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

SATN-INDEX
1 June 1979

SHARP APL TECHNICAL NOTES

SATN-0	January 1, 76		SATN Introduction
SATN-1	January 1, 76		TASKID
SATN-2	February 14, 79	(Rev. ~~4~~ 5)	Control Messages
SATN-3	January 1, 76		□*OUT*
SATN-4	April 1, 78	(Rev. 2)	N-tasks and B-tasks
SATN-5	August 1, 78	(Rev. 2)	Batch APL
SATN-6	January 1, 76		Execute
SATN-7	January 1, 76		Latent Expression
SATN-8	March 1, 79	(Rev. 2)	*HSPRINT*
SATN-9	August 1, 78	(Rev. 1)	Usage Inquiry System
SATN-10	June 1, 78	(Rev. 2)	*SORTREQ*
SATN-11	January 1, 76		)*RESET*
SATN-12	January 1, 76		)*COPY*
SATN-13	March 10, 78		Early Warnings
SATN-14	August 15, 78	(Rev. 2)	Package - A New Variable Type
SATN-16	April 20, 76		File System Must-Write Buffers
SATN-17	June 30, 76		Formatting Primitive
SATN-18	July 1, 76		□*FMT*
SATN-19	January 1, 77		Fileprint
SATN-20	June 1, 78	(Rev. ~~4~~)	System Variables, Session Variables and System Functions
SATN-21	June 1, 78	(Rev. 1)	□*WS* and □*FD*
SATN-22	January 1, 79	(Rev. 2)	APL Workspace Transfer
SATN-23	July 15, 78	(Rev. 1)	Comparison Tolerance
SATN-24	March 23, 77		)*SYMBOLS*
SATN-25	May 15, 77		Extensions to Argument Passing
SATN-26	September 10, 77		Enhancements to the File System
SATN-28	July 11, 77		Terminal Control
SATN-29	June 15, 78		System Time and Timestamps
SATN-30	January 1, 79		Numeric Display
SATN-31	February 1, 79		Line Editing in SHARP APL
SATN-32	March 30, 79		Shared Variables
SATN-33	March 26, 79		Event Trappinng

SHARP AI TECHNICAL NOTES

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

SHARP APL TECHNICAL NOTES

SATN-0	January 1, 76		SATN Introduction
SATN-2	October 21, 79	(Rev. 4)	Control Messages
SATN-4	April 1, 78	(Rev. 2)	N-tasks and B-tasks
SATN-5	February 1, 78	(Rev. 3)	Batch APL
SATN-8	March 1, 79	(Rev. 2)	*HSPRINT*
SATN-9	November 1, 80	(Rev. 2)	Usage Inquiry System
SATN-10	June 1, 78	(Rev. 2)	*SORTREQ*
SATN-19	January 1, 77		Fileprint
SATN-22	March 11, 80	(Rev. 3)	APL Workspace Transfer
SATN-23	July 15, 78	(Rev. 1)	Comparison Tolerance
SATN-28	July 11, 77		Terminal Control
SATN-29	June 15, 78		System Time and Timestamps
SATN-34	September 10, 80		Replication
SATN-35	September 15, 80		Extended Upgrade and Downgrade
SATN-36	October 1, 80		Direct Definition
37	MAY 1, 82	(Rev 1)	IBM 3270 User Guide
39	June 1, 81		S-tasks
40	June 20, 81		Complex Nos
41	June 20, 81		Composition & Enclosure
42			Determinant ◇ −.× ⍵
43			⍎ ⍕ DPS
44			Enhancements to event handling
45			Language extensions of May '83
46			Enhancements in Update #1
47			IBM 3270 Users Guide
48			HCPRINT

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

SHARP APL TECHNICAL NOTES

TITLE: SATN Introduction

ABSTRACT: The structure and raison d'etre of this new family of documents is presented.

KEYWORDS: WS 1 *SATN*

[illegible] Sharp Corporation
[illegible] Street West
Toronto, Ontario [illegible]
[illegible]

[illegible]
[illegible]

SHARP AR TECHNICAL NOTES

TITLE: [illegible]

ABSTRACT: [illegible]

[illegible]

SHARP APL is a dynamic environment that has been steadily improving and expanding for several years. It seems the ability to extend the APL environment has always outstripped the ability to document the extensions. Sometimes new facilities are released promptly, but with little or no documentation. Other times their release is considerably delayed waiting for proper documentation. The intent of SHARP APL technical notes (SATNS) is to fill the gap between "no documentation" and "good documentation" and to minimize the time between "implemented" and "released". Following the publication of a SATN or of a series of related SATNS will come a proper manual complete with extensive examples and guidelines.

SATNS will usually be written by those responsible for the development being documented. They will tend to be technical and terse, with few examples and guidelines. However, an experienced SHARP APL user should be able to read a SATN, experiment a bit, and then put the new capabilities to work in solving his problems.

A SATN will often refer to other SATNS or manuals. Sometimes a SATN will reference as yet unwritten SATNS about as yet unimplemented features. An example of this would be a family of related developments that are each distinct and useful enough to warrant being released separately. Occasionally a SATN will be reprinted to incorporate corrections and additions. In these cases it will be reprinted with the same SATN number, but with a new publication date.

Notification of new SATNS will be made through `1 NEWS` and the newsletter. SATNS and an index of SATNS are available in WS `1 SATN`.

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

SATN 1
1 JAN 76

SHARP APL TECHNICAL NOTES

TITLE: TASKID

ABSTRACT: APL tasks are now assigned a TASKID (a small integer) at initiation. The port number (i.e. communication link information) of a task is no longer used as a task identifier.

KEYWORDS: PORT
TASK
)MSG
)PORTS

L.F. Sharp Associates
145 King Street West
Toronto Ontario M5H 1J8
(416) 368-0321

SHARP API TECHNICAL NOTES

The use of port number as an APL task identifier has several drawbacks. The most significant is that they tend to be reassigned quickly. For example, in sending messages to PORT 249003, between one message and the next the user being communicated with could have signed off and a new user signed on.

The source of the problem is using a number which is information about the communication link to also serve as a task identifier. To solve this problem we are introducing TASKIDS into the system. Everywhere that port numbers were used as task identifiers, TASKIDS will now be used. At initiation, each task will be assigned a TASKID. TASKIDS will be monotonically increasing integers that wrap around to 1 at some maximum number. The maximum number is chosen so as to ensure that TASKIDS are not reassigned in less than several days.

The introduction of TASKIDS affects the system as follows:

1. TASKID is reported at sign-on:

```
)1234567:LOCK
  567)   17.14.07  12/08/75  EXAMPLE
```

2. TASKID is reported at sign-off:

```
      )OFF
567  17.40.03 12/08/75  EXA
CONNECT .....
```

3. The argument to)*MSG* is TASKID:

```
      )MSG  567  READY?
SENT
567:  NO
```

4. The result of 2 ⎕*WS* 3 now contains both PORT and TASKID. PORT remains the 10th element and TASKID is now the 11th element.

5. The report from)*PORTS* now contains a third column for TASKID.

```
      )PORTS  1234567
249003 EXA 567

      )PORTS EXA
249003 EXA 567
```

6. A successful ⎕*RUN* returns the TASKID of the new task.

7. The first column of the result from ⎕*RUNS* is TASKID.

8. The arguments to ⎕*BOUNCE* are TASKIDS.

I.P. Sharp Associates
2 First Canadian Place,
Suite 1900,
Toronto, Ontario M5X 1E3
(416) 364-5361
Telex 0622259

SHARP APL TECHNICAL NOTES

SATN-2
1 MAY 82
Rev. 5

TITLE: **CONTROL MESSAGES**

ABSTRACT: A convention is established to distinguish control information from data.

KEYWORDS: *□OUT*
HSPRINT

The advent of *⎕OUT* and *HSPRINT* (SATN-8) forced a solution to a problem that has always existed. The problem is how to distinguish between data and information about the data in a convenient, systematic manner. To solve this problem for *⎕OUT* and *HSPRINT* a convention has been adopted that will be consistently used in future developments that pose the same problem.

A control message is a 22 element character vector with the following format:

Position	Field Names	Contents
1	Delim	`¯1↑⎕AV`. This prints on a terminal as the canonical unprintable character ⎕ .
2-5	Message Number	A four digit number identifying the type of control message. Numbers start at 0000 and are always leading zero filled.
6		Blank.
7-10	Component Count	A four digit number specifying the number of following components controlled by this control message. Numbers start at 0000 and are always leading zero filled.
11		Blank.
12-22	Mnemonic	Text describing the control message.

NOTE:

Only the first three fields are checked by *HSPRINT*.

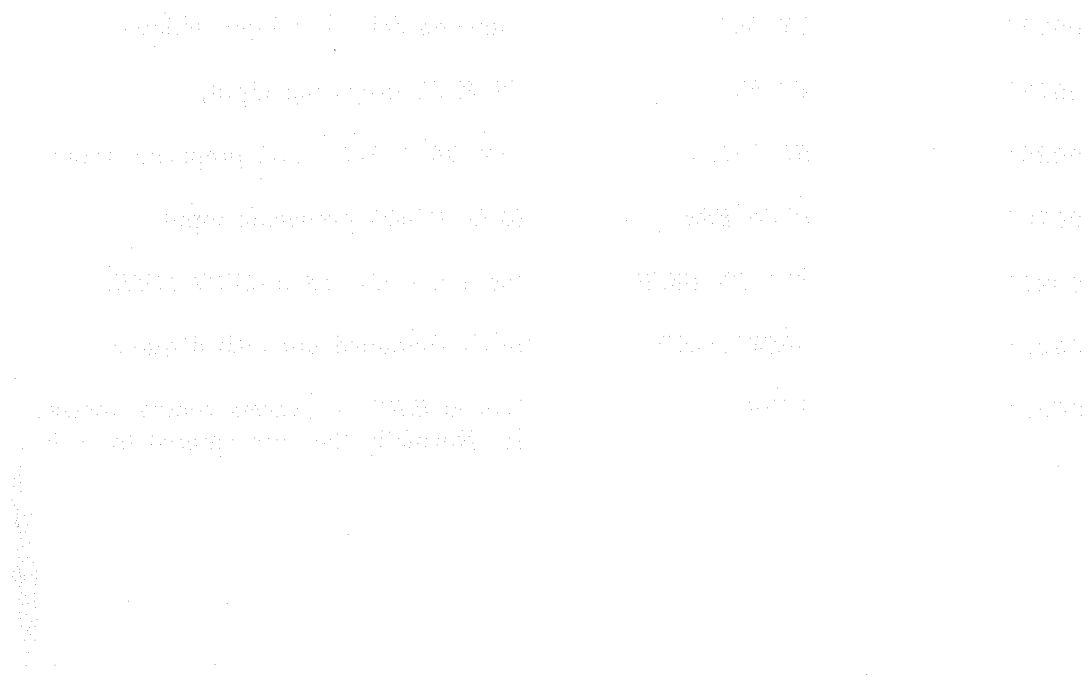
Any deviation from the fixed format means the object is not a control message, and hence is data. Note that nothing prevents data from accidentally looking exactly like a control message. Certain systems recognize control messages. To other systems which are not checking for them, control messages are simply character data.

Control messages that are of universal significance will be assigned message numbers ascending from 0000. Users with applications that can benefit from the control message convention are urged to adhere strictly to the format and to assign their message numbers ascending from 9000.

A list of control messages can be found in *WS 1 SATN*. The following is a list of current control messages:

Number	Mnemonic	Description
'0000'	*FORMAT*	Arguments to □*FMT* follow.
'0001'	*MIXED*	Mixed output follows.
'0002'	*PAGEN*	New page number follows.
'0003'	*QUADP*	⍞ output follows.
'0004'	*TRANSLATE*	A translate table follows.
'0005'	*DIGITS*	New value for □*PP* follows.
'0006'	*WIDTH*	New value for □*PW* follows.
'0007'	*TITLE*	A page title follows.
'0008'	*SUBTITLE*	A page subtitle follows.
'0009'		
'0010'	*PRTUCTM*	Switch to *PRINT/NOPRINT* user control messages.
'0011'	*PAGE*	Skip to beginning of a new page.
'0012'	*CARRIAGE*	Vertical forms control.
'0013'	*PRTARBOUT*	Switch to print/noprint *ARBOUT* data
'0014'	*ARBOUT*	*ARBOUT* argument follows.
'0015'	*ARBIN*	Prompt and result follows.
'0016'	*PRTARBIN*	Switch to *PRINT/NOPRINT ARBIN* data.
'0017'	*FICHE*	*HSPR* to process file for microfiche printing.
'0018'	*HFILE*	File created by *HSPRINT*.
'0019'	*QUAD ER*	□*ER* follows.
'0020'	*INPUT*	Normal input follows.
'0021'	*QUAD*	□ input follows.
'0022'	*QUAD PRIME*	⍞ input follows.

'0023'	*FN DEFN*	Function definition input follows.
'0024'	*WS FULL*	*WS FULL* preparing input.
'0025'	*ST FULL*	*SYMBOL TABLE FULL* preparing input.
'0026'	*DEFN ERR*	*DEFN ERROR* preparing input.
'0027'	*NOT IN DEFN*	Input not allowed in *DEFN MODE*.
'0028'	*OPEN QUOTE*	Input contained unmatched quote.
'0029'	*PINP*	Causes *HSPR* to process control messages 19 through 28. Normally they are ignored by *HSPR*.



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

SHARP APL TECHNICAL NOTES

TITLE: *□OUT*

ABSTRACT: A system function is defined that designates a file to which terminal output along with necessary control information is appended.

KEYWORDS: Control Message

SHARP API TECHNICAL NOTES

The system function □*OUT* designates a file to which data to be printed on the terminal is appended along with any necessary control information. The □*OUT* function also controls whether data to be printed on the terminal is in fact printed.

```
R←□OUT PRYN, TIENUMBER
```

The argument to □*OUT* is a two element numeric vector or an empty vector.

The first element specifies whether or not user output is printed on the terminal. A value of 1 specifies that user output is printed and a value of 0 specifies that user output is not printed. If an *NTASK* or *BTASK* (SATN-4) specifies a *PRYN* value of 1 and does terminal output, the output is discarded.

The second element is the tienumber of the file to which the user's output is to be appended. A value of 0 for tienumber specifies that no user output file is being used.

An argument of `⍳0` has no effect, but does return a result.

The result of □*OUT* is a two element vector of the previous settings.

Note that if an output file has been designated that simple expressions like `1+1` can cause an error (e.g. *FILE TIE ERROR*, or *FILE FULL* error). Also note that □*OUT* `0 0` will discard all user output.

Only user output is appended to the output file. Error reports, messages from the operator or another user, and system command reports are not appended to the output file.

Format of the Output File

Most user output simply results in appending the data to the output file. When required, a control message is appended preceding the data component or components.

A control message is a 22 element character vector with a fixed format described in SATN-2. Currently four types of control messages are automatically appended along with their related data to the output file. It is expected that eventually more control messages will be automatically appended to the output file.

The following are examples of the current four types:

1. $\square FMT$ output

If $\square FMT$ is the only expression on the line, then the conversion of its arguments to a character matrix is not done for the $\square OUT$ file. Instead, a control message is appended followed by components containing the left and right arguments to $\square FMT$ For example:

```
'I5' □FMT ⍳4
```

would append:

Component 1 -	'□0000 0002 *FORMAT* '	Control Message
Component 2 -	'I5'	Character left argument
Component 3 -	1 2 3 4	Numeric right argument

If the right argument to $\square FMT$ is a list then one component is appended for each list element in the right argument. For example:

```
'5A1,2X,I5' □FMT (2 5⍴'NUTS BOLTS';2 5⍴⍳10)
```

would append:

Component 1 -	'□0000 0003 *FORMAT* '
Component 2 -	'5A1,2X,I5'
Component 3 -	2 5⍴'NUTS BOLTS'
Component 4 -	2 5⍴⍳10

2. Mixed output

One component is appended after the control message for each list element.
For example:

```
'YOUR SCORE IS ';RIGHT;' RIGHT AND ',100-RIGHT;' WRONG.'
```

would append (assuming right is 75):

Component 1 - '□0001 0005 *MIXED* '

Component 2 - '*YOUR SCORE IS* '

Component 3 - 75 Numeric

Component 4 - ' *RIGHT AND* '

Component 5 - 25 Numeric

Component 6 - ' *WRONG.*'

3. ⎕ Output

When ⎕ is assigned a value a control message will be appended and then the value will be appended.

```
⎕← 'PLEASE ENTER YOUR NAME'
```

would append:

Component 1 - '□0003 0001 *QUADP* '

Component 2 - '*PLEASE ENTER YOUR NAME*'

4. Arbitrary output

A control message is appended followed by a component with the arbout argument.

```
ARBOUT 7 ⍵ RING BELL ON ASCII TERMINAL
```

would append:

Component 1 - '⍞0014 0001 *ARBOUT* '

Component 2 - 7 Numeric

The following example illustrates many characterictics of the use of ⎕*OUT*:

```
      ∇FOO
[1]    'OUTF' ⎕CREATE 7
[2]    ⎕OUT 0 7
[3]    A←⍳2
[4]    A
[5]    ⎕←'ABC'
[6]    ⍞←'DEF'
[7]    'I5'⎕FMT (A;2+⍳2)
[8]    ⎕←B←'I2'⎕FMT (A;2+⍳2)
[9]    A;B;3
[10]   'SURPRISE' ⎕APPEND 7
[11]   A←⎕OUT 1 0
     ∇
```

After executing *FOO* the file *OUTF* would contain the following 15 components:

Component	Value	Explanation
1	`1 0`	Integer vector from line 2.
2	`1 2`	Integer vector from line 4.
3	`'ABC'`	Character vector from line 5.
4	`'0003 0001 QUADP '`	Line 6.
5	`'DEF'`	Line 6.
6	`'0000 0003 FORMAT'`	Line 7.
7	`'I5'`	Line 7.
8	`1 2`	Value of *A* from line 7.
9	`3 4`	2+⍳2 from line 7.
10	`1 3` `2 4`	Character matrix from line 8.
11	`'0001 0003 MIXED '`	Line 9.
12	`1 2`	Line 9.
13	`1 3` `2 4`	Character matrix from line 9.
14	`3`	Line 9.
15	`'SURPRISE'`	Line 10.

Printing Files

The *HSPRINT* facility (SATN-8) is available for printing on a high speed printer files with the format created by using ⎕*OUT*. *HSPRINT* is much like *FILEPRINT*, but is easier to use, has more capabilities and is more efficient. The main difference from *FILEPRINT* is that *HSPRINT* can format and print numeric data. This means that expensive data conversion that must be done by the APL system to create a *FILEPRINT* file need not be done for an *HSPRINT* file. In addition to the control messages automatically appended to a ⎕*OUT* file, *HSPRINT* recognizes many others (e.g. a control message to cause printing to start on a new page).

Workspace 1 *PRINTOUT* (SATN-8) can be used to print a file on a terminal exactly as *HSPRINT* would print it on a high speed printer.

Application ideas

1. □*OUT* can be used to "spool" terminal output. An example would be a system that uses considerable cpu and connect to create a report that is currently printed as the system progresses. Such a system is subject to expensive reruns if there are minor line problems, the paper does not feed properly, or if some detail in the report is not correct. By capturing the output on file and not printing it on the terminal, the critical cpu and connect part of the job can be done more quickly with less chance of interruption or failure. The report can then be printed on the terminal by a simple function that is easy and cheap to restart or rerun. In addition, if some detail is wrong (e.g. a mis-aligned heading) it can be fixed and printed correctly without a complete rerun. In addition to making it easy to convert old systems to use "spooling", □*OUT* makes it easier to design new systems using this powerful technique.

2. □*OUT* can be used to capture output from systems that were not designed to provide data for further manipulation. An example would be a system that prints a report that contains data needed for some other application. Without □*OUT* it would be necessary to modify the application (which might be impossible with locked functions!). Using □*OUT* the report can be captured on file and then the data can be manipulated as desired.

3. □*OUT* can be used to provide sophisticated support for CRT terminals. With the output on file, images that have been lost can be recalled. This is an example where the output is desired both on the terminal and in the file.

4. □*OUT* can be used to easily convert existing systems to run as *NTASKS* or *BTASKS*. It also considerably simplifies the design and implementation of new systems to run as *NTASKS* or *BTASKS*.

5. □*OUT* can be used to easily allow programs currently printing reports on the terminal to use *HSPRINT*. It also considerably simplifies the design and implementation of new systems that use *HSPRINT*.

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

SATN 4
1 APR 78
Rev. 2

SHARP APL TECHNICAL NOTES

TITLE: N-tasks and B-tasks

ABSTRACT: System functions are presented that allow APL execution without a terminal. The facility for batch APL is also briefly described.

KEYWORDS: □*RUN*
□*RUNS*
□*BOUNCE*
Tasks
T-tasks
N-tasks
B-tasks
Batch APL

U.S. Sharp Document
440 King Street West
Toronto, Ontario M5V 1L6
(416) 599-2581

SHARP AP TECHNICAL NOTES

TASK - T-task, N-task, B-task

An active APL workspace is a task. There are now three types of tasks in SHARP APL:

T-task - A task with a terminal attached to it. A T-task is created when a terminal is signed onto the system. This is the traditional task in APL.

N-task - A task with no terminal attached. An N-task is created by the ⎕*RUN* function. Thus an N-task is created by another task.

B-task - A task with no terminal attached that is created by the system B-task scheduler from a request submitted by a user. A request to create a B-task is made with the request function in workspace `1 BTASKREQ`. B-tasks are automatically scheduled by the system to use system resources not required by N-tasks or T-tasks, and are normally run during periods of light system load. A B-task is a batch APL task.

An important aspect of N-tasks and B-tasks is that they cost less than T-tasks. This is reasonable as neither requires communication links and B-tasks do not consume prime time cpu units.

APPLICATIONS

The N-task and B-task are formal recognition of a pattern of using APL which has become ever more prevalent since the introduction of the SHARP APL file system. Often an APL task can take all its input from files and place all of its output in files. The final result might be completely contained within the system or might be a file for printing on the highspeed printer or for transferring to an external medium. Throughout this activity (minutes to perhaps hours) a terminal is unnecessary and is in fact harmful. A terminal that is "just sitting there" has at least three drawbacks:

1. It must be guarded to prevent interruption or damage to the running task.

2. It leaves the task subject to abnormal completion due to communication line noise or failure.

3. It pointlessly ties up a communication link and terminal.

To facilitate converting systems that currently run as T-tasks to run as N-tasks or B-tasks, and to make it easier to design new applications as N-tasks or B-tasks the system function ⎕*OUT* (SATN-3) has been made available. This system function designates a file to which terminal output, along with necessary control information, is appended.

USER NUMBERS

Each task has two user numbers associated with it: the user number it is signed on with, and the sign-on number of the task that initiated it. In many cases (always for T-tasks) the initiator number is the same as the sign-on number.

Only one T-task is allowed on the system with the same sign-on number at one time. Up to four N-tasks can be simultaneously active on the same sign-on number. The B-task scheduler automatically controls the number of simultaneously active B-tasks.

CREATING AN N-TASK

The ⎕*RUN* system function is used to initiate an N-task.

```
R←⎕RUN 'SIGN-ON LOADWSID CONTWSID CPULIM CONLIM'
```

The argument to ⎕*RUN* is a character vector consisting of five fields, separated by blanks.

Field	Name	Description
1	*SIGNON*	The user number and lock to be signed on. If only a : appears in the sign-on field then the sign-on number and lock of the task executing the ⎕*RUN* are used.
2	*LOADWSID*	The name of the workspace to be loaded at sign-on, or :. If the *LOADWSID* is elided, (only : appears), then a copy of the active workspace is used, and a return code of `0 0` is given to the new copy. If the *LOADWSID* is specified, the workspace must contain a non-empty ⎕*LX* (SATN-7).
3	*CONTWSID*	The name of the workspace the active workspace (if non-empty) is saved into when the task terminates. The library number, if present, **must** match the sign-on number.
4	*CPULIM*	The number of cpu units (must be < `100000`) the task can use before it is forced off. This field is used by the scheduler to control a task and should be realistically estimated.
5	*CONLIM*	The number of seconds (must be < `100000`) the task can run before it is forced off. This field is used by the scheduler to control a task and should be realistically estimated.

An example of a properly formed N-task creation would be:

```
⎕RUN '1234567:SECRET LRS:TUMS LRSEND:ABC 300 600'
```

This would sign on user `1234567:`*SECRET* and load workspace `1234567` *LRS*`:`*TUMS*. The ⎕*LX* in the workspace would then be executed. When the task terminates, the workspace would be saved in `1234567` *LRSEND*`:`*ABC*. If more than 10 minutes of connect time or more than 300 cpu units had elapsed, the task would be terminated.

Other examples of properly formed statements would be:

```
□RUN '1234567:SECRET 1234567 A:B 1234567 B:C 300 600'

□RUN ': LRS:TUMS LRSEND:ABC 300 600'

□RUN ': START END 600 1000'

□RUN '1234567:SECRET 7654321 NEW END 600 1000'

□RUN '1234567:SECRET : END 400 1250'

□RUN ':: END 800 1800'
```

Note that the last example splits the active workspace into two tasks. The result of the □*RUN* could be tested in each task to determine if it was the original or the new task.

RESULT OF □*RUN*

The result of □*RUN* is a 2 element numeric vector.
Execution of □*RUN* normally takes 5 to 10 connect seconds.

First Element		Second Element	
Meaning	**Value**	**Value**	**Meaning**
Successful	0	*N*	*TASKID* assigned the new task. If *N* is 0, this task is the new task resulting from a □*RUN* with an elided *LOADWSID*.
Bad argument	1	*N*	Origin 0 point of discovery of bad argument
Too many N-tasks running	2	0	
A user can be the sign-on number or initiator of only 4 N-tasks	3	0	
Workspace quota exhausted	4	0	
No room in *LOADWSID* for □*SP*	5	0	
No room in SYMBOL TABLE of *LOADWSID* for session variables	6	0	
Sign-on number invalid	10	0	
Sign-on number locked out	11	0	
LOADWSID not found	12	0	
Improper □*LX*	13	0	
System resources exceeded	14	0	
User not allowed to □*RUN*	15	0	
Hardware read error on load of *LOADWSID*	16	0	

CREATING A B-TASK

A request for a B-task creation is made in much the same way as a fileprint request is made. A request function from workspace 1 *BTASKREQ* is used to provide the system B-task scheduler the information (essentially the same as the argument with □*RUN*) necessary to create the B-task. When system resources are available, the B-task scheduler will execute a □*RUN* with the arguments provided in the request as if the requestor himself had executed the □*RUN*. See SATN-5 for detailed information about B-tasks.

SESSION VARIABLE □*SP* (see SATN-20 and SATN-5)

The session variable □*SP* (Session Parameter) can be used to pass parameters, data and functions forward into the *LOADWSID* of an N-task or B-task. The local □*SP* of a task doing a □*RUN* is copied forward into the *LOADWSID* of the new N-task. The local □*SP* of a task submitting a B-task request becomes part of the request and when the B-task schedulor does the □*RUN* to start the B-task, that □*SP* is copied forward into the *LOADWSID* of the B-task.

□*RUNS*

```
R←□RUNS
```

Result is an integer matrix. Each row contains information about a task that has the same sign-on number or was initiated by the same sign-on number. The first row of □*RUNS* will always contain the information about the task executing the □*RUNS*.

Column	Description
1	*TASKID*
2	sign-on number
3	sign-on number of task which initiated this task
4	type of task (0 for T-task; 1 for N-task; 2 for B-task)
5	cpu units consumed
6	connect time in seconds
7	*CPULIM* as specified in initiating □*RUN*
8	*CONLIM* as specified in initiating □*RUN*

Do not assume a row length of 8 as new columns may be added.

The result of `I28` (terminal type) for an N-task or B-task is `¯1`.

⎕BOUNCE

```
R←⎕BOUNCE TASKIDS
```

A task may terminate any task with the same sign-on number, or any task initiated by his sign-on number.

The right argument is a vector or scalar of the TASKIDS of tasks to be terminated. The result is a vector or scalar of 1's and 0's corresponding to TASKIDS. A 1 indicates a BOUNCE was issued and 0 indicates it was not. If a BOUNCE was issued the corresponding task will be forced off by the system as soon as possible.

TERMINATION OF N-TASKS AND B-TASKS

When an N-task or B-task terminates it is saved (unless the workspace is empty as a result of clearout) in the *CONTWSID* specified in the *⎕RUN* argument. An N-task or B-task can terminate for any of several reasons, among which are: terminal input is requested, *CPULIM* is exceeded, *CONLIM* is exceeded, an error is encountered, scheduled shutdown, and unscheduled shutdown. An N-task or B-task will also be terminated by a system crash in which case the recovered workspace (unless empty) is saved in *CONTWSID*. B-tasks are automatically restarted (by loading *CONTWSID* and resuming with *⎕LX*) after a shutdown or crash, if the request specifications so specified. N-tasks are not automatically restarted after a shutdown or crash.

If an N-task or B-task terminates due to an error (e.g. syntax error or file full) the *CONTWSID* can be loaded by a T-task and the fault can be corrected in the interactive T-task environment.

ACCOUNTING INFORMATION

The to-date fields in *⎕AI* and in the text printed at sign off report **only** resources used by T-tasks. To avoid possible misinterpretation, *⎕AI* when used by an N-task or B-task reports 0 in the cumulative fields for cpu and connect. The session fields are of course correct for the task executing *⎕AI*. Users can find out their cumulative N-task and B-task resource usage through the usage inquiry system (SATN-9).

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

SATN 5
1 FEB 80
Rev. 3

SHARP APL TECHNICAL NOTES

TITLE: BATCH APL

ABSTRACT: System facilities are presented that allow delayed execution of APL without a terminal, in a batch scheduling environment.

KEYWORDS: Workspace 1 *BTASKREQ*
]*RUN*
]*RUNS*
]*BOUNCE*
Tasks
T-tasks
N-tasks
B-tasks
Processing specs
Scheduling
Restartability

SHARP AR TECHNICAL NOTES

SHARP APL B-TASKS

Tasking facilities in SHARP APL were described in detail in SATN-4 with the introduction of N-tasks (No-terminal tasks created by the user via the ⎕*RUN* function). B-tasks (Batch APL tasks) are similar to N-tasks, and share many of their applications. However, the two have the following important differences:

- A B-task is created by the system B-task scheduler from a request submitted by the user. Therefore, the user's request does not actually initiate the task.
- B-tasks can be automatically restarted after a system failure or shutdown.
- B-task execution costs are less than those of N-tasks, as the tasks tend to run during off-peak times. This makes B-tasks especially attractive for non-interactive jobs not requiring immediate turnaround.

HOW B-TASKS ARE SCHEDULED

B-tasks are automatically scheduled by the B-task scheduler according to the system resources available. For this reason, they will normally run during periods of light T-task and N-task load. As tasks are scheduled completely automatically with no manual intervention, SHARP APL Operations has no control over when or how fast a task will be executed, nor can they predict when a B-task will be scheduled. By means of the processing specifications in the B-task request, however, the user can specify some additional conditions to be used in scheduling the task. These will be discussed in more detail in subsequent sections of this SATN.

CREATING A B-TASK

While a ⎕*RUN* is used to initiate an N-task, a B-task is requested via request functions in workspace `1 BTASKREQ`. The functions *BTASKREQ* and *AUTOBREQ* provide both an interactive and non-interactive means of submitting a request.

I. Interactive request submission

The *BTASKREQ* program may be used to interactively submit a request by prompting the user for the required information. Its syntax is:

```
Z←BTASKREQ
```

where the result `Z` is a character vector containing the request number, and the time and date it was filed. The request number is important, as it permits the user to refer to the task by means of the utility functions also present in workspace `1 BTASKREQ`.

BTASKREQ first prompts the user with the message *PROCESSING SPECS*: . Processing specifications are special codes which are used in controlling scheduling, priority, task restarting, and so forth, and should be entered separated by either commas or blanks. As an example, a user might wish to designate that his B-task be run only after a certain time of day, or not until the following morning. This can be accomplished by entering a special processing specification designed to perform the desired effect.

Special codes may be implemented according to the needs of your application, by prior arrangement with your marketing representative and the SHARP APL Computer Centre. If your application could benefit by their use, please contact your local SHARP APL representative for more information.

Processing specifications of general use are described in detail in the next section of this SATN. If no specifications are desired in your request, simply press return at the prompt
PROCESSING SPECS: .

The next prompt requests the input of the ⎕*RUN* parameters to be used in initiating the B-task. The parameters are currently *SIGNON LOADWSID CONTWSID CPULIM CONLIM*, and should be entered with each field separated by one or more blanks. These parameters are identical to those used in a ⎕*RUN* for an N-task, and are explained in detail in SATN-4. The only exception is that the load workspace for a B-task must be named explicitly, and may not be elided as in an N-task.

If you are unsure of the parameters, or the order in which they should be entered, simply press carriage return at the prompt ⎕*RUN PARAMETERS*: . The function will then prompt you separately for each of the fields required.

There is a limit to the number of B-task requests that a user may submit. A user may not have more than a total of twenty-one requests either pending scheduling, or actually running. (Note that this is independent of the number of processed or withdrawn requests.) If the quota has not been exhausted, then the B-task request is submitted, and will be scheduled by the system scheduler when appropriate. Note that the *CPULIM* and *CONLIM* fields are used by the scheduler to control the task, and should be **realistically** estimated. A limit of 0 means no limit and should be used judiciously.

I. P. Sharp Associates reserves the right to enhance the tasking facilities available by the addition of task parameters to the argument of ⎕*RUN* and to the B-task request functions.

II. Non-interactive request submission

The *AUTOBREQ* function permits the submission of a B-task request without any user intervention. It is useful for submitting requests under program control, especially from N-tasks and other B-tasks. The syntax of the function is:

```
Z←AUTOBREQ R
```

where *R* is a character vector containing the request information. The result *Z* is a numeric scalar containing the request number, or 0 if the request was not successful.

The argument *R* designates the processing specifications to be used, separated by either commas or blanks, followed by the null symbol, ∘, and the ⎕*RUN* parameters in correct sequence. If no processing specifications are required, the request should begin with the ⎕*RUN* parameters, optionally preceded by a ∘.

If the function result is 0 , the request was invalid and was not submitted. The reasons for this include:

- The argument was not a non-empty character vector.
- An invalid processing specification was given.
- The user has exhausted his quota of queued or active B-task requests (currently 21).

III. Examples of use

Some examples of using *BTASKREQ* and *AUTOBREQ* are given below. The requests submitted in each case would be identical. For illustrative purposes, the request number is the same in all examples.

```
      Z←BTASKREQ
PROCESSING SPECS: NORMRST
⎕RUN PARAMETERS: carriage return
SIGNON: 1234567:SECRET
LOADWSID: 7654321 NEW
CONTINUE WSID: END
CPU LIMIT: 600
CONNECT LIMIT: 6000
      Z
BTASK REQ. NO. 23547 FILED 11.45.38  11 AUG 1978

      BTASKREQ
PROCESSING SPECS: NORMRST
⎕RUN PARAMETERS: 1234567:SECRET 7654321 NEW END 600 6000
BTASK REQ. NO. 23547 FILED 11.45.38  11 AUG 1978

      Z←AUTOBREQ 'NORMRST∘1234567:SECRET 7654321 NEW END 600 6000'
      Z
23547
```

PROCESSING SPECIFICATIONS

Certain commonly-required processing specifications are available to all users. These may be entered in response to the prompt *PROCESSING SPECS*: in *BTASKREQ*, or before a ∘ symbol in the argument to *AUTOBREQ*. Each valid specification is outlined below, followed by a description of its use:

1. *NORMRST*

Signifies that the B-task is restartable after a normal system shutdown. See the next section on task restartability for more information.

2. *ABNORMRST*

Signifies that the B-task is restartable after a system crash. See the next section on task restartability for more information.

3. *COPYSP*

Signifies that the value of □*SP* in the requestor's workspace is to be propagated into the B-task workspace on initial startup of the B-task. The default value of □*SP* (currently '' - an empty character vector) is propagated into the B-task workspace on all restarts after system shutdown or a crash. The default value of □*SP* is also propagated into the B-task workspace on initial startup of the task if *COPYSP* is not specified.

4. *NBTIME*(hh:mm)
NBTIME(hh:mm hh:mm)

Signifies that the B-task should not be scheduled before the time given in parentheses. The time should be specified in hours and minutes. A second time designation is optional, and, if present, is taken to be an upper bound on the time interval whose lower bound is given by the first specification. A single specification is treated as if a second parameter of 24:00 had been provided. For example, *NBTIME*(17:30) may be used to ensure that the B-task is not scheduled before 17:30. *NBTIME*(03:00 08:00) ensures that the B-task will be scheduled only between the hours of 03:00 and 08:00. The time boundaries defining the scheduling window need not refer to the same day (but must be within 24 hours). For example, *NBTIME*(21:00 25:30) or *NBTIME*(21:00 1:30) specifies that the task should be scheduled only between the hours of 21:00 and 1:30 the following morning. Leading zeroes within time fields may be omitted, if desired, and periods or blanks may be used in place of colons.

5. *NBDATE*(mm/dd/yy)

Signifies that the B-task should not be scheduled before the date given in parentheses. The date should be specified as month, followed by day and year. For example, *NBDATE*(8/11/78) may be used to ensure that the B-task is not scheduled before 11 August 1978. Leading zeroes within date fields may be omitted, if desired. Also, blanks may be used in place of slashes to separate the fields.

6. *NBTS*(yyyy/mm/dd hh:mm)

Signifies that the B-task should not be scheduled before the timestamp given in parentheses. The timestamp should be specified in the same format as that used by the system function □*TS*. For example, *NBTS*(1978/8/11 17:30) may be used to ensure that the B-task is not scheduled before 17:30 on 11 August 1978. The century information is optional, and leading zeroes within the date and time fields may be omitted. Also, blanks may be used in place of slashes or colons to separate the fields.

7. *NBBTASK*(req.no.)

Signifies that the B-task should not be scheduled before the B-task whose request number is given in parentheses has been successfully started and has completed. For example, *NBBTASK*(1080) may be used to ensure scheduling after B-task request 1080 has terminated, with a successful □*RUN* initiation code as indicated by the *INQ* function. The B-task requests need not have been submitted by the same user number. If you would like a certain B-task to be scheduled only after another B-task has successfully **completed**, then have the first B-task submit a request for the second, using the *AUTOBREQ* program, before its normal termination.

8. *NBSORT*(req.no.)

Signifies that the B-task should not be scheduled before the file sort whose request number is given in parentheses has been successfully processed. For example, *NBSORT*(1729) may be used to ensure scheduling after file sort request 1729 has completed normally. The file sort request need not have been submitted by the same user number submitting the B-task request.

9. *NBOR*(message text)

Signifies that the B-task request should not be scheduled before manual operator release, conditional on the message text given in parentheses. This permits a B-task to be initiated pending any event, including the completion of a batch job or even a telephone call. An example of the use of the *NBOR* processing specification might be:
NBOR(*PLEASE ENSURE THAT THE ABC COBOL FILE FEED HAS COMPLETED FIRST*).

Any of these processing specifications can be used in combination if desired. If more than one specification is given, then the B-task will not be scheduled until **all** of the conditions imposed by the specifications are simultaneously satisfied. For example, scheduling after 17:30 **and** after sort request 1729 has successfully completed, can be achieved by using both *NBTIME* and *NBSORT* in the following manner: *NBTIME*(17:30),*NBSORT*(1729).

Because multiple scheduling conditions must be simultaneously satisfied, using the processing specifications *NBTIME* and *NBDATE* is not the same as using the *NBTS* specification with an equivalent timestamp. In the first case, the task may be scheduled any day on or after the specified one, only if it is **also** past the specified time on that day. Hence, if the B-task is not scheduled on the day specified, the earliest subsequent time that it may be scheduled is after the given time on the following day. With a single *NBTS* specification, the task may be scheduled any time on or after the designated timestamp. If the task cannot be scheduled on the day specified, it will become available for scheduling immediately the following day.

If an application requires a processing specification not listed here, please contact your local SHARP APL representative.

TASK RESTARTABILITY

The SHARP APL environment provides extremely effective and time-saving recovery mechanisms for the file system, and for APL workspaces running under T-tasks, N-tasks, or B-tasks. For all types of tasks, in the case of a normal system shutdown, integrity is guaranteed between the active files and workspace at the time of the shutdown. This means that for most applications, restart should be as simple as re-tying any previously tied files and then resuming at the point of interruption. If the interruption occurred in a critical program section, it may be necessary to re-establish any file holds that might have been in effect, and then resume at the start of the sensitive section rather than at the point of suspension.

Dealing with the case of a system crash, which occurs very infrequently but without warning, is somewhat more complicated. APL workspaces are updated at least every four seconds, and so the recovered workspace can be out of date by anywhere from zero to at most that amount. The file system, however, is recovered and updated by a more complex procedure. Hence some activity in the workspace, such as internal program counters, might not be reflected in the recovered workspace, while the file system might reflect this processing in the form of updated data in a file. (See SATN-16, "File Subsystem Must-write Buffers", for more information.)

Careful APL coding techniques, including the writing of fully restartable APL statements, can effectively overcome the obstacles presented by system failures. Programmers who make use of ⎕*LX* to provide workspace autostarting should note that ⎕*LX* can cause the restarting of *CONTINUE* workspaces saved due to a system shutdown or crash. This is sometimes dangerous if the package is not restartable, and can result in the inadvertent re-execution of a possibly damaging program. For this reason, it is often advisable for non-restartable packages to employ latent expressions which destroy themselves. For example, the latent expression:

```
⎕LX←'⎕LX←'''' ◇ MAIN∆PROGRAM'
```

will function correctly the first time through, but should the workspace be saved by a system crash and then reloaded, an empty ⎕*LX* will be encountered and the *MAIN∆PROGRAM* routine not re-executed.

The technique of altering the ⎕*LX* of a workspace at load time is valuable for all three types of SHARP APL tasks. In the above case, if some sort of restart other than *MAIN∆PROGRAM* had been desired, the ⎕*LX* could have been set appropriately. For many well-programmed applications, it is sufficient to restart using:

```
⎕LX←'TIE∆FILES ◇ →1↓⎕LC'
```

which would call *TIE∆FILES* to re-tie any files that might have been in use at the time of the interruption, and then resume at the suspended line.

With the submission of a B-task request, users may specify that their task is restartable under any combination of the following conditions:

- The B-task is restartable after a normal system shutdown. This implies synchronized file system and workspace states.
- The B-task is restartable after a system crash. This implies close but not guaranteed file system and workspace states.
- The B-task is not restartable under any conditions.

When submitting a B-task request, no restart is the default and is taken to be the desired state unless the user specifies otherwise. Restart after a normal system shutdown can be accomplished by entering the processing specification *NORMRST* when prompted; if restart after a system crash is desired, specify *ABNORMRST*. Both types of restart can be obtained by entering *NORMRST*,*ABNORMRST*. Note that if a task is not restartable after a system shutdown (that is, *NORMRST* is not specified in the request), then it will not be scheduled unless enough time remains in the operating day. If this is the case, the task will remain in the scheduling queue until the following day, when the system will again try to schedule it.

Restartable B-tasks should employ the techniques of altering the □*LX* of the load workspace, as described above. When recovered, the B-task's active workspace will be saved in the continue workspace designated in the B-task request. The B-task scheduler will then reschedule and correctly restart the task, executing the desired altered latent expression. Thus, unlike N-tasks, B-tasks running during system interruption are provided with complete restart facilities. This considerably enhances the application range of batch tasks.

SUPPLEMENTARY INQUIRY FUNCTIONS

In addition to the *BTASKREQ* and *AUTOBREQ* programs, a number of user inquiry functions are provided in workspace `1 BTASKREQ`. These permit the submittor of a B-task request to perform a variety of operations, including withdrawing a request, or monitoring its execution and completion status. All functions are origin independent.

Z←ACTIVE

It returns as its explicit result the request numbers of B-task requests that were submitted by you (or for your account number by another user) and are currently running.

DISPLAY N

The argument is a numeric scalar or vector of request numbers submitted by you or for you. The *DISPLAY* program prints the contents of each request, including its processing specifications, □*RUN* parameters, and current status. The "status" indication may be one of the following:

QUEUED	Request is currently in the scheduler queue, pending scheduling.
ACTIVE	Request is currently running.
LAPSED	Request has been processed (successfully or unsuccessfully).
ERROR	Scheduling status error. Please report this to the SHARP APL Computer Centre immediately.

If *COPYSP* was given as one of the processing specifications and the request is still queued, then the value of □*SP* to be passed to the B-task when it is scheduled, is also displayed.

As a protection, any account number or workspace password in the request is blanked with underscores (_) in the output of *DISPLAY*.

Z←INQ N

The argument is a scalar or vector of request numbers that were submitted by you or for you. The result *Z* is a four-column matrix with as many rows as there are elements in *N*. Each row contains the request number, request status, and two additional elements of information. The request status integers have the following meaning:

0	B-task request not on file.
1	Request is queued.
2	Request is active.
3	Request is lapsed.
4	Reserved.
5	Scheduling status error. This should be reported to the SHARP APL Computer Centre immediately.

If the request is active (status code 2), then column three of the appropriate row will contain the TASKID of the running task. If the request is lapsed (status code 3), then the last two columns indicate if the task started successfully. If both columns are zero, the initiating *□RUN* was successful; however, this does not guarantee that the task **terminated** successfully. (It may have terminated due to an error encountered during its execution, or due to the exceeding of its cpu or connect limits. The continue workspace of the task can be examined as a convenient means of determining successful task completion.)

If the entries in the last two columns for a lapsed request are non-zero, then they represent a two-element error code similar to the result of an improper *□RUN*. In addition to the *NTASK □RUN* errors outlined in **'The SHARP APL Reference Manual'**, there are a number of other possible error conditions for B-tasks:

58	0	The B-task could not be scheduled, as the *□SP* specified through *COPYSP* was too large.
59	0	A system crash occurred while the B-task was active, and the task could not be restarted as no workspace was saved by the Crash Recovery program.
60	0	The B-task could not be scheduled, as the B-task request specified using the *NBBTASK* processing specification was not successfully initiated.
61	0	The B-task could not be scheduled, as the sort request specified using the *NBSORT* processing specification did not complete successfully.
62	0	A system crash occurred while the B-task was active, and the task was not restartable because *ABNORMRST* was not specified in the original request.
63	0	The B-task could not be scheduled, as an error occurred in the processing specifications designated in the request.

Except as noted, the last two columns of the result matrix are zero and have no representative meaning.

Z←LAPSED

It returns as its explicit result the request numbers of B-task requests submitted by you or for you that have completed (successfully or unsuccessfully), but are still on file. Lapsed tasks remain on file for inquiry by their submittor for a period of at least three days after they were scheduled.

Z←QUEUED

It returns as its explicit result the request numbers of B-task requests that were submitted by you or for you and are awaiting scheduling by the B-task scheduler. Once scheduled, the tasks become *ACTIVE* and eventually *LAPSED*.

Z←REQUESTS

It returns as its explicit result the request numbers of all B-task requests submitted by you or for you, whether *QUEUED*, *ACTIVE*, or *LAPSED*. Hence executing the statement *DISPLAY REQUESTS* will produce a complete listing of all B-task requests submitted by you or for you that are currently on file.

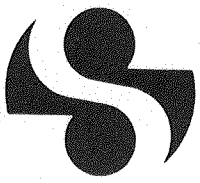
Z←WITHDRAW N

The argument is a scalar or vector of request numbers submitted by you or for you. The *WITHDRAW* program retracts each *QUEUED* request in its argument and changes its state to *LAPSED*, thus bypassing the scheduling of the tasks. The result *Z* is a Boolean vector with as many elements as there are in *N*. Each 1 in *Z* indicates the successful retraction of the corresponding request in *N*.

In addition to the above facilities, the system functions provided for N-tasks, ⎕*BOUNCE* and ⎕*RUNS*, apply equally well to B-tasks. As *WITHDRAW* will operate only on a task that is queued, the program will not remove one once it has been scheduled. If it is desired to terminate an active task, ⎕*BOUNCE* may be used. Its argument of TASKID's can be determined from the *INQ* function, which provides a link between a B-task request number and its corresponding TASKID if the request is active. Alternately, the TASKID's can be obtained from the result of ⎕*RUNS*, which returns an integer matrix containing information about tasks that have the same sign-on number, or were initiated by the same sign-on number, as the task executing the ⎕*RUNS*.

For more details, see the manual '**Batch Tasks in SHARP APL**' or the '**SHARP APL Reference Manual**'.

SATN-6
1 JAN 76



I. P. Sharp Associates Limited Suite 1400, 145 King Street West, Toronto, Ontario M5H 1J8 • (416) 364-5361

SHARP APL TECHNICAL NOTES

TITLE: *EXECUTE*

ABSTRACT: *THE PRIMITIVE FUNCTION ⍎ IS DEFINED. EXECUTE TAKES A CHARACTER AGUMENT AND EXECUTES IT AS AN APL EXPRESSION.*

KEYWORDS: 3 □*FD*
)*SI*
HYDRANT
DIAMOND

EXECUTE IS A MONADIC PRIMITIVE FUNCTION WHICH TAKES A SCALAR OR VECTOR CHARACTER ARGUMENT. ITS RESULT IS THE RESULT OF THE EXECUTION OF ITS ARGUMENT. FOR EXAMPLE, THE RESULT OF ⍎'1+1' *IS* 2. *THE GRAPHIC* ⍎ *IS PRONOUNCED HYDRANT.*

EXECUTE OF A DIAMONDIZED LINE IS PERMITTED. THE VALUE OF ⍎'A◇B⍝C' *IS B AND MAY OR MAY NOT BE DEFINED DEPENDING ON B (WHICH MIGHT E.G. BE A NILADIC FUNCTION NOT RETURNING A VALUE). THE VALUE OF* ⍎'A^B◇' *IS ALWAYS UNDEFINED.*

THE VALUE OF ⍎'A←B' *IS B BUT DOES NOT PRINT. SIMILARLY,* ⍎'→5' *DOES NOT PRINT.*

BRANCHING FROM WITHIN ⍎ *IS PERMITTED. THERE ARE THREE CASES OF BRANCHING WITHIN EXECUTE:*

1. →0 *HAS THE EFFECT OF EXITING FROM THE EXECUTED LINE AND CONTINUING ON TO THE NEXT EXPRESSION. NOTE THAT THIS IS PARTICULARLY USEFUL IN EXECUTING DIAMONDIZED STATEMENTS THAT EXIT EARLY.*

2. →*L (WHERE L IS A VALID LINE IN THE FUNCTION DOWN THE SI STACK FROM THE EXECUTE) HAS THE EFFECT OF EXITING FROM THE EXECUTED LINE AND STARTING EXECUTION ON LINE L OF THE FUNCTION. NOTE THAT →L COULD CAUSE THE EXIT FROM SEVERAL LEVELS OF EXECUTE (E.G.* FOO[1] ⍎'⍎''→L'''). *ALSO NOTE THAT AS EACH LEVEL IS EXITED IT IS CHECKED THAT THE* ⍎ *WAS THE LEFTMOST FUNCTION IN THE STATEMENT. FOR EXAMPLE* FOO[1] ⍎'2+⍎''→L''' , *WOULD RESULT IN A SYNTAX ERROR.*

3. →*N (WHERE N IS OUT OF RANGE FOR THE FUNCTION DOWN THE SI STACK FROM THE EXECUTE) HAS THE EFFECT OF EXITING FROM THE FUNCTION. AS EACH LEVEL IS EXITED IT IS CHECKED THAT THE → IS THE LEFTMOST FUNCTION. FOR EXAMPLE* FOO[1] ⍎'2+⍎''→9999''' , *WOULD RESULT IN A SYNTAX ERROR.*

NAKED → IS TREATED IDENTICALLY WITHIN ⍎ *AS ELSEWHERE BUT BEWARE THAT IN IMMEDIATE EXECUTION → IS NOT EQUIVALENT TO* ⍎'→' *FOR THE SAME REASON THAT → IS NOT EQUIVALENT TO TYPING* ⎕ *AND REPLYING → OR EXECUTING A FUNCTION WHICH CONTAINS →.*

LISTS ARE PERMITTED AS THE RESULT OF EXECUTE. THUS ⍎'A;B;C' *IS PERMITTED. THE RESULT IS A LIST, AND SUBJECT TO THE NORMAL RESTRICTIONS ON LISTS. FOR EXAMPLE* 'I5'⎕FMT⍎'1;2;3' *IS EQUIVALENT TO* 'I5'⎕FMT(1;2;3)

EXECUTE IS SUBJECT TO THE SAME RESTRICTIONS AS A SINGLE LINE OF MATRIX INPUT TO 3 ⎕FD, *EXCEPT THAT TOTALLY BLANK LINES ARE PERMITTED. A WS FULL ERROR CAN OCCUR IF THERE IS NOT SUFFICIENT ROOM TO BUILD THE EXPRESSION. A SYMBOL TABLE FULL ERROR CAN OCCUR IF THE SYMBOL TABLE BECOMES FULL BUILDING THE EXPRESSION. STATEMENTS MAY BE LABELLED BUT LABELS ARE IGNORED.*

IF AN ERROR IS DETECTED DURING AN EXECUTED STATEMENT A MULTIPLE ERROR REPORT OCCURS SHOWING EACH STACKED ⍎ AS WELL AS THE ORIGINAL EXPRESSION. THUS ⍎' ⍎''1÷0'' ' PRINTS AS

```
DOMAIN ERROR
      1÷0
       ∧
EXECUTE ERROR
      ⍎'1÷0'
      ∧
EXECUTE ERROR
      ⍎' ⍎''1÷0'' '
      ∧
```

IF THE STACK CONTAINS PENDING ⍎ LINES THESE SHOW UP IN)SI AS ⍎ MUCH THE WAY ⎕ DOES. SIMILARLY 2 ⎕WS 2 MAY CONTAIN ⍎ ON SOME ROWS IN THE FIRST COLUMN.

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

SATN 7
1 JAN 76

SHARP APL TECHNICAL NOTES

TITLE: Latent Expression

ABSTRACT: The system variable $\square LX$ is described. $\square LX$ is a replacement for and a considerable extension to the autostart facility. $\square LX$ provides an expression to be executed when a workspace is loaded.

KEYWORDS: Quad variable
)*LOAD*
)*SAVE*;
Autostart
Execute

SHARP AR TECHNICAL NOTES

An early innovation of SHARP APL was the ability to save a workspace with a statement to be executed when the workspace was loaded.

The mechanism was to place the APL expression to be executed after the *)SAVE* command, separated from it by ;

For example:

```
)SAVE MYWS; TIEFILES ◇ PROCESS
```

The ability to provide this argument to *)SAVE* is discontinued and a new method for obtaining the same effect is provided. Furthermore, the autostart of old workspaces saved in this manner will not be honoured.

The new procedure will be to store in system variable *⎕LX* (latent expression) the desired APL expression.

Thus the example above would now be achieved by:

```
⎕LX←'TIEFILES ◇ PROCESS'
)SAVE MYWS
```

This new approach has two major advantages:

1. It allows programmatic modification of the autostart expression.

2. It is possible to autostart workspaces saved by line drops, walkaway, *⎕BOUNCE*, *⎕RUN* termination, *)CONTINUE*, as well as *)SAVE*.

Like all ⎕ variables, *⎕LX* may be used just like any other variable. It may be undefined or localized or may contain numeric or character data of any rank. If *⎕LX* is empty or undefined no action is taken when the workspace is loaded. If *⎕LX* is defined the system acts as if the user had entered *⍎⎕LX*. An unusual condition occurs when the workspace was saved in ⎕ or ⍞ mode. Instead of executing *⎕LX* on loading the workspace, the ⎕ or ⍞ on top of the stack is converted to an execute of

```
'⍎⎕LX ◇ ⎕' or          '⍎⎕LX ◇ ⍞'
```

This allows the restart procedure invoked by *⎕LX* to finish and then accept input from the user for the original ⎕ or ⍞ if desired. Authors of packages who are attempting complete restartability should consider this case carefully. A *⎕LX* which handles the case, as well as the more familiar one of a function suspended between lines, is

```
⎕LX←'TIEFILES ◇ →1↓⎕LC'
```

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

SATN 8
1 MAR 79
Rev. 2

SHARP APL TECHNICAL NOTES

TITLE: *HSPRINT*

ABSTRACT: *HSPRINT* is a BG program that prints APL files on a high speed printer at the Toronto data centre or a remote location. If desired the output can be transferred to a magnetic tape or an APL file.

KEYWORDS: Workspace 1 *HSPRINT*
□*OUT*
Control Messages
High speed printer
Remote printer
Tape
HSPRINT to file

INTRODUCTION

Since SHARP APL was first offered as a timesharing service there has been a need for a highspeed print facility. In the past this was filled by *FILEPRINT*.

With the advent of ⎕*OUT* (SATN-3) and N-task/B-task capability (SATN-4) an economical and convenient method of printing the resultant files was needed. *HSPRINT* answers this need. It incorporates and extends many of the *FILEPRINT* features as well as offering new capabilities. No further development will be done on *FILEPRINT* although it will continue to be supported at its present level.

Some of the features available in *HSPRINT* are:

1) Multiple files in a single request
2) Explicit error messages and error detection
3) Private forms capability
4) "Infinite" depth pages
5) ⎕*FMT* control message relieves the APL task of the cpu required to do the formatting
6) Request queue manipulation and interrogation
7) "Remote" printer facility
8) Output to tape or APL file

HSPRINT INPUT FILES

An *HSPRINT* input file component is either **data** or a **control message**. A data component is any APL object that can be put to file and is not a control message. Control messages are more fully documented in SATN-2 but are reviewed here as being a 22-element character vector with the first element as ¯1↑⎕*AV* followed by a message number, a component count and a mnemonic.

For example, suppose you do the following:

```
      'SCRATCH' ⎕CREATE 86
      (X←'THIS IS TEST DATA') ⎕APPEND 86   ⍝ CHARACTER DATA
      (Y←⍳5) ⎕APPEND 86                    ⍝ NUMERIC DATA
      A←⎕OUT 1 86                          ⍝ PRINT ON TERMINAL AND OUTPUT TO FILE
      FS←'⎕  :  :     /  /⎕,T1,6(ZI2,X1)'  ⍝ FORMAT STRING
      FS ⎕FMT ⎕TS[1 6⍴ 4 5 6 3 2 1]        ⍝ PRINT TIMESTAMP
08:09:55 21/06/76
      A←⎕OUT 0 86                          ⍝ SUPPRESS TERMINAL PRINTING
      X;Y;'END OF DATA'                    ⍝ HETEROGENEOUS OUTPUT TO FILE
      A←⎕OUT 1 0                           ⍝ PRINT ON TERMINAL NO OUTPUT TO FILE
      ⎕SIZE 86
1  10  6312  50496
```

Note that the resultant file has 9 components.

Let's examine the contents of this file:

```
      ⎕READ 86 1                    ⍝ CHARACTER DATA
THIS IS TEST DATA
      ⎕READ 86 2                    ⍝ NUMERIC DATA
1  2  3  4  5
      ⎕READ 86 3                    ⍝ CONTROL MESSAGE
⍬0000 0002 FORMAT
      ⎕READ 86 4                    ⍝ LEFT ARGUMENT TO ⎕FMT
⎕  :  :     /  /⎕,T1,6(ZI2,X1)
      ⎕READ 86 5                    ⍝ RIGHT ARGUMENT TO ⎕FMT
   8   9  55  21   6  76
      ⎕READ 86 6                    ⍝ CONTROL MESSAGE
⍬0001 0003 MIXED
      ⎕READ 86 7                    ⍝ FIRST LIST ELEMENT
THIS IS TEST DATA
      ⎕READ 86 8                    ⍝ SECOND LIST ELEMENT
1  2  3  4  5
      ⎕READ 86 9                    ⍝ THIRD LIST ELEMENT
END OF DATA
```

The output from the *HSPRINT* program would look like this:

```
      1142021 DATE 06/21/76,CLOCK 8/41/45      PAGE 1

THIS IS TEST DATA
1  2  3  4  5
08:09:55 21/06/76
THIS IS TEST DATA1  2  3  4  5END OF DATA
```

Note that *HSPRINT* has properly displayed character data, numeric data, and has performed the ⎕*FMT* and heterogenous output. *HSPRINT* also inserted a default title consisting of the user number who submitted the request, the timestamp when the request was processed, and has automatically numbered the pages.

HSPRINT CONTROL MESSAGES

You can control many aspects of the printed result by using the Control Message programs in workspace 1 *HSPRINT*. All programs in this workspace are unlocked. Control message functions have the form:

data **program name** filenumber (optional magic number)

For example:

```
      ∇WIDTH[⎕]∇
    ∇ X WIDTH Y
[1]   (22↑⎕AV[255+⎕IO],'0006 0001 WIDTH') ⎕APPEND Y
[2]   X ⎕APPEND Y
    ∇
```

Following is a list of all the *HSPRINT* Control Message Programs together with a description of their use.

CTL MSG NO: 0005 *X DIGITS Y*

Purpose: Allows setting of ⎕*PP* at print time.
Constraints: *X* must be in same domain as for ⎕*PP*.
Example: 16 *DIGITS* 86
Default: 10

CTL MSG NO: 0006 *X WIDTH Y*

Purpose: Allows setting of ⎕*PW* at print time.
Constraints: *X* must be in same domain as for ⎕*PW*.
Example: 132 *WIDTH* 86
Default: 132

CTL MSG NO: 0007 *X TITLE Y*
0008 *X SUBTITLE Y*

Purpose: Establishes a new title or subtitle, or controls printing of title or subtitle. A value of 0 means suppress printing.
Constraints: *X* must be a character array or a non floating point number.
Examples:
```
'NEW TITLE' TITLE 86          ⍝ SET UP NEW TITLE
'NEWER SUBTITLE' SUBTITLE 86  ⍝ AND SUB TITLE
1 TITLE 86                    ⍝ PRINT TITLE
0 SUBTITLE 86                 ⍝ BUT DO NOT PRINT SUBTITLE
```

Defaults: *TITLE*: (user number) (timestamp)
SUBTITLE: blank
TITLE: 1 (print title)
SUBTITLE: 0 (do not print subtitle) ⍝ *NOTE THIS DEFAULT*

CTL MSG NO: 0012 *X CARRIAGE Y*

Purpose: Establishes a new forms control vector, with printed output only where there are 1's in the vector. Forces a skip to top of form.
Constraints: *X* must be a numeric vector or scalar. If a scalar, it must be 1, and this has the meaning "infinite" depth form. If a vector the only legal values are 0 or 1 and only the first 250 elements are used. There must be at least one writable line (i.e. at least one 1).
Example: (60⍴1 0) *CARRIAGE* 86 ⍝ *DOUBLE SPACED*
Default: 1 0,58⍴1

CTL MSG NO: 0002 *X PAGEN Y*

Purpose: Controls automatic page numbering and printing.

Constraints: *X* must be a numeric non floating point array. The first element controls printing of page numbers where a 0 means suppress printing. The second element may be elided, but if present establishes a new page number on the next page. Page numbers are maintained even if printing is suppressed and they must be non-negative. They are printed on the first line of a title and are located on the far righthand side of the page, (i.e. at ⎕*PW* - 9 for 9 positions). Note that page numbers will overwrite the title in those positions.

Example:
```
1 10 PAGEN 86      ⍝ PRINT PAGE NUMBERS AND RESET IT TO START AT 10
0 PAGEN 86         ⍝ DO NOT PRINT PAGE NUMBERS
```

Default: 1 1

CTL MSG NO: 0011 *PAGE Y*

Purpose: Forces a skip to top of form for next printed line. Only the first one of multiple *PAGE* messages with no intervening output is effective.

Constraints: Ignored if "infinite" form is in use.

Example: *PAGE* 86

Default: n/a

CTL MSG NO: 0016 *X PRTARBIN Y*
0013 *X PRTARBOUT Y*
0010 *X PRTUCTM Y*

Purpose: Controls printing of ⎕*ARBOUT*, ⎕*ARBIN* or user control message data.

Constraints: *X* must be a non floating point 0 or 1 where 0 means suppress printing. A 1 causes the Control Message to be printed followed by the data.

Example: 1 *PRTARBOUT* 86

Default: 0

Note: User control messages are those established by the user for their own convenience. The message numbers must be 9000 or greater and *HSPRINT* handles them in a similar way to the handling of the *ARBIN* and *ARBOUT* control messages.

CTL MSG NO: 0004 *X TRANSLATE Y*

Purpose: Establishes a new translate table. Three standard translate tables are maintained: *FAST*, *FULL* and *PN2*. Specifying one of these when you submit your request is sufficient to cause its use. These tables and some functions to manipulate them can be found in workspace 1 *HSPRINT* as the group *TRANS*.

Constraints: *X* is either
1) a character vector naming one of the pre-defined tables or
2) a 128 element integer vector of your own table.

Example:
```
'FAST' TRANSLATE 86
'FULL' TRANSLATE 86
'PN2'  TRANSLATE 86
```

Default: *'FAST'*

Warning: A full understanding of the vector option is recommended before using that feature. Refer to Appendix B of this SATN for further details.

The remaining control messages, 0, 1, 3, 14 and 15 are produced by the system when a □*OUT* file is present and output is to be appended to the file.

To continue our example with the file *SCRATCH* tied to 86, suppose you wanted to suppress title printing but allow automatic page numbering beginning with page number 10. One way to do this is as follows:

```
'CONTROL' □CREATE 55      ⍝ CREATE FILE
0 TITLE 55                ⍝ SUPPRESS TITLES
1 10 PAGEN 55             ⍝ PRINT PAGE NUMBERS
```

Now when you submit your request you would include both file numbers. *HSPRINT* will process the files in the submitted order. Of course the control information need not be in a separate file.

SUBMITTING *HSPRINT* REQUESTS

Workspace 1 *HSPRINT* contains a program for submitting *HSPRINT* requests. Users may copy and save this program, *HSP*, into their own workspace. The syntax of the program is:

```
Z←A HSP B
```

The program can operate in 3 different modes:
1) Interactive request submittor
2) Automatic request submittor
3) Request validator.

The Result

A successful submission results in a character vector of the form:

```
'REQ. NO 123456 FILED 1456180 11.27.58 28 JAN 1976.'
```

Otherwise the result will be an error code, *N*, that can be used as an origin 1 index of the variable *HSPERRORS*, in workspace 1 *HSPRINT*, and *HSPERRORS*[*N*-~□*IO*;] is also displayed.

The Left Argument

This is used to pass the file tie numbers for the files to be submitted. It also controls the mode of operation as follows:

First Element	Mode	Example
< 0	validation	`¯6 55 86 HSP 'ERASE,TIED∘PHT WILL PICKUP'`
= 0	interactive	`0 55 86 HSP ''`
> 0	automatic	`55 86 HSP 'LHG'`

The left argument must be a numeric scalar, vector, or matrix.

Regardless of its original shape the argument is internally made into a 4-column matrix with the columns having the following meaning:

column 1	Column 2	Column 3	Column 4
File tie numbers of files to be submitted.	Magic numbers for file tie numbers in col 1.	If zero set BG access. If not do not.	Magic number for BG to use.

Examples

Left Argument	Internal Form
scalar `86`	`86 0 0 0`
vector `55 86`	`55 0 0 0` `86 0 0 0`
matrix `86 1234`	`86 1234 0 0`
matrix `86 1234 1` `55 0 0`	`86 1234 1 0` `55 0 0 0`
matrix `86 1234 1 5678` `55 0 0 5678`	same as given

Only the first 4 columns of a matrix left argument are used. In the above examples the mode would be automatic submission. In the other two modes the left argument is ignored, except for mode determination. For example, the following are all equivalent in establishing validation mode for the file tied to 6:

`¯6` or `¯6 55` or `(,¯6)` or `2 3⍴ ¯6 86 1234 1 55 0 0`

In **interactive mode** you are prompted for file number(s). There are two forms of response:

1) *I J K* where *I J K* are file tie numbers separated by at least 1 blank, e.g. *FILENUMBER(S)*: 55 86. This is handled internally as though a vector of 55 86 had been passed.
2) *I , J K, L M N, O P Q R* where a comma delimits a row of the matrix and a blank delimits the numbers in the row. For example:
   FILE NUMBER(S): 55, 86 1234, 66 1234 1, 77 1234 0 5678
   This is handled internally as though a matrix of
   ```
   55    0 0    0
   86 1234 0    0
   66 1234 1    0
   77 1234 0 5678
   ```
   had been passed in automatic mode.

Validation mode does not submit an *HSPRINT* request. It validates the right argument (see below), but no file checking is done. In **automatic** and **interactive modes** *HSP* will attempt a □*SIZE* on each file and reject any empty ones. It may also attempt a □*RDAC* / □*STAC* depending on column 3 of the matrix. This may lead to a file access error and function suspension (since these errors can not be programmatically detected).

The Right Argument

This is used to pass the print specifications and delivery instructions to the *HSPRINT* program. It is ignored in interactive mode since the information is captured interactively. However, for validation or automatic modes it must be a non-empty character array which is internally ravelled for processing. There are two forms for the right argument:

1) **specs∘delivery instructions**. In this form **specs** are validated, errors reported and missing items are set to their default values. **specs** may contain 0 or more items. For example:
   ```
   55 86 HSP 'FAST∘RBE WILL PICK UP'
   55 86 HSP 'FULL,8LPI,STD2∘DLF WILL PICK UP'
   ```

2) **delivery instructions**. In this form default specs are provided, e.g.
   ```
   55 86 HSP 'EBI WILL PICK UP'
   ```

Anything to the left of the first ∘ is **specs** and everything else is **delivery instructions**.

Specs consists of a list of items, with defaults supplied for any not provided. The items you supply may appear in any order and more than once. Only the last occurrence of any item is used. Following is a list of these items, the default values being given first. (Note that only the first 4 characters of the choice are significant.)

Item	Choices	Meaning
file ties	← *UNTIED*|	- Files in the left argument are untied or left
	TIED	tied when the request has been submitted.
file disposition	*NOERASE*|	- Files in the left argument are not or are
	← *ERASE*	erased after being *HSPRINT*ed.
print train	*FAST*|	- Print train to mount on the highspeed printer,
	FULL|*PN2*	- and translate table to use. (See Appendix B)
lines per inch	6*LPI*|8*LPI*	- Vertical print density.
forms separation	*NODECOLLATE*|	- Do not or do remove carbons from forms.
	DECOLLATE	
copies	*DO*1|*DO*2|..	- Number of times to process request.
	*DO*8|*DO*9	
forms	*STD*1|*STD*2|	- Stock tab 1,2,
	*STD*3|*STD*4|*STD*5	3,4 or 5 part paper;
	*ULD*1|*ULD*2|	unlined stock tab;
	*CON*1|*CON*2	console paper;
	*REV*1	reversed stock tab 1 part;
	CAMC|	camera copy paper;
	PRIVATE FORM	see below.
destination	*PRINTER*|	- output printed at the Toronto data centre
	←*REMO*(nodename)|	- at a remote location
	TAPE(parameter list)|	- transferred to a tape
	FILE	- or put into an APL file (see Remote Printing and Tape Output)

In addition to the above, three other items may be included in *SPECS*.

1) *'QUIT'* anywhere in *SPECS* will cause an immediate exit from the *HSP* program with an explicit result of 10.
2) ←*'HELP'* will display a brief summary of the choices for *SPECS* and will either exit, if you were in automatic or validation modes, or continue interactively.
3) *'SHOW'* will display the request before submission and prompt with *'OK TO SUBMIT?'*.

For example:

```
55 86 HSP 'TIED,ERASE∘RHL WILL PICKUP' is equivalent to:
55 86 HSP 'TIED,ERAS,FAST,6LPI,NODE,DO1,STD1,PRIN∘RHL WILL PICKUP'.
```

PRIVATE FORM

A private form is a form not in the list of forms choices in *SPECS* and is supplied to operations by the customer.

The procedure for establishing a private form is:

1) provide operations with
    a) 2 samples of the complete form and a carriage tape if non-standard,
    b) a list of authorised user numbers,
    c) the **text** you wish to establish;
2) operations will then issue a unique 4 character **forms identification** for use only by the authorised users and will then activate the form in the system. They will advise you of its ID and you may then use this in future *HSPRINT* submissions where appropriate.

For example let **text** be:

```
'USE BROOK CO. SPECIAL FORMS AND CARRIAGE CONTROL TAPE'
```

and let the **forms identification** be *BRCO*. Then if you entered:

```
55 86 HSP 'BRCO∘HOLD FOR PICKUP AT DATA CENTRE.'
```

a request of the following form would be submitted:

```
55 86 HSP 'BRCO∘HOLD FOR PICKUP AT DATA CENTRE.
USE BROOK CO. SPECIAL FORMS AND CARRIAGE CONTROL TAPE'.
```

Note that **text** may be an empty string.

Remote Printing

With the growth of the Sharp network and the wider geographic destribution of customers it has become even more desirable to support some form of highspeed report printing other than at the Toronto data centre. Both *REMO* (nodename) and *FILE* are part of this support.

REMO (**nodename**)

A *REMO* node is a Sharp office, or customer location, where local printing facilities are available. A remote nodes table is maintained in component 8 of the file `99` *HSPSYSTEM*. Associated with each node are up to three user numbers that are privileged to use the *'RSHOW'* function in *WS* `1` *HSPRINT*. This function allows those users to interrogate the queue file for requests directed to that node. Its use is described in the workspace (type *HOWRSHOW*).

The local printing facility may be a 30 cps terminal, a 120 cps terminal such as a GE 1230, DIABLO 1641 or AJ 860, or a bi-synch device such as a 2780 or 2968. All of these are attached directly to the Sharp network.

Additionally, we have the capability to, on an offline basis, communicate over the dialed telephone network with other mainframes, or stand alone bi-synch devices, by using the Toronto data centre Mohawk in some emulation mode.

This means, for example, that a Calgary user can submit a request specifying *REMO(CALG)*. The Calgary office will print the report locally.

FILE

Another method of printing reports in a remote location is to put the output into an APL file.

When your request has been processed, a file is created with the library number of the submittor, and a name of *'HSPRTXXXXXX'*, where *XXXXXX* is the request number, left zero filled.

The file itself, called an *HFILE*, consists of a control message as the first component, followed by character vector components, each one being a page of the *HSPRINT*.

The page components contain sufficient trailing linefeeds and a carriage return to space the terminal paper to 'top of page'.

Warning: When reading consecutive components from the file assign your output to ⍞. Otherwise the APL system will insert an extra, undesired, line between components. The following function, from *WS* `1` *HSPRINT*, will print an *HFILE* on the terminal.

```
    ∇ HPRT FN;I;J
[1]   I←1+1↑J←2↑⎕SIZE FN ◇ J←1↓J
[2]   ⍞←⎕READ FN,I ◇ →(J>I←I+1)⍴⎕LC
    ∇
```

The function *HFILES* in workspace `1` *HSPRINT* will return a matrix of *HFILE* names.

An *HFILE* may itself be submitted for highspeed printing. The control message at the front identifies an *HFILE* to the *HSPRINT* program and causes titling, page numbering, etc. to automatically be suppressed. This is necessary since the file will already contain all needed titles and page numbers.

This means that users can now process a highspeed print file, inspect the result, and if satisfied, get hard copy either at the terminal or on a highspeed printer, as well as keeping the report on line for reference.

TAPE (parameter list)

For knowledgeable users who want tape output and who want to alter the default tape output characteristics (defaults are the first choice), the following options are available:

LABELS|*NOLABELS*	(IBM OS labels or no labels)
TAPEMARK|*NOTAPEMARK*	(only meaningful if *NOLABELS*)
1600*BPI*|800*BPI*|6250*BPI*	(density)
VARIABLE|*FIXED*	(record length variable or fixed)
RECSIZE=2048	(max bytes per logical record)
BLOCKSIZE=2048	(max bytes per physical block)

The above options should be separated by semicolons (;).

e.g. *TAPE*(*NOLABELS*;*NOTA*;*FIXED*;*RECS*=133;*BLOCKSIZE*=1995)

Note that *FIXED* implies that the blocksize is an exact multiple of record size. *VARIABLE* means that trailing blanks will be clipped and the physical block will contain a variable number of logical records (variable blocked but not spanned). Records have ASA carriage control as their first byte.

WARNINGS a) This facility is definitely not recommended for the unsophisticated user.

b) Volume switching is **not** supported. Only one reel of output will be generated.

REQUEST FORMAT AND QUEUE INTERROGATION

One of the features of *HSPRINT* is the ability to interrogate the request queue to determine the status of your request. The following programs to assist you in this can be found in workspace 1 *HSPRINT*.

Syntax	Action
R←MINE	Returns a vector of request numbers of all requests submitted by you in the last few days.
R←ALL	Returns a vector of all request numbers submitted by you that are still in the queue. **Warning:** since the queue may be quite large this may take quite a while and be costly.
R←WITHDRAW X	Attempts to withdraw request numbers in *X* subject to: 1) your user number matching that of the original submittor and 2) the request being unprocessed. Returns a vector of sucessfully withdrawn numbers.
R←SHOW X	Returns a character vector of requests for numbers in *X* that were submitted by you. Requests are delimited by a `'∘'` (uppercase *J*).
R←STATUS X	Returns a character vector of the status of each request number in *X* that was submitted by you. Present status codes are: 0 unprocessed 1 request processed 2 error seen 4 withdrawn 8 in process 16 transmitted (set by operator) 32 submitted (by operations, to be printed)

An *HSPRINT* request, (i.e. the result of `'SHOW'`), consists of 7 distinct fields separated by a delimiter (*DL*←¯1↑⎕*AV*).

These are:

Field	Usage
1) *STATUS WORD*	Indicates the of status of the request (see status function for meanings).
2) *ID AND COUNT*	User number and timestamp of submission. When the request has been processed the number of printed lines will be inserted at the right hand end of this field. When appropriate, a third field indicating time when transmitted, is also inserted.
3) *FILES TO PROCESS*	The names of the files to be processed in the order submitted.
4) *SPECS*	Processing specifications
5) *DELIVERY INSTRUCTIONS*	Where to deliver the output
6) *MAGIC NUMBERS*	BG magic numbers for each file in *FILES TO PROCESS*, i.e. the 4th column of the left argument matrix.
7) [*ERROR REPORT*]	Optional error report. Present only if *HSPRINT* detected an error. The field consists of 3 sub fields: 1) *MSGXX* where *XX* is a number that can be used as an origin 1 index into the variable '*BGHSPRINTERRORS*' in workspace 1 *HSPRINT*. 2) The name of the file that was being processed when the error was detected. 3) Component number of file if applicable.

For example:

```
 0001[]   1142021 10.07.24  14 JUL 1976          000396[]
1142021 TEST1      []NOER,FAST,6LPI,NODE,DO1,STD1,PRIN[]
LOC WILL PICK UP[]0000000000
```

APPENDIX A

At publication time the variables *STATUSWORD*, *HSPERRORS*, and *BGHSPRINTERRORS* in workspace 1 *HSPRINT* have the following values:

STATUSWORD

PROCESSED
ERROR SEEN
WITHDRAWN
IN PROCESS
TRANSMITTED
SUBMITTED

HSPERRORS

INVALID LEFT ARGUMENT
INVALID FILE TIE NUMBER(S)
HELP GIVEN IN AUTOMATIC MODE
EMPTY FILES REMOVED. NONE LEFT TO SUBMIT
INVALID RIGHT ARGUMENT
INVALID SPECS
UNAUTHORISED USE OF PRIVATE FORM
UNKNOWN REMOTE NODE NAME

BGHSPRINTERRORS

WS FULL
FILE TIE ERROR
FILE ACCESS ERROR
FILE INDEX ERROR
FILE FULL
FILE NAME ERROR
FILE DAMAGED
FILE TIED
FILE SUBSYSTEM S ERROR
FILE SUBSYSTEM T ERROR
FILE TIE QUOTA USED UP
FILE QUOTA USED UP
FILE RESERVATION ERROR
ERROR IN TRANSLATION
STACKED CONTROL MESSAGE
INVALID CONTROL MESSAGE NUMBER
INVALID CONTROL MESSAGE NUMBER OF COMPONENTS
INVALID TRANSLATE TABLE
CHARACTER DATA INVALID IN THIS CONTEXT
DOMAIN ERROR
RANK ERROR
CARRIAGE TAPE MUST HAVE A WRITABLE LINE
EXCESSIVE EXECUTION TIME
INVALID DATA TYPE IN THIS CONTEXT
RECURSIVE TITLING/SUBTITLING DETECTED
UNABLE TO RENAME
UNABLE TO STAC

APPENDIX B

HSPRINT TRANSLATE TABLES

In APL there is a set of 256 possible characters represented internally by the hexadecimal values 00 through *FF*, and usually called *Z*-codes or ⎕*AV*. Each element of a character component of a print file must be in this set. ⎕*AV* can be thought of in terms of 4 subsets:

1) Terminal control characters. These can be found in 1 *WSFNS* and consist of *BS*, *UBS*, *CR*, *LF*, *NL*, and *ID*.

2) Graphics that can be directly entered or printed on a standard APL terminal. These can be found by displaying the variable *APLCHARACTERS* in workspace 1 *HSPRINT*.

3) Graphics that cannot be directly entered or printed on a standard APL terminal but can be printed on the high speed printer. Some of these can be found in the group *NONAPLGRAPHICS* in workspace 1 *HSPRINT*.

4) Illegal characters.

Output printed on the highspeed printer will generally appear as anticipated, assuming the correct translate table and print train, when the character components being printed are in groups 1, 2 or 3 above. Those characters in group 4 will generally be printed as ⎕. From a user's perspective, the translate tables can be viewed as defining a mapping from ⎕*AV* into a particular print train used on the printer at the computer centre.

Each translate table is a 128 element vector. However, each integer represents the packing of 4 hexadecimal values, each in the range of 00 through *FF*. Functions to manipulate these tables can be found in workspace 1 *HSPRINT* in the group *TRANS*.

The translate table in its unpacked form can be viewed as a 2 row by 256 column table, *T*. Similarly, the print train can be viewed as a 256 element vector, *P*, of printable and control characters. Assuming 0 origin, let *J*∈⎕*AV*; then *T*[;⎕*AV*⍳*J*] produces an integer pair, *M N* , where *M* is a direct index into *P*, and *N* is an index into the columns of *T* . Thus *T*[;*N*] produces another integer pair, *M' N'*, where *M'* indexes *P* and *N'* indexes *T*; and so on. As a result, a vector of indices into *P* is generated recursively, halting either when ' '=⎕*AV*[*N*] (*N*=152), or when recursion is 10 levels deep.

For example, let *J* be ϕ. Using the *FAST* translate table we find that 49↔⎕*AV*⍳*J* and *T*[;49]↔206 33; furthermore *T*[;33]↔79 152 and *T*[;152]↔64 152. Thus our indices to *P* are 206 79 64 and our output would appear as ϕ. If, however, we had used the *PN*2 translate table then *T*[;49]↔214 157; *T*[;157]↔231 152; *T*[;152]↔64 152 and our indices would be 214 231 64 which prints out as ⎕.

Creation of your own translate tables or modifications to the existing ones requires an understanding of how they function as well as a familiarity with the character sets available on the various print trains.

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

SATN 9
Revision 2
1 November 80

SHARP APL TECHNICAL NOTES

TITLE: Usage Inquiry System

Author: John D. Burger (JDB)

ABSTRACT: Describes how the use of APL system resources can be reported with functions in workspace 1 *USAGE*.

KEYWORDS: Workspace 1 *USAGE*
Resource Usage

INTRODUCTION

The functions in workspace 1 *USAGE* are provided to enable users to access details of the usage of system resources. All users may access data relating to their own usage of resources. In addition, the same functions may be used, where appropriate access has been obtained, to access the data relating to usage by other specified users. Thus a project supervisor could monitor the usage of the users numbers within his control.

All the data collection functions are self-contained, and may be copied into a users's workspace to facilitate the construction of any form of reporting or monitoring package.

Access to usage data for other user numbers is controlled by the use of CODES and MASKS. For the user who is interested only in accessing his own usage data, use of CODES and MASKS is unnecessary and he may safely ignore the following three sections.

CODES

Every user number in the APL system has a branch-code and a customer-code. Together, the branch-code and customer-code is called a CODE. A CODE uniquely defines an invoicing group of user numbers for billing purposes, and appears on the monthly APL invoice. A CODE is composed of two integer numbers, branch-code and customer-code. Both numbers are composed of several parts.

The branch-code contains information related to I.P. Sharp Associates, and is made up of three parts. The first part is the I.P. Sharp company number, which is one or more digits. Two digits are added to the right of the company number to define the I.P. Sharp office within the company. A further two digits are added to the right of both the company and the office to define the department within the I.P. Sharp office.

The customer-code consists of customer-related information, and has two parts. The first part is the customer number, usually of three or more digits. Three digits are appended to the right of the customer number to define an invoice sub-group for that customer number. A sub-group number is meaningless without a customer number to the left of it.

For example the CODE:

14231 4321001

uniquely defines one billing or invoice group to which user numbers may be assigned. Inspection of the branch code, 14231, reveals that the invoice group is within department 31 of office 42 of company 1. Inspection of the customer code reveals that this is invoice group 001 for customer number 4321. Note that to maintain the correct number of digits fields are zero filled.

MASKS

A MASK is a means of indicating to the *USAGE* functions a range of CODES.

None of the parts of a CODE (company, office, department, customer, invoice) may ever have a zero value. This is utilised in the MASK. If the rightmost fields in either the branch-code or customer-code are set to zero, then this indicates that any value is acceptable in these fields.

Thus a branch-code mask of:

`14231` indicates department 31 of office 42 of company 1

`14200` indicates any department in office 42 of company 1

`10000` indicates any department in any office of company 1

`00000` indicates any possible company, office and department combination

And a customer-code mask of:

`4321001` indicates invoice group 001 for customer 4321

`4321000` indicates all invoice groups for customer number 4321

`0000000` indicates all possible customer and invoice group combinations

Of course `00000` and `0000000` are identical to `0`, and so `0` may be used as the most general mask for either the branch code or the customer code masks.

A branch-code mask and a customer-code mask may be combined to create a MASK. Thus to indicate that all CODES in company 3 for customer 1234 are acceptable the MASK would be:

```
30000 1234000
```

and to indicate all CODES for customer number 1234 the MASK would be:

```
0 1234000
```

ACCESS TO USAGE DATA

All users have access to their own usage data. Access to more than the user's usage data may be arranged through your local SHARP APL representative.

Access is controlled by the use of MASKS. If the accessor has no access MASKS then he has the default access to his own usage data. Each additional access mask denotes a range of CODES to which the accessor has access. Usage data for users assigned to CODES that fall within these MASKS is then available to the accessor through the use of the functions in workspace `1` *USAGE*.

FUNCTIONS IN WORKSPACE 1 *USAGE*

Syntax	Description
R←ACCESS	The result is an *N*×2 matrix of the user's access control masks. If the result is an empty matrix, then the user has access only to his own data.
R←ACCOUNTS MASK	The right argument *MASK* is one or more masks. *MASK* may be any shape provided that the ravel of *MASK* results in pairs. If the argument is empty, or does not have an even number of elements, it is zero filled to be one or more pairs in length. The result is an *N*×3 numeric matrix, one row for each user number, available to the accessor, whose CODES fall within the MASK. The first column is the user numbers and the second and third columns the corresponding CODES.
R←BCCC NOS	The argument *NOS* is a scalar or vector containing the user numbers for which the CODES are required. The result is an *N*×3 numeric matrix, one row for each user number in the argument. Column one is the user numbers, and columns two and three the corresponding CODES. If any of the numbers in the argument are not in the APL system, or are unavailable to the accessor, then the corresponding row is zero filled.
BFDEFN	Defines several variables which may be used to index the result of the function *USAGE*.
BF1DAYDEFN	Defines several variables which may be used to index the result of the function *STORAGE*.
R←CUSTADDR CODE	The argument *CODE* is a vector or matrix of the *CODE*(*S*) for which the customer address is required. More than one code may be specified as an *N*×2 matrix. The result is an 8×50 character matrix of the customer address, or an *N*×8×50 character array if the argument is a matrix. If the *CODE* does not exist, or is unavailable to the accessor, then the matrix is blank filled.

R←GROUPS CODE

The argument *CODE* is a vector or a matrix. If it is a vector, it consists of one or more *CODES*, and if the argument is a matrix, each row may contain one or more *CODE*s.

The result is an *N*×2 numeric matrix with each row representing a unique *CODE*. *GROUPS* returns all *CODES* within a given branch-code or customer-code.

The following are valid uses of *GROUPS*:

a) `GROUPS 10101 4356000` ⍝ return all branch-codes and customer-codes for customer `4356` in location `10101`

b) `GROUPS 10101 4356000 10111 547000` ⍝ all branch-codes and customer-codes for `4356` in `10101` and `547` in `10111`

c) `GROUPS 10000 4356000 20000 4356000` ⍝ all branch-codes and customer-codes for `4356` in countries `1` and `2`

d) `GROUPS 3 2⍴0 4356000 0 543000 10101 0` ⍝ all branch-codes and customer-codes for customers `4356` and `543` in all countries and for all customers in branch `10101`

R←NAMES NOS

The argument *NOS* is a scalar, or vector, containing the user numbers for which the signon names are required.

The result is an *N*×22 character matrix of names, one row for each user number in the argument. The format of the result is similar to the format of *⎕NAMES* or *⎕LIB*.

If any of the numbers in the argument are not in the APL system, or are unavailable to the accessor, then the corresponding row is blank filled.

R←OFFHIST MYDA

The argument *MYDA* is a vector of

{MONTH NUMBER} {YEAR} {[DAY NUMBERS]} {[USER NUMBERS]}

All days are reported if no days are specified and the current users number is assumed if no user numbers are specified.

The result *R* is a matrix of sign-off records for all requested user numbers for which the accessor may access data.

Hence the following uses would be valid.

a) `OFFHIST 8 79` ⍝ return all records for the accessor for August 1979

b) `OFFHIST 6 1979 25 26 27` ⍝ records for 25, 26 and 27th June 1979

c) `OFFHIST 10 78 2381946 4511127` ⍝ specified users for October 1978

c) `OFFHIST 8 79 1 2 4511127 3918261` ⍝ users on 1st and 2nd of August 1979

R←FILE OFFHIST MYDA The left argument *FILE* may be the number of a file which is already tied or a file name which will be created if possible, if it doesn't already exist.

The right argument *MYDA* is a vector of

{MONTH NUMBER} {YEAR} {[DAY NUMBERS]} {[USER NUMBERS]}

All days are reported if no days are specified and the current users number is assumed if no user numbers are specified.

The result *R* is the tie-number of the file to which sign-off records, as requested in the right argument, have been appended in blocks of one hundred, or less, records.

The following would be valid dyadic uses.

a) `10 OFFHIST 8 79` ⍝ append to file tied to 10 the requested records

b) `'OHRECS' OFFHIST 1 79` ⍝ create a file called *'OHRECS'* and add records

c) `'OHRECS' OFFHIST 2 79` ⍝ add more records to file *'OHRECS'*

In the case where data is requested for a period for which there is no data on-line then a single 'pseudo record' is returned. The pseudo record for no data on-line consists of a single zero filled record.

R←OHONLINE The result is an *N*×2 numeric matrix where each row is a month year pair indicating that *OFFHIST* data is on-line and available for that time period.

OHPRINT MYDA

The function *OHPRINT* formats into report form data from the function *OFFHIST*. The right argument is a vector of

{MONTH NUMBER} {YEAR} {[DAY NUMBERS]} {[USER NUMBERS]}

All days are reported if no days are specified and the current user's number is assumed if no user numbers are specified.

The following are valid uses of *OHPRINT*.

a) `OHPRINT 8 79` ⍝ print records for the accessor for August 1979

b) `OHPRINT 6 79 25 26 27` ⍝ print records for the 25, 26 27th of June, 1979

c) `OHPRINT 10 79 2381946 4511127` ⍝ print records for specified users for October 1979

d) `OHPRINT 8 79 1 2 4511127 3918261` ⍝ print records for specified users on specified days in August 1979

R←FILE OHPRINT MYDA

The result *R* is the tie number of the file to which the sign-off report has been appended. The left argument may be the number of a file which is already tied or a file name which will be created (if possible) if it doesn't already exist.

The following are valid dyadic uses of the function *OHPRINT*.

a) `10 OHPRINT 8 79` ⍝ append the offhist report to the file tied to number 10

b) `'OHRECS' OHPRINT 1 79` ⍝ create or tie the file *'OHRECS'* and append a report for January 1979

c) `'OHRECS' OHPRINT 2 79` ⍝ append another report to the file *'OHRECS'*

R←ONLINE

The result is an *N*×2 numeric matrix where each row is a month year pair indicating that *USAGE* data is on-line and available for that time period.

PRINT MYA

Prints out a formatted report of the usage data as specified in the argument *MYA*. *PRINT* calls the functions *BFDEFN*, *BF1DAYDEFN*, *NAMES*, *STORAGE*, *UPDATES* and *USAGE* to obtain the necessary data.

The argument must be a vector of at least two elements, these elements denoting the month and year for which data is required.

- If the argument is of just two elements then the report is for the accessor's usage data.
- If the argument is of more than two elements then the report is for those user numbers that follow the month and year.

Blank rows in the report indicate non-existent, or unavailable, user numbers.

R←QUOTAS A

The result is a 9-element numeric vector containing information about the user's number. When the right argument *A* is empty, (`''`), the result is:

R[1]	1 if the user is classified as 'internal', 0 otherwise.
R[2]	The workspace quota.
R[3]	Number of workspaces saved.
R[4]	CPU limit (0 means 'no limit').
R[5]	User's file system volume class.
R[6]	File quota.
R[7]	Number of files created.
R[8]	File reservation limit.
R[9]	Total file bytes reserved.

Workspace information alone may be obtained by using a right argument of `'W'`. In this case *R*[5 6 7 8 9] will be zeros. Similarly, file information may be obtained by using `'F'` as the right argument, giving zeros in *R*[1 2 3 4]. *QUOTAS* executes more quickly if `'F'` or `'W'` is specified.

R←STORAGE MYA

The argument *MYA* indicates the month, year and user numbers for which the last days storage data is required. The argument must be a vector of at least two elements, denoting the month and year for which data is required.

- If the argument is of two elements then the result is a numeric vector of storage data for the accessor.
- If the argument is of more than two elements then the result is a numeric matrix of storage data, one row for each user number following the month and year.

The values contained in the columns of the matrix, or elements of the vector, in the result may be indentified by displaying the function *BF1DAYDEFN*.

Non-existent, or unavailable, numbers are indicated by zero filled rows.

R←UPDATES MY

The argument *MY* must be a two element vector indicating the month and year for which the status of update is required.

The result is a 5×3 numeric matrix. Column one indicates the updating status, column two the date of update, in *MMDDYY* format, and column three the time of update in *HHMMSS* format. The status is defined as:

```
¯1 - unavailable.
 0 - update in progress.
 1 - valid.
```

The first row relates to **storage updates**, second to **sessions** (e.g. connect and CPU), third to *FILEPRINT*, fourth to *HSPRINTS* and fifth to **sorts**.

R←USAGE MYA

The argument *MYA* must be of similar form as the argument to *STORAGE*: a two, or more, element vector.

The result is of the same shape and form as the result of *STORAGE*. However, the data returned relates to **cumulative month to date** usage data, rather than last days storage data.

The values contained in the result may be identified by displaying the function *BFDEFN*.

R←WHOIS USER

The argument *USER* is a scalar, vector or matrix, containing the user numbers or sign-on names for which the sign-on names, branch codes and customer codes are required. Multiple sign-on names in the argument may be in vector form separated by semicolons (;), or in matrix form where each row is a sign-on name.

The result is an *N*×38 character matrix of names, one row for each user number or sign-on name in the argument. The format of the result is like that of ⎕*NAMES* plus two fields, each eight wide, of *CODE* information.

If the argument is numeric, any numbers that are not in the APL system, or are unavailable to the accessor, result in the corresponding rows being blank filled.

Information for deleted users is not normally printed when the argument is a sign on name, but these records may be displayed by preceding the sign-on name with a tilde (~).

The following are valid uses of the *WHOIS* function:

a) *WHOIS* 1234567
⍝ get information for account 1234567

b) *WHOIS* 1234567 7654321
⍝ get information for account numbers 1234567 and 7654321

c) *WHOIS* '*ABC*'
⍝ get information for all users whose sign-on name begins with '*ABC*', ignoring deleted accounts

d) *WHOIS* '*ABCD;EFGH*'
⍝ get information for all users whose sign-on name begin with either '*ABCD*' or '*EFGH*', ignoring deleted accounts

e) *WHOIS* '*ABCD;~EFGH;IJK*'
⍝ get information for all users whose sign-on name begins with '*ABC*', ignoring deleted accounts, or '*EFGH*', including deleted accounts, or '*IJK*', ignoring deleted accounts

f) *WHOIS* 2 3⍴'*ABCDEFGHI*'
⍝ get information for all users whose sign-on name begins with '*ABC*', '*DEF*', or '*GHI*', ignoring deleted accounts

```
    ∇ BF1DAYDEFN;I
[1]  ⍝  column numbers for the last days storage data
[2]    I←⎕IO ⍝                          ∘  use current index origin
[3]    BF1DUSER←I+0 ⍝                   ∘  user number
[4]    BF1DWSS←I+1 ⍝                    ∘  number of workspaces saved
[5]    BF1DWSBYTES←I+2 ⍝                ∘  number of bytes of workspaces saved
[6]    BF1DBSSBYTES←I+3 ⍝               ∘  number of bytes of BSS stored
[7]    BF1DCOL←4 ⍝                      ∘  the number of columns in the 1 day blocks
[8]    BF1DAYSHP←BF1DAYL,BF1DCOL ⍝∘     the shape of 1 day blocks
    ∇
```

```
    ∇ BFDEFN;I
[1]   ⍝  column numbers for the month-to-date usage data
[2]   ⍝  cpu times are in 300ths of units
[3]   ⍝  connect times are in 300ths of a second
[4]     I←⎕IO ⍝                    ∘ use current index origin
[5]     BFUSER←I+0 ⍝               ∘ user number
[6]   ⍝  columns 1 to 6 are reserved (origin - 0)
[7]     BFCPU←I+7 ⍝                ∘ terminal cpu time this month
[8]     BFCON←I+8 ⍝                ∘ terminal connect time this month
[9]     BFCHRS←I+9 ⍝               ∘ characters this month
[10]    BFNCPU←I+10 ⍝              ∘ N-task cpu time this month
[11]    BFNCON←I+11 ⍝              ∘ N-task connect time this month
[12]    BFBCPU←I+12 ⍝              ∘ B-task cpu time this month
[13]    BFBCON←I+13 ⍝              ∘ B-task connect time this month
[14]    BFBSSBD←I+14 ⍝             ∘ BSS byte days this month (÷1000 IFF 1=2|BFSTRUPD)
[15]    BFWSDYS←I+15 ⍝             ∘ WS days this month
[16]    BFWSBDYS←I+16 ⍝            ∘ WS byte days this month
[17]    BFSTRUPD←I+17 ⍝            ∘ storage accounting flags
[18]    BFSOHUPD←I+18 ⍝            ∘ SOH update indicator
[19]    BF30CPS←I+19 ⍝             ∘ time connected at 30 cps
[20]    BFLINES←I+20 ⍝             ∘ number of lines fileprinted this month
[21]    BFLNUPD←I+21 ⍝             ∘ fileprint update indicator
[22]    BFFORMS←I+22 ⍝             ∘ the number of forms mounts this month
[23]    BFTRAINS←I+23 ⍝            ∘ the number of train mounts this month
[24]    BFMISC←I+24 ⍝              ∘ miscellaneous fileprint charges this month (cents)
[25]    BFLNSMIN←I+25 ⍝            ∘ the number of minimum cost fileprints and HSPRINTs
[26]    BFPRINTN←I+26 ⍝            ∘ the number of fileprints and HSPRINTs
[27]    BFPLY←I+27 ⍝               ∘ the number of extra ply lines (lines×ply) for HSPRINT and
                                     fileprint
[28]    BFHSUPD←I+28 ⍝             ∘ the HSPRINT update indicator
[29]    BFSRTUPD←I+29 ⍝            ∘ the file sort update indicator
[30]    BFSRTN←I+30 ⍝              ∘ the number of file sorts this month
[31]    BFSRTMIN←I+31 ⍝            ∘ the number of minimum cost file sorts this month
[32]    BFSRTOPS←I+32 ⍝            ∘ the number of FILEOPS during sorting this month
[33]    BFSRTSZE←I+33 ⍝            ∘ the file size total sorted this month
[34]    BFTPELNS←I+34 ⍝            ∘ the number of lines printed to tape
[35]    BFDEC←I+35 ⍝               ∘ the number of prints decollated
[36]    BFFCOL←36 ⍝                ∘ the number of columns in monthly data
[37]    BFFSHP←BFFL,BFFCOL ⍝       ∘ the shape of monthly data blocks
    ∇
```

```
    ∇ OHDEFN;I
[1]  ⍝  column numbers for the OFFHIST data
[2]  ⍝  cpu times are in 300ths of units
[3]  ⍝  connect times are in 300ths of a second
[4]   I←⎕IO ⍝          ∘  use current index origin
[5]   OHUSER←I+0 ⍝     ∘  user number
[6]   OHDATE←I+1 ⍝     ∘  date in form MMDDYY
[7]   OHTIME←I+2 ⍝          ∘  time in form HHMMSS
[8]   OHTASKID←I+3 ⍝        ∘  TASKID number
[9]   OHNODE←I+4 ⍝     ∘  node/line number
[10]  OHTTYPE←I+5⍝     ∘  task type. 0-T, 1-N, 2-B
[11]  OHBAUD←I+6 ⍝     ∘  nominal terminal baud rate
[12]  OHMODE←I+7 ⍝     ∘  sign-off mode. 1-bounce, 2-drop, 3-crash, 4-crash (but not billed)
[13] ⍝  columns 8 and 9 are reserved (origin - 0)
[14]  OHCPUS←I+10 ⍝         ∘  cpu during session
[15]  OHCPUC←I+11 ⍝         ∘  cpu cumulative
[16]  OHCONS←I+12 ⍝         ∘  connect time during session
[17]  OHCONC←I+13 ⍝         ∘  connect time cumulative
[18]  OHCHARS←I+14 ⍝        ∘  character transmission during session
[19]  OHCHARC←I+15 ⍝        ∘  character transmission cumulative
[20]  OHWS←I+16 ⍝           ∘  number of workspaces stored
[21]  OHCR←I+17 ⍝           ∘  number of carriage returns during session
    ∇
```

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

SATN 10
1 JUN 78
Rev. 2

SHARP APL TECHNICAL NOTES

TITLE: SORTREQ

ABSTRACT: A facility for sorting SHARP APL files.

KEYWORDS: 1 *FILESORT*
BG sort

SHARP AR TECHNICAL NOTES

The workspace 1 *FILESORT* contains functions which allow users to submit requests for BG sorts of SHARP APL files.

Request Format

A request takes the form of a character vector. In each case the parameter name and an = sign precede the parameter setting.

Parameter Name	Default
OUTFILE=	(mandatory)
KEYS=	ascending
INROWS=	rows in first component
OUTROWS=	*INROWS*
SEQ=	⎕*AV*
INLOCK=	0
OUTLOCK=	0
INCOMPA=	1↑⎕*SIZE*
INCOMPZ=	1↑1↓⎕*SIZE*
INTIE=	share
OUTTIE=	share
OUTDROP=	minus

OUTFILE= defines the output file and must be specified. The library number, if specified, must match the user number. For example:

```
OUTFILE=1234567 FILEO
```

KEYS= defines the sort keys to be used. These consist of a parenthesized set of triplets, each triplet of the form:

```
N M X
```

where *N* is the origin 1 starting column
M is the number of contiguous columns, and
X is either *A* or *D*, indicating that the file is to ascend or descend on this key.

The BG sort program imposes restrictions so that:

- for numeric files +/*M*1,*M*2, ... must be less than or equal to 12.

- for character files +/*M*1,*M*2, ... must be less than or equal to 256 and not more than 12 triplets may be specified.

- further for character files no *N* may exceed 4092
  for integer no *N*+4×*M*-1 may exceed 4092
  for floating point no *N*+8×*M*-1 may exceed 4092

For example *KEYS*= (1 3 *A* 7 4 *D* 12 8 *A*)
would be syntactically correct but not acceptable for numeric files since (3+4+8)>12.

Warning:

The BG sort program is "unstable". This means that records which have identical key fields but differ elsewhere might not have the same relative ordering before and after sorting. In order to be able to fully predict the final sequence of records, enough keys must be specified to ensure that no two records have all key fields identical. Thus, if a file is already ordered on a particular key and further sorting is required, that key field must be included in the *KEYS*= parameter to ensure that the file remains properly ordered on that key.

INROWS= defines the maximum number of rows expected in any component of the input file. Input is restricted to scalars, vectors and matrices of constant data type and constant number of columns. Output is always matrices.

It is important that this estimate not be low since this will cause the request to be rejected by the BG sort.

If omitted, the number of rows in the first input component becomes the default.
For example: *INROWS*=20

OUTROWS= defines the number of rows per component of the output file.

If omitted it defaults to *INROWS*.
For example: *OUTROWS*=40

SEQ= specifies a collating sequence. If omitted it defaults to □*AV*. Only the relevant character set need be specified. the sequence terminates at any repeated character.
For example: *SEQ*= 01234567890

This example would cause blank to have the lowest collating value, 0 the next lowest and so on up to 9. Thereafter the remaining elements of □*AV* would resume their normal sequence.

INLOCK= specifies magic number for BG access to input file.

OUTLOCK= specifies magic number for BG access to output file

INCOMPA= specifies initial input component.

INCOMPZ= specifies 1+ final input component number.

INTIE= share or full specifies nature of tie of input file done by BG.

OUTTIE= share or full specifies nature of tie of output file done by BG.

OUTDROP= minus or none or plus specifies whether output file is to have all components dropped initially and if so from which end.

An example of a complete request argument might be:

```
OUTFILE=12345 ASDF OUTROWS=4 KEYS=(5 2 A) SEQ=   OUTTIE=FULL
```

If the input file was the single component

```
ABCDEF
GHIJKL
012345
A B C
E
X    Y
```

the output file would be the components

```
E
X    Y
A B C
ABCDEF
```

and

```
GHIJKL
012345
```

i.e. sorted on the last 2 columns with blank sorting ahead of its normal position in ⎕*AV* because of its presence in the *SEQ=* parameter.

The file 12345 *ASDF* would be ⎕*TIE*d if possible, otherwise ⎕*CREATE*d. Any existing components would be dropped.

Request Submission Functions

The request is filed using the dyadic function

REQUESTNUMBER←REQUEST SORTREQ TIENUMBER

which takes the request as a left argument.

The simplest form of the right-hand argument is the input file tie number or tie number and magic number. When fully specified the right-hand argument is a 2×4 matrix. Row 1 refers to the input file, row 2 to the output file. Columns 1 and 2 are the current tie and magic numbers.

Column 3: 0 - set BG access if needed
1 - do not set BG access

Column 4: the magic number for which BG access is to be set.

The result of *SORTREQ* is a request number which may be used as an argument to *SORTINQ* and *SORTWITHDRAW*. It also may be used in discussing the fate of your request with the APL operator.

SORTREQ unties the input file just before submitting the request.

When the request is filed a leading blank and the submittor's account number are catenated at the left and the input filename at the right. You may display the filed request by using the function:

REQUEST←SORTINQ REQUESTNUMBER

The character *?* is returned if the sort request number does not belong to this user.

When the sort is complete the leading character is replaced by an alphabetic. *'A'* signifies successful sort; other alphabetics signify some error. The nature of the error may be displayed by the function

ERRORTEXT←SORTCODE ERRORCODE

An asterisk is used to indicate a withdrawn sort request. Withdrawal may be made only by a task with the same account number as the submittor. The result of the function

REQUEST←SORTWITHDRAW REQUESTNUMBER

is the request itself. If it begins with an asterisk the request has been withdrawn. Otherwise the withdrawal has been too late and the sort *ERRORCODE* (or successful completion code) is present.

The function:

SORTMINE

returns the request numbers of all recent sort requests.

Documentation on all functions and request parameters is available in 1 *FILESORT* in *DESCRIBE*.

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

SATN 11
1 JAN 76

SHARP APL TECHNICAL NOTES

TITLE:)*RESET*

ABSTRACT:)*RESET* clears the SI stack.

KEYWORDS: →

SHARP AR TECHNICAL NOTES

The)*RESET* command takes no argument. Executing)*RESET* causes the SI stack to be totally discarded.)*RESET* has the same effect as executing → for each ⋆ (suspended function) on the SI stack. The edit-line at each SI level (the line recalled for editing by)*N*) is also discarded.

Previously, it was frequently necessary to display the SI stack and then to enter several naked branches to clear it. Clearing the SI stack is a critical procedure that is necessary since useless data might cause *WS FULL* errors. It also ensures that execution is done in a "clean" environment. Simple faults can go undetected and cause major problems if execution is done with old data and functions left on the SI stack. For example, a reference to a variable that would, if the SI stack had been cleared, cause a value error, thereby pointing out the fault immediately, could instead use an old incorrect value from the SI stack.

The old multi-step procedure can now be done by simply entering)*RESET*. As it is now easy to clear the SI stack, *WS FULL*s and program faults due to old data on the SI stack should be less frequent.

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

SATN 12
1 JAN 76

SHARP APL TECHNICAL NOTES

TITLE:)*COPY*

ABSTRACT:)*COPY* now runs about 5-12 times faster.

KEYWORDS: Efficiency

SHARP/APL TECHNICAL NOTES

)COPY has been completely rewritten to achieve a significant increase in efficiency. Previously a copy of a large workspace with many functions cost more than $5.00. Now it costs about $.50. This increase in efficiency is almost completely due to a new method for copying functions. Previously functions were copied by displaying them and then redefining them. The process (and cost) was equivalent to performing `1 ⎕FD` in the source workspace and `3 ⎕FD` in the sink workspace. The new copy moves the function in internal format and is considerably more efficient.

The new copy differs from the old in the following ways:

1. The new copy will never cause *SI* damage. If copying a function would damage the *SI* stack the function is reported as not copied. *)RESET* can clear the *SI* stack so that the function can be copied.

2. All objects that can be copied are copied. All objects that are not found or not copied are reported.

3. The new copy does not have the side effect of changing the data type of numeric literals in functions as did the old copy. For example, the old copy would change `F[1] A←1000⍴1.0` to `F[1] A←1000⍴1` , whereas the new copy will not change the line.

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

SATN-13
10 MAR 78

SHARP APL TECHNICAL NOTES

TITLE: EARLY WARNINGS

ABSTRACT: This SATN provides effective dates and details of system changes that affect the system interface with the user or might make old functions fail or work differently.

KEYWORDS:

SHARP ART TECHNICAL NOTES

Most development in SHARP APL is carefully kept transparent to old functions and to the system interface with the user. Occasionally it is necessary or desirable to make system changes that are not transparent. Such system changes will be made infrequently and will be announced considerably in advance by *)HI* messages, *NEWS* items and by revisions of this SATN. When possible, use of a part of the system scheduled for change will be monitored to allow contact of users who will be affected.

The following are the general principles which guide decisions on non-transparent system changes:

1. Compatibility (when possible and reasonable) with the APL community. A useful guide for this is the IBM APL Language manual, publication GC26-3847.

2. General trend away from SYSTEM COMMANDS, and toward their replacement by SYSTEM FUNCTIONS and SYSTEM VARIABLES.

3. Removal of old mechanisms that have been superseded by new mechanisms.

4. Simplification of the system and the documents necessary to describe it. In particular, the removal of mechanisms of marginal or questionable utility.

A: Changes effective on or after 1 MAY 1978.

1. *)DIGITS* *)TABS* *)ORIGIN* and *)WIDTH* will no longer be valid system commands. Their use will cause an *INCORRECT COMMAND* report. They have been replaced respectively by the system variables *⎕PP*, *⎕HT*, *⎕IO*, and *⎕PW* (see SATN-20).

2. System commands will not be allowed in function definition. A system command entered in function definition mode will be treated as a function line. The use of) or *)N* to bring a line into the function is considered part of the function definition editor and will continue to have the same effect.

3. *)ERASE* will report *NOT ERASED*, rather than damage the STATE INDICATOR and report *SI DAMAGE*.

4. Carriage return will not be allowed in character constants during input from the terminal. A carriage return following an odd number of quotes will close the quote and end the line.

5. Load of an old workspace saved using the old semicolon autostart feature will not autostart.

6. *)VARS* and `1 ⎕WS 'N'` will not report system variables.

7. *⎕RUN* and *⎕FD* error results will indicate origin-1 error locations. Previously *⎕RUN* returned an origin-0 error index and *⎕FD* returned an origin-dependent error index.

B: Changes effective on or after 1 OCT 1978.

1. 2 □*WS* 3 elements for *ORIGIN*, *DIGITS*, *WIDTH*, *TABS*, *FUZZ* and *LINK* will not be maintained and will contain ¯1. They have been replaced respectively by the system variables □*IO*, □*PP*, □*PW*, □*HT*, □*CT*, and □*RL* (see SATN-20).

2. The I-BEAMS for *ORIGIN*, *DIGITS*, *WIDTH*, *TABS*,, *FUZZ* and *LINK* will cause *DOMAIN ERROR*. The functions in 1 *WSFNS* have been changed to use the appropriate system variables and should be copied to replace old versions which contain the I-BEAMS.

3. The *AUTOLOAD* and *QUIETLOAD* functions will cause *DOMAIN ERROR*. They should be replaced by □*LOAD* and □*QLOAD*.

4. The I-BEAM used in the old *DELAY* function from 1 *WSFNS* will cause a *DOMAIN ERROR*. The old *DELAY* function should be replaced by □*DL* or by the new *DELAY* function from 1 *WSFNS* which uses □*DL*.

5. The I-BEAM used in the old *ARBOUT* function in 1 *WSFNS* will cause a *DOMAIN ERROR*. It is replaced by □*ARBOUT*.

6. System variables will not be allowed as labels. Beginning a line with a system variable followed by a colon will result in a line without a label. Such a line will cause a *SYNTAX ERROR* when executed. An old function line with a system variable label will continue to work, but redefining the line will result in the system variable no longer being a label.

7. Numeric vectors printed on the terminal or by *HSPRINT*, as well as those generated by monadic ⍕, will provide a single blank between elements rather than the current two. In addition, the number of blanks separating elements of the last coordinate of integer or floating point arrays of rank two or greater will be decreased by one.

8. □*TS* will include the century in its first element (i.e. 1978 rather than 78).

9. □*TS*, □*RDCI*, workspace timestamps, *SCHEDULE* in 1 *NEWS*, communication with operators, and most other references to time, will be in GMT. See SATN-29 for details.

10. Input to □ and ⍞ of the five characters *O* backspace *U* backspace *T* will cause an INTERRUPT at the □ or ⍞.

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

SATN 14
15 AUG 78
Rev. 2

SHARP APL TECHNICAL NOTES

TITLE: Package - A New Variable Type.

ABSTRACT: A package is a collection of named objects (i.e. user-defined functions or variables). A number of system functions are provided to manipulate packages, including functions to assemble them from objects in the workspace and to define objects in the workspace from objects in the package.

KEYWORDS:

SHARP PAPER TECHNICAL NOTES

NAMELISTS

A **namelist** is a scalar, vector or matrix character array of names of APL objects. If a vector, the names in the namelist are separated by blanks; if a matrix, one name appears in each row with optional leading and trailing blanks. This construction is frequently used in conjunction with the workspace-manipulating system functions such as ⎕*WS* and ⎕*FD*.

Many of the system functions that manipulate packages also require, or accept optionally, an argument of this type, or return one as a result. When used as an argument, duplicate names are permitted; the duplicates are ignored.

⎕*PACK*

```
P←⎕PACK NL
```

⎕*PACK* assembles a package *P* from a namelist *NL*. Each object named in *NL* must be a variable (including system variables such as ⎕*IO*), a user-defined function, a keyword (such as Δ*FMT* - but not a system function such as ⎕*FMT*), or be undefined. *P* becomes a package of these named objects with their values at the time of assembly.

DOMAIN ERROR will occur if a name in *NL* denotes a locked function in a sealed workspace.

```
P←NL ⎕PACK V
```

When used dyadically, ⎕*PACK* assembles a package *P* containing the variable *V* with the name specified by *NL*. *DOMAIN ERROR* will occur if *NL* is not a namelist containing a single name.

⎕*PDEF*

```
⎕PDEF P
```

⎕*PDEF* defines in the workspace each of the named objects in *P*. Previously defined objects of the same names are erased and acquire the new definition. In the case of localized objects it is the local definition that is affected rather than the global definition.

```
NL ⎕PDEF P
```

When a namelist is provided as a left argument to ⎕*PDEF*, only those objects named in *NL* acquire a new definition from *P*.

DOMAIN ERROR will occur if:

a) A name in *NL* does not denote an object in *P*.

b) A selected object is
   1) on the)*SI* stack
   2) a label
   3) a locked function in a sealed workspace.

For example:

```
      A←'ABC'
      BB←1 2 3
      T←⎕PACK 'A BB'
      )ERASE A BB
      'BB' ⎕PDEF T


      A
VALUE ERROR

      A
      ∧
      BB
1 2 3
      )ERASE BB
      ⎕PDEF T
      A
ABC
      BB
1 2 3
```

⎕PPDEF

```
      NL2←⎕PPDEF P
      NL2←NL1 ⎕PPDEF P
```

⎕*PPDEF* is similar to ⎕*PDEF*, except that

1) Only those objects which are undefined in the workspace acquire definitions.
2) The result is a namelist matrix of those names which were rejected because an object of that name was defined in the workspace.

⎕PEX

```
      P1←NL ⎕PEX P2
```

A complementary function to ⎕*PSEL*, ⎕*PEX* returns a package *P*1 with those objects named in *NL* expunged from the package *P*2. Names in *NL* which do not denote objects in *P*2 have no effect.

⎕PNAMES

```
NL←⎕PNAMES P
```

⎕*PNAMES* returns a namelist matrix of the names of the objects in its argument.
If the argument to ⎕*PNAMES* is not a package, then ⎕*PNAMES* returns an empty character vector.
Thus the rank of ⎕*PNAMES* can be used to determine whether *P* is a package or a simple variable.

For example:

```
⎕PNAMES ⎕PACK 'X XYZW AS'
```

would result in the 3 4 matrix:

```
X
XYZW
AS
```

```
⎕PNAMES ⎕PACK ''
```

would result in `0 0 ⍴''` (an empty character matrix)

```
⎕PNAMES 'ASDF'
```

would result in `''` (the empty character vector).

The names within a package are arranged in an arbitrary sequence at the time the package is assembled. Once that package is assembled (and not modified by insertions or delections), the order in which its names are listed does not change.

⎕PSEL

```
P1←NL ⎕PSEL P2
```

⎕*PSEL* returns a package *P1* of objects selected from a package *P2* based on namelist *NL*. If any name in *NL* does not denote an object in *P2*, a *DOMAIN ERROR* occurs.

For example:

```
P2←⎕PACK 'A B CD'
P1←'CD A' ⎕PSEL P2
```

would result in a package *P1* consisting of two objects *CD* and *A*.

⎕PINS

```
P1←P2 ⎕PINS P3
```

⎕PINS returns a package P1 which is the result of inserting the package P3 into package P2.

For example:

```
A←⍳5
P2←⎕PACK 'A B BC'
A←'ASDF'
P3←⎕PACK 'A DEF'

P1←P2 ⎕PINS P3
```

would result in a package P1 containing the objects *A*, *B*, *BC*, *DEF*. Moreover, *A* would have the value *'ASDF'* because that was its value in package P3.

⎕PLOCK

```
P1←⎕PLOCK P2
P1←NL ⎕PLOCK P2
```

⎕PLOCK returns as a result a package with the same contents as its right argument P2, but with all the functions inside locked. If a namelist left argument is provided, only the specified functions become locked. Functions which had been locked remain so.

If any name in *NL* does not denote a function in P2 a *DOMAIN ERROR* occurs.

For example:

```
P←⎕PLOCK ⎕PACK 'FN1 FN2'
```

produces a package *P* whose functions *FN*1 and *FN*2 are locked, even though in the workspace they may well not have been locked, and remain unlocked.

In the past, locking functions in SHARP APL has had the effect of "locking in keywords" such as *ΔFI*, *FE*, etc. This feature is not supported by ⎕PLOCK. This has implications for users of ⎕PLOCK who continue to use outmoded keywords rather than the corresponding system functions ⎕FI, ⎕READ, etc. The implications are:

1) copying a function may also require copying of the keywords it references.

2) **loss of system security** - keywords may be)ERASEd and replaced with user-defined functions or variables. Some systems depend on the integrity of keyword references within locked functions. If yours does, either

   a) don't use ⎕PLOCK,
   or
   b) convert to system functions.

⎕*PNC*

R←*NL* ⎕*PNC P*

⎕*PNC* returns an integer vector indicating the class, within the package *P*, of each name in *NL*.

The values reported by ⎕*PNC* have the following significance:

¯1 ↔ Object undefined but name in use
0 ↔ Object not in package (Name available)
1 ↔ (Reserved)
2 ↔ Variable
3 ↔ Function
4 ↔ Invalid name

The result of ⎕*PNC* is analogous to that of ⎕*NC*, which returns the class, within the workspace, of each name in its argument. Although the interpretations of their results differ in several ways, the major differences lie in the class ¯1 (which does not occur in ⎕*NC*), and the class 1 (which indicates a label in ⎕*NC* and therefore has no significance to ⎕*PNC*).

R←⎕*PNC P*

When used monadically, ⎕*PNC* returns its elements in the same order in which names are returned by ⎕*PNAMES*. Thus, for a named variable *P*:

⎕*PNC P* ↔ (⎕*PNAMES P*) ⎕*PNC P*

⎕*PVAL*

R←*NL* ⎕*PVAL P*

⎕*PVAL* returns the value of the variable named by *NL* within the package *P*.

Domain error will occur if:

a) The name in *NL* does not denote a **defined** variable in *P*.
b) *NL* is not a namelist containing a single name.

PACKAGE MANIPULATION

As well as these system functions for manipulating packages, packages may be used as arguments to:

1) user-defined functions
2) ⎕*APPEND*, ⎕*APPENDR* and ⎕*REPLACE*

They may be returned as a result of:

1) user-defined functions
2) ⎕*READ*
3) 6 ⎕*WS*
4) ⍎ and ⎕

Names of packages may appear anywhere names of variables are permitted:

1) on system commands such as)*COPY* and)*ERASE*
2) returned by system commands such as)*VARS* and)*SIV*
3) in namelists such as arguments to ⎕*FD*, ⎕*WS* or ⎕*PACK*
4) in namelist matrices returned as the result of system functions such as ⎕*WS*.

Packages may not be used as arguments to primitive functions such as + × ÷. Displaying a package produces the text **PACKAGE**. For example:

```
      1;⎕PACK 'ABC';2
1**PACKAGE**2
```

PACKAGE APPLICATIONS

Although we foresee users finding many applications for packages, two major needs inspired their development:

1) **The maintenance of packages on files**

   The introduction of ⎕*FD* immediately led to a number of systems that achieved workspace management by reading character components from file, defining them by 3 ⎕*FD*, invoking the defined function and erasing it when finished. This technique has three major drawbacks:

   1. 3 ⎕*FD* is relatively expensive,
   2. access to the file must be controlled if the functions are to be treated as locked,
   3. functions and variables must be ⎕*READ* one at a time.

   By storing the functions within packages on file, the system designer can greatly reduce the cost of reading functions from files, and simplify system security at the same time.

Timings on typical applications indicate that ⎕*PACK* is comparable in cost to 1 ⎕*FD*, but that ⎕*PDEF* operates at one fifth the cost of 3 ⎕*FD*.

A sample timing:
A workspace contained 59 functions totalling 12.5*K* storage. The cost of

```
NL←1 ⎕WS 1   ⍝ namelist matrix of all functions
P←⎕PACK NL   ⍝ assemble into a package
```

was about 2.7 cpu units;
while a loop:

```
Q←1 ⎕FD NL[I←I+1;]   ⍝ loop over all rows of NL
```

cost 2.0 cpu units.
When the appropriate data was appended to file

```
⎕PDEF ⎕READ FILE,1   ⍝ redefine all functions
```

cost 0.36 cpu units;
while a loop

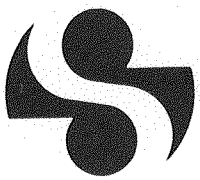
```
3 ⎕FD ⎕READ FILE,I←I+1   ⍝ loop to redefine all functions
```

cost 2.1 cpu units.

2) The mixture of data types within a single file component.

It is occasionally necessary to maintain parallel files in order to hold information of mixed data type. These techniques are at best awkward, and occasionally susceptible to race conditions when shared by several users. Packages provide a solution to this problem.

SATN-15
20 APR 76



I. P. Sharp Associates Limited Suite 1400, 145 King Street West, Toronto, Ontario M5H 1J8 • (416) 364-5361

SHARP APL TECHNICAL NOTES

TITLE: *INDEX*

ABSTRACT: *THE APL PRIMITIVE FOR INDEX FETCH AND STORE HAS BEEN REWRITTEN TO BE MORE EFFICIENT.*

KEYWORDS: *EFFICIENCY*

THE MAIN OBJECTIVE IN REWRITING INDEX WAS TO MAKE IT FASTER THAN THE OLD ONE.

A SECOND OBJECTIVE WAS TO REMOVE THE FOLLOWING SIGNIFICANT PROBLEM: IF X[Y]←Z CAUSED AN INDEX ERROR THEN X WAS PARTIALLY MODIFIED. THE NEW INDEX DETECTS AND REPORTS THE INDEX ERROR WITHOUT AFFECTING X.

ANALYSIS OF THE OLD INDEX SHOWED THAT THE BEST APPROACH TO THE PROBLEM WOULD BE TO PROVIDE SPECIAL ROUTINES TO HANDLE THE MOST OFTEN USED CASES (I.E. VECTORS AND MATRICES).

IN PARTICULAR, REFERENCING ROWS OF A MATRIX COULD BE MADE CONSIDERABLY MORE EFFICIENT BY TAKING ADVANTAGE OF MACHINE INSTRUCTIONS THAT ALLOW A ROW TO BE MOVED ESSENTIALLY AS A SINGLE ENTITY RATHER THAN ELEMENT BY ELEMENT.

BASED ON THESE CONSIDERATIONS, THE FOLLOWING OBJECTIVES WERE SET (AND REALIZED):

1) PROVIDE SPECIAL CASES FOR VECTORS FOR ALL DATA TYPES (INTEGER, CHARACTER, BOOLEAN AND FLOATING POINT). THE OLD INDEX PROVIDED SPECIAL CASES ONLY FOR INTEGER AND CHARACTER. THE FOLLOWING IS AN INDICATION OF THE IMPROVEMENTS:

BOOL[INT]	56°/° FASTER
FP[INT]	300°/° FASTER
INT[INT]	NO CHANGE
CHAR[INT]	NO CHANGE

2) PROVIDE SPECIAL ROUTINES FOR MATRIX INDEXING. EACH CASE IS INCLUDED IN THE FOLLOWING TIMING TABLE. <S> INDICATES A SCALAR AND <V> A VECTOR.

°/° FASTER

	INT	CHAR	BOOL	FP
M[S;S]	3	NO CHANGE	8	4
M[S;V]	17	24	11	13
M[S;]	252	308	411	58
M[V;S]	11	18	9	5
M[V;V]	37	36	21	36
M[V;]	534	1042	587	282
M[;S]	328	263	48	51
M[;V]	77	74	43	70

3) *MAKE THE GENERAL FETCH AND STORE ROUTINES HANDLE TRAILING SEMICOLONS THE SAME WAY AS IN THE MATRIX CASES. THIS MAKES THE MANIPULATION OF PLANES MUCH MORE EFFICIENT, AS INDICATED BY THE FOLLOWING.*

%/% FASTER

	INT	*CHAR*	*BOOL*	*FP*
A[S;;]	250	273	350	249
A[V;;]	350	445	400	510
A[;S;]	90	80	45	73

4) *GENERAL CLEANUP OF THE ALGORITHM ACHIEVING A CLEANER LOGICAL STRUCTURE, IMPROVED READABILITY, AND - BY APPLYING MODULAR STRUCTURE AND A USEFUL SET OF SUBROUTINES - MAKING IT EASY TO MODIFY OR IMPROVE IN THE FUTURE.*

WHILE THE IMPROVEMENT IN THE GENERAL CASE (WHERE THE ALGORITHM WAS NECESSARILY THE SAME) WAS UNDER 5%, THE IMPROVEMENT IN MANY COMMON CASES (PARTICULARLY WHERE ENTIRE ROWS, COLUMNS OR PLANES ARE FETCHED OR STORED) WAS SIGNIFICANT.

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

SATN 16
20 APR 76

SHARP APL TECHNICAL NOTES

TITLE: File System Must-Write Buffers.

ABSTRACT: A significant change has been made to the file system's use of memory buffers. The effect of the change is threefold:

1. Makes all file operations more cpu efficient,
2. Makes some special sequences of operations less efficient in elapsed time, and
3. Makes it **much** easier to design restartable systems.

KEYWORDS: Must-write buffers
Restartability
Data integrity
Efficiency

SHARP AR TECHNICAL NOTES

Previous to 22 March 1976 a □*APPEND* or □*REPLACE* to a file **often** resulted in the data simply being stored in a memory buffer without the file on disk being updated. The buffer would be marked as MW (must-write) and the system ensured that the buffer would be written to disk within 10 seconds. The advantage of this was that a sequence of small appends or replaces often resulted in simply adding data to the memory buffer without having to do writes to disk for each operation. The effect of this technique was most significant with frequent appends or sequential replaces of small components. With medium or large components the effect was much less. The support and maintenance of MW buffers was a cpu burden to all file operations.

As there was no control over the order in which MW buffers were written, a system crash could leave files in a state that could be **very** difficult to understand (or explain), making it extremely difficult to design systems which were restartable (or even had data integrity) after a crash. Many system designers thought they understood the implications of MW buffers, but in fact did not; this led to the dangerous position of critical systems which were considered to maintain data integrity and to be restartable after a crash, that in fact were not.

Perhaps the best way to "explain" the effect of MW buffers is to give an example.

```
    ∇F
[1]   'T' □CREATE 1
[2]   A □APPEND 1
[3]   B □APPEND 1
[4]   C □APPEND 1
[5]   D □REPLACE 1 1
[6]   E □REPLACE 1 2
[7]   F □REPLACE 1 3
    ∇
```

If execution of *F* finished and the system crashed within 10 seconds, the file *T* could contain almost anything. The following chart (where ~ indicates component does not exist) indicates several of the possibilities.

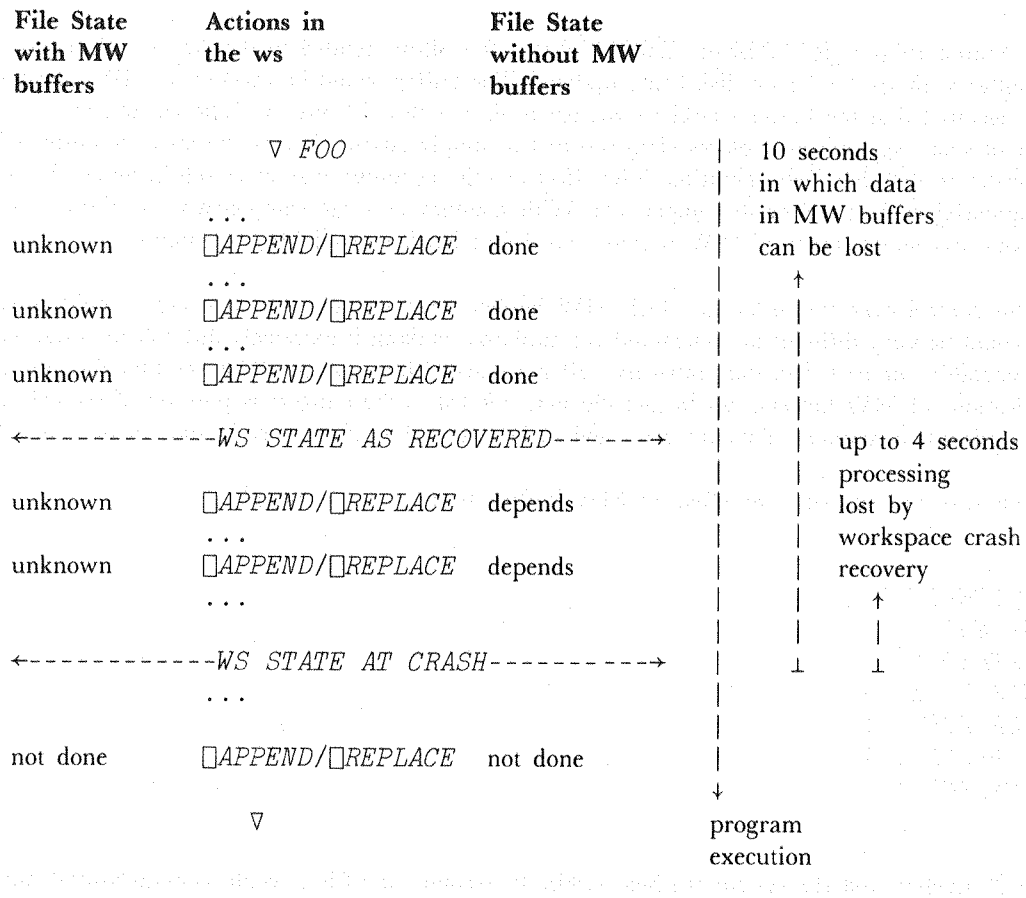
*CN*1	~	*A*	*D*	*A*	*A*	*A*	*D*	*D*
*CN*2	~	~	~	*B*	*E*	*B*	*B*	*B*
*CN*3	~	~	~	~	~	*F*	*F*	*C*

If applications involved several files with directories, the situation after a crash could be an inconsistent, unrecoverable mess.

Henceforth, the file system will **not** use MW buffers. This results in an overall decrease of at least 10% in cpu requirements for file operations, and in a sometimes significant increase in elapsed time for the special cases mentioned earlier. The most important result of not using MW buffers is that the state of files after a crash is much easier to understand. In the earlier example file *T* would simply contain the three components *D*, *E* and *F*.

Now, (without MW buffers) when a function has finished a file operation and is moving on to its next step, the operation is in fact completely finished and is on disk (i.e. it is not buffered in memory such that it will disappear in the event of a crash). This allows data integrity to be maintained, in that file operations are done in the order the system designer specifies, rather than in a random order as was the case with MW buffers.

Not using MW buffers also considerably simplifies the relationship between the workspace recovered after a crash and the files the workspace had been using. The following chart and notes indicate how the relationship is simplified.



NOTES: . . . indicates activity not affecting files.

"unknown" indicates the data from the append or replace may or may not be in the crash recovered file.

"depends" is different from "unknown" in that if one of them is shown to be done then all those before it are also done.

"unknown" operations could have been executed by the program as much as 10 seconds before the crash.

"unknown" or "depends" operations could have been executed by the program as much as 4 seconds after the workspace state as recovered.

With MW buffers it is necessary to trace both forward and backward from the recovered workspace state to resolve the file state, whereas without MW buffers it is only necessary to trace forward until an operation that was not done is found.

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

SATN 17
30 JUNE 76

SHARP APL TECHNICAL NOTES

TITLE: Formatting Primitive

ABSTRACT: The Formatting primitive ⍕ provides an easy and efficient means of converting numeric APL arrays to character. The function is useful in the manipulation of combined alphabetic and numeric data, and supplements some of the formatting and report generating facilities already available to SHARP APL users through ⎕*FMT* .

KEYWORDS: ⍕
Thorn
⎕*FMT*
Formatting
Format Control Vector
Format Phrase

SHARP AR TECHNICAL NOTES

INTRODUCTION

The format primitive function ⍕ (⊤ and ∘ overstruck, pronounced 'Thorn') provides a convenient way to form the character image of an APL array. Thorn may be used both monadically and dyadically. Its monadic definition formats an array and returns a ⎕*PP*-dependent result which, if printed, resembles the result of printing the array directly. When used dyadically, ⍕ uses its left argument to provide detailed control over the width and precision of its formatted result. The function enhances the format capabilities of ⎕*FMT*, as well as facilitating certain features already available. A detailed comparison of ⍕ and ⎕*FMT* is provided at the end of this SATN.

MONADIC USE

The syntax of monadic thorn is:

```
R← ⍕ A
```

Where the argument *A* may be any APL expression. The result *R* is a character array which, if printed, would appear identical to what would be printed if the expression itself were entered. The result may be displayed directly, or conveniently stored or massaged by other APL operators.

The result of ⍕ , unlike the corresponding terminal output of the same argument, is independent of the value of ⎕*PW* in the active workspace. While a vector expression *V* might take several lines to display, the result of ⍕*V* would simply be a long character vector, without embedded carrier returns. Monadic thorn is dependent on the setting of ⎕*PP* , however, just as terminal display is. For example:

```
      ⎕PP←10 ◇ ⍴⎕←⍕*1
2.718281828
11

      ⎕PP←16 ◇ ⍴⎕←⍕*1
2.718281828459045
17
```

```
      ⎕PP←3 ◇ ⍴⎕←⍕ 3000000.3 30000.003 300.00003 3.0000003
3E6  3E4  300  3
16
      ⍴⎕←⍕10
10
2
```

As illustrated, the result of formatting a numeric scalar is a character vector. If the argument is a numeric vector or array, then the result is a character image having the same rank as the argument. The shape of the argument and that of the result will be identical with the exception of the last dimension of the result, which is extended to hold the image of the argument plus additional spaces to separate values:

```
      A←? 4 4 ⍴9 ◇ ⍴⎕←A
  2  7  5  5
  2  1  7  7
  9  4  5  8
  1  1  5  7
4  4
      (,R≠' ')/,R
2755217794581157
```

```
      R←⍕A ◇ ⍴⎕←R
  2  7  5  5
  2  1  7  7
  9  4  5  8
  1  1  5  7
4  12
```

If the argument is already a character expression, then the result is simply the same as the argument. For example:

```
      A←'⍕  IS A FORMAT PRIMITIVE'
      A                                    ⍕A
⍕  IS A FORMAT PRIMITIVE          ⍕  IS A FORMAT PRIMITIVE
      ⎕AV∧.=⍕⎕AV
1
```

Thorn also makes it convenient to use catenation instead of heterogeneous output, in order to combine literal statements on the same line as numeric results:

```
      T←'THE ANSWER IS ' ◇ R←1.61
      T;R                                  T,⍕R
THE ANSWER IS 1.61                THE ANSWER IS 1.61

      (⍕T),⍕R
THE ANSWER IS 1.61
```

Using catenation has the added advantage of making the resulting expression explicitly available, so that it may be printed or used within another expression. For example, with T and R as above:

```
      C←1                                  C←0
      ⍴⎕←C/T,⍕R                            ⍴⎕←C/T,⍕R
THE ANSWER IS 1.61
18                                0
```

DYADIC USE

The syntax of dyadic thorn is:

```
R← FC ⍭ A
```

Where *A* may be any APL expression of numeric type. The result *R* contains a character array image of *A*, formatted according to the format controls designated by *FC*. Each vertical slice of the array *A* is formatted independently. Variations in the left argument can be used to provide varying degrees of control over the detail of the result.

The format control specifications must be either an integer vector of *N* pairs of numbers, or a single integer value. In general, integral pairs are used to control the result. The number of pairs given must equally divide ¯1↑⍴*A*; for example, 1 or ¯1↑⍴*A* pairs are acceptable values of *N*. The *I*th pair of numbers in *FC* is used to control the format of column *I* of the right argument, and every *N*th column after that.

The first element of a pair of numbers in *FC* is the width indicator. The width indicator determines the total width in the result of the image of a number field in *A*. The second element of a pair is the precision indicator. It controls the decimal-point precision of the image of a field in *A*. Several classes of image formats are available; these are chosen according to the sign of the precision indicator.

A positive precision indicator signifies fixed-point decimal format, the number of digits to the right of the decimal point being equal to the specified precision. This corresponds to the *F*-format phrase of ⎕*FMT*. For example:

```
P← 2 ¯8.3 64 ∘.* ¯1 2 3
9 1 ⍭P                          10 2 ⍭P
 ¯.5      4.0      8.0          ¯.50     4.00     8.00
 ¯.1     68.9   ¯571.8          ¯.12    68.89  ¯571.79
  .0   4096.0 262144.0            .02  4096.00 262144.00

'F9.1' ⎕FMT P
 ¯0.5      4.0      8.0
 ¯0.1     68.9   ¯571.8
  0.0   4096.0 262144.0
```

Note the difference that ⍭ does not provide a leading zero for fractional values, while an equivalent ⎕*FMT* expression does.

If the precision indicator is zero, each number is rounded to an integer. The decimal point itself is suppressed in the output:

```
9 0 ⍭P                          10 0 ⍭P
  1        4        8             1        4        8
  0       69     ¯572             0       69     ¯572
  0     4096   262144             0     4096   262144
```

This is analagous to the *I*-format phrase of ⎕*FMT* .

If the precision indicator is negative, then the corresponding field will be formatted using exponential (or scientific) notation, similar to ⎕*FMT*'*S E*-format. The absolute value of the precision indicator controls the minimum number of significant digits that will appear to the left of the exponent. This should be at least six smaller than the field width, to allow for a possible negative sign, a decimal point, and the characters of the exponent itself. For example:

```
9 ¯1 ⍭P                         10 ¯2 ⍭P
```

```
 5.0E¯1   4.00E0   8.00E0          5.00E¯1   4.000E0   8.000E0
¯1.2E¯1   6.89E1  ¯5.72E2        ¯1.20E¯1   6.889E1  ¯5.718E2
 1.6E¯2   4.10E3   2.62E5          1.56E¯2   4.096E3   2.621E5
```

If, in either fixed-point, integer, or exponential format, the width indicator is zero, then the width is automatically chosen to be whatever is required to represent the widest value among those controlled by that number pair, allowing for one leading blank. All other values formatted by that number pair are right-justified and padded on the left with as many spaces as are necessary to fill the remaining positions in the field:

```
      0 1 ⍕P                          0 ¯3 ⍕P
  .5     4.0     8.0          5.000E¯1  4.0000E0  8.0000E0
 ¯.1    68.9   ¯571.8        ¯1.205E¯1  6.8890E1 ¯5.7179E2
  .0  4096.0 262144.0          1.563E¯2  4.0960E3  2.6214E5
```

More detailed control over the result can be obtained by using a longer format control vector, which must be conformable in length with the array being processed, according to the rules stated earlier. For example:

```
     (6⍴ 0 1)⍕P
  .5    4.0      8.0
 ¯.1   68.9   ¯571.8
  .0 4096.0 262144.0
```

Note the difference between this and the previous fixed-point example. In the first case, a single number pair was used to format all three columns of the argument. Hence all columns in the result have the same width, chosen to be long enough to accomodate the widest number. In the second example, each column was formatted using its own width and precision indicators. Thus the width of each column was chosen independently of the others.

The following examples further illustrate the use of selective format control on the columns of the argument:

```
      0 3 11 ¯3 20 0 ⍕P
 .500   4.0000E0                      8
¯.120   6.8890E1                   ¯572
 .016   4.0960E3                 262144

      (18⍴ 8 2 5 0 9 ¯2 3 0 0 0 10 4)⍕,P

    .50    4  8.000E0  0 69 ¯571.7870      .02 4096  2.621E5

      0 2 5 0 ⍕⍕10⋆ 6 2 0 1
3141592.65  314       3.14   31
```

Note that, in the last example, the third member of the argument was formatted according to the automatic width chosen to accomodate the larger first value, as both these elements were controlled by the same number pair.

If only a single number is used as the format control, it is treated as the precision indicator of a number pair with a width indicator of zero. For example, if *FC* is zero, the result will be formed with an automatic width and integer format:

```
      0⍕P
      1       4       8
      0      69    ¯572
      0    4096  262144

      ∧/,(3⍕P)= 0 3 ⍕P
1
```

If the width indicator for a column is inadequate to hold a member of that column, then the corresponding image in the result will be filled with asterisks:

```
       7 3 ⍕P
  .500  4.000  8.000
¯ .120 68.890*******
  .016**************
```

On higher dimensional arrays, the format control vector applies to each of the planes defined by the latter two coordinates of the array. For example:

```
     (,6,[1.5] ⍳3)⍕? 2 2 6 ⍴2
  1.0  2.00 1.000   2.0  1.00 1.000
  2.0  2.00 2.000   1.0  2.00 2.000

  1.0  1.00 2.000   2.0  1.00 1.000
  1.0  1.00 2.000   2.0  2.00 2.000
```

A specified width indicator need not provide for separating blanks between adjacent fields in the result. For example, a matrix may be formatted closely-packed if desired:

```
      L←? 4 20 ⍴4
      1 0 ⍕L                              1 0 ⍕L=2
14231133423411331212              00100000010000000101
33443132344213234421              00000001000100100010
43431342242122124414              00000001101011010000
33242214132243241444              00101100001100100000
```

If a non-zero precision indicator specifies more significant digits than are available in the internal representation of a value being formatted, then the indeterminate spurious positions will be filled with underscores:

```
      J← 1 3 100 1000 ∘.×⋆,1
      25 20 ⍕J                          25 ¯20 ⍕J
   2.7182818284590451____            2.7182818284590451_____E0
   8.154845485377135_____            8.154845485377135______E0
 271.82818284590451______            2.7182818284590451_____E2
2718.2818284590451_______            2.7182818284590451_____E3
```

Thorn can be used to conveniently format tabular output, incorporating row and column headings if desired. For example:

```
      DATA←0.01× 5 5 ⍴-/? 25 2 ⍴50000

      C←' ',[1] (5 7 ⍴'TRIAL -'), 2 0 ⍕ 5 1 ⍴⍳5
      R←2⍬, 5 ¯10 ↑ 5 1 ⍴'ABCDE'
      C,R,[1] (,10,[1.5] 2 0 3 3 2)⍕DATA

              A       B       C       D       E
TRIAL - 1 ¯312.04    ¯37  ¯85.950    ¯.210  275.59
TRIAL - 2 ¯155.78     ¯9  ¯70.720 ¯187.860 ¯175.32
TRIAL - 3   48.90     42  217.480  118.960 ¯104.57
TRIAL - 4  249.87   ¯344 ¯152.200 ¯117.310   59.15
TRIAL - 5  129.94     51 ¯279.470  306.000 ¯165.04
```

OTHER FORMAL CHARACTERISTICS

In addition to the definitions already provided, ⍕ possesses some other relevant characteristics. While these will likely not concern the general user, they may be of interest in certain applications, and are included here for completeness.

1. If the width specification of the Ith number pair of N number pairs is zero, and the corresponding precision indicator is P, then the elements of the data columns of array A controlled by that number pair are given by:

   ```
   C←,((¯1↑⍴A)⍴I=⍳N)/A
   ```

   Further, the smallest width required to represent the chosen columns is:

   ```
   for P≥0:   P+(C∨.<0)+(P≠0)+⌈/0,1+⌊10⍟C+C=0
   for P<0:   4+(C∨.<0)+(P≠¯1)+|P
   ```

2. The expressions `(FC1,FC2)⍕A,B` and `(FC1⍕A),FC2⍕B` are equivalent by the distributive law if `FC1` and `FC2` are full control vectors and completely specify each column of A and B, respectively. Additionally, if FC is a scalar or a vector with `(⍴FC)∊ 1 2`, then the expressions `FC⍕A,B` and `(FC⍕A),FC⍕B` are equivalent.

3. When a precision indicator requires that the low-order significant digits of the data to be formatted be discarded, then the affected values are rounded by increasing their magnitude by five in the leftmost digit position being discarded. Low-order digits close to five may be rounded either up or down, due to possible inaccuracies associated with the inexact internal representation of decimal fractions in computers.

COMPARISON WITH ⎕FMT

The format capabilities of ⍕ and ⎕FMT overlap to a certain extent. There are, however, some important differences between the two facilities:

1. ⍕ is generally more cpu-efficient than a comparable use of ⎕FMT to generate the same result.

2. ⍕ cannot format more than one argument at a time, while ⎕*FMT* can take a list (with objects separated by semicolons) as its right argument.

3. Data to be formatted using ⎕*FMT* is restricted to rank two or less; ⍕ is capable of handling arrays of any rank.

4. The format controls of ⍕ are specified by numeric scalars or vectors, while ⎕*FMT* format statements are character type.

5. ⍕ has no facilities corresponding to the qualifiers and decorations of ⎕*FMT*. There is also no equivalent to ⎕*FMT*'s filler, positioning, and character codes if using thorn.

6. ⎕*FMT* field widths are determined explicitly by the components of the format phrase used as its left argument. Hence, it is not directly capable of performing operations equivalent to the monadic use of ⍕, or to the dyadic use with a width indicator of zero.

7. The result of ⎕*FMT* is always a matrix, while the result of ⍕ can be a vector, matrix, or higher-dimensional array, depending on the rank of the data being formatted.

8. A non-explicit call to ⎕*FMT*, where the result is not stored or manipulated, reduces the chances of *WS FULL* reports because the formatted output is printed as it is computed, and not built up in the workspace. ⍕ always returns its full result to the workspace, and is thus more susceptible to overhead problems in certain situations.

ERROR MESSAGES

Incorrect use of ⍕ can result in certain error messages. These are outlined below, for both the monadic and dyadic uses of the operator:

1. **Monadic**: *R*← ⍕ *A*

WS FULL	The result of ⍕ is too large for the active workspace to hold.

2. **Dyadic**: *R*← *FC* ⍕ *A*

RANK ERROR	*FC* is a matrix or a higher-dimensional array.
LENGTH ERROR	The number of pairs of integers in *FC* is not a multiple of the last dimension of *A*.
DOMAIN ERROR	Either or both arguments are character type; a precision indicator is not an integer value; or a specified width indicator is negative or nonintegral.
NONCE ERROR	A specified or implicit width indicator is larger than 255 .
WS FULL	The result of ⍕ is too large for the active workspace to hold.

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

SATN 18
1 JUL 76

SHARP APL TECHNICAL NOTES

TITLE: □*FMT*

ABSTRACT: This SATN describes the following enhancements to the SHARP APL report formatting facility (□*FMT*):

1) Use of parentheses to simplify coding of repetitive format phrases;
2) *K* qualifier to scale data before formatting;
3) *G* format phrase (formatting via a pattern specification);
4) Alternative text delimiters to ⍞;
5) *S* qualifier to allow substitution of user - specified characters for standard □*FMT* symbols;
6) Generalization of *X* format phrase; *T* format phrase to allow absolute column tabbing.

KEYWORDS: □*FMT*
Parentheses
K qualifier
G Format
Delimiters
S Qualifier
X Format
T Format
Tabbing

1) PARANTHESES

The left argument to □*FMT* may contain parentheses to group repetitive format phrases. This facility simplifies the coding requirements of the left argument and enables the user to control the scanning of the left argument when the data is exhausted.

The format phrase to be repeated is coded in the normal fashion and enclosed in parentheses. The desired repetition factor is then coded immediately to the left of the left parenthesis. An omitted repetition factor is equivalent to 1. Parentheses may be nested to a maximum of eight levels.

Example 1.1

```
'LI5,X3,F8.2,X3,F8.2,X3,F8.2'
```

may be simplified to

```
'LI5,3(X3,F8.2)'
```

When the data in the right argument has been exhausted, non-editing format phrases (*X*,*T*, literal text) are processed until any of the following is encountered:

a) An edit code (*A E F G I*) with a non-zero repetition factor;
b) A left parenthesis;
c) A right parenthesis with an associated repetition factor which is not yet exhausted;
d) The end of the argument.

The user may insert otherwise superfluous parentheses to control the insertion of literal text, thus avoiding having to trim the result.

Example 1.2

```
X←□TS[1 3ρ□IO+2 1 0]
```

Without employing parentheses in the left argument, a date in the form DD/MM/YY may be generated by:

```
0 ¯1↓'ZI2,□/□' □FMT X
```

However, by using appropriate parentheses, the user can construct a simpler, equivalent statement:

```
'ZI2,(□/□)' □FMT X
```

2) SCALING OF DATA (*K* Qualifier)

The *K* qualifier may be used to treat numeric data as if it had been multiplied by an integral power of 10. This scaling is performed internally by □*FMT* and eliminates the need to perform arithmetic on the right argument to □*FMT*. The syntax for the qualifier is:

Kω

where ω is an integer (negative, zero or positive).

The effect of the *K* qualifier is the display of the data as if it had been multiplied by 10*⍵. For ease of coding, negative integers may be denoted by either a high minus (¯⍵), or by a middle minus (-⍵).

Example 2.1

```
      'K¯3F8.2' ⎕FMT 1234.5
1.23

      'K2I6' ⎕FMT 8.76
876
```

3) PATTERN CONTROLLED FORMATTING (*G* FORMAT PHRASE)

The *G* format phrase allows the user to specify a pattern which is to be used to control the display of the data. The required syntax of the *G* format phrase is:

```
⍺ ⍵ G⍞ PATTERN TEXT ⍞
```

where ⍺ is an optional repetition factor and ⍵ optional qualifiers. The pattern text may contain characters from the following classes:

(1) '*DIGIT SELECTORS*' (9 *OR* Z);
(2) '*MESSAGE TEXT*' (any characters other than digit selectors, or the right delimiter of the pattern text).

The result of the *G* format phrase is the same length as the pattern text. In general, digit selectors cause the corresponding digits of the data to be mapped to the result; message text acts as decorative text.

The following rules apply to the generation of the result:

[*G*1] Before formatting, the data is scaled by the (optional) *K* qualifier and rounded to an integer;

[*G*2] A 9 digit selector causes the corresponding data digit to be mapped to the result;

[*G*3] A Z digit selector corresponding to a non-zero data digit behaves as a 9 digit selector;

[*G*4] A Z digit selector corresponding to a zero data digit causes a zero to appear in the result if and only if digits appear on both the left and the right - this amounts to leading and trailing zero suppression;

[*G*5] The integer is padded with leading zeroes if necessary;

[*G*6] If there are fewer digit selectors than data digits, field overflow occurs and the result is filled with asterisks (*);

[*G7*] Leading or trailing message text is mapped directly to the result;

[*G8*] Message text appearing between digit selectors is mapped to the result if and only if digits are selected on both sides of the message text.

The following additional rules are concerned with the use of *M N P Q R* qualifiers in conjunction with the *G* format phrase. The application of the rules above is carried out before considering the effects of optional qualifiers.

[*G9*] *M* or *P* text is inserted immmediately to the left of the leftmost digit selected;

[*G10*] *N* or *Q* text is inserted immediately to the right of the rightmost digit selected;

[*G11*] *R* text is inserted in the result only in positions which were left unchanged by the pattern and other qualifiers;

[*G12*] *M N P Q* text is not inserted in the result if the integer is zero and the pattern did not contain at least one 9 digit selector;

[*G13*] Sufficient digit selectors or message text must be included in the pattern to accomodate *M N P* or *Q* text or the presence of a negative sign;

[*G14*] Use of *B C L* or *Z* qualifiers is not permitted with the *G* format phrase.

Example 3.1

```
      'G⍞(999) 999-9999⍞' ⍞FMT 4163632051 4163645361
(416) 363-2051
(416) 364-5361


      'G⍞99/99/99⍞' ⍞FMT I25
06/16/76


      'K2G⍞Z,ZZZ,ZZZ DOLLARS AND 99 CENTS ⍞' ⍞FMT 1874.38
    1,874 DOLLARS AND 38 CENTS
```

4) TEXT DELIMITERS

The left argument to ⎕*FMT* often must contain character strings which necessarily are delimited by a character reserved for that purpose. In particular, literal text, *G* format phrases, *M N P Q R* decorators and the *S* qualifier, all incorporate character strings. In addition to ', the following (pairs of) characters may be validly used as string delimiters:

< > (less than, greater than)
⊂ ⊃ (subset, superset)
¨ ¨ (dieresis)
⎕ ⎕ (quad)

The delimiter chosen to indicate the start of the string implies the delimiter **required** to indicate the end of the string. The enclosed characters may include any character other than the required ending delimiter.

Example 4.1

```
'<BALANCE FORWARD: >,F8.2'          is equivalent to

'⎕BALANCE FORWARD: ⎕,F8.2'
```

5) STANDARD SYMBOL SUBSTITUTION (*S* QUALIFIER)

There are several characters which are used by ⎕*FMT* for standard data representation or specific required syntax. These standard symbols are identified in the following table, along with the standard use of the symbol and the applicable format phrase(s).

Symbol	Use	Format phrases
9	Digit selector	*G*
Z	Digit selector with leading and/or trailing zero suppression	*G*
*	Field overflow fill character	*F G I*
.	Decimal point	*F*
,	*C* Qualifier insert character	*F I*
_	Non-significant digit position marker	*F G I*
0	Z Qualifier fill character	*F I*
0	Leading zero fill character	*G*

The *S* qualifier allows the user to specify substitute characters for the above standard symbols. The syntax of the *S* qualifier is:

```
S⎕ PAIRS OF CHARACTERS ⎕
```

Each pair of characters defines a substitution in this manner: the first character of a pair must be a standard symbol from the table above; the second character of a pair is a character chosen by the user to be substituted for the symbol.

The following additional rules apply:

[*S1*] Symbol pairs need only be specified for standard symbols which require substitution:

[*S2*] The pairs may be in any order:

[*S3*] There must be an even number of characters between delimiters:

[*S4*] Only one substitution per standard symbol is allowed (i.e. '*S*␣9*X**09*Y*␣' is invalid since the '9' appears twice):

[*S5*] '9' and 'Z' may not be replaced by the same character.

Example 5.1

(Interchange Role of Decimal and Comma)

```
      'S␣.,,.␣CF15.2' ␣FMT  1024.86 ¯1618033.9 1234
      1.024,86
¯1.618.033,90
      1.234,00
```

Example 5.2

(Segregate Fractional Portion of Numbers)

```
      'S␣.|␣CF15.2' ␣FMT 1024.86 ¯1618033.9 1234
      1,024|86
¯1,618,033|90
      1,234|00
```

Example 5.3

(Free '9' and 'Z' For Use in Message Text)

```
      'S␣98Z?␣ G␣TOTAL SIZE 9 SHOES: 888␣' ␣FMT 309
TOTAL SIZE 9 SHOES: 309
```

Any otherwise unused character could have been substituted for 'Z'. The superfluous blank preceding *G* improves readability.

6) COLUMN SKIPPING AND TABBING (*X* AND *T* FORMAT PHRASES)

The *X* and *T* format phrases allow the user to specify the result column where the display of the next formatted field will begin. The *X* format phrase provides column skipping relative to the current column in the result; the *T* format phrase provides column skipping to an origin 1 absolute column number.

SYNTAX OF *X* FORMAT PHRASE

$\alpha X \omega$

α is optional repetition factor (≥ 0)
ω is integer (negative, zero or positive)

The effect of an *X* format phrase is to cause the display of the next field to begin $|\alpha \times \omega$ columns to the left or right of the current result column. The direction of the skip is determined by the sign of ω ; negative values cause a skip to the left, positive values cause a skip to the right. For ease of coding, negative values of ω may be denoted by high minus ($\bar{}$) or middle minus (-)

SYNTAX OF *T* FORMAT PHRASE

$\alpha T \omega$

α is optional repetition factor (≥ 0)
ω is optional absolute column number (origin 1)

The effect of the *T* format phrase is to cause the display of the next field to begin at column ω. If the column number is omitted in the format phrase, the display of the next field will begin in the column following the rightmost column previously specified.

The *X* or *T* format phrase may specify skipping to any column of the result. $\square FMT$ will extend each row of the result (with blanks) as required. When the result of the current format phrase overlaps a previous result, the characters from the previous result are overlaid **unless** the overlay character is a blank resulting from left or right justification or use of the *B* or *L* qualifiers.

For example:

```
      'I5, T1, I5' ⎕FMT 1 2⍴12345 99
12399
```

(The three blanks to the left of '99' are the result of right justification and therefore do not overlay the '123').

Example 6.1

(To conveniently place data in specific result columns)

```
Q←3 1⍴ 132 756 459                                  ⍝ QUANTITY
P←3 1⍴ 0.48 6.79  6.8                               ⍝ UNIT PRICE


'T10,I5,T25,F8.2,T45,CF10.2' ⎕FMT (Q;P;Q×P)
  132                     0.48                    63.36
  756                     6.79                 5,133.24
  459                     6.80                 3,121.20
```

Example 6.2

(To process right argument in arbitrary sequence)

```
I←2 3⍴ 418 687 589 931 847 527                      ⍝ INTEGERS
R←2 3⍴ 0.92 6.54 4.16 7.02 9.11 7.63                ⍝ REALS


     '3(I5,X10),T6,3(F10.2,(X5))' ⎕FMT (I;R)
418         0.92  687         6.54  589         4.16
931         7.02  847         9.11  527         7.63
```

(i.e. interleaves columns)

Example 6.3

(To superimpose text)

```
     'F7.2,X¯3,⍞:⍞,T' ⎕FMT 1 4⍴10+0.15×0 1 2 3
10:00  10:15  10:30  10:45
```

Note the use of *T* without a column number to ensure the next field display does not overlap.

I. P. Sharp Associates Limited Suite 1400, 145 King Street West, Toronto, Ontario M5H 1J8 • (416) 364-5361

SHARP APL TECHNICAL NOTES

TITLE: *FILEPRINT*

ABSTRACT: *INFORMATION STORED IN AN APL FILE MAY BE PRINTED ON A HIGHSPEED PRINTER AT THE COMPUTER CENTRE OR WRITTEN TO MAGNETIC TAPE.*

NOTE: USE OF HSPRINT (SATN-8) IS STRONGLY RECOMMENDED INSTEAD OF FILEPRINT. DEVELOPMENT ON FILEPRINT IS FROZEN. THIS SATN IS PROVIDED TO ASSIST IN MAINTENANCE OF SYSTEMS WHICH HISTORICALLY HAVE USED FILEPRINT.

KEYWORDS: *WS 1 FILEPRINT*
PRINTREQ
AUTOPREQ
HSPRINT

Мои домашние задания

INFORMATION STORED IN AN APL FILE MAY BE PRINTED ON THE HIGH SPEED PRINTER AT THE COMPUTER CENTRE, OR WRITTEN TO MAGNETIC TAPE IN EBCDIC WITH ASA CARRIAGE CONTROL AND A RECORD LENGTH OF 133. PAGE LENGTH AND LINE SPACING CAN BE VARIED TO ACCOMODATE SPECIAL FORMS LIKE INVOICES OR ADDRESS LABELS. PAGE TITLES AND SUBTITLES CAN BE GENERATED AUTOMATICALLY, WITH OR WITHOUT PAGE NUMBERS, TIME AND DATE. YOU MAY REQUEST PRINTING OF FILES AT ANY TIME FROM YOUR TERMINAL OR APL PROGRAM; PRINTING IS DONE WHEN THE HIGH SPEED PRINTER IS AVAILABLE.

A FILE TO BE PRINTED IS BUILT BY APPENDING COMPONENTS THAT ARE CHARACTER-VALUED SCALARS, VECTORS OR MATRICES. WHEN THE FILE IS PRINTED EACH SCALAR OR VECTOR IS IS PRINTED ON A SINGLE LINE, AND EACH MATRIX IS PRINTED ON AS MANY LINES AS THE MATRIX HAS ROWS. THE LAST ROW OF ONE COMPONENT IS FOLLOWED IMMEDIATELY BY THE FIRST ROW OF THE NEXT COMPONENT; THE GROUPING OF INFORMATION INTO COMPONENTS HAS NO EFFECT ON THE PRINTED RESULT. PRINT LINE LENGTH IS LIMITED TO 132 POSITIONS. THE FIRST 132 CHARACTERS OF LONGER ROWS ARE PRINTED NORMALLY. THE REMAINING DATA IS PLACED ON SUBSEQUENT LINES INDENTED BY SIX POSITIONS FROM THE LEFT MARGIN. IN GENERAL, CHARACTER DATA WILL APPEAR ON THE HIGHSPEED PRINTER JUST AS IT WOULD ON A STANDARD APL TERMINAL. THIS INCLUDES BACKSPACES, USER-CONSTRUCTED OVERSTRIKES, IDLES AND LINEFEEDS.

NUMERIC FILE COMPONENTS ARE USED ONLY TO CONTROL PAGE FORMAT OR TYPE FONT SELECTION AND CANNOT BE PRINTED DIRECTLY. THE ⎕FMT SYSTEM FUNCTION (SATN-18) PROVIDES A PARTICULARLY EASY WAY TO CONVERT NUMERIC VALUES TO CHARACTER VALUES SUITABLE FOR PRINTING.

SUBMITTING A PRINT REQUEST

A PRINT REQUEST IS SUBMITTED BY EXECUTING EITHER <PRINTREQ> OR <AUTOPREQ> FROM WS 1 FILEPRINT. INTERACTIVE USERS SHOULD USE <PRINTREQ> . USERS WHO WISH TO AUTOMATE THEIR SUBMISSIONS SHOULD USE <AUTOPREQ> .

WHEN A PRINT REQUEST HAS BEEN SUCCESSFULLY PRINTED THE BATCH PROGRAM WILL ⎕READ AND ATTEMPT TO ⎕REPLACE THE FIRST COMPONENT OF THE PRINT FILE, SO THAT THE USER CAN DETERMINE VIA ⎕RDCI, IF AND WHEN THE FILE WAS PRINTED.

USER 1000 (BATCH) MUST HAVE SUITABLE ACCESS TO THE PRINT FILE. IF THERE IS NO ENTRY FOR USER 1000 IN THE ACCESS MATRIX BOTH <PRINTREQ> AND <AUTOPREQ> WILL SET ⎕TIE, ⎕READ, ⎕REPLACE, AND (IF ERASE IS SPECIFIED) ⎕ERASE PERMISSION. IF THERE IS ANY ENTRY FOR USER 1000 THEN NO CHANGE IS MADE TO THE ACCESS MATRIX.

USING <PRINTREQ>

THE SYNTAX OF THIS FUNCTION IS < R←PRINTREQ >. FOR SUCCESSFUL SUBMISSIONS THE RESULT RETURNED WILL BE A CHARACTER VECTOR OF THE FORM:

REQ. NO. <REQUEST NUMBER> FILED <TIMESTAMP>

WHERE <REQUEST NUMBER> IS THE REFERENCE NUMBER OF YOUR REQUEST AND <TIMESTAMP> IS THE DATE OF SUBMISSION. FOR EXAMPLE:

REQ. NO. 12345 FILED 10.52.15 25 SEP 1976.

IN ALL OTHER CASES THE RESULT WILL BE AN EMPTY CHARACTER VECTOR.

THE <PRINTREQ> FUNCTION PROMPTS YOU FOR INPUT. THE FOLLOWING IS A DISCUSSION OF THESE PROMPTS AND VALID RESPONSES. SEE ALSO EXAMPLES 1 AND 2.

1) <FILE NUMBER>. IF A FILE IS TIED TO THE GIVEN NUMBER, 'PRINT REQUEST FOR <FILE ID>' IS DISPLAYED AND THE NEXT PROMPT IS ISSUED. IF AN INVALID NUMBER IS ENTERED OR NO FILE IS TIED TO THE GIVEN NUMBER THE MESSAGE 'INVALID FILE NUMBER' IS DISPLAYED AND THE FUNCTION EXITED.

2) <PRINT SPECS>. SIX PARAMETERS ARE REQUIRED FOR COMPLETE SPECIFICATION, AS FOLLOWS:

PARAMETER	RESPONSE
FORMS TYPE	CON1 CON2 STD1 STD2 STD3 STD4 STD5 REV1 ULD1 ULD2 SPEC TPN (WHERE N IS AN INTEGER AND 1≤N≤30)
PRINT TRAIN	FAST FULL PN2
LINES PER INCH	LNS6 LNS8
DECOLLATE	DECO NODE
ERASE	NOER ERAS
REPEATS	RPN (WHERE N IS AN INTEGER AND 1≤N≤10)

DEFAULT VALUES OF <STD1 FAST LNS6 DECO NOER RP1> ARE ESTABLISHED FOR THESE PARAMETERS WHEN THE PROMPT IS ISSUED AND THEY REMAIN IN EFFECT UNTIL OVERRIDDEN BY YOUR INPUT.

RESPONSES MAY BE ENTERED IN ANY ORDER, MAY OCCUR ANY NUMBER OF TIMES (ONLY THE LAST ENTERED FOR EACH PARAMETER IS USED), ARE DELIMITED BY BLANKS OR COMMAS, AND ONLY THE FIRST FOUR CHARACTERS ARE SIGNIFICANT.

INVALID RESPONSES WILL BE DISPLAYED IN THE FORM

INVALID : <LIST OF INVALID RESPONSES>

AND THE PROMPT 'PRINT SPEC: ' IS REISSUED, THUS RESETTING ALL PARAMETERS TO THEIR DEFAULT VALUES.

A RESPONSE OF <EMPTY LINE> (RETURN, TAB RETURN, OR SPACE RETURN) TO THE PROMPT WILL ESTABLISH DEFAULT PRINT SPECS.

USE AND MEANING OF EACH PARAMETER

A) FORMS TYPE. THE COMPUTER CENTRE MAINTAINS A SUPPLY OF CERTAIN FREQUENTLY USED FORMS TYPES. THESE ARE:

CON1	1 PART CONSOLE PAPER
CON2	2 PART CONSOLE PAPER
STD1	1 PART LINED STOCK TAB PAPER
STD2	2 PART LINED STOCK TAB PAPER
STD3	3 PART LINED STOCK TAB PAPER
STD4	4 PART LINED STOCK TAB PAPER
STD5	5 PART LINED STOCK TAB PAPER
REV1	REVERSED (BACK OF) 1 PART LINED STOCK TAB PAPER
ULD1	1 PART UNLINED STOCK TAB PAPER
ULD2	2 PART UNLINED STOCK TAB PAPER
SPEC	ANY FORMS TYPE NOT ONE OF THE ABOVE. SPECIAL FORMS SHOULD BE SUPPLIED BY THE USER, AND THE <COMMENTS> PROMPT (SEE BELOW) WILL BE TRIGGERED.
TP1M	FOR 2780 TRANSMISSIONS
TPN	WRITE TO MAGNETIC TAPE FOR LATER OFF-LINE TRANSMISSION OR SHIPMENT. BLOCK TAPE <N> PRINT LINES PER PHYSICAL TAPE RECORD. WARNING: USERS SHOULD BE AWARE OF THE CHARACTERISTICS OF THE TRANSMISSION DEVICES IN ORDER TO ENSURE THAT THE SPECIFIED BLOCKING FACTOR IS WITHIN THE CAPABILITY OF THE DEVICE.

THE DEFAULT SETTING IS <STD1>.

B) PRINT TRAIN. THREE PRINT TRAINS ARE AVAILABLE AT THE COMPUTER CENTRE; FAST, FULL, AND PN2. (SEE SECTION ON CONTROLLING TYPEFONT SELECTION.) THE DEFAULT SETTING IS <FAST> . DEFAULT TRANSLATE TABLES WILL AUTOMATICALLY BE PROVIDED BY THE BATCH PROGRAM IF THIS PARAMETER IS <FULL> OR <PN2>.

C) LINES PER INCH. THE LINE PRINTER WILL PRINT WITH A VERTICAL DENSITY OF 6 OR 8 LINES PER INCH. (SEE SECTION ON CONTROLLING PAGE FORMAT). THE DEFAULT SETTING IS <LNS6> - 6 LINES PER INCH.

D) DECOLLATE. THE OPERATIONS STAFF WILL DECOLLATE THE OUTPUT (REMOVE THE CARBON) ON MULTIPLE PART FORMS, IF <DECO> IS SPECIFIED. <DECO> IS THE DEFAULT SETTING.

E) ERASE. AFTER THE FILE HAS BEEN PRINTED SUCCESSFULLY IT CAN BE AUTOMATICALLY ERASED ON YOUR BEHALF IF YOU SO REQUEST. THE DEFAULT SETTING IS <NOER>.

F) REPEATS. PRINT THE FILE <N> TIMES. THIS FEATURE SHOULD BE USED WHEN <N> ORIGINALS ARE REQUIRED. THE DEFAULT SETTING IS <RP1>.

TWO OTHER VALID RESPONSES ARE <QUIT> AND <COMM>. <QUIT> MEANS DO NOT SUBMIT A PRINT REQUEST AND EXIT THE FUNCTION. <COMM> WILL TRIGGER THE <COMMENTS> PROMPT - SEE BELOW.

3) <DELIVERY INSTRUCTIONS>. THE EXPECTED RESPONSE IS THE DELIVERY ADDRESS (DELIVER TO) AND DELIVERY METHOD (DELIVER VIA). NO VERIFICATION IS DONE ON THIS DATA. AN <EMPTY LINE> ENDS THIS PROMPT. HOWEVER, AT LEAST ONE LINE OF DATA MUST BE ENTERED.

REQUESTS SUBMITTED WITH UNCLEAR DELIVERY INSTRUCTIONS WILL BE PRINTED, THE SUBMITTOR WILL BE BILLED, AND THE OUTPUT HELD AT THE COMPUTER CENTRE UNTIL CALLED FOR OR UNTIL THE COMPUTER STAFF DISCARDS IT.

4) <COMMENTS>. THIS PROMPT IS ONLY ISSUED
A) IF THE FORMS TYPE WAS <SPEC> OR
B) IF THE RESPONSE <COMM> WAS INCLUDED IN THE PRINT SPECIFICATION.

FOR FORMS TYPE <SPEC> THE RESPONSE SHOULD BE UNAMBIGUOUS INSTRUCTIONS TO THE OPERATIONS STAFF DETAILING THE SPECIAL FORMS TO BE USED AND, WHERE NECESSARY, ALIGNMENT INFORMATION.

FOR THE <COMM> OPTION IN PRINT SPECS THE RESPONSE SHOULD BE SUPPLEMENTARY USEFUL INFORMATION FOR THE OPERATIONS STAFF TO SUCCESSFULLY COMPLETE YOUR PRINT REQUEST. PLEASE NOTE THAT THIS OPTION MUST NOT BE USED FOR DELIVERY INSTRUCTIONS.

AN EMPTY LINE ENDS THE <COMMENTS> PROMPT.

WARNING: THE TOTAL NUMBER OF TEXT LINES ENTERED FOR DELIVERY INSTRUCTIONS AND COMMENTS MAY NOT EXCEED 26 AND THE WIDTH OF EACH LINE MAY NOT EXCEED 130 CHARACTERS.

5) WHEN THE REQUEST HAS BEEN ENTERED THE USER'S PRINT FILE IS UNTIED AND THE FUNCTION TERMINATES.

THE FOLLOWING ARE TWO EXAMPLES USING THE <PRINTREQ> FUNCTION. IN BOTH CASES IT IS ASSUMED THAT THE FILE <1234567 PRINTFILE> IS TIED TO FILE TIE NUMBER 5. NOTE THAT USER RESPONSES BEGIN IMMEDIATELY TO THE RIGHT OF, AND ON THE SAME LINE AS, THE PROMPT.

EXAMPLE 1

```
      )LOAD 1 FILEPRINT
SAVED   16.04.10 02/20/75
      PRINTREQ
FILE NUMBER: 5
PRINT REQUEST FOR - 1234567 PRINTFILE
PRINT SPECS: <EMPTY LINE>
DELIVERY INSTRUCTIONS:
ROGER D. MOORE
I.P. SHARP ASSOCIATES LIMITED
SUITE 1400
145 KING STREET WEST
TORONTO M5H 1J8.
PLEASE MAIL WHEN COMPLETED.
<EMPTY LINE>        (TO END PROMPT)
REQ. NO. 12345 FILED  10.52.15  25 OCT 1976
```

IN THIS EXAMPLE THE USER REQUESTED THE DEFAULT PRINT SPECS (<EMPTY LINE> AFTER THE PROMPT <PRINT SPEC:>). NOTE ALSO THE LAST LINE DISPLAYED.

EXAMPLE 2

```
      X←PRINTREQ
FILE NUMBER: 5
PRINT REQUEST FOR - 1234567 PRINTFILE
PRINT SPECS: STD3,ERASE
DELIVERY INSTRUCTIONS:
ROGER D. MOORE
I.P. SHARP ASSOCIATES LIMITED
SUITE 1400
145 KING STREET WEST
TORONTO, ONTARIO M5H 1J8.
CALL 364-5361 EXT 276 WHEN READY AND HE WILL PICK UP.
<EMPTY LINE>
```

IN THIS EXAMPLE STOCK TAB LINED 3 PART PAPER, DECOLLATED, WAS REQUESTED AND THE FILE WAS TO BE ERASED AFTER PRINTING. THE DEFAULT VALUES WERE ASSUMED FOR PRINT TRAIN AND LINES PER INCH. NOTE THE ABSENCE OF THE LAST DISPLAY LINE SINCE THE RESULT WAS ASSIGNED TO X IN THE FUNCTION CALL.

ERRORS
------

1) *WS FULL - PRINT REQUEST NOT SUBMITTED.*

2) *SYMBOL TABLE FULL - PRINT REQUEST NOT SUBMITTED.*

FOR BOTH THE ABOVE ERROR MESSAGES THE FUNCTIONS RETURN RESULTS FOR NON-SUBMISSION (0 FOR <AUTOPREQ> AND '' FOR <PRINTREQ>).

3) *UNABLE TO SUBMIT PRINT REQUEST - PLEASE CONTACT APL OPERATIONS IMMEDIATELY.*

USING <AUTOPREQ>

THE SYNTAX OF THIS FUNCTION IS:

```
      R←A AUTOPREQ B
```

THE LEFT ARGUMENT MUST BE THE FILE TIE NUMBER OF THE FILE TO BE PRINTED. THE RIGHT ARGUMENT MUST BE A CHARACTER VECTOR OF THE FORM:

<PRINTSPECS><DELIM><DELIVERY INSTRUCTIONS><DELIM><COMMENTS>

WHERE:

<PRINTSPECS> IS A STRING OF RESPONSES THAT WOULD BE ENTERED TO THE PROMPT <PRINT SPECS:>.

<DELIM> IS CARRIAGE RETURN

<DELIVERY INSTRUCTIONS> IS A STRING OF DATA OF THE FORM <TEXT><DELIM><TEXT><DELIM>... AND WHICH HAS <DELIM> AS THE LAST ELEMENT.

<COMMENTS> IS A STRING OF DATA OF SIMILAR FORM TO <DELIVERY INSTRUCTIONS>.

NOTE THAT INCLUSION OF <COMMENTS> IS PERMITTED ONLY IF FORMS TYPE = <SPEC> OR THE <COMM> OPTION IS SPECIFIED IN <PRINTSPECS>.

FOR SUCCESSFUL SUBMISSIONS THE RESULT RETURNED IS <REQUEST NUMBER>, ELSE THE RESULT RETURNED IS ZERO.

EXAMPLE

THE FOLLOWING PROGRAM BUILDS A FILE TO BE PRINTED. IT GENERATES A THOUSAND-ENTRY TABLE OF ENGLISH-TO-METRIC CONVERSIONS, PLACED IN A FILE NAMED <METRICON>. SINCE THE WHOLE TABLE DOES NOT FIT COMFORTABLY INTO A WORKSPACE, IT IS GENERATED ONE HUNDRED LINES AT A TIME. THE PRINT REQUEST IS SUBMITTED AUTOMATICALLY BY THE FUNCTION.

```
      ∇ METRIC
[1]   'METRICON 70000' ⎕CREATE 5
[2]   FACTOR← 2.54 30.48 0.914402 1.60935 ◇ P←0
[3]   '     UNITS       CM/IN       CM/FT        M/YD
          KM/MI' ⎕APPEND 5
[4]  LN: Q←P+⍳100
[5]   TAB←'I10,4F12.3' ⎕FMT (Q;Q∘.×FACTOR)
[6]   TAB ⎕APPEND 5
[7]   →(1000>P←P+100)⍴LN
[8]   X←5 AUTOPREQ 'STD3,DECO,NOER',CR,'ROGER D.
      MOORE',CR,'SUITE 1400',CR,'145 KING ST. WEST',CR,
      'TORONTO, ONTARIO M5H 1J8',CR,'CALL 364-5361 EXT 276
      WHEN READY AND HE WILL PICK UP',CR
[9]   ∇
```

CONTROLLING THE PAGE FORMAT

UNLESS OTHERWISE SPECIFIED, EACH PAGE OF PRINTER OUTPUT HAS THIS FORMAT:

- BOTTOM LEFT CORNER OF THE FIRST PRINT POSITION ONE-HALF INCH BELOW PAPER PERFORATION AND ONE INCH FROM LEFT EDGE OF FORM.
- TITLE LINE, HOLDING THE FILE ID AND PAGE NUMBER
- BLANK LINE
- SUBTITLE LINE, HOLDING THE TIME AND DATE
- BLANK LINE
- 53 LINES OF INFORMATION FROM THE FILE, SINGLE SPACED.

YOU MAY SKIP THE REST OF THIS SECTION UNLESS YOU NEED A DIFFERENT PAGE FORMAT OR NON-STANDARD TYPEFONT.

PAGE ALIGNMENT, PAGE TITLES AND SUBTITLES , LINE SPACING AND TYPEFONT SELECTION ARE CONTROLLED BY NUMERIC FILE COMPONENTS. THE PAGE NUMBER IS INCREMENTED AUTOMATICALLY; NUMBERING MAY BE RESET TO 1 AT ANY POINT. PRINTING OF PAGE NUMBERS, DATE, AND TIME CAN BE SUPPRESSED. THE TITLE AND SUBTITLE LINES, UNLESS SUPPRESSED, ARE ALWAYS THE FIRST TWO LINES PRINTED ON EACH PAGE AFTER THE TOP MARGIN.

YOU CAN GET MOST FORMATS USEFUL FOR STANDARD 11-INCH FORMS BY USING ONE OR MORE OF THESE FUNCTIONS FROM WS 1 FILEPRINT. THEY OPERATE BY APPENDING ONE OR TWO COMPONENTS TO THE FILE DESIGNATED BY THE RIGHT ARGUMENT.

SYNTAX	EFFECT
<PAGE> FILE-NUMBER	PRINT THE NEXT LINE OF CHARACTERS ON A FRESH PAGE.
<SINGLE> FILE-NUMBER	PRINT THE NEXT LINE OF CHARACTERS ON A FRESH PAGE, AND SINGLE SPACE ALL FURTHER LINES EXCEPT FOR TITLE AND SUBTITLE LINES. 53 LINES OF FILE INFORMATION ARE PRINTED ON EACH PAGE. (THIS IS THE STANDARD PAGE FORMAT.)
<DOUBLE> FILE-NUMBER	PRINT THE NEXT LINE OF CHARACTERS ON A FRESH PAGE, AND DOUBLE-SPACE ALL FURTHER LINES. 27 LINES OF FILE INFORMATION ARE PRINTED ON EACH PAGE.
'STRING' <TITLE> FILE-NUMBER	USE THE LEFT ARGUMENT AS A TITLE LINE FOR ALL SUBSEQUENT PAGES. IF THE TITLE IS SHORTER THAN 119 CHARACTERS, IT IS FOLLOWED BY THE WORD 'PAGE' AND THE PAGE NUMBER. OTHERWISE, PRINTING OF THE PAGE NUMBER IS SUPPRESSED.
'STRING' <SUBTITLE> FILE-NUMBER	USE THE LEFT ARGUMENT AS A SUBTITLE LINE FOR ALL SUBSEQUENT PAGES. IF THE SUBTITLE IS SHORTER THAN 119 CHARACTERS, IT IS FOLLOWED BY THE DATE AND TIME OF PRINTING. OTHERWISE, PRINTING OF THE DATE AND TIME IS SUPPRESSED.
TABLE <FONT> FILE-NUMBER	USE THE TYPE FONT DEFINED BY THE PACKED INTEGER VECTOR <TABLE>. THE DEFAULT OR PREVIOUSLY SPECIFIED TRANSLATE TABLE IS OVERRIDDEN BY THE NEW ONE. ALL SUBSEQUENT PRINTER OUTPUT IS TRANSLATED ACCORDING TO THE TABLE.

THE <TITLE> AND <SUBTITLE> FUNCTIONS DO NOT FORCE A SKIP TO A FRESH PAGE, BUT RATHER TAKE EFFECT WHEN THE NEXT PAGE BEGINS. USE THE <PAGE> FUNCTION TO FORCE A FRESH PAGE. YOU CAN SUPPRESS UNWANTED PAGE NUMBERING, OR TIME AND DATE, EVEN IF YOUR TITLE OR SUBTITLE LINES ARE SHORT, BY EXTENDING THEM WITH BLANKS TO MAKE THEM AT LEAST 119 CHARACTERS LONG.

REPEATED EXECUTION OF THE <PAGE> FUNCTION PRODUCES A SINGLE SKIP TO A FRESH PAGE, NOT MULTIPLE BLANK PAGES. TO PRODUCE BLANK PAGES, ALTERNATE BETWEEN EXECUTING <PAGE> AND APPENDING A LINE OF BLANKS.

FOR FORMATTING PAGES IN WAYS NOT HANDLED BY THE FUNCTIONS ABOVE, NUMERIC COMPONENTS MAY BE APPENDED DIRECTLY. THE FOLLOWING VALUES MAY BE USED:

1 PRINT THE NEXT LINE OF CHARACTERS ON A FRESH PAGE.

2 SUPPRESS TITLE AND SUBTITLE PRINTING ON SUBSEQUENT PAGES. PAGE NUMBERS ARE KEPT CURRENT, EVEN THOUGH NOT PRINTED. PRINT LINES THAT WOULD NORMALLY BE OCCUPIED BY THE TITLE AND SUBTITLE LINES WILL BE FILLED FROM FILE COMPONENTS.

3 TAKE THE NEXT LINE OF CHARACTERS IN THE FILE AS A TITLE LINE FOR SUBSEQUENT PAGES. IF THE TITLE IS SHORTER THAN 119 CHARACTERS, IT IS FOLLOWED BY THE WORD 'PAGE' AND THE PAGE NUMBER.

4 TAKE THE NEXT LINE OF CHARACTERS IN THE FILE AS A TITLE LINE FOR SUBSEQUENT PAGES, AND RESET PAGE NUMBERING TO 1. IF THE TITLE IS SHORTER THAN 119 CHARACTERS, IT IS FOLLOWED BY THE WORD 'PAGE' AND THE PAGE NUMBER.

5 TAKE THE NEXT LINE OF CHARACTERS IN THE FILE AS A SUBTITLE LINE FOR SUBSEQUENT PAGES. IF THE SUBTITLE IS SHORTER THAN 119 CHARACTERS, IT IS FOLLOWED BY THE DATE AND TIME.

6 TAKE THE NEXT COMPONENT (WHICH MUST BE A VECTOR OF 128 INTEGERS) AS A TYPE FONT SELECTION TRANSLATE TABLE.

VECTOR OF 0'S AND 1'S
USE THE VECTOR FOR VERTICAL FORMAT CONTROL. THE LENGTH OF THE VECTOR INDICATES THE LENGTH OF THE PAGE; 1'S INDICATE LINES TO BE PRINTED ON, AND 0'S INDICATE LINES TO BE LEFT BLANK. WHEN THE LAST PRINTABLE LINE ON A PAGE IS REACHED A PAGE SKIP IS FORCED THAT POSITIONS THE PAPER AT THE FIRST LINE POSITION OF THE FOLLOWING PAGE, AND VERTICAL FORMAT CONTROL STARTS OVER AT THE BEGINNING OF THE VECTOR. THE VECTOR MUST CONTAIN AT LEAST THREE 1'S AND BE OF LENGTH 250 OR LESS.

EXAMPLES:

1)

```
      1 0 1 0,53⍴1
```

IS THE STANDARD PAGE FORMAT ASSUMED BY THE FILE PRINTING FACILITY.

2)

```
      1 0 1 0,54⍴1 1 1 1 1 0
```

DESIGNATES DOUBLE SPACED LINES FOR A TITLE AND SUBTITLE, FOLLOWED BY LINES FOR FILE INFORMATION SINGLE SPACED IN NINE GROUPS OF FIVE.

3)

```
      66⍴1                    (SCALE OF 6 LINES PER INCH)
```

DESIGNATES PRINTING ON EVERY LINE OF AN 11-INCH PAGE, ASSUMING THE OPERATOR HAS BEEN GIVEN SPECIAL INSTRUCTIONS TO POSITION THE FORM. IN CONJUNCTION WITH ANOTHER FILE COMPONENT CONTAINING THE SCALAR 2 (WHICH SUPPRESSES TITLES AND SUBTITLES) THIS COULD BE USED WHEN NO AUTOMATIC PAGE CONTROL OF ANY KIND IS WANTED.

4)

```
      88⍴1                    (SCALE OF 8 LINES PER INCH)
```

DESIGNATES PRINTING ON EVERY LINE OF AN 11-INCH PAGE, ASSUMING THE OPERATOR HAS BEEN GIVEN SPECIAL INSTRUCTIONS TO POSITION THE FORM. THIS VERTICAL FORMAT CONTROL CAN ALSO BE USED WHEN NO AUTOMATIC PAGE CONTROL IS WANTED.

5) *MODIFICATION OF THE METRIC CONVERSION PROGRAM USING TITLE, SUBTITLE AND VERTICAL FORMAT CONTROLS*:

```
      ∇ METRICB
[1]   'METRICONB 70000' ⎕CREATE 5
[2]   FACTOR← 2.54 30.48 0.914402 1.60935
[3]   P←0
[4]   (1 0 1 0,54⍴1 1 1 1 1 0) ⎕APPEND 5
[5]   '      METRIC CONVERSION TABLES' TITLE 5
[6]   '     UNITS       CM/IN       CM/FT        M/YD
      KM/MI' SUBTITLE 5
[7]  LN:Q←P+⍳100
[8]   TAB←'I10,4F12.3' ⎕FMT (Q;Q∘.×FACTOR)
[6]   TAB ⎕APPEND 5
[7]   →(1000>P←P+100)⍴LN
    ∇
```

ERRORS

CHARACTER-VALUED COMPONENTS OF RANK GREATER THAN TWO, OR NUMERIC COMPONENTS OTHER THAN THOSE DESCRIBED ABOVE, CAUSE AN ERROR MESSAGE TO BE PRINTED, BUT ARE OTHERWISE IGNORED.

IF THE FILE IS TIED WHEN THE FILE PRINT PROGRAM ATTEMPTS TO PRINT IT, IT IS NOT PRINTED. THE PRINT REQUEST IS DEFERRED UNTIL THE NEXT TIME THE FILE PRINTING FACILITY IS EXECUTED.

CONTROLLING TYPE FONT SELECTION

UNLESS OTHERWISE SPECIFIED, THE STANDARD TYPEFONT USED ON THE PRINTER IS <FAST>. THIS MEANS YOU MAY TREAT THE TEXT DESTINED FOR <FILEPRINT> AS IF IT WERE DESTINED FOR YOUR APL TERMINAL. NOTE THAT THIS INCLUDES THE POSITIONING AND CONTROL CHARACTERS <BS, UBS, LF, CR, ID> AND <NULL>. YOU MAY SKIP THIS SECTION UNLESS YOU NEED A DIFFERENT TYPE FONT.

BY DEFINITION, A PRINT FILE IS COMPOSED OF TWO CLASSES OF COMPONENTS:

NUMERIC COMPONENTS USED FOR CONTROL, AND
CHARACTER COMPONENTS OF THE ACTUAL DATA TO BE PRINTED.

THE NUMERIC COMPONENTS FALL INTO THREE CATEGORIES:

1. *SCALAR INTEGERS FROM THE SET* 1 2 3 4 5 6;
2. *BOOLEAN VECTORS FOR VERTICAL FORMAT CONTROL;*
3. 128 *ELEMENT VECTORS OF TRANSLATE TABLES.*

THE PREVIOUS SECTION EXPLAINED THE USE OF SCALAR VALUES 1 *THROUGH* 5 *AND THE BOOLEAN VECTOR; THIS SECTION DEALS WITH THE USE OF SCALAR* 6 *AND THE TRANSLATE TABLES.*

IN ORDER TO USE A NON-STANDARD TYPE FONT YOU MUST:

1) *APPEND A SCALAR* 6 *TO YOUR PRINT FILE, THUS SIGNALLING TO THE <FILEPRINT> PROGRAM THAT THE NEXT COMPONENT IS A TRANSLATE TABLE;*

2) *APPEND A* 128 *ELEMENT INTEGER VECTOR TRANSLATE TABLE IMMEDIATELY AFTER THE SCALAR* 6 *COMPONENT;*

3) *ADVISE THE APL OPERATOR WHICH TYPE FONT TO USE VIA THE SPECIAL INSTRUCTIONS PORTION OF <PRINTREQ>.*

NOTE THAT THE SPECIFIED TRANSLATE TABLE IS USED FOR ALL SUBSEQUENT CHARACTER COMPONENTS IN THE PRINT FILE OR UNTIL IT IS REPLACED BY ANOTHER TABLE.

FOUR USEFUL TRANSLATE TABLES CAN BE FOUND IN <1 FILEPRINT>, NAMELY <APLFAST, APLFULL, COURIER29> AND <PNFAST>. THERE ARE THREE PRINT TRAINS AT THE COMPUTER CENTRE: FAST, FULL AND PN2.

TRANSLATE TABLE	OUTPUT
COURIER29	OUTPUT APPEARS AS IF THE COURIER29 TYPE BALL WERE ON THE TERMINAL. SPECIFY FULL TRAIN.
APLFAST	APL CHARACTER SET. SPECIFY FAST TRAIN.
PNFAST	OUTPUT PRINTED THE WAY IT PRINTED BEFORE WE HAD APL PRINT TRAINS. SPECIFY PN2 TRAIN.
APLFULL	INTERMIXED COURIER AND APL FONTS THROUGHOUT THE PRINT FILE. SPECIFY TO THE OPERATOR THAT THE FULL TRAIN IS TO BE MOUNTED.

TRANSLATE TABLES

IN APL THERE IS A SET OF 256 POSSIBLE CHARACTERS REPRESENTED INTERNALLY BY THE HEXADECIMAL VALUES 00 THROUGH FF AND USUALLY CALLED Z-CODES OR ⎕AV. EACH ELEMENT OF A CHARACTER COMPONENT OR A PRINT FILE MUST BE IN THIS SET. ⎕AV CAN BE THOUGHT OF AS FALLING INTO 4 SUBSETS:

1) TERMINAL CONTROL CHARACTERS. THESE CAN BE FOUND IN 1 WSFNS AND CONSIST OF <BS>, <UBS>, <CR>, <LF>, <NL>, AND <ID>.

2) GRAPHICS THAT CAN BE DIRECTLY ENTERED OR PRINTED ON A STANDARD APL TERMINAL. THESE CAN BE FOUND BY DISPLAYING THE VARIABLE <APLCHARACTERS> IN WS 1 FILEPRINT.

3) GRAPHICS THAT CANNOT BE DIRECTLY ENTERED OR PRINTED ON A STANDARD APL TERMINAL BUT CAN BE PRINTED ON THE HIGH SPEED PRINTER. SOME OF THESE CAN BE FOUND IN THE GROUP <NONAPLGRAPHICS> IN WS 1 FILEPRINT.

4) ILLEGAL CHARACTERS.

IN TERMS OF OUTPUT FROM THE HIGH SPEED PRINTER, ASSUMING THAT THE CORRECT TRANSLATE TABLE AND PRINT TRAIN ARE USED, CHARACTER COMPONENTS WHOSE ELEMENTS ARE IN 1), 2), OR 3) WILL GENERALLY APPEAR AS ANTICIPATED, WHILE THOSE IN 4) WILL GENERALLY BE PRINTED AS ▯ . FROM A USER'S PERSPECTIVE, THE TRANSLATE TABLES CAN BE VIEWED AS DEFINING A MAPPING FROM ⎕AV INTO A PARTICULAR PRINT TRAIN USED ON THE PRINTER AT THE COMPUTER CENTRE.

EACH TRANSLATE TABLE IS A 128 ELEMENT VECTOR. HOWEVER, EACH INTEGER REPRESENTS THE PACKING OF 4 HEXADECIMAL VALUES, EACH IN THE RANGE OF 00 THROUGH FF. FUNCTIONS TO PACK AND UNPACK THESE TABLES CAN BE FOUND IN WS 1 FILEPRINT.

THE TRANSLATE TABLE IN ITS UNPACKED FORM CAN BE VIEWED AS A 2 ROW BY 256 COLUMN TABLE, T . SIMILARLY, THE PRINT TRAIN CAN BE VIEWED AS A 256 ELEMENT VECTOR, P , OF PRINTABLE AND CONTROL CHARACTERS. ASSUMING 0 ORIGIN, LET J←⎕AV; THEN T[;⎕AV⍳J] PRODUCES AN INTEGER PAIR, M N , WHERE M IS A DIRECT INDEX INTO P, AND N IS AN INDEX INTO THE COLUMNS OF T . THUS T[;N] PRODUCES ANOTHER INTEGER PAIR, M1 N1 , WHERE M1 INDEXES P AND N1 INDEXES T ; AND SO ON. AS A RESULT, A VECTOR OF INDICES INTO P IS GENERATED RECURSIVELY, HALTING EITHER WHEN ' '=⎕AV[N] (N=152) , OR WHEN RECURSION IS 10 LEVELS DEEP.

FOR EXAMPLE, LET J BE ϕ. USING THE APLFAST TRANSLATE TABLE WE FIND THAT 49↔⎕AV⍳J AND T[;49]↔206 33; FURTHERMORE T[;33]↔79 152 AND T[;152]↔64 152 . THUS OUR INDICES TO P ARE 206 79 64 AND OUR OUTPUT WOULD APPEAR AS ϕ . IF, HOWEVER, WE HAD USED THE COURIER TRANSLATE TABLE THEN T[;49]↔214 157; T[;157]↔231 152; T[;152]↔64 152 AND OUR INDICES WOULD BE 214 231 64 WHICH PRINTS OUT AS ▯ .

CREATION OF YOUR OWN TRANSLATE TABLES OR MODIFICATIONS TO THE EXISTING ONES REQUIRES AN UNDERSTANDING OF HOW THEY FUNCTION AS WELL AS A FAMILIARITY WITH THE CHARACTER SETS AVAILABLE ON THE VARIOUS PRINT TRAINS.

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

SATN 20
1 AUG 79
Rev. 4

SHARP APL TECHNICAL NOTES

TITLE: System variables, session variables and system functions

ABSTRACT: System variables and session variables are similar to variables and system functions are similar to APL primitives.

KEYWORDS:

SYSTEM VARIABLES

System variables resemble ordinary variables in many ways. They require symbol table entries and may be assigned values, referenced, localized and grouped like ordinary variables. System variables have default values that may be changed by assignment.

Any value can be assigned to a system variable. If an "invalid" value is assigned to a system variable the next implicit use of that variable will cause an error. For example:

```
      ⎕IO←'*' ◇ ⍳5
⎕IO ERROR
      ⎕IO←'*' ◇ ⍳5
                ^
```

A system variable may be localized by including it in a function header. The local system variable thus created applies to the function localizing it and all lower level functions which do not themselves localize that system variable. Any operation which explicitly or implicitly references a local system variable which has not been assigned a value will cause an error.

The main use of system variables is in the dynamic control or interrogation of the workspace environment. For example, origin independence is accomplished by making ⎕*IO* a local variable.

System variables in the receiving workspace of a)*COPY* command are not affected by the)*COPY* unless they or groups containing them are explicitly requested in the command.

The following is an alphabetic list of system variables:

⎕*CT*	(Comparison Tolerance)
⎕*ER*	(Event Report)
⎕*IO*	(Index Origin)
⎕*LX*	(Latent Expression)
⎕*PP*	(Print Precision)
⎕*PW*	(Print Width)
⎕*RL*	(Random Link)
⎕*SC*	(State Change)
⎕*TRAP*	(Trap Definitions)

⎕*CT* - Comparison Tolerance

⎕*CT* determines the maximum relative difference between two floating point numbers considered equal by the system. ⎕*CT* may be set to any value *X* in the range `0≤X<16*¯8` . Any ⎕*CT* setting outside this range will result in a ⎕*CT* error report during any comparison. The default value of ⎕*CT* is `1.419697593739419E¯14`. The most useful values are the default and zero (which implies exact equality). See SATN 23.

⎕*ER* - Event Report

⎕*ER* is set by the system to contain information about the last error that has occurred. See SATN-33.

⎕IO - Index Origin

⎕IO determines the index origin. ⎕IO may be set to either 0 or 1. Any other value assigned to ⎕IO will cause a ⎕IO error report during origin dependent functions such as indexing and ⍳. The default value of ⎕IO is 1.

⎕LX - Latent Expression

When a workspace is loaded in which ⎕LX is defined, one of the expressions ⍎⎕LX, ⍎'⍎⎕LX ◇ ⎕', or ⍎'⍎⎕LX ◇ ⍞' is evaluated, depending upon whether the workspace was saved in immediate execution, ⎕-input mode, or ⍞-input mode. The default value of ⎕LX is an empty character vector. See SATN 7.

⎕PP - Print Precision

⎕PP is the number of significant digits to be printed when numbers are displayed. Permissible values of ⎕PP are the integers from 1 to 16. The default value of ⎕PP is 10.

⎕PW - Print Width

⎕PW controls print width. ⎕PW may be set to any integer from 30 to 250. Any other value will cause a ⎕PW error during any operation involving printing to a terminal. The default value is 132.

⎕RL - Random Link

⎕RL is the random link for the query and deal functions. Each time ? is used the "most local" ⎕RL is changed to preserve randomness. The default value of ⎕RL is 16807, which is 7*5.

⎕SC - State Change

⎕SC is set to 1 by the system whenever a change in the shared-variable state for the task occurs. A reference to ⎕SC will delay the task until a change in the shared-variable state occurs. See SATN-32.

⎕TRAP - Trap Definitions

⎕TRAP is a character matrix or vector of 0 or more trap definitions. A trap definition defines the response to take when a specific event occurs. See SATN-33.

SESSION VARIABLES

A session variable is a system variable with the property that its local value (or lack of a value) is copied forward into the new workspace from a)*LOAD*,)*CLEAR*, □*LOAD*, or □*QLOAD*.

A "session" begins at sign-on for a TTASK, or at □*RUN* time for an NTASK or BTASK, and session variables are assigned their default values at this time. The exception to this is that a task doing a □*RUN*, or a B-task request has its □*SP* copied forward into the *LOADWSID* of the new task.

The following is an alphabetic list of session variables:

□*HT* (Horizontal Tabs)
□*SP* (Session Parameter)

□*HT* - Horizontal Tabs

□*HT* controls the use of the TAB character in terminal I/O, and may be a numeric scalar or vector.

If □*HT* is a scalar, it must be a non-negative integer, meaning that there is a tabstop set at the left margin and every □*HT* spaces thereafter.

If □*HT* is a vector it must be non-negative, monotonically increasing integers. The elements give the column numbers, relative to 0 (the left margin), at which tabstops are set.

Thus, □*HT*←*N* and □*HT*←*N*×⍳⌈□*PW*÷*N* are functionally equivalent in either index origin.

□*HT*←⍳0 indicates no tabstops are set, and is the default. □*HT*←0 also indicates no tabstops are set.

Entering a TAB character to the right of the rightmost defined tabstop results in a *CHAR ERROR* report.

□*SP* - Session Parameter

□*SP* is provided to allow the user to pass information from workspace to workspace. It is never inspected (i.e. evaluated) by the system, and may be assigned any value (including, of course, a PACKAGE). The default □*SP* is the empty character vector.

Because its intended use (i.e. passing information to a newly-loaded workspace) makes it especially useful to an N-task, during execution of □*RUN* the local □*SP* is copied from the initiator's workspace to become the initial value of □*SP* for the new task. Similarly a B-task request will save the local □*SP* as part of the request and it will be copied forward into the *LOADWSID* of the B-task. See SATN 5 for details on the use of □*SP* for B-tasks.

SYSTEM FUNCTIONS

System functions are like primitives. They do not need to be copied in from another workspace. The following is an alphabetic list of system functions:

R← ⎕*AI*	Accounting Information
X ⎕*APPEND Y*	File append
R← ⎕*APPENDR Y*	File append returning component number
R← ⎕*ARBIN Y*	Arbitrary input - see SATN 28
⎕*ARBOUT Y*	Arbitrary output - see SATN 28
R← ⎕*AV*	Atomic Vector
R← ⎕*AVAIL ''*	File system availability
R← ⎕*BOUNCE Y*	Force tasks *Y* off - see SATN 4
R← ⎕*CR Y*	Canonical representation of function named in *Y*
X ⎕*CREATE Y*	File create
R← ⎕*DL Y*	Delay *Y* seconds
⎕*DROP Y*	File drop of *Y* components
X ⎕*ERASE Y*	File erase
R← ⎕*EX Y*	Expunge (erase) objects named in *Y*
R←*X* ⎕*FD Y*	Function definition - see SATN 21
⎕*FF*	File primitive
⎕*FHOLD Y*	File hold with permission
R← ⎕*FI Y*	Build numbers from character string
R←*X* ⎕*FMT (A;B;....;Z)*	Format numbers into character data - see "SHARP APL Report Formatting" and SATN 17
R← ⎕*FX Y*	Function establishment from canonical representation in *Y*
⎕*HOLD Y*	File hold
R← ⎕*LC*	Line counter stack
R← ⎕*LIB Y*	File names in library *Y*
⎕*LOAD Y*	Load workspace *Y*

The following is a description of the system functions which are not covered in the "SHARP APL File System Manual", "SHARP APL Report Formatting", or in other SATNS.

R← ⎕*AI* Accounting Information

⎕*AI* returns a numeric vector as follows:

Element	Contents
1	User number
2	CPU this session
3	Connect time this session
4	Keying time this session
5	Characters this session
6	Cumulative CPU to date
7	Cumulative connect to date
8	Zero (unused at present)
9	Cumulative characters to date

Keying and connect times are in milliseconds. CPU is in milliunits. As new fields may be added, a length of 9 should not be assumed. Related information is available in SATN 9 (Usage Inquiry System).

R← ⎕*AV* Atomic Vector

⎕*AV* returns a vector of 256 unique characters that are ordered by their hexadecimal representations so that `1↑⎕AV` is `X'00'` and `¯1↑⎕AV` is `X'FF'`. Many of these characters are unprintable and will print as `'⎕'` (squish-quad), the canonical illegal graphic. The location of recognizable printing characters within ⎕*AV* is implementation dependent. Therefore any program which depends on the location of printable characters within ⎕*AV* is implementation dependent. This practice is accordingly not recommended.

⎕*AV* may be used in the processing of untranslated files (in EBCDIC, ASCII, etc.) or the preparation of such files. It is also useful in the construction of special purpose translate tables for HSPRINT (SATN 8).

R← ⎕*CR Y* Canonical Representation

⎕*CR* returns the canonical representation of the defined function named by its argument. The canonical representation is the same as the matrix representation returned by `2 ⎕FD Y`. If the argument does not name an unlocked defined function the result is an empty character matrix.

DOMAIN ERROR is reported if the argument is not character.
RANK ERROR is reported if the argument is not a vector or a scalar.

R← ⎕*DL Y* **Delay**

⎕*DL* causes execution to be suspended for at least *Y* seconds. ⎕*DL* returns, as a scalar, the number of seconds actually delayed.

R← ⎕*EX Y* **Expunge**

⎕*EX* erases the most local definition of the objects named by its character matrix argument. Each row of the argument represents one name, scalar and vector arguments are treated as one row matrices.

The explicit result is a boolean vector with one element per row of the argument, indicating whether the name is now available, (i.e. was expunged or was not found). 0 in the result indicates that the name is not available for one of the following reasons: the name was not well formed, the name represents a label, group, halted function, or a locked function in a sealed workspace.

DOMAIN ERROR is reported if the argument is not character.
RANK ERROR is reported if the argument rank is greater than matrix.

R← ⎕*FI Y* **Format Input**

⎕*FI* converts character vectors and scalars to numeric vectors. It returns a zero for substrings containing non-numeric characters. For example:

```
      ⎕FI '1 2 1.234 7.1E¯3'
1 2 1.234 0.0071

      ⎕FI '123 QWE 123QWE 12.12 1⎕23 '
123 0 0 12.12 0

      ⎕FI 'THIS IS A NUMBER 129856'
0 0 0 0 129856
```

R← ⎕*FX Y* **Function Establishment "Fix"**

⎕*FX* defines the function whose canonical representation is supplied as the argument. The result is the name of the function if it was established, or a numerical scalar index (origin 0) of the row which prevented establishment. If ⎕*FX* fails there is no effect upon the environment. ⎕*FX* will fail if the argument is not a valid canonical representation or if the name of the function conflicts with the name of a label, group, halted function, or a locked function in a sealed workspace. Note that ⎕*FX Y* differs from 3 ⎕*FD Y* in the following respects:

1. ⎕*FX* accepts only the matrix representation
2. ⎕*FX* will replace the current definition of an object.

DOMAