of $<\omega$
$\alpha\equiv\omega$	*Match*	Are α and ω identical?

- **New operators** Create a function as *composition* of two functions, and execute it for each cell of array arguments.

$X\ \alpha\ddot{\circ}\omega\ Y$	OVER	For each cell: $\alpha X\omega Y$
$X\ \alpha\overset{\cdot\cdot}{\circ}\omega\ Y$	ON	For each cell: $(\omega X)\alpha(\omega Y)$
$X\ \alpha\overset{\cdot\cdot}{}\omega\ Y$	WITH	For each cell: ω' $(\omega X)\alpha(\omega Y)$

- **Complex numbers** Square root of ¯1 is 0J1.
  Extended domain of + - × ÷ ⋆ ⍟ ∧ ∨ ! etc.
  New ○ functions for Cartesian and polar complex representations.

- **S-Tasks** AP-1 (an auxiliary processor) uses shared variable rather than terminal to provide complete control of an APL task.

- **IBM 3270 family of terminals** New support for full-screen manager and 3279 colour terminals (for in-house systems under MVS with VTAM).

- **APL access to OS data sets** AP-370 is compatible with IBM's TSIO (for in-house systems under MVS).

- **Speedups to inner product**

- **User-settable CPU limits**

Enclosed arrays

In APL, a data object (unless it is a *package*) is a rectangular array. An array can have any number of axes; each axis can have any length. Each of the elements within an array is a scalar: something that has no axes. None of the foregoing has changed with the introduction of enclosed arrays.

To enclose an array is to encode it as a scalar. Its inner structure is concealed, and it appears as a single structureless point. The format function ⍕ has not yet been extended to enclosed arrays, so expressions such as ⎕←<*A* or <*A* at present yield the message ***ARRAY***.

Since an array that has been enclosed thereby becomes a scalar, it can then be an element within another array. For example:

```
N1←?3 5⍴100          Create
N2←÷?N1              three simple
C←'DATA TABLES'      arrays.

Z←(<C),(<N1),<N2     Enclose each
                     and catenate.
```

Although the inner structure of each enclosed element remains invisible as long as the element is enclosed, it is possible to reverse the process and thereby *disclose* the structure hidden within it. For example:

```
      >Z[1]          Disclose one
DATA TABLES          element.

      >Z[2]
14 76 46 54 22
 5 68 68 94 39
52 84  4  6 53
```

Generally speaking, you can disclose only one scalar at a time. However, it is all right to disclose several at once *if,* when disclosed, they all have the same type and shape. In this example, *N*1 and *N*2 are both numeric matrices with shape 3 5, so you can disclose Z[2 3] by an expression such as >Z[2 3] (or >1↓Z etc.) and get an array having shape 2 3 5.

All the structural functions, such as ⍴ ↑ ↓ ⌽ ⊖ , [] ⍳ etc., apply to an enclosed array in the usual way. For example, to find where in *Z* an enclosed encoding of *N*1 may be found, execute *Z*⍳<*N*1.

Homogeneity of arrays. The domain of , (catenation) is restricted; you are not permitted to join numeric data and character data in the same array. There is a similar restriction on catenation of enclosed data. All the elements of an array have to be of the same type: *numeric* or *character* or *enclosed.*

Computation with enclosed arrays. A function that involves computation (such as + × ⊥ etc.) cannot be applied directly to an enclosed array. You have to construct an expression that discloses each of the elements in turn and does the computation on the disclosed value. Three new operators (described in the next section) make that easy.

The new function *match* (written *A*≡*B*) is not restricted to enclosed arrays, but it does simplify working with them. The result of match is always a scalar 1 or 0. It is 1 when both arrays are identical. An enclosed scalar matches another enclosed scalar if, when each is disclosed, the results are of the same type, rank, shape, and value. A simple array (that is, one made up of numbers or characters) never matches an enclosed array; however, an empty array matches another empty array of the same shape, without regard to type.

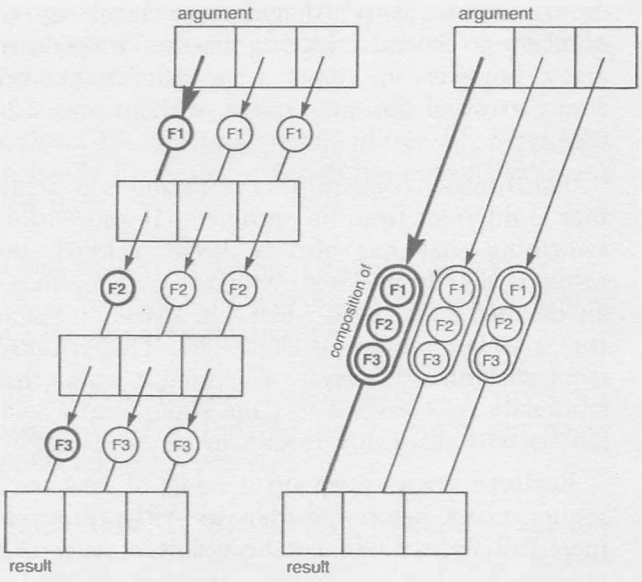
The function *enclose* always produces a result that is different from its argument. If you enclose something that has already been enclosed, the result is doubly enclosed. Moreover, <*X* produces an enclosed scalar even when *X* is already a scalar (say a scalar number or character). This is called *strict* enclosure. However, *dis*closing a scalar that is already *simple* gives you the scalar unchanged; that is why disclosure is said to be *permissive.*

Enclosed arrays open up a range of new techniques never before possible in APL. Probably their full impact will not be visible at once. Although this release contains the foundations, it does not yet include all of the facilities that have been proposed. The implications of being able to enclose an array have been debated in the APL community over several years. Certain theoretical issues remain unresolved. I.P. Sharp Associates has elected to move cautiously, implementing only those features about which there is strong agreement, and deferring until subsequent releases others which, although attractive, have implications that need further exploration.

Operators

From its earliest days, APL has included both *functions* and *operators*. A function works with data; by contrast, an operator is something that *creates* a function. (In the expression +/*A* the symbol / denotes an operator whose argument is the function +. The result of +/ is the function *summation*. A function produced by an operator is called a *derived function*. The derived function summation takes as its argument the array *A*.)

The June '81 release of SHARP APL includes the first new operators since \ (scan) was added in 1969. They are ⍤ (OVER), ⍥ (ON), and ⍣ (WITH). These three provide various forms of *composition*. A composition operator takes two functions as arguments, and creates a derived function. Each of the composition operators generates a derived function that is *ambivalent*. For example, if you want the floor of the log of *X*, ⌊⍤⍟ creates a function floor-of-log, and ⌊⍤⍟ *Y* applies it to an array *Y*. If you want the floor of the base-*X* log, *X* ⌊⍤⍟ *Y* creates the same derived function, but this time its dyadic definition applies.



Drawing: Janet Okagaki

The derived function produced by a composition operator applies *independently to each cell* of the argument array. (A *cell* is the array needed for a single evaluation of a function. For all the functions defined on scalar arguments, "cell" is the same as "element". Certain functions, such as enclose or ravel, treat the entire argument as a single cell.)

Having composition apply to each cell (rather than to the array as a whole) is called *close* composition. To appreciate the importance of close composition, consider the expression ⍴⍣>*Y*. It finds the shape that each element of *Y* has when disclosed. That is *not* the same as ⍴>*Y*, which would mean "first disclose all elements of *Y*; then find the shape of that result". By contrast, ⍴⍣>*Y* discloses an element of *Y*, immediately finds its shape, and then encloses it. That triple sequence is repeated for each element of *Y*.

When the functions combined by the composition operator are both *scalar* functions, the effect of composition is no different from what you would get without composition. Because ⌊ and ⍟ are both scalar functions, ⌊⍥⍟*X* gives you the same result as ⌊⍟*X*. It is when you apply a composition operator to a *non*-scalar function that the derived function thus created has properties you could not get in any other way (or only by writing an explicit iteration).

⍤ **OVER** [*X*] α⍤ω *Y*

Derived function executes for each cell:

α [*X*] ω *Y*

(where [ ] indicate the optional left argument)

Examples: Find the reciprocal of each sum of a cell of *X* and a cell of *Y*:

X ÷⍤+ *Y*

Enclose each cell produced by summing a cell of *X* with a cell of *Y*:

X <⍤+ *Y*.

For dyadic expressions such as *X* α⍤ω *Y*, the function ω to the right of ⍤ is restricted in the June '81 release to a scalar function. When that restriction is removed, expressions such as the following become possible:

Generate independent random deals, each having a length unrelated to the lengths of the others, and enclose each:

X <⍤? *Y*

⍤ ON [X] α⍤ω Y

Derived function executes for each cell:
[(ωX)] α ωY

⍤ (ON) differs from ⍥ (OVER) only when the derived function is used dyadically. With α⍥ω, the function on the *right* is interpreted ambivalently, whereas with α⍤ω the one on the *left* is.

Examples: Sum the logs of *X* and *Y*:

```
X +⍤⊛ Y
```

Form an array by disclosing corresponding elements of *X* and *Y*, and catenating each pair of arrays:

```
X ,⍤> Y
```

The array *X* consists of enclosed numeric vectors of varying lengths. Find the sum of each:

```
1 ⊥⍤> X
```

(This solution uses 1⊥*X* for "sum of *X*". In principle, you should be able to write +/⍤> *X*, but the June release does not yet permit a derived function to be the argument of a composition operator.)

A new style of vector-to-matrix—form a rectangular array from an array *A* of enclosed vectors (whose lengths are independent):

```
(⌈/, ⍴⍤> A) ↑⍤> A
```

Generate *R* repetitions of each element of *A* (where *R* is a scalar, so that the same number of repetitions is required throughout):

```
      R←4
      A←⍳3
      R ⍴⍤+ A
1 1 1 1
2 2 2 2
3 3 3 3
```

(In the foregoing, + is used as an identity function, since for real numbers, *A* ↔ +*A*.)

⍨ WITH [X] α⍨ω Y

Derived function executes for each cell:
ω′ [(ωX)] α ωY

Like ⍤, WITH generates a derived function which applies ω to each cell of its arguments, and then α to each of those results. But ⍨ goes further. For each argument cell, after executing ω, it retransforms that result by the *inverse* of ω, (shown here as ω′). Such a sequence turns up in many situations: before applying some function, you first convert the arguments, and then convert back the results.

Examples: Find the antilog of sums of the logs of *X* and *Y*:

```
X +⍨⊛ Y
```

Form an array in which each element is an enclosed vector containing somebody's name (given name followed by surname) from enclosed vectors *GIVEN* and *SUR* (which contain the given names and the surnames respectively, with each name having its own length):

```
GIVEN ,⍨> SUR
```

Join two arrays *X* and *Y* (of the same type and shape) along a new first axis:

```
X ,⍨< Y
```

The operator WITH requires the system to supply the inverse of the function to the right of the ⍨ symbol. At present, the system recognizes inverses for the following monadic functions: < > * ⊛ ~ + - ÷ ⌹; the last five are self-inverse.

Complex numbers

The square root of ¯1 used to cause a *DOMAIN ERROR*, but now the system knows better and reports 0*J*1. A complex number is a point on the number plane. It's just one point, but it takes a pair of numbers to describe it. In APL, the point whose position is at 4 on the real axis and 3 on the imaginary axis is written 4*J*3. Complex numbers become a fourth internal type for the representation of numbers. As with the other

internal types (Boolean, integer, and floating point) conversion between them is automatic. Each element of a complex array requires 16 bytes.

You cannot hold ¯5 apples in your hand, but negative numbers are very handy as intermediate results in all kinds of business calculations. So too with complex numbers: you will never see a warehouse with 5J1 cases stacked in it, but complex values are useful intermediate steps in the solutions of equations that turn up in engineering, in economic modelling, and in other fields as well.

With the June '81 release, SHARP APL is extended to return complex results where called for, and to accept complex arguments for any of the functions whose domains normally include them (such as power, logarithm, sine, cosine, GCD, LCM, Gamma function, and so on). The fact that multiplication and addition are defined on complex numbers is handy for work in graphics, because translations and transformations can now be written directly, without the need for the trigonometry previously required.

Several functions have been added expressly for conversion between various representations of a complex number. They are obtained by new left arguments to the ○ function. For example:

```
      X←4J3 ¯7      Two-element complex
                    vector
      9○X           Real part
4 ¯7
      11○X          Imaginary part
3 0
      9 11 ∘.○ X    Real and imaginary parts
 4 ¯7
 3  0

      10○X          Magnitude
5 7                 (same as |X)

      12○X          Arc (or phase angle)
0.6435011088 3.141592654

      10 12∘.○ X    Magnitude and arc
5           7
0.6435011088 3.141592654

      CART ← 9 11 ∘.○ X
      POLAR←10 12 ∘.○ X

      ¯9 ¯11 +.○ CART
4J3 ¯7              Resolve real and
                    imaginary parts

      ¯10 ¯12 ×.○ POLAR
4J3 ¯7              Resolve magnitude and
                    arc
```

Comparisons such as "smaller" or "larger" make sense only with respect to a line, and are in principle undefined for numbers on the complex plane. For that reason complex numbers are outside the domain of functions that directly or indirectly require a decision about which of two numbers is larger. That excludes complex arguments for the dyadic functions < ≤ > ≥ ⌈ ⌊ and for the grading functions ⍋ and ⍒. Gene McDonnell and Doug Forkes have proposed ways in which monadic ⌊ and ⌈ (floor and ceiling) might be defined for complex numbers, and perhaps also residue, encode, and decode, but their work has not yet produced complete agreement, and so these functions do not yet accept complex arguments.

S-tasks and a general interface to APL

When you sit at an APL terminal and type instructions, what you type is relayed to a terminal controller, which collects the message formed by your keystrokes, and then passes it to the APL system for processing. The responses from APL go back to the terminal controller, are suitably divided into transmissions, and then forwarded to your terminal. This intermediary is as discreet as possible, and usually you are not at all aware of it. You think of yourself as communicating directly with APL.

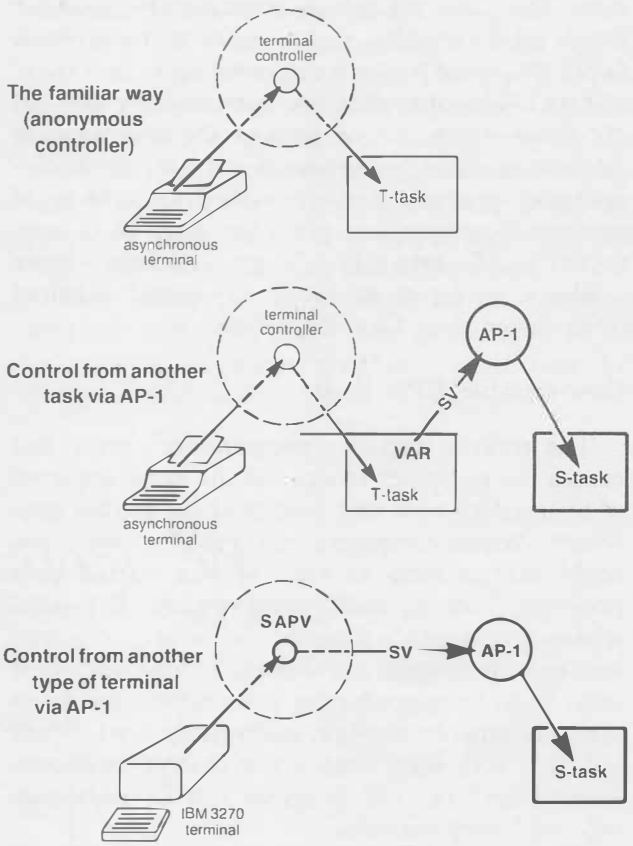
An *auxiliary processor* is a program running on the same central computer as APL, not itself part of APL, but able to share a variable either with an APL workspace or with another non-APL program. AP-1 (auxiliary processor 1) is able to communicate with the APL interpreter in the same way as the anonymous terminal controller. That is, AP-1 can start an APL task, send it instructions and receive from it responses. But instead of receiving its input from a terminal, or sending its output to a terminal, AP-1 communicates through a shared variable. A task whose work is controlled through AP-1 is called an S-task.

With whom does AP-1 share a variable? If the variable is shared with another APL workspace, then that workspace in effect controls a second active workspace. In some ways that is like an N-task, but an S-task is considerably more responsive. An S-task can do anything that a T-task can do; indeed, as far as the APL interpreter is concerned, there is no difference between S-tasks and T-tasks.

If you share a variable with AP-1 (by an expression such as `1 ⎕SVO 'VAR'`) you can send it instructions just as you might type them from a terminal. When you set a value of the variable you've shared, you transmit a message to AP-1, which passes it on to the APL interpreter. When you refer to the variable you've shared, you read the reply that AP-1 brings you from the interpreter. The first thing you need to send is a sign-on message so that you can start an APL task. AP-1 passes your request to the APL system, and returns to you the usual system response: a blot. If you then supply a valid sign-on, you have a new task at your command. Not only can the S-task be given any expression to execute, but it can be interrupted, interrogated, asked to load a new workspace, and so on; it can even be made to continue after its parent task signs off, and to accept control again when the parent returns. The development of S-tasks makes possible a whole new approach to the design of applications, using concurrent workspaces, in some ways independent and in other ways closely linked.

IBM 3270-series terminals

When AP-1 shares a variable, it is not limited to sharing with an APL workspace. It can share just as well with another background program. In particular, it can share with the terminal controller for IBM 3270-series terminals (called SAPV). These video-screen terminals (the model 3279 has seven-colour video) require considerable processing at the central computer, and a rate of transmission to refresh their screens far higher than is feasible over ordinary telephone lines. They are usually connected by hard-wired coaxial cables directly to the computer, or to a remote terminal controller (which is in fact a small computer). For this reason they cannot be used at all over the SHARP dial-up network. Support for them is available only at in-house installations using VTAM for terminal control, and running under the MVS operating system. A user of SHARP APL who is seated at an IBM 3279 terminal is in fact running an S-task rather than a T-task—but sees nothing different, other than the characteristics of the terminal itself.



Drawing: Janet Okagaki

Access to OS Data Sets

SHARP APL now offers AP-370 (also called PJAM, for "Pseudo-JCL Access Method"). By sharing a variable with AP-370, you can pass instructions to the MVS operating system.

PJAM is capable of reading and writing OS data sets (that is, non-APL files). This makes it possible to run on SHARP APL installations programs written using the IBM processor called TSIO. If you are in such an installation, you can now choose to work with standard files, generated perhaps in COBOL or FORTRAN environments, without having them converted to APL files.

Speed-ups to inner products

Inner products are extraordinarily useful. Some enthusiasts claim that all of accounting can be represented by a few matrix equations. APLers have found the extended matrix operations that inner products provide (such as ∨.∧ or +.∧) elegant to write but sometimes disappointingly slow to execute. The June '81 release contains new code for inner product which makes most of them work faster. (∧.= and ∨.≠ were speeded up in an earlier release.) The ones that are spectacularly affected are those where one or both of the arguments is Boolean, and the workspace is not too full. Boolean inner products are often used as a form of partitioned summation, or in the analysis of connectivity of networks. These Boolean inner products now run as much as several hundred times faster than they did before.

User-settable CPU limit

The endless loop is a programmer's error that causes the computer to execute the same sequence of instructions over and over, without getting anywhere. When computers ran rather slowly, you could suspect such an error if you started your program running and nothing had happened within a reasonable time. As the central computer has become bigger and faster, a loop (or some other form of unproductive calculation) can do an immense amount of work, and run up a whopping bill, in a very short time. On a modern computer, such "bugs" in your program can be embarassing—and very wasteful.

Not to worry! Help is at hand. The new system command *)CPULIM* permits you to set a limit on the amount of CPU usage since your last keyboard entry, and also on the total accumulated CPU charge during a task. The limits apply only to T-tasks. The command *)CPULIM* takes one or two arguments:

)CPULIM 10	*Sets per-entry limit to 10.*
)CPULIM 2 50	*Sets per-entry limit to 2, and cumulative limit to 50.*

When the computer notes that the limit has expired, it generates an interrupt, event 1003. The effect is the same as signalling BREAK (or ATTN) twice. You can then take steps to find out why you have used more CPU units than you expected. Whatever you do next (diagnose the current program, resume its execution, or something else entirely) the computer starts a fresh timing of your per-entry CPU consumption. It is possible to trap event 1003, but inadvisable: if your recovery expression instructs the computer to resume execution, you have defeated the computer's warning.

To remove the CPU limit, execute *)CPULIM* 0 0.

For more information...

Operators. K.E. Iverson, *ACM Transactions on Programming Languages and Systems,* Vol. 1, No. 2, Oct. 79, pp. 161-176.

Operators and Enclosed Arrays. Bob Bernecky & K.E. Iverson, *Proceedings, APL Users Meeting, Toronto 1980.* I.P. Sharp Associates. pp. 319-331.

SATN-41. Composition and Enclosure. K.E. Iverson.

Proposal for a Complex APL. Paul Penfield Jr., *APL79 Conference Proceedings,* ACM, pp. 47-53.

SATN-40. Complex Numbers. E.E. McDonnell.

SATN-39. The SHARP APL S-Task Interface. R.H. Lathwell.

SATN-38. PJAM User Guide. Paul Jackson.

SATN-37. IBM 3270 Terminal User Guide. J.D. Burger.

SIGNOFF MESSAGE CHANGES

After the first file system was implemented in APL 10 years ago, the development group printed the message *FILES UNTIED* every time a user signed off. It seemed time, in the light of the new enhancements to SHARP APL, to let the message disappear.

technical supplement 33

NETWORK SOFTWARE CHANGES

Roger Moore, Toronto

A node has from one to six links which connect it to the rest of the network. Every node which has more than one network link requires some routing protocol to dispose of packets addressed to other nodes. A routing protocol takes as input some address information from the packet and some tables stored in the node. These tables may be fixed or may change as a result of commands or advice from other nodes in the network.

Both the old and the new routing protocol start by numbering the links of a node from one to six. The usual output of the routing algorithm is a link number along which the packet can be transported to some other node. Other possible outputs are an indication that the packet is destined for this node, or that there is no link which can be used to forward the packet. The old routing algorithm used a fixed length vector which had one entry for every node number in the network. The 16-bit address field of a packet was decoded by `NODELINE← 256 256⊤ADDRESS`. The expression `RT[NODELINE[0]]` gave a link number for the destination node. For packets such that `OWNNODE=NODELINE[0]`, `NODELINE[1]` addressed a particular terminal (or T-task) attached to the node.

The original network routing protocol was designed for a network with star connectivity. In a star network there is only one path between any pair of nodes. If some link in that path fails, the two nodes cannot communicate. In June 1977 the restriction was slightly relaxed to tolerate simple loops. With considerable awkwardness, traffic flow could be routed to avoid a failed link. This was a little cumbersome as the software imposed some consistency requirements on routes. The route from node A to node B can be represented as a vector of node numbers `A2B`.

To be meaningful this vector must have the following properties: `A=1↑A2B` ; `B=¯1↑A2B` and all rows of `A2B[(¯1↓⍳⍴A2B)∘.+0 1]` must represent physical links in the network. The old routing protocol imposed two additional constraints:

1] If traffic from *A* to *B* was using route `A2B` then traffic from *B* to *A* was required to use the route `⌽A2B`.

2] If traffic from *A* to *B* was using route `A2B` and `C∈A2B` held, then traffic from *A* to *C* had to use route `(A2B ⍳ C)↑A2B`.

These constraints required some care in calculating and altering the routing vectors of the ninety network nodes. An APL workspace was developed to maintain a matrix with the route tables for all nodes. The workspace expanded a manually specified loop solution and performed the obvious calculation on the star connected portions of the network. One minor motivation in eliminating this routing system was that a *WORKSPACE FULL* appeared in May when the function attempted to make a third copy of the matrix with rho of `126 126`.

The current routing protocol uses a scheme which is biased towards timesharing applications. Communication within the network is assumed to be between two points. The most common example is a terminal which is logically connected to a single APL T-task. (This T-task may be sharing variables with other tasks but this is beyond the scope of the network.) This logical connection between a terminal and an APL T-task is referred to as a 'virtual call'. The name is derived by an analogy with the public telephone network which usually establishes a physical connection between two telephones. Every call in the network is assigned a unique number in the zero to 32767 range. Packets are routed by call number rather than by destination node number. The original 16-bit address field is decoded by
`32768 2⊤ADDRESS`. The first element is the call number and the second element is the direction of travel (towards the zero or one end). The routing algorithm is:

```
ORGDST ← VCT[VCT[;2]⍳⌊ADDRESS÷2;
                    0 1 =2|ADDRESS]
```

If the packet is legitimate, no index error will result and the packet will have been received from `ORGDST[0]`. An illegitimate packet is discarded (with a warning message to the network logging system). A legitimate packet is forwarded to `ORGDST[1]`. `ORGDST[1]` when ≤6 is assumed to be the number of a network link within the node. When `ORGDST[1]≥8`, the packet is destined for the current node. `ORGDST[1]-8` is the line

number `256|(2 ⎕WS 3)[9+⎕IO]` within the node (assuming destination is a terminal). A node begins operation with `0 3↔⍴VCT`. As calls are established from, to or through the node, rows are added to the *VCT* matrix. Call termination via *)OFF*, *LINES DOWN* etc. will remove the row from the matrix. Termination of a call within a node involves waiting until all data packets for that call to or from the adjacent nodes have been destroyed. This avoids a problem with the original routing protocol where a packet from an old call could be delayed by retransmission and later inserted into a subsequent call.

The current protocol has different consistency requirements than the original protocol. The representation assures that the route from the one end to the zero end is exactly the reverse of the zero to one route. Calls between a particular pair of nodes may follow different paths. This is a result of abandoning node number as a packet address.

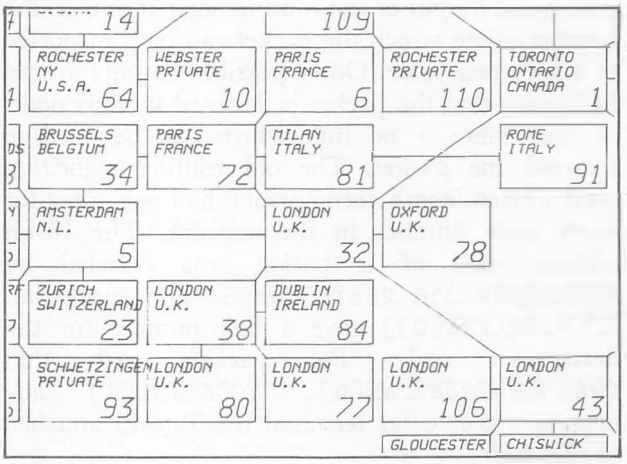
The important requirement introduced by the current protocol is the uniqueness of call numbers along a route. This is enforced by two methods. The first method is to avoid:

```
VCT←VCT,[0] ZERODIRECTION,ONEDIRECTION,
                                NEWCALLNUM
```

when *NEWCALLNUM*∈*VCT*`[;2]` holds. The second method is to delay the deletion of a *VCT* row until the adjacent nodes in the zero and one directions have indicated that they have stopped forwarding data packets with the call number being deleted.

New Call Setup

Call setup in the current protocol is adaptive. A broadcast call setup packet is created in the originating node. This call setup packet gives some information about the terminal which originated the call and names the destination node. The broadcast call setup packet is sent across all of the operational links from the originating node. A node receiving a broadcast call setup packet checks to see if it is the destination node. If not, the call setup packet is again forwarded down all network links from the receiving node except the link from which it was received. If the broadcast call setup packet is received by the destination node, the destination node delays for half a second and then accepts the incoming call. It is possible that the destination node or some intermediate node has received more than one copy of the original call setup packet. This may indicate that there is more than one route from the originating node to the destination node. The choice between multiple routes is made by assigning a weight to every link in the network. For two routes *R1* and *R2* if `+/WEIGHTS[R1] < WEIGHTS[R2]` holds then *R1* will be selected. For the case where the weight sums are equal, the route which is seen first will be selected. The delay at the destination node is to allow time for a low weighted route to be selected even if the setup packet is slightly delayed. The scheme requires that the weights be `>0`. Thus the sum around a loop will be greater than zero. For example (see below) if an attempt is made to set up a route from node 43 to node 38, node 32 will receive setup packets from nodes 43 and 1. It will forward copies to nodes 72 and 5 as well as 38. Routes involving the 32 5 34 72 32 loop will have a greater weight sum than those which exclude it and so bubble routes are rejected.



LINES DOWN is generated after trying every link in the network which is accessible from the originating node. Any endnode which is not the destination node rejects the call setup packet by sending a negative reply back up the link from which it received the setup packet. When a simple fanin node (such as 16 in the network diagram) receives negative replies from all its endnodes, it can send a negative reply up towards node 5 (assuming the call setup packet originally passed from 5 to 16). When a call setup packet is rejected by a loop member node because the link weight sum is not a new minimum, the loop member sends a negative reply along the link from which it received the rejected call setup packet. Failure of a link while a reply to call setup is expected implies a negative reply to any call setups along that link. Assuming all nodes which receive the call setup packet reject it within a finite time, the *LINES DOWN* message will eventually appear on the user's terminal. (see page 15)

Future Extensions

Some future extensions to the routing protocol are being considered. It is theoretically possible to change the route of an established call. This involves establishing *VCT* entries along the new route and eventually destroying the *VCT* entries in the old route. The operation must be performed in a consistent sequence to avoid loss of data. The usual reason for doing this would be in response to failure of a network link. Another reason might be to balance the load on the links of the network.

A somewhat simpler extension would be a change in the call setup method. The present APL system is served by two network nodes. Under normal circumstances, a T-task user would prefer to use the node which gives the best response. If the node which would normally give the best response is responding with some rude message such as *APL PROBABLY DOWN*, automatic selection of the alternate node would be desirable.

FACTORIAL CORRESPONDENCE ANALYSIS

Nathalie Merlin, Paris

The correspondence analysis method is used to analyse tables of data whose items consist of observed or collected frequencies of two phenomena. This base data matrix is called a contingency table. 'Classical' statistics provides a statistical test for a relationship between the phenomena, but rarely provides us with the means to describe this relationship, which is the object of this method.

If you denote the two phenomena as A and B, and let:
$A[1],\ldots,A[N]$ be the N levels of A and
$B[1],\ldots,B[P]$ be the P levels of B,
the $N\times P$ events $A[I]\cap B[J]$ $(I=1,\ldots,N\ ;\ J=1,\ldots,P)$ can be arranged in an $N\times P$ contingency table. The event $A[I]\cap B[J]$ is called the (I,J)th event. Let $K[I;J]$ represent the cell frequencies.

The N rows of the table can be represented as a cloud of points in P-dimensional space, the coordinates of the Ith point being $K[I;]\div+/K[I;]$, each having a mass $(+/K[I;])\div+/+/K$.

Similarly, the P columns can be represented in N-dimensional space, the coordinates of the Jth point being $K[;J]\div+/K[;J]$, with mass $(+/K[;J])\div+/+/K$.

Chi-square is used to calculate the relation between the observations, or, in other words, distance between the observations.

The analysis allows, within the structure of the first factorial axes, a simultaneous representation of:

- the relation between the columns of the table
- the relation between the rows of the table
- the proximity between the rows and columns.

The matrix manipulation functions of SHARP APL allow the easy implementation of the correspondence analysis method which will be added to the SHARP APL multivariate statistics public libraries.

The functions used for correspondence analysis can be found in the workspace 1843198 *CORAN*. More information is online in the *DESCRIBE*. Further inquiries are welcome and should be addressed to your local representative, or to the author.

FACTORS - CATEGORIES SOCIO-PROFESSIONNELLES
7 PRINCIPAL CATEGORIES SOCIO-PROFESSIONNELLES

J1	QLT	POID	INR	F1	COR	CTR	F2	COR	CTR	F3	COR	CTR	F4	COR	CTR
*01	978	26	70	¯139	83	¯80	175	132	222	244	255	708	344	508	1612
*02	703	8	31	¯36	3	¯7	¯385	335	¯159	347	272	330	¯204	94	¯312
*03	971	96	94	210	519	453	184	401	871	56	38	614	34	13	591
*04	992	149	207	326	904	1092	22	4	158	¯99	84	¯1678	6	0	159
*05	644	138	33	¯101	328	¯316	60	114	407	¯73	170	¯1143	¯32	33	¯808
*06	936	96	29	¯69	176	¯149	130	623	615	¯14	8	¯156	¯59	129	¯1037
*07	974	296	198	¯236	972	¯1578	¯8	1	¯123	¯8	1	¯267	0	0	¯16
*08	705	19	29	¯38	8	¯16	¯201	223	¯186	239	315	509	¯169	159	¯582
*09	999	46	142	9	0	9	¯474	863	¯1071	¯130	65	¯676	135	70	1130
*10	998	127	110	207	590	592	¯117	189	¯733	122	206	1759	¯32	14	¯736

FACTORS - MODES D'HEBERGEMENT EN VACANCES
7 PRINCIPAL MODES D'HEBERGEMENT EN VACANCES

J1	QLT	POID	INR	F1	COR	CTR	F2	COR	CTR	F3	COR	CTR	F4	COR	CTR
c01	989	146	260	329	729	1086	175	205	1260	87	51	1442	25	4	663
c02	906	109	67	¯78	116	¯221	138	368	861	¯60	69	¯855	¯135	353	¯3122
c03	996	72	218	408	655	943	¯131	68	¯664	¯263	272	¯3059	24	2	443
c04	988	411	130	¯64	153	¯352	¯143	778	¯1737	38	56	1072	3	0	148
c05	594	60	45	113	202	238	¯9	1	¯40	115	212	1228	¯106	180	¯1822
c06	949	124	186	¯301	724	¯917	141	158	936	¯85	58	¯1308	35	10	861
c07	750	43	50	¯213	471	¯382	163	274	637	¯15	2	¯138	15	2	212
c08	881	36	44	¯83	66	¯134	111	120	397	34	11	275	265	683	3504

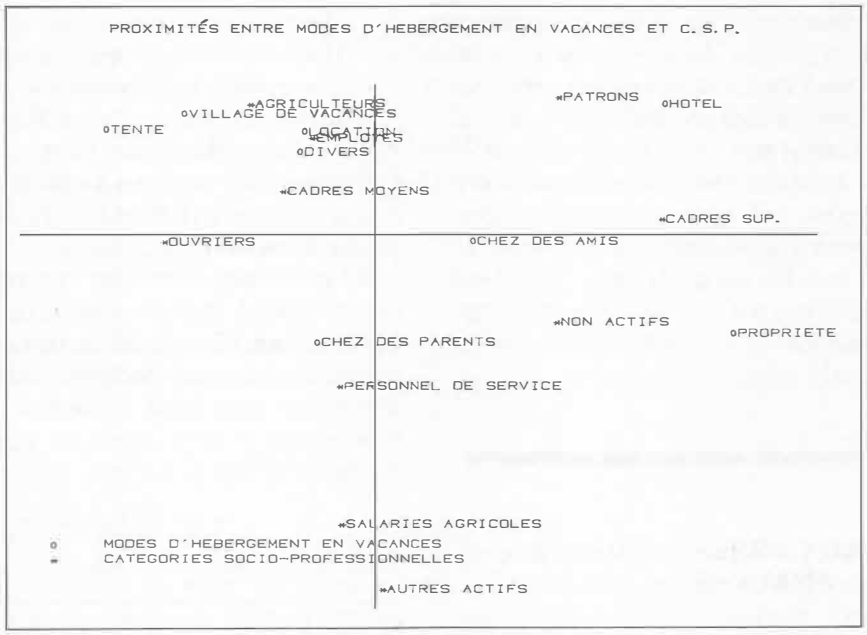
Fα : *COORDINATES OF FACTOR α*
COR: *RELATIVE CONTRIBUTION OF FACTOR α TO THE SQUARED DISTANCE FROM ROW I TO THE CENTER OF GRAVITY OF THE CLOUD*
CTR: *RELATIVE CONTRIBUTION OF ELEMENT I TO THE AXIS α*

EIGEN VALUE	%	RUNNING TOTAL	HISTOGRAM OF THE EIGEN VALUES
0.044431560	53.008	53.008	**********
0.02024917	24.221	77.229	**********
0.00880492	10.532	87.760	**********
0.00546530	6.537	94.298	*******
0.00238147	2.849	97.146	***
0.00214511	2.566	99.712	***
0.00024064	0.288	100.000	

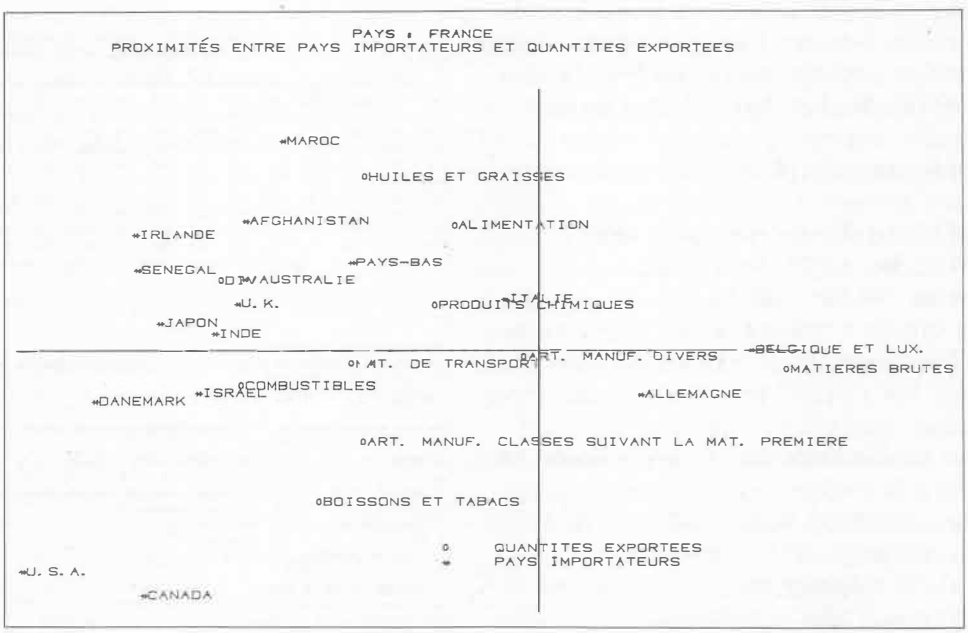
CPU : 25.023

The proximity of two 'catēgories socio-professionnelles' points indicates a resemblance in their 'modes d'hēbergement' profiles, as seen in the case of the 'agriculteurs' and the 'employēs'. In the same way, we can also say that 'village de vacances' and 'tente' have similar 'socio-professionnel' profiles.

The simultaneous representation of the observations and of the variables shows us which variables are responsible for those proximities. For example, the proximity of 'village de vacances' and 'tente' can be explained by a higher than average proportion of 'ouvriers'.



This graph gives a quick visual comparison between French exports to several countries for several commodities. It also points out the resemblance between the import statistics of Germany, Italy, Belgium and Luxembourg on the one hand, and of the United States and Canada on the other, as one would expect.



SHARED VARIABLES IN APL SYSTEMS
PART 1 of 2

Roland H. Pesch, Palo Alto

Two factors limited the use of shared variables within APL systems when they were first introduced by IBM;

- the lack of facilities for inspecting the shared variable state, and
- the lack of APL tasks not connected to terminals.

Neither of the facilities implemented in SHARP APL as □*SC* and □*SVS* was available to APL tasks, though equivalent (and perhaps more comprehensive) facilities have always been available for non-APL processors using the shared variable processor [1]. At least □*SVS* (current state of the shared variable) is necessary for any processor that is intended to serve more than one user (barring elaborate semaphoring arrangements involving □*SVC*, and requiring, like □*HOLD*, the cooperation of both partners) since otherwise there is no way of determining which users need to be served.

The presence of APL non-terminal tasks is the other crucial factor for full use of shared variables within APL; without it, there is no way of guaranteeing there will be someone to share with, at least without permanently tying up a terminal.

It is probably due to these two limitations of the original APL environment that there exists the widespread notion that shared variables have something to do with files. R.H. Lathwell, their originator, and others have long stated publicly that, far from being an adjunct to a particular file system, shared variables are a communications discipline; in 1973, Lathwell wrote 'The function of the shared variable subsystem is analogous to the function of a central switching office of a telephone system' [2]. However, the confusion persists, as perusal of the U.S. APL standards committee (X3J10) discussions of this subject reveals [3].

We, however, as users of SHARP APL, have had one of these two facilities even before the introduction of shared variables to our system, and have now had the other (along with the rest of the shared variable apparatus) for over two years. Further to help wean us from the conventional prejudices about shared variables, we have long had a completely independent means of accessing files.

As might be expected, shared variables have been eagerly seized upon by SHARP APL users as a means of communication between simultaneously active workspaces. Two major varieties of applications have emerged:

- auxiliary tasks intended to serve a single user, and
- central processors serving a community of users.

1) Single-Partner Auxiliary Processor (AP)

The simpler of these two, of course, is the single-partner AP. Such a processor may use □*SC* and □*SVS*, but it doesn't require them.

A typical example of a function to control such an AP, with no reference to **what** the processor will actually be doing, is the function *NTASKSV* below. It is the sort of function one might use for a quick reduction to the cost of an application, taking advantage of the lower N-task rates [4]. (This particular example is a slight variation on a function originally written by S.J. Taylor and S. Garland):

```
          ∇ R←NTASKSV EXP; A; ANS
[1]        A←(1 2 ⍴□AI[□IO],99) □SVO 'ANS'
[2]       →(0∧.=A←□RUN ': : NCRASH 0 0')↑NTASK
[3]        ('□RUN ERROR ',⍕A) □SIGNAL(0≠1↑A)/888
[4]        A← 0 0 1 0 □SVC 'ANS'
[5]        R←ANS
[6]       →0
[7] NTASK: □TRAP←'∇ 0 E ANS←□ER'
[8]        A←□SVN 99
[9]        A←□AI[□IO] □SVO 'ANS'
[10]       ANS←⍎EXP
[11]       □TRAP←'∇ 0 S ∇ 2001 D CLEAR'
[12]       ⍎
          ∇
```

NTASKSV behaves (from the outside) approximately like ⍎, save that errors are returned as the explicit result. This might require a test of some sort when the function is called.

On line 1, the variable *ANS* is offered to the (future) N-task. It is important to make the share offer **before** starting the N-task; otherwise there would be a race between the two tasks. The N-task might conceivably finish executing before the T-task got around to matching the share, in which case the T-task would wait forever. Another possible solution is to go ahead and start the N-task first, but to include additional code requiring it to ensure that a full share has been established before continuing (see note 5).

Line 2 attempts to establish the associated N-task, and selects appropriate behavior for each copy of the active workspace. On line 3, the T-task makes sure the other task started; on line 4, it arranges to wait for the answer from the N-task; and on line 5 it does so. Meanwhile, the N-task has been executing lines 7 through 12. On line 7, it ensures that the T-task will be informed of any error. On lines 8 and 9, it establishes its identity and matches the share from the T-task; on line 10 it fulfils its purpose in life. Line 11 sets a trap to avoid a saved workspace, if everything else has worked up to now. Line 12 makes sure this trap takes effect: simply exiting the function might have destroyed the trap if some pendent function had localized it.

Possible problems

What is wrong with this function? Very little, actually. One slight problem (the possible race between the T-task and the N-task) has already been noted above. An error report is quite different from the expected result, making it easy to tell them apart (though the test may well be whether or not the next function or statement gives a *DOMAIN ERROR* somewhere...) In any event, this sort of function would only be inserted in a system that already works, since its purpose is just to reduce costs.

One other problem might plague any shared variable application involving two tasks on the same account number: what if the desired clone ID (99 in this case) were already in use? If a single programmer controls all use of a given account, this problem can be avoided by establishing a convention that guarantees unique clone IDs. Software in a public library at least should make no assumptions about other systems. As there is no facility to inquire which clone IDs are in use, I know of no way to guarantee no conflict; I can only suggest that public libraries accept a convention that guarantees no conflict with other public software.

Probably the simplest way to choose a unique identifier is to take advantage of the one APL has already assigned: that is, to use the AP's task ID. For the moment, it is safe for the T-task at least to use 0 as its clone ID, but when multiple T-tasks are allowed on a single account they, too, may have to subscribe to some such convention. Using the task ID has the (slight) additional advantage of avoiding conflict with clone IDs of less than 1000 (though of course they may not avoid conflict with each other).

Space Limitations

Another use for a single-partner AP is when more space is required for a computation, or the software to support it, than is available in the main task's active workspace (although the final result does fit in that workspace). SUPERPLOT is subject to the problem, since it is often used inside workspaces that contain a fair bit of MAGIC, and perhaps data as well. As a result, the SUPERPLOT implementation uses an associated task. The following function is executed whenever a user requests a plot:

```
      ∇ Z←⍺⍺PLOT ⍵; SHPCK; ⎕RBS; MSG; ⎕NS; ⎕IN; QUADER; ⍺⍺; SP
    : EN; ⎕; L; M; RUN; ⎕PRAP; DA; XPK; QAS; OLDK; CL
[1]       ⍺((∧/⍺/⍵=0)∨0<⍴⍵)/'''NO DATA TO PLOT''⎕←¯1'
[2]       ⍺(∧/1=⍴⍵)/'''ONLY 1 POINT''⎕←¯1' ⋄ ⍺(' '=1+0+.⍵)/'''NOT
    NUMERIC DATA''⎕←¯1'
[3]       ⍺(0=⎕NC 'XQN')/'''NO TERMINAL TYPE DEFINED. USE
    <SUPERPLOT> FIRST''⎕←¯1'
[4]       ⍺((⍵RBTOF≠0)∧~⍵RBTOF∈⎕WUMS)/'''NO FILE TIED TO ''.▼⍵RBTOF
    ⋄ ¯1'
[5]       ⍺(0≠⍵RBTOF)/'EN←.(⎕WNAMES)[⎕WUMS⍳⍵RBTOF;]⋄(1≠(⎕SIZE
    ⍵RBTOF)[2])/''WARNING. FILE IS NOT EMPTY'''
[6]     L1: ⍺(0=⎕NC '⍙')/'⍙←''''''
[7]       ⎕PRAP←'P 1001 1004 S P 73 C →NOS P 777 1000 C →INT '
[8]       →L4×⍳('Q'∈⍙) ⋄ 'SUPERPLOT V3.4 18/10/80'
[9]     L4: XPK←⎕PACK '' ⋄ →L2×⍳0=⎕NC 'SPK' ⋄ XPK←⎕PACK SPK
[10]    L2: DA←(1,1↓⍴DATA)⍴DATA ⋄ DA←⍉DASHES
[11]      SP←⎕PACK ⎕PNAMES ⎕READ 315 12 ⋄ ⍺A←⎕WA
[12]      CL←⎕SVN 0
[13]      ⍝ START THE NTASK
[14]      →ERR×⍳0=¯1↑RUN←⎕RUN '': 3269878 SPLOT:PROD CONTINUE 1000 0'
[15]      →CRASH×⍳¬(1↓RUN)∈⎕RUNS[;1]
[16]      ⍙←1 2 ⍴DAI[1],1↓RUN) ⎕SVO 'SHPCK'
[17]      ⍙← 0 1 1 1 ⎕SVC 'SHPCK'
[18]      SHPCK←⎕PACK 'MQ CL SP ⍺A XQN ⍙ A EN DOW BER LWD RC DA
    ⍵RBTOF XPK QAS TT BLAB CLAB QRLAB QCLAB QLAB ETN'
          OLDK←KEYB 'NOMSG'
[19]      →L5×⍳('B'∈⍙) ⋄ STWITCH
[20]    L5: ⎕PDEF SHPCK ⋄ SHPCK←''          ⍝ UNPACKS ⍵RBS MSG QUADER
[21]      →(MSG←0,⍳5)/M1,M1,M2,M3,M4,M5
[22]    M5: ⍺QUADER ⋄ →L5
[23]    M4: QUADER ⋄ →END
[24]    M3: QUADER ⋄ →OUT
[25]    M2: 'ERROR IN SUPERPLOT' ⋄ QUADER ⋄ →OUT
[26]    M1: ⍵OUTPUT ⍵RBS ⋄ →END×⍳¬MSG ⋄ →L5
[27]   ERR: 'ERROR IN NTASK SUBMISSION'
[28]      ¬((1+RUN)=0,⍳16)/(M1,M1,M2,M3,M4,M1,M1,M1,M1,M1,M1,M1,M1
    ,M1,M14,M15,M1)
[29]    M1: SPNTASKERR←RUN ⋄ 'CONTACT YOUR SHARP APL REPRESENTATIVE'
    ⎕SIGNAL 778
[30]    M2: 'SYSTEM CAPACITY FOR NTASKS EXHAUSTED. TRY AGAIN LATER'
    ⎕SIGNAL 778
[31]    M3: 'YOU HAVE 4 NTASKS RUNNING ALREADY. THAT IS THE SYSTEM
    LIMIT' ⎕SIGNAL 778
[32]    M4: 'TYPE '')OPR PLEASE INCREASE MY WS QUOTA''. TRY AGAIN WHEN
    YOU RECEIVE A REPLY' ⎕SIGNAL 778
[33]   M14: 'SYSTEM CAPACITY FOR TASKS EXHAUSTED. TRY AGAIN LATER'
    ⎕SIGNAL 778
[34]   M15: 'YOU CANNOT USE NTASKS. CONTACT YOUR SHARP
    REPRESENTATIVE.' ⎕SIGNAL 778
[35]    NOS: 'SYSTEM CAPACITY FOR SHARED VARIABLES REACHED. TRY AGAIN
    LATER' ⎕SIGNAL 778
[36] CRASH: 'ERROR IN NTASK '.▼RUN ⋄ →N1
[37]    INT: ⍙←⎕BOUNCE 1↓RUN ⋄ ⍙←⎕SVR 'SHPCK' ⋄ ⍵CONTROL 'R' ⋄ ⍺(0≠⎕NC
    'OLDK')/'0 0 ⍴KEYB OLDK' ⋄ ⎕SIGNAL 778
[38]    END: ⍙←⎕SVR 'SHPCK' ⋄ ⍵CONTROL 'R'
[39]    OUT:  0 0 ⍴KEYB OLDK
          ∇
```

Unlike our last example, this one uses an AP for a particular, predetermined purpose; therefore a completely detailed reading of the function requires knowledge of the application. Nevertheless, some features are of interest even without such knowledge. Notice (lines 16, 17, 18, and 21) that the interface with the N-task consists of a single variable, *SHPCK*, a package containing both control information and data. This may seem an obvious thing to do to those of us who grew up with SHARP APL, but it may be a revelation to people used to communicating with associated or auxiliary processors through two (or more) different variables: one to provide data, and one to tell it what to do. In an early paper on shared variables, R.H.

Lathwell noted that there were two ways of passing to a sharing partner a multiplicity of information: through a multiplicity of shares, or through a predetermined sequence of them [6]. We have just seen a third means, and now also have a fourth (enclosed arrays, available on the June release). A choice among the four alternatives is still based on taste and context, but it strikes me as a very elegant alternative to be able to manage all communications with an AP through a single shared variable link.

Another interesting feature is embodied in lines 21-27: the problem of distinguishing error reports from results (see *NTASKSV* above) is solved here by always including information (*MSG*) which specifies how the partner's response is to be interpreted. In some contexts, it might be sufficient to provide just two values (or the possibility of two values) in the response, one for errors and one for results. Here, *QUADER* may be used either for error or status reports, or for executing something in the T-task workspace; hence *MSG* is also required. *MSG* and either *QUADER* or *ARBS* would suffice, but the author of the function found it clearer to use distinct names for normal results and exceptional conditions.

Notice that communications take place in more than one step; lines 23 and 27 go back to wait for more from the N-task. This permits both tasks to work simultaneously after the first exchange (one deciding how to push a pen, the other pushing it), thus improving response time.

Before leaving my major example of the use of an AP to overcome space limitations, I should point out that this won't always work, even if there is space for the result; the shared variable processor may not have room to send the result back. Currently, shared variable sets will not succeed (the error is *INTERFACE CAPACITY EXCEEDED*) for objects over 48K bytes. This size is installation dependent, and can change weekly even on a single installation; the minimum space required on distributed systems allows a maximum shared variable size of about 3500 bytes.

Modularity

One further effect of incorporating an associated APL task into the design of a system is an increase in the modularity of the system. In the example above, user interaction and application are almost completely segregated from the inner workings of SUPERPL●T; having defined the communications link and functional capabilities of the associated task, it could be entirely replaced by different software.

In an experimental text markup and photocomposition system called TEX, I use two APs (so far...): one for hyphenating words in the text, and one for generating numerical information controlling spacing. This (and the use of direct definition throughout, which also contributes enormously to modularity) enables me to easily replace either of these two major components of the system, and frees me from worrying about name conflicts between the parts. Both APs are controlled by essentially the same functions, shown below for the case of the hyphenation task:

```
HSTART◇ ωFSVAR 'H' ◇◇ HCTRL 'H'
FSVAR◇ ∨/(2=⍺⌷⎕SVO⍵)⌷⎕SVC ⍵ ◇ ⍺FSVAR⍵,0⍴⌷⎕SC ◇ 1
HCTRL◇ 2=⌷⎕SVO ⍵,0⍴T←⍬⍵ ◇ OFF ◇ HCTRL ⍵,0⍴⍬⍵,'←HYPHENATE T'
OFF◇ ⍬⌷⎕TRAP←'∇ 2001 D CLEAR'
```

(the version of direct definition used is that used in 12 *COURSE*: ◇ is the separator, and the selector statement comes first).

These functions operate on a slightly different principle than both of the examples we have already seen: they assume the APs are started as soon as the main workspace is loaded, they wait until they are needed, and they go away when the link with the main workspace is destroyed, rather than at the completion of one task.

One particular kind of application that takes advantage of cooperating, but different, workspaces as an approach to modularity is a solution to a question that has long plagued designers of utility functions.

For ease of use, a utility should be a single function; but utilities are supposed to encourage modularity, so they should themselves be modular. P.C. Berry gave an example of the use of the kind of associated task we have been discussing to solve this problem [8].

Testing AP's

Useful as APs can be, it's often a tedious task to test them and ensure they're working properly. I'm sure most of you are familiar with the problems of testing an N-task: trying to deduce the cause of a crashed workspace makes one suspect the term 'batch APL' may be more appropriate than one had thought. The classic solution, of course, is to run the intended N-task as a T-task for debugging purposes. This does, however, tie up two terminals; in a busy office that can be awk-

ward. Fortunately, the recently released S-task facility provides an alternative.

I like to use the set of functions in [note 7] to use S-tasks. These functions allow simultaneous execution in two workspaces by always returning to T-task immediate execution. The major functions are *SSTART*, *SDO*, *STELL*, and *SGET*. The first three are dyadic, taking as the left argument the name of the shared variable controlling the S-task, and as a right argument instructions for the S-task. *SGET* is monadic; its argument is the control variable. Briefly: *SSTART* is used only to start an S-task, and its argument should be a signon command (e.g. ')1234567:*LOCK*'); otherwise it is identical to *SDO*. *SDO* passes to the S-task a request to do something, and waits for a response (like walking to your second terminal, keying in something, and staying to see what happens). *STELL* is half of *SDO*; it tells the S-task to do something, but does not wait for the reply. *SGET* is the other half. *SATTN*, *SQAI*, *SDISC*, and *SREC* (all with the same syntax as *SGET*) provide access to attention, 3↑⎕*AI*, disconnect, and reconnect, respectively.

NOTES

[1] See the description of the Shared Variable Processor in the IBM APL standard: Falkoff, A.D., and D.L. Orth, 'Development of an APL Standard' (**APL79 Conference Proceedings**, part 2, also **APL Quote Quad**, vol. 9, no. 4-part 2; June 1979), p. 440-445. I.P. Sharp shared variables conform to that description. The following describes, in the terms of that standard, I.P. Sharp's extensions to the facilities described there:

'The extension to the left argument of ⎕*SVO* and the result of ⎕*SVQ*, and ⎕*SVN*, provide the second integer for the processor ID which the IBM standard requires to be zero (such extension was in mind at the time).

'⎕*SVS* is implemented by augmenting the ACV operation so that it returns the access state in addition to the control vector.

'⎕*SC* is the wait/signal mechanism described there made explicitly available to APL programs — an addition to the interpreter requiring no change to the shared variable processor definition.' (R.H. Lathwell, personal communication, May 1981)

[2] Lathwell, R.H., 'System Formulation and Shared Variables' (**IBM Journal of research and development**, vol. 17, no. 4, July 1973, p.356)

[3] The following is taken from a message by a member of the mailbox group *STD*:

'As far as file systems are concerned, standardizing the shared variable interface without standardizing the variables to be used and their meanings, capabilities, and behavior won't help anybody to write portable code.'

Through much of the long discussion of which this was part, lip service was paid to the idea that shared variables are a communications discipline, but the discussion kept coming back to files.

[4] The T-task overhead associated with executing an expression through this function seems to be quite small. My measurements put it at about 0.1 CPU units.

[5] The following version of *NTASKSV* uses enclosed arrays to implement a version of the solution seen in *NΔPLOT* to the problem of distinguishing errors from results. It also subscribes to the convention of using a task ID as a clone ID, thereby at least avoiding conflict with other processors subscribing to this convention.

```
          ∇ R←NTASKSV EXP; A; ANS
 [1]       →(0∧.=A←⎕RUN ': : NCRASH 0 0')↑NTASK
 [2]        ('⎕RUN ERROR ',⍕A) ⎕SIGNAL(0≠1↑A)/888
 [3]        A←(1 2 ρ⎕AI[⎕IO],1↓A) ⎕SVO 'ANS'
 [4]        A← 0 0 1 0 ⎕SVC 'ANS'
 [5]        R←>''ρ⍎A←ANS
 [6]       →('E'⎕>''ρA)↓0
 [7]        ('NTASK ERROR ',⍕↑,R) ⎕SIGNAL 887
 [8] NTASK: ⎕TRAP←'∇ 0 E ANS←(<''E''),<⎕ER'
 [9]        A←⎕SVN ''ρ⎕RUNS
[10] DWELL:→(2=⎕AI[⎕IO] ⎕SVO 'ANS')↑OK
[11]       →⎕SC↑DWELL
[12]    OK: ANS←(<''),<⍎EXP
[13]        ⎕TRAP←'∇ 0 S ∇ 2001 D CLEAR'
[14]
          ∇
```

[6] Lathwell, Op. cit., p. 355-356

[7] The following functions may be used to control an S-task. They always return control to the T-task executing them.

```
SSTART◇ 2=1 ⎕SVO α ◇ αSSTART ω,0ρ⎕SC ◇ αSDO ω,0ρ 1 ⎕SVC α

SDO◇ αSTELL ω ◇◇ '' SCOLLECT α

STELL◇ ρ⍴ρ⍕ α,'←⎕AV[4ρ⎕IO],ω'

SGET◇ '' SCOLLECT ω

SCOLLECT◇ STEST 4↑T←''SCHOP ⍕ω ◇ α,4↓T ◇ (α,4↓T) SCOLLECT ω

STEST◇ 2|(⎕AV⍳ω[⎕IO+3])-⎕IO

SCHOP◇ 6>ρω
     ◇ ((4↑L),(4↓α),4↓L) SCHOP (2+ρL←2↓(256⊥(⎕AV⍳2↑ω)-⎕IO)↑ω)↓ω
     ◇ α
```

```
SATTN◊ ppρωSCMD 0 0 2 0 ◊◊ '' SCOLLECT ω

SQAI◊ ppρωSCMD 0 0 1 0 ◊◊ ⍕ω

SDISC◊ ppρωSCMD 0 0 3 0 ◊◊ '' SCOLLECT ω

SREC◊ 2=1 ⎕SVO ω
    ◊ SREC ω,0ρ⎕SC
    ◊ '' SCOLLECT ω,0ρωSCMD 0 0 4 0,0ρ1 ⎕SVC ω

SCMD◊ ⍕ α,'←⎕AV[⎕IO+ω]'
```

[8] Berry, P.C., Talk entitled '*S-task as an independent environment for a major subfunction*', I.P. Sharp Technical Meeting 81.

CONTEST 11 RESULTS

Phil Chastney, Gloucester

The winners are named in the last paragraph, so check the end if you want to set your mind at rest.

Well, now that you know whether you won (unlikely, considering the ratio of readers to winners), how did your entry compare?

Competition 11 attracted quite a lot of mail, quite a number of entries, required a megabyte of file space, and has been the most tremendous fun to adjudicate.

The object (somewhat sloppily described) was to construct APL expressions, which used primitive functions and operators, and no others, which used precisely four 4's and no other constants, and which evaluated to give successive natural numbers. The criterion for winning was then to see how far through the natural numbers each entry went, before having to make a break in the sequence.

There was some ambiguity in the description of the problem. For one thing "four 4's" was an ellipsis for "four uses of the scalar value 4". One or two entries used 0.4 and 44, but in no cases were these illegal constants indispensible to the entry, and in no case did they affect the final result.

A separate expression was required for each natural number, so that `⍳*/4 4 4 4` and `+\(4*4)⍴4*4` were disqualified; `+\444⍴⍬⍬''` set something of a record for being disqualified on three separate counts.

The rest of the ambiguity was deliberate. As for previous competitions, a SHARP APL environment could be assumed, but the status of that environment, and in particular *⎕IO* was not defined, which had a strange effect on people more used to working with exact specifications. Many worried messages came from people exhibiting this 'programmer mentality', and it was difficult being noncommital in reply, without explicitly remarking that "If you don't know you don't need to know, you've missed the point".

Some entries consisted of the generated numbers themselves.

Some entries consisted of expressions, appearing as rows of character matrices, or as function lines. A gem of an entry from Maurice Elliot and Jaime Menendez read

```
    ∇ FOUR4S
[1]   (⌶⌈○4+4)-4÷4
[2]  ⍝ REPEAT THE ABOVE LINE AS OFTEN
    ∇                    AS YOU LIKE!
```

Julian Coward submitted an entry, aptly titled *CHEAT44A*, which required repeated execution of `?-(⍴,4)-(⌈⊛4)*((+/×\⍳4)-(⌈⊛4))` adding the comment that "the numbers will not be in order".

But the favourite form of entry (used in half the 40 entries) was a generating function. Only the executable result of such a function need conform to the rules: the generating function itself could use the full power of APL.

The central problem with a generating function is how to generate the next number from the last. Each generation formula, then, is comprised of essentially three parts: some starting formula, a transitional part to move to N from (N-1), and a tidying-up part to ensure that the result is integral and scalar. So, let us have a look at the principal types of generating function.

	TIDY UP	*{REPEATED PORTION}*	*LEAD IN*
[1]	`+/`	`+\`	`4 4=4 4`
	`(4÷4)↓`	`+\`	`×4 4`
[2]	`⌊⊛`	`○`	`4=4=4=4`
	`(○4÷4)⊛`	`○`	`4÷4`
[3]		`⌊⊛○*`	`4=4+4+4`
[4]		`+/⌈○○÷⍳`	*VARIOUS*
[5]	`(-+/×4 4)+(-×4)`	`-\`	`⌽⍳4`
[6]		`(×⎕UL)+`	`4=4=4=4`

⎕*UL* is not a primitive function, so that last entry was disqualified. All the same, it is rather neat.

The first pattern closely resembles the "successor function" from the foundations of mathematics, and was easily the most popular - minor variations being submitted by Julian Coward, Linda Furrow, Douglas Holmes, Morten Kromberg, David McKnight, Gary Powell, Bruce Roe and Danal Estes.

The second pattern came as something of a surprise to me - the whole thing looks so like unary notation, it could serve as an alternative model for arithmetic. Variations were submitted by Chris Baron, Roger Hui (who deserves special mention for the clarity and completeness of his comments), D. Mitchell, Roland Pesch, Roger Taylor and A.L. Tritter.

Patterns three, four and five bear the individual stamps of Arthur Whitney, Maurice Elliott and Ross Wilkinson, respectively.

There has got to be something about working in Melbourne that makes -\-\...-\⍳4 look like a natural act. Not only does their southern hemisphere bath-water drain away in the wrong direction, but Ross also came up with a totally original method of hitting the result from above, using {+/≤\⍳}*M*, where M is some simple expression evaluating to a number larger than N.

Now the fun begins.

Having devised a generating function, how far will it go? The imaginative leap to the use of generating functions seems to have come easier to people who trained originally in mathematics. The dark side of the "mathematician's mentality" is that, having discovered a general method of solution, they lose interest in the details of implementing that solution within given system limits.

Particular problems encountered were

- *WSFULL* during generation,
- *DOMAIN ERROR* during generation,
- *DOMAIN ERROR* when attempting execution,
- *DOMAIN ERROR* during execution,
- *WSFULL* during execution.

Of these, the commonest hang-up was the limiting of the right argument of ⍎ to a little over 32,000 bytes.

You will have noticed that many of the entries could have as easily been restricted to a single constant. This is significant, because it means the big numbers can be evaluated from a combination of expressions (and my thanks to Mike Powell and A.L. Tritter for the interesting observation that, apart from physical limits, it is possible to combine expressions to generate all the rationals). Combining expressions was an approach suggested by Chris Baron and Morten Kromberg, and implemented by John Harris, Roger Hui and Hazel O'Hare (demonstrating thereby a strong correlation between inspired programming, and the letter 'H'). By developing his first entry, Maurice Elliott was able to speed up the execution, without significantly extending his previous limit of 60,000 or thereabouts. (Describing that first entry as "slow" would be something of an understatement, since it required 912 CPU units to evaluate the expression for 10,000.) Sam Sexton likewise managed to pass 60,000, while Uri Shamir beat the lot of them by special-casing the first 10,000 numbers, to attain a limit of 75,981.

And that looked like the final result ..... except for a couple of entries from Arthur Whitney, submitted in the last 48 hours of the contest. Either he was just too busy to submit them earlier, or it took him until the last minute to get the things working properly, but those last two entries were just brilliant. The first generated executable expressions for all integers up to 2*31, and (with ⎕*CT*←0) the second carried on quite happily to ⍎⍕⍳0, thus wrapping the subject up for all time.

So ***Arthur Whitney*** gets a book to take with him to Hong Kong, while ***Uri Shamir*** collects the $50 prize for the best non-I.P. Sharp entry. The winning entries may be seen in 999 *CONTEST*.

WHETHER CONDITIONS

Stephen Taylor, London

Reading other people's programs, I've found three favourite ways of writing a conditional branch:

```
→CONDITION↑LABEL
→CONDITION⍴LABEL
→CONDITION/LABEL
```

I don't know any compelling reason for preferring one to the the other two. My own practice is to hide my choice in an *IF* function, and many other people do the same.

WHETHER CONDITIONS

The choice makes a difference though if you want to use your *IF* function on non-scalar arguments. We'll take our three versions

```
IFT: ω↑α
IFR: ωρα
IFC: ω/α
```

and try them instead of *IF* in the expression `→⎕LC IF TIENOS∈⎕NUMS`. Here we see that *IFC* is equivalent to `→⎕LC IF∨/TIENOS∈⎕NUMS`; that *IFR* is the same as `→⎕LC IF∧/TIENOS∈⎕NUMS`; and that *IFT* won't work at all.

What lesson you draw from that depends on your own stylistic preferences. You may prefer to use ↑ to prevent any ambiguity from accidentally using a vector argument. You may like to use ρ so that the condition is always interpreted as restrictively as possible. Or / for just the opposite reason. Whichever you choose, you should be conscious of its effects, especially if you're using an *IF* function.

I use /, for quite another reason: I can use my *IF* with ⍎ as well as →. Thus

```
⍎EXPRESSION IF CONDITION
```

So far in the argument, using an *IF* function seems to have as many drawbacks as advantages: as a utility it's trivial, and the underlying idiom is easily mastered. Hardly worth the potential trouble with *SYNTAX ERROR*s (did not copy the function) and non-scalar arguments. But it turns out that judicious use of *IF*: `ω/α` will allow some useful further constructs. Take for example the common 'print and branch' job:

```
[5]   ASK:NUMBER←ASKI 'NUMBER:'
[6]   →(NUMBER<100)ρOK
[7]   'NUMBER OUT OF RANGE'
[8]   →ASK
[9]   OK:
```

We can construct a 'print and branch' function *BECAUSE*: `0/α : ××/ρω : α,0ρ⎕←ω` which will allow us to write the above code as

```
[5]   ASK: NUMBER←ASKI 'NUMBER:'
[6]   →ASK BECAUSE 'NUMBER OUT
          OF RANGE' IF NUMBER<100
```

There are several APL models for the IF-THEN-ELSE constructs found in other programming languages [1]. I haven't yet seen one that fitted in with APL's right-to-left evaluation as harmoniously as an IF-BUT construction:

```
      →REPLACE BUT APPEND IF I>ρDIR
⍎'X ⎕REPLACE T,C' BUT 'C←X ⎕APPENDR
      T' IF C≥1↑1↓⎕SIZE T
```

BUT turns out to be very simple too,
`BUT: α : ××/ρω : ω.`

Lastly, we can construct an 'execute and branch' function corresponding to *BECAUSE*. We might use it for special cases.

```
[1]   →0 AFTER 'Z←⍳0' IF 0∈ρX
```

or even

```
[1]   →0 AFTER 'Z←⍳0' BECAUSE
          'NOTHING TO DO' IF 0∈ρX
```

AFTER is equally simple,

```
      AFTER: 0/α : ××/ρω : α,0ρ⍎ω.
```

I started using these constructs in my code with the intention of making it more readable for posterity. I quickly found I was helping myself. The adage is true: short readable code offers less scope for error.

[1] R.C. Metzger, Branching in APL, *I.P. Sharp Newsletter Technical Supplements 22 - 26.*

SYSTEM VARIABLES, Part II

Robert Metzger, Rochester

We learned in the first part of this series that whenever you use APL, you use system variables, even if you're not aware of it. The Output Control system variables (⎕*PP*, ⎕*PW*, ⎕*HT*) make it possible for APL to provide convenient default output. In this article we will explore the Computation Control system variables.

There are three Computation Control system variables: ⎕*RL*, ⎕*CT*, ⎕*IO*. The Output Control system variables control the form in which you will see your results. In contrast, these variables control the actual content of your results.

⎕*RL* stands for Random Link. It is used only by the monadic and dyadic functions denoted by the symbol *?*. These are called Roll and Deal. These functions generate 'pseudo-random' numbers, and ⎕*RL* is the 'seed' from which the APL system can create such numbers. We say 'pseudo-random' because while the numbers pass statistical tests of randomness, they are produced by an algorithm.

This algorithm is described in the SHARP APL Reference Manual on pages 126, 178, and 261. The algorithm both uses and resets ⎕*RL*. The default value of ⎕*RL* is 16807.

If ⎕*RL* has the same value, and you invoke Roll or Deal with the same argument on two occasions, you will always get the same result. You can see this by doing the following:

```
⎕RL←16807 ◇ ?10 ◇ ⎕RL ◇ ?10 ◇
        ⎕RL←16807 ◇ ?10
```

This indicates the circumstances under which you will want to set ⎕*RL*.

1. If you want to ensure that you get the same results you did at another time, you can set ⎕*RL* to the same value. This could happen if you wanted to reproduce the results of an experiment.

2. If you want to ensure that you get different results each time you use a program, you should set ⎕*RL* to a different value. This could happen if you were using random numbers in a simulation, or for test data.

Two common ways to set ⎕*RL* to a different (random) value each time you set it are given below:

```
⎕RL←×/1+3↓⎕TS
⎕RL←⎕TS+.*2
```

⎕*TS* is used because the values of the minute, second, and millisecond elements change rather quickly.

The function below creates an array whose dimensions and values are both chosen randomly. The argument specifies the upper bounds for the length of each dimension (and thus also the rank). ⎕*RL* is localized so that the choosing of these random numbers does not affect its global environment. To create a matrix with 50 rows maximum, and 100 columns maximum, you say *BUILD* 50 100. It is useful for creating test data.

```
    ∇ ARRAY←BUILD BOUNDS;⎕RL;SHAPE
[1]   ⎕RL←⎕TS+.*2
[2]   SHAPE←?BOUNDS
[3]   ARRAY←SHAPE⍴(×/SHAPE)?×/SHAPE ∇
```

⎕*CT* stands for Comparison Tolerance. It is used by the relational functions < ≤ = ≥ > ≠ , and those functions which are based upon them: ⌈*X*, ⌊*X*, *X*⍳*Y*, *X*∈*Y*, *X*|*Y* . Its purpose is to make the limitations of machine arithmetic less obvious to you, the user. It makes APL work more like mathematics, and less like an electronic device.

One problem handled by ⎕*CT* is that a computer can store only a finite number of digits to represent a number. But some numbers, like the repeating fraction 0.3333... need an infinite number of digits to represent them. Related to this problem is the fact that computers do arithmetic in base 2 (or base 16), not base 10. A fraction which terminates in base 10 may repeat endlessly in base 2.

⎕*CT* reduces the effects of these problems. It defines the difference between two numbers which you consider so close as to make them equal. The default is about 1.4*E*¯14. This means that if *X*-*Y* is less than this number, then *X*=*Y* will give a 1. It has the effect of ignoring the 2 least significant decimal digits. You can observe the effect of ⎕*CT* in the following expressions.

```
      ⎕PP←16
      ⎕←T←⍉' .',(12⍴'3'),' .',
         (14⍴'3'),' .',16⍴'3'
0.333333333333 0.33333333333333 0.3333333333333333
      3×T
0.999999999999 0.99999999999999 0.9999999999999999
      1=3×T
0 1 1
```

Because the default value of ⎕*CT* works so nicely, it is a rare occasion when you would want to change it. But it does happen. One reason is to speed up the Index Search (⍳) and Membership (∈) functions. When dealing with floating point (8 byte) numbers, these functions must use ⎕*CT* to detect 'almost equal' elements. If ⎕*CT*←0, which means no tolerance, a simpler algorithm can be used to compare the elements.

When would you be willing to give up tolerant comparison? When the floating point numbers you are using are really just integers which can't be represented in the fixed decimal (4 byte) format. These would be integers greater than -1+2*31 and less than 2*52. These integers are represented without loss of precision in floating point format. A use of such integers is encoding textual information. Using an alphabet of 27 symbols (letters and blank), 6 letters can be encoded into a 4 byte integer, since (2*31)>27*6. Using the same alphabet, 11 letters can be encoded into an 8 byte integer, since (2*52)>27*11.

The functions listed below will encode and decode words up to 11 characters long.

```
    ∇ NUMBERS←ENCODE WORDS
[1]   NUMBERS←27⊥⍉¯1+' ABCDEFGHI
        JKLMNOPQRSTUVWXYZ'⍳(
        (¯1↓⍴WORDS),¯11)↑WORDS ∇
```

```
    ∇ WORDS←DECODE NUMBERS
[1]   WORDS←' ABCDEFGHI
        JKLMNOPQRSTUVWXYZ'[⍉1+
        (11⍴27)⊤NUMBERS] ∇
```

Two situations in which you might want to encode words in this way are:
1. Analysis of natural language texts;
2. File directories in which the identifiers are character data.
The function listed below sets *⎕CT←0* in order to minimize the cost of searching lists of encoded words.

```
    ∇ INDEX←LIST INDEXOF NUMBERS;⎕CT
[1]   ⎕CT←0
[2]   INDEX←LIST⍳NUMBERS ∇
```

You can see how these functions work from the example below.

```
      NAMES←4 11⍴' AUGUSTINE
          IRENAEUS   ORIGEN TERTULLIAN'
      ENCODE NAMES
505120722197 101195956252 224981942
       154120440096716
      DECODE ENCODE NAMES
 AUGUSTINE
  IRENAEUS
    ORIGEN
TERTULLIAN
      (ENCODE NAMES) INDEXOF
        ENCODE 'IRENAEUS'
2
```

⎕IO stands for Index Origin. It is probably the most frequently used system variable. It is referred to by those functions which create or use indices, and by the axis operator. These functions are listed below.

```
⍳Y     ?Y     ⍋Y     ⍒Y
X⍳Y   X?Y   X⍋Y   X⍒Y   X[Y]   X⍉Y
X,[A]Y   X/[A]Y   X\[A]Y   ⌽[A]Y   X⌽[A]Y
                          ⌿[A]Y   ⍀[A]Y
```

The default value of *⎕IO* is 1. The only possible values it can take are 0 or 1.

When should you use origin 1? When is origin 0 more appropriate? Three situations seem to be easier to handle in origin zero.
1. Computation of indices,
2. Mathematical calculations (such as series),
3. Computer hardware oriented programs (systems programs).

An example of the first case is a statement used in the first part of this series.

```
TABS[1+,⍉0 1∘.+POSITIONS+2×¯1+
      ⍳⍴POSITIONS]←(2×⍴POSITIONS)⍴ESCAPE,TABSET
```

If it were written for origin 0, it would be simplified to the following.

```
TABS[,⍉0 1∘.+POSITIONS+2×
      ⍳⍴POSITIONS]←(2×⍴POSITIONS)⍴ESCAPE,TABSET
```

An example for the third case is the creation of codes to be transmitted with *⎕ARBOUT*.

```
[∘]  ⎕ARBOUT ¯1+ARBBIT⍳CHARACTERS
```

The ASCII code system starts with 0, as do many numbering sequences which relate to hardware. In origin 0, it is more simply stated.

```
[∘]  ⎕ARBOUT ARBBIT⍳CHARACTERS
```

Most indexing situations which don't fall into one of these categories seem more naturally expressed in origin 1. (At least to people who think in origin 1). There is one situation which definitely easier to express with *⎕IO←1*. This is when an index must be converted to a count. An example of this would be

```
(TEXT⍳' ')↑TEXT
```

which would be the following in origin 0.

```
(1+TEXT⍳' ')↑TEXT
```

Another case would be computing the length of partitions marked by a boolean vector.

```
    ∇ LENGTH←PARTNLENGTH PARTN
[1]   LENGTH←(1⌽PARTN)/⍳⍴PARTN
[2]   LENGTH←LENGTH-0,¯1↓LENGTH ∇
```

In origin 0, the first expression would have to be the following:

```
LENGTH←(1⌽PARTN)/1+⍳⍴PARTN
```

While you can tamper with where you start counting, you can't change the fact that having 0 items means you're empty-handed. This is a difference between ordinal and cardinal numbers.

How do we convert between expressions which yield results in origin 1, origin 0, and either origin? As you have seen from the examples above, to convert from origin 1 to 0 you subtract 1. To convert from origin 0 to 1 you add 1. Making

origin dependent statements work in either origin is a little more complicated.

To convert an origin 0 statement to work in either origin, you must add ⎕IO, i.e., *INDEX*+⎕*IO*. The statement below does this in computing a 4-way branch.

```
[∘]  →(LA,LB,LC,LD)[⎕IO+2⊥(2=ρρX),
       (0=1↑0ρX)]
```

To convert an origin 1 statement to work in either origin, you must subtract the negation of ⎕*IO*, i.e., *INDEX*-~⎕*IO*. The statement below selects the last column of a matrix.

```
MATRIX[;(¯1↑ρMATRIX)-~⎕IO]
```

To convert a statement which yields a result in either origin to give an origin 0 result, you must subtract ⎕*IO*, i.e., *INDEX*-⎕*IO*. The function below uses a matrix of indices as the left argument to select individual elements from an array right argument.

```
    ∇ ELEMENTS←INDICES FROM ARRAY
[1]   ELEMENTS←(,ARRAY)[⎕IO+(ρARRAY)⊥
        ⍉INDICES-⎕IO] ∇
```

An example of the use of this function would be the following:

```
(3 2ρ1 4 2 3 4 2) FROM 4 4ρ⍳16
```

To convert a statement which yields a result in either origin to give an origin 1 result, you must add the negation of ⎕*IO*, i.e., *INDEX*+~⎕*IO*. The statement below gives a sequence of integers starting at 1, no matter what value ⎕*IO* has.

```
INTEGERS←(~⎕IO)↓⍳(~⎕IO)+LIMIT
```

This concludes this series on system variables. These variables provide a great deal of flexibility for the APL programmer in coding algorithms. If you take the time to learn about them, your efforts will be rewarded with easier programming and more effective programs. That's what APL is all about.

GRAPHICS FOR THE ACTUARY

Jim Spurgeon, Toronto

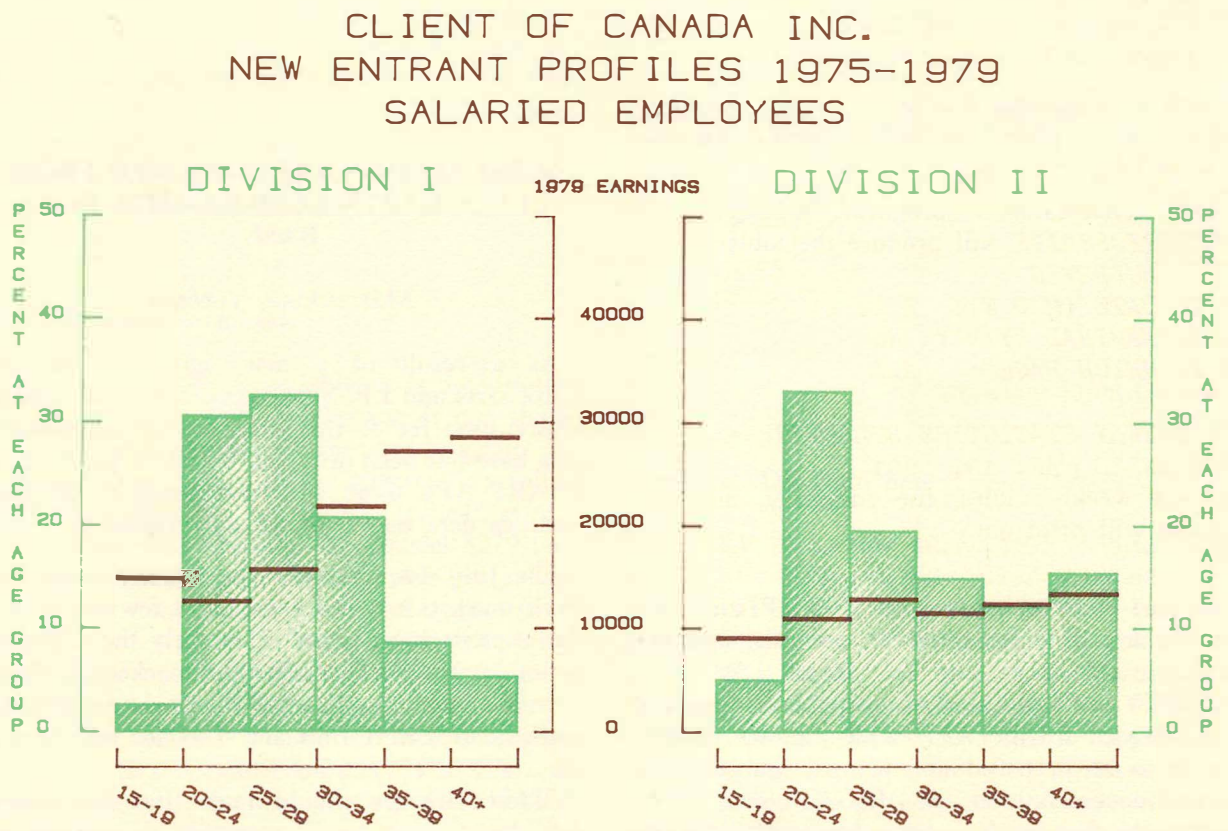
For some time now, actuaries have been using colour plots. Consulting actuaries in particular have found them useful to illustrate reports for their clients. They so effectively present complex and subtle actuarial concepts that some actuaries make them a regular part of all reports. They have been used to present:

- Comparisons of current to proposed employee benefits for various classes of employees;
- Comparisons of benefits across industry segments;
- Projections of future actuarial liabilities, assets, and plan contributions for a pension fund;
- Distribution of employees by age and salary.

For consulting actuaries employed by firms which maintain offices in several cities, there is an extra benefit to producing graphics on the SHARP APL network. Provided the proper equipment exists locally (i.e., HP7221 plotters), plots produced by actuaries in one office can be examined and used by any of their colleagues with access to the network, enhancing the speed and effectiveness of communications between offices.

The graph below is an example of the type of plot being created by actuaries. This one was used to compare the distribution by age of new entrants to two employee groups, and to illustrate the distribution of salary levels among the new employees. While the facts could be shown in tabular form, the plot provides a much more effective means of communicating the information.

More examples are available in the workspace 800 *ACTPLOT*. To see these plots and obtain information on colour graphics contact your I.P. Sharp representative.



API WEEKLY STATISTICAL BULLETIN

Brian Hart, Houston

The American Petroleum Institute (API), the major U.S. oil industry trade organization located in Washington DC, now prepares its weekly statistical bulletin using SHARP APL. The bulletin reports industry activities each week ending 7am Friday, and is released in the middle of the following week.

Activities (including refinery input, capacity figures, product stocks, import figures for crude oil, and petroleum products) are reported for the total U.S., and for most tables are broken down into PAD and refining districts as well.

A major benefit to our users is that all of the weekly tables of the bulletin are available online the instant they are released by the API. The tables are:

- U.S. Refinery Operations
- U.S. Refinery Output
- Stocks of Selected Products
- Estimated Domestic Production and Imports of Crude Oil
- Location of Crude Oil Stocks
- Imports of Petroleum Products

as well as a summary table (in several formats) and a footnotes page reporting revisions to the prior week's figures, if any.

The conversational program *BULLETIN* in 121 *PETROSERIES* will produce the tables:

```
      BULLETIN
ENTER DATE (M D Y):  3 13 81
WIDE TERMINAL (Y/N)?  NO
PRINT WHICH PAGES:   ALL

API WEEKLY STATISTICS BULLETIN: *1
WEEK ENDED MAR. 13, 1981
```

(All six weekly tables, the summary, and the footnote will print out.)

As part of PETROSERIES, over 500 API time series are available for retrieval, analysis, and reporting, with the help of MAGIC. As with *BULLETIN* itself, the data is available the instant it is released by the API, which allows custom reports to be prepared and printed immediately. Further documentation, the *MAGIC User's Manual* and the *Energy Newsletter* are available from your local I.P. Sharp office (check the box on the inside back cover).

EXPANDED SERVICE TO THE AVIATION INDUSTRY

The aviation industry is faced with significant changes in the eighties. Since May 1, a new group in Applications Software, **Aviation Products** under the management of **Brian Oliver**, has had the responsibility of developing or coordinating the development of large scale software packages.

Initial projects for the department include a total operations information system for small and medium sized carriers, and a graphics system to generate airline route system maps.

In addition to product development, the department will assume responsibility for the organization of I.P. Sharp Aviation Seminars, and for the quarterly Aviation Newsletter.

For further information regarding aviation software products, or if you would like to be added to the Aviation Newsletter mailing list, please contact the Aviation Products Department at our head office in Toronto.

SUBSCRIPTION FEE REMOVED FROM EUROCHARTS COMMODITIES DATA BASE

Marc Odho, Toronto

As a result of a new agreement between Eurocharts and I.P. Sharp Associates, the monthly subscription fee to the EUROCHARTS COMMODITIES data base has been dropped. Effective June 1st, all SHARP APL users will have access to the commodities data base without any surcharge.

On July 1st, data for the Chicago and New York markets will be available a few hours after the markets have closed. Currently the U.S. data is put up 16-20 hours after the market has closed.

In conjunction with more timely data, additional markets such as T-bills and T-bonds will be added.

These changes should make the commodities data base much more attractive to existing and potential users. A revised edition of the commodities manual is now available.

I.P. SHARP TECHNICAL MEETING

Lib Gibson, Toronto

The lead article of this newsletter describes a new release of SHARP APL installed on June 20th. It represents probably the largest parcel of significant enhancements to the language ever to be released at one time. It's a tough problem to provide thorough training in such features to a widely dispersed population such as I.P. Sharp employees. Add to this the need for training in more and more specialized application areas and in programming skills, and you have the raison d'etre of I.P. Sharp's first-ever Technical Meeting.

Held in May in Toronto, the meeting brought together representatives from our 50 offices around the world and a variety of specialists from corporate headquarters. Besides the formal sessions, working groups, and birds-of-a-feather sessions, there was incessant discussion during every waking minute—and most participants stretched their waking minutes to the limit!

An important contribution to the technical expertise around I.P. Sharp is the informal exchange of ideas. This sort of communication abounds through the mailbox. A meeting such as this has greatly reinforced this information flow and facilitated the sharing of this common resource pool in every office, large and small. It was not only productive, but highly stimulating, to bring such a talented group of people together in one place. And everyone left for home a little 'sharper'.

FINANCIAL PLANNING SOFTWARE

Keith Iverson, formerly Regional Manager for the U.S. Mid-Atlantic Region, has joined Applications Software in Toronto. He will be specializing in financial planning systems and modelling languages.

Keith has been with the company for over ten years, holding positions in Toronto, London England, and New York City. He is now living in Philadelphia, and will be moving back to Toronto in August.

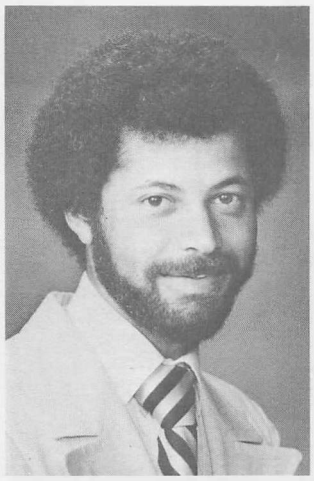
HONG KONG

The Hong Kong office "went live" early in July under the management of **James Sinclair.** James, born in London England, obtained his B.Sc. (Eng.) in Computing Science from Imperial College, University of London. James' first contact with APL was through friends who were using the language, however, it was not until he saw an actual APL system while working as an assembler programmer one summer vacation, that he understood the language's appeal. "It was not only the power of the language, but also the manner in which APL people talked about the *problem* rather than the internals of the computer system. I saw APL as a means of learning more about applications—computers never interested me for computers' sake."

James then worked with a North London timesharing bureau "who appreciated APL" through university, and joined I.P. Sharp in 1978. A year later he joined the newly formed marketing group in London, since when he has coordinated I.P. Sharp's presence at two major UK exhibitions.

James will be assisted by **Arthur Whitney**.

TORONTO



Ted Talbot

Ted Talbot has been appointed manager of the Toronto branch. Ted embarked on a career in APL almost 10 years ago, immediately following his graduation from the University of Waterloo Computer Science program. Four years of implementing business applications in APL led to a position at I.P. Sharp late in 1976.

Ted soon became Project Leader, then Support Group Manager. After three years he decided to seek his fortune in Florida as the Miami Branch Manager. Now back in Canada as the Toronto Branch Manager, Ted aims to 'provide the best service to our clients by continually creating new business challenges, and by developing the staff to meet them.'

MID ATLANTIC REGION

Fred Sencindiver, previously Branch Manager of the Washington D.C. office, has been appointed Manager of the U.S. Mid-Atlantic Region. A graduate of Antioch College, Fred has been with the company since July 1977, and previously held positions with The Computer Company and the Corporation for Public Broadcasting.

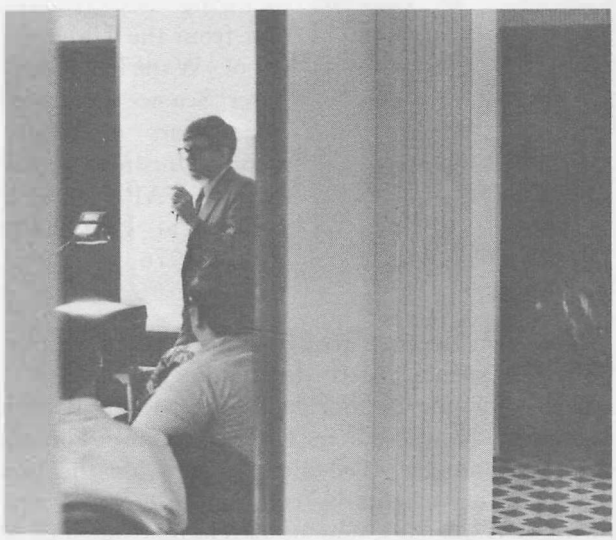
EXPANDING AUDIENCE FOR APL MYTHS

Peter Meekin, COMPRO Associates,
New Palz, NY

The 1981 Digital Equipment Users Symposium (DECUS) met in Miami Beach in May. It included four sessions, chaired by Susan Abercrombie, that dealt with topics ranging from 'What is APL?' to 'APL Hierarchical Analysis and Retrieval Techniques'.

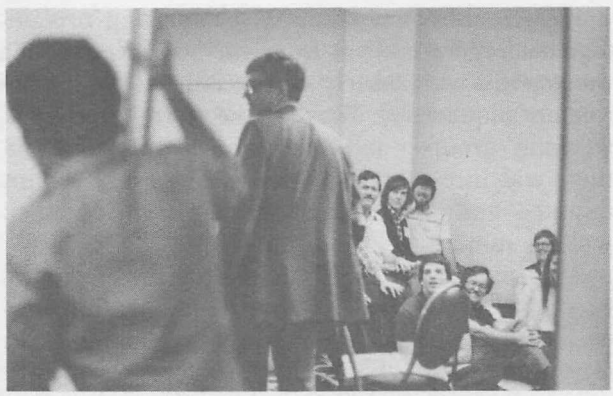
A special feature was an excellent evening talk given by **Dr. Kenneth E. Iverson** of I.P. Sharp on 'APL Myths'. The wide interest in APL among the DECUS members was clearly evidenced by the enthusiastic crowd of about 200 attendees crowded around the door of a room designed for less than 80 people.

At one point during Dr. Iverson's talk, hotel workmen began to remove the walls: they used brawn, crowbars (2), and as much discretion as possible under the circumstances, but eventually everyone could see.



1. Note the wall on the right.

2. Dr. Iverson keeps talking as the walls come down.



3. The audience, still crowding to see, moves out of the way as the last wall is removed.

The text of Dr. Iverson's paper appears in the DECUS newsletter, *The Special Character Set*, which is sent to members of DECUS and the APL Special Interest Group. Address inquiries to Roger Matus:
Digital Equipment Computer Users Society
Marlboro, Mass 01752.

Photos by R. Matus

APL82 HEIDELBERG

The Congress will be held in Heidelberg, West Germany from July 26 to July 30 1982. It is being sponsored by the APL CLUB of Germany and the German Cancer Research Centre, in cooperation with ACM's special interest group (SIGAPL), and the German Association for Informatics (GI).

The purpose of APL82 is to identify and propagate current applications of, and research in APL, and to encourage progress and development in the language and its use in many and varied fields. It will provide a forum for the exchange of information and ideas, and for the discussion of future research and development.

A call for papers has been issued. The deadline is September 1st, 1981. Further details appear in workspace 1 *APL*82, or may be obtained from either:

The Conference Chairman: **Ken Waller**
Postfach 101248
D-6900 Heidelberg 1/W. Germany

or

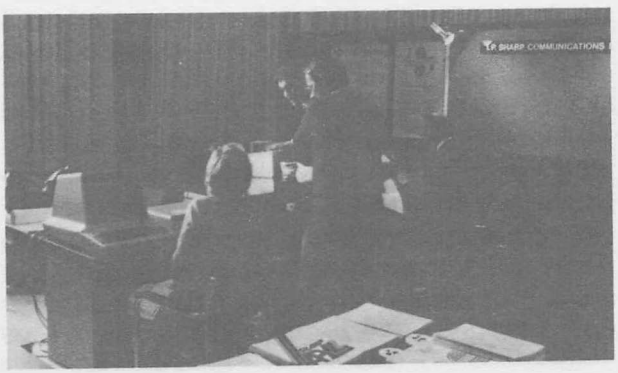
The Programme Committee Chairman:
Herrn Prof. **Dr. W. Janko**
Universitaet Karlsruhe
Institut fuer Betriebswirtschaftslehre-unternehmung
Postfach 6380
D-7500 Karlsruhe 1 / W. Germany.

THIRD DUTCH APL MINI CONGRESS

On May 26, some 350 people attended the Congress which was held in the Phillips recreation centre in Eindhoven. Five papers were presented by different users and suppliers of APL, including Fred Perkins of I.P. Sharp's London office, whose paper was entitled 'APL, a language of the future'.

The exhibition area provided us with an opportunity to demonstrate 3 different terminals, 2 flatbed plotters and a word processor, all of which were connected to a 'mobile node', a temporary node in the I.P. Sharp communications network (connected by one 1200 baud line to the permanent node in Amsterdam).

It was encouraging to see so many people at such a congress, in a country where, a few years ago, APL was virtually unknown. Everybody seemed to enjoy the day, including the free lunch, of course — and we look forward to the next congress. The rate of growth in attendees at these congresses so far has been somewhat similar to the use of APL: appropriately exponential!



PUBLICATIONS CHECKLIST

Jane Minett, Toronto

If you want to bring yourself up to date on applications software, public data bases, and SHARP APL, make sure you get copies of the following new and revised publications from your nearest I.P. Sharp office, or write to the Publications Department in Toronto.

Applications Software brochure (*revised*), June 81, 7pp. This brochure contains a summary of the major application software products available to our timesharing customers. This software may be rented or purchased by in-house SHARP APL customers.

SUPERPLOT Reference Card, (*new*), March 81. Business graphics at your fingertips, for the non-programmer.

CONSOL Reference Manual (*new*), March 81, 207pp., $12. See the March 81 issue of the I.P. Sharp Newsletter for a discussion of CONSOL, a system for maintaining data and producing business reports according to the structure of a particular organization.

CONSOL Illustrations (*new*), March 81, 37pp., $4. A handy illustrated guide to the CONSOL command language.

STARS Steward's Manual, (*new*), April 81, 47pp., $5. A programmer's tool for systems which require a case management facility.

SHARP APL File System, (*revised*), April 81, 39pp., $4. The 1981 edition of one of our oldest publications (9 years).

Batch Tasks in SHARP APL, (*revised*), February 81, 33pp., $4. Changes made to the batch task facility in November 80 are reflected in this edition.

Commodities Data Base, (*revised*), March 81, 39pp., $4. This is a users' guide to the Commodities data base, which contains prices and volumes for all commodities traded on the London, New York, and Chicago futures markets.

U.S. Consumer and Producer Price Index Data Bases, (*revised*), March 81, 32pp., $4. This is a users' guide to the CPI and PPI data bases which contain index values for U.S. consumer foods and commodities.

EASY, Econometric Analysis System brochure, (*new*), March 81, 5pp. For economists and other social sciences analysts, EASY facilitates the quantitative analysis of time series data.

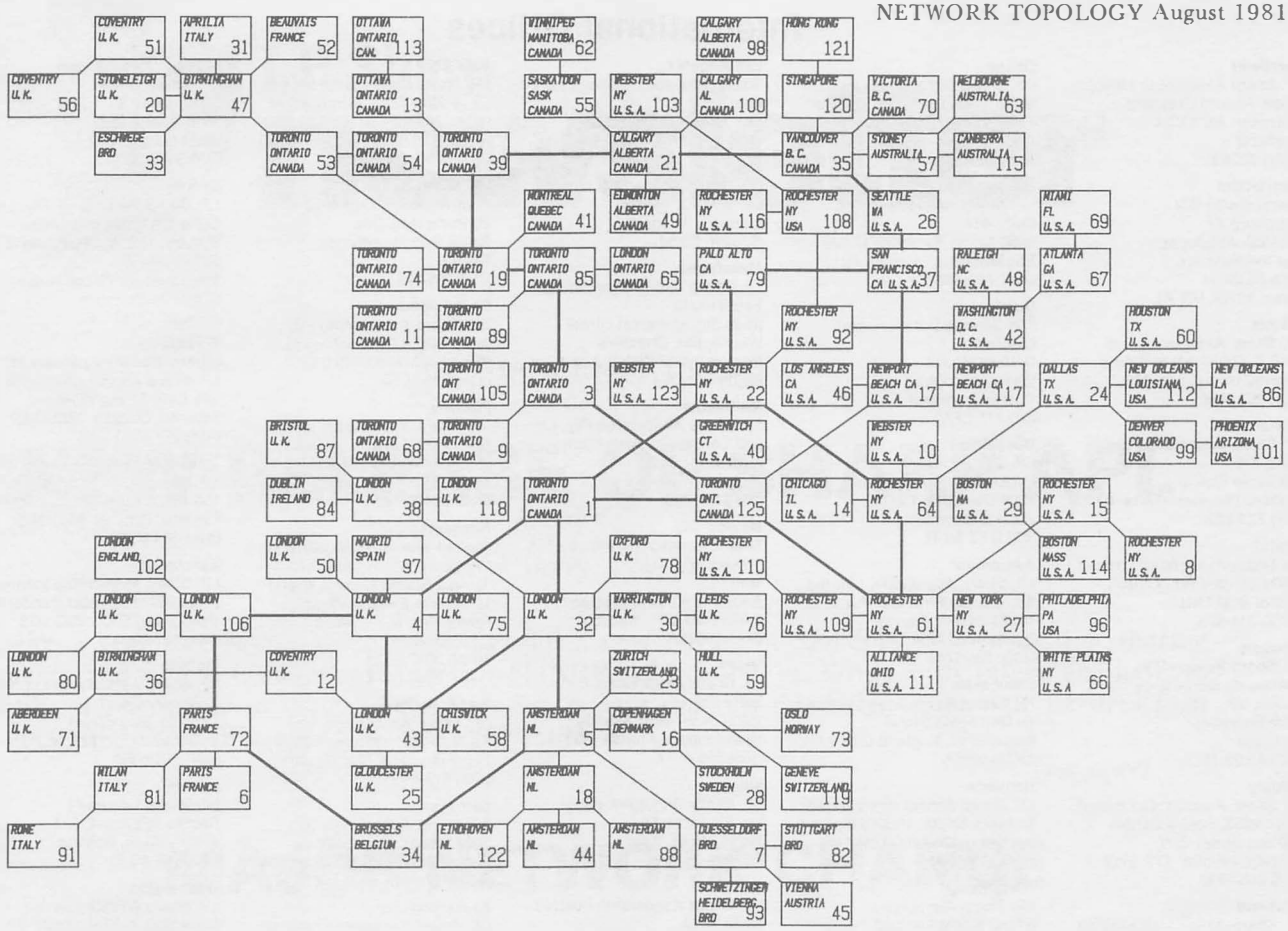
EASY, Econometric Analysis System manual, (*preliminary edition*), December 80, 93pp. The preliminary release of the user's guide to the EASY system.

SAGA, SHARP APL Graphics Aids, (*preliminary edition*), January 81, 37pp. The preliminary release of the user's guide for SAGA, a tool for systems designers writing graphics applications.

NEW NEWSLETTER
Quarterly Financial & Economic News

The Data Base Group is introducing the *I.P. Sharp Quarterly Financial and Economic Newsletter*. It is designed to keep you abreast of current developments and enhancements to the financial and economic data bases in the areas of data availability, data base software, and documentation. The newsletter should prove useful to those users interested in regular coverage of the expanding financial and economic data bases.

If you are interested in receiving the newsletter, which is scheduled to begin publication in October, please check the appropriate box on the "update" form on the next page.



UPDATE

MAILING REQUEST

- ☐ Please amend my mailing address as indicated.
- ☐ Please add the following name(s) to your Newsletter mailing list.
- ☐ Please send me a Publications Order Form.
- ☐ Please add my name to the Energy Newsletter mailing list.
- ☐ Please add my name to the Aviation Newsletter mailing list.
- ☐ Please add my name to the Financial Newsletter mailing list.
- ☐ Please send me information about your courses in

(city) _____

Name: _____

Title: _____

Co.: _____

Address: _____

_____

_____

Telephone: _____

The Newsletter is a regular publication of I.P. Sharp Associates. Contributions and comments are welcome and should be addressed to: I.P. Sharp Newsletter, 145 King Street West, Toronto, Canada M5H 1J8.

Jeanne Gershater, *Editor*
Mary Kopfensteiner, *Circulation*

Printed in Canada
July 1981
ISSN 0226 854X

I.P. Sharp Associates **Head Office:** 145 King Street West, Toronto, Canada M5H 1J8 (416) 364-5361

International Offices

Aberdeen
I.P. Sharp Associates Limited
5 Bon Accord Crescent
Aberdeen AB 12DH
Scotland
(0224) 25298

Amsterdam
Intersystems B.V.
Kabelweg 47
1014 BA Amsterdam
The Netherlands
(020) 86 80 11
Telex: 18795 ITS NL

Atlanta
I.P. Sharp Associates, Inc.
1210 S. Omni International
Atlanta, Georgia 30335
(404) 586-9600

Boston
I.P. Sharp Associates, Inc.
Suite 415
148 State Street
Boston, Massachusetts 02109
(617) 523-2506

Bristol
I.P. Sharp Associates Limited
5 Whiteladies Rd., Clifton
Bristol BS8 1NN
(0272) 314-486

Brussels
I.P. Sharp Europe S.A.
Avenue du General de
Gaulle, 39
1050 Bruxelles
Belgique
(02) 649 99 77

Calgary
I.P. Sharp Associates Limited
Suite 2660, Scotia Centre
700-2nd Street S.W.
Calgary, Alberta T2P 2W2
(403) 265-7730

Canberra
I.P. Sharp Associates Limited
16 National Circuit
Barton, ACT 2600
Australia
(062) 73-3700

Chicago
I.P. Sharp Associates, Inc.
2 North Riverside Plaza
Suite 1736
Chicago, Illinois 60606
(312) 648-1730

Cleveland
I.P. Sharp Associates, Inc.
(216) 431-6861
(local call, switched through
to Rochester office.)

Copenhagen
I.P. Sharp ApS
Ostergade 24B
1100 Copenhagen K
Denmark
(01) 11 24 34

Coventry
I.P. Sharp Associates Limited
7th Floor B Block
Coventry Point, Market Way
Coventry, England CV1 1EA
(0203) 21486/7

Dallas
I.P. Sharp Associates, Inc.
Suite 1148, Campbell Centre
8350 Northcentral Expressway
Dallas, Texas 75206
(214) 369-1131

Denver
I.P. Sharp Associates, Inc.
Suite 416
5680 South Syracuse Circle
Englewood, Colorado 80111
(303) 741-4404

Dublin
C/o Gamma Data Systems
Limited
Dollard House
Wellington Quay
Dublin 2, Ireland
(01) 711 877

Düsseldorf
I.P. Sharp GmbH
Leostrasse 62A
4000 Dusseldorf 11
West Germany
(0211) 57 50 16

Edmonton
I.P. Sharp Associates Limited
Suite 2358, Principal Plaza
10303 Jasper Avenue
Edmonton, Alberta T5J 3N6
(403) 428-6744

Gloucester
I.P. Sharp Associates Limited
29 Northgate Street
Gloucester, England GL1 2AN
(0452) 28106

Hamilton
I.P. Sharp Associates Limited
14 Hess South, Hess Village
Hamilton, Ontario L8P 3M9
(416) 527-3801

Hong Kong
I.P. Sharp Associates
(Hong Kong) Limited
Admiralty Centre
Tower One, Suite 606
Hong Kong
5-294341

Houston
I.P. Sharp Associates, Inc.
Suite 375, One Corporate Square
2600 Southwest Freeway
Houston, Texas 77098
(713) 526-5275

Kitchener/Waterloo
I.P. Sharp Associates Limited
3 Menno St.
Waterloo, Ont. N2L 2A4
(519) 884-5420

London, Canada
I.P. Sharp Associates Limited
Suite 510, 220 Dundas Street
London, Ontario N6A 1H3
(519) 434-2426

London, England
(European Headquarters)
I.P. Sharp Associates Limited
132 Buckingham Palace Road
London SW1W 9SA, England
(01) 730-0361
Telex: 8954178 SHARP G

Los Angeles
Suite 1230, 1801 Century Park
East
Los Angeles, Ca. 90067
(213) 277-3878

Madrid
I.P. Sharp Associates Ltd.
Serrano 23, Piso 8
Madrid - 1, Spain
(91) 276 70 54

Manchester
I.P. Sharp Associates Limited
Paul House
89-91 Buttermarket Street
Warrington, Cheshire
England WA1 2NL
(0925) 50413/4

Melbourne
I.P. Sharp Associates Pty. Ltd.
520 Collins St., 13th Floor
Melbourne 3000
Victoria, Australia
(03) 614-1766

Mexico City
Teleinformatica de Mexico S.A.
(Agent)
Mail to:
Arenal N 40, Chimalistac
Mexico 20 D.F., Mexico
(905) 550-8033

Miami
I.P. Sharp Associates, Inc.
Suite 240
15327 N.W. 60th Avenue
Miami Lakes, Florida 33014
(305) 556-0577

Milan
I.P. Sharp Srl (agent I.S.I.)
Via Eustachi 11
20129 Milan, Italy
(02) 271-6541

Montreal
I.P. Sharp Associates Limited
Suite 1610
555 Dorchester Boulevard W.
Montreal, Quebec H2Z 1B1
(514) 866-4981

New York City
I.P. Sharp Associates, Inc.
Mail to:
Suite 2004
200 Park Avenue
New York, N.Y. 10169
(212) 557-1200

Newport Beach
I.P. Sharp Associates, Inc.
Suite 1135
610 Newport Center Drive
Newport Beach, Ca. 92660
(714) 644-5112

Oslo
I.P. Sharp A/S
Dronningens gate 34
Mail to: P.Boks 486 Sentrum
OSLO 1, Norway
(02) 41 17 04

Ottawa
I.P. Sharp Associates Limited
Suite 600, 265 Carling Ave.
Ottawa, Ontario K1S 2E1
(613) 236-9942

Palo Alto
I.P. Sharp Associates, Inc.
Suite 201, 220 California Ave.
Palo Alto, Ca. 94306
(415) 327-1700

Paris
I.P. Sharp Sarl.
Tour Neptune, Cedex 20
20 Place de Seine
92086 Paris-la-defense
France
(1) 773 57 77

Philadelphia
I.P. Sharp Associates, Inc.
Suite 604, 437 Chestnut St.
Philadelphia, Pa. 19106
(215) 925-8010

Phoenix
I.P. Sharp Associates, Inc.
Suite 503
3033 N. Central Avenue
Phoenix, Arizona 85012
(602) 264-6819

Rochester
(United States Headquarters)
I.P. Sharp Associates, Inc.
(United States Headquarters)
1200 First Federal Plaza
Rochester, N.Y. 14614
(716) 546-7270
Telex: 0097 8473
0097 8474

San Francisco
I.P. Sharp Associates, Inc.
Suite C-415, 900 North Point St.
San Francisco, Ca. 94109
(415) 673-4930

San Jose
I.P. Sharp Associates, Inc.
3028A Scott Boulevard
Santa Clara, California 95050
(408) 727-9446

Saskatoon
I.P. Sharp Associates Limited
Suite 208, 135 21st Street E.
Saskatoon, Sask. S7K 0B4
(306) 664-4480

Seattle
I.P. Sharp Associates, Inc.
Suite 217
Executive Plaza East
12835 Bellevue-Redmond Rd.
Bellevue, Washington 98005
(206) 453-1661

Singapore
I.P. Sharp Associates (S) Pte. Ltd.
Suite 1501, CPF Building
79 Robinson Rd.
Singapore 0106
Republic of Singapore
2230211

Stockholm
I.P. Sharp AB
Kungsgatan 65
S111 22 Stockholm, Sweden
(08) 21 10 19

Stuttgart/Boeblingen
I.P. Sharp GmbH
Schafgasse 3
7030 Boeblingen
West Germany
(070 31) 2 30 14

Sydney
I.P. Sharp Associates Pty. Ltd.
Suite 1351, 175 Pitt Street
Sydney, N.S.W., Australia 2000
(02) 232-6366
Freight to: c/-Greenaways
Customs Services
Alexandria, N.S.W.

Toronto
(International Headquarters)
I.P. Sharp Associates Limited
145 King Street West
Toronto, Ontario M5H 1J8
(416) 364-5361

Toronto (Special Systems Div.)
I.P. Sharp Associates Limited
156 Front Street W., 5th Floor
Toronto, Ontario M5J 1G6
(416) 364-5361

Vancouver
I.P. Sharp Associates Limited
Suite 902, 700 West Pender St.
Vancouver, B.C. V6C 1G8
(604) 687-8991

Victoria
I.P. Sharp Associates Limited
Chancery Court
1218 Langley Street
Victoria, B.C. V8W 1W2
(604) 388-6365

Vienna
I.P. Sharp Ges.mbH
Rechte Wienzeile 5/3
A-1040 Wien, Austria
(0222) 57 65 71

Washington
I.P. Sharp Associates, Inc.
Suite 305, 2033 K Street N.W.
Washington, D.C. 20006
(202) 293-2915

White Plains
I.P. Sharp Associates, Inc.
180 East Post Road, LL-6
White Plains, New York 10601
(914) 328-8520

Winnipeg
I.P. Sharp Associates Limited
Suite 208
213 Notre Dame Avenue
Winnipeg, Manitoba R3B 1N3
(204) 947-1241

Zurich
I.P. Sharp A.G.
Fortunagasse 15
8001 Zurich
Switzerland
(01) 211 84 24

SHARP APL Communications Network

APL OPERATOR VOICE (416) 363-2051 COMMUNICATIONS (416) 363-1832

Local dial access is available in all locations listed above and in:

• Alliance • Ann Arbor • Austin • Baltimore • Birmingham • Buffalo • Clewiston (Fl) • Dayton • Des Moines • Des Plaines • Detroit • Ft. Lauderdale • Greenwich (Ct) • Halifax • Hartford • Hull • Knoxville • Laurel • Leeds • Liverpool • Lyndhurst • Minneapolis • New Orleans • Oxford • Quebec City • Raleigh • Red Deer • Regina • Rome • Santa Ana • Schwetzingen • Sunnyvale • Syracuse • Towanda • Ukiah • Warrington

Our private, packet-switched network connects with the Value Added Networks in:

• Alaska • Argentina • Bahrain • Bermuda • Finland • Hawaii • Israel • Luxembourg • Mexico • New Zealand • The Philippines • Portugal • Puerto Rico • Taiwan

In the continental United States the SHARP APL Network is interconnected with the Value Added networks to provide access in 170 more cities, and in Canada with 40 more. In all, with the 80 cities served by the I.P. Sharp Network listed above, SHARP APL is accessible from close to 300 places via a local phone call. Please ask at your nearest I.P. Sharp office for a complete list of access points and access procedures. Our private network also connects with the worldwide Telex network via the Rochester, N.Y. and Amsterdam codes.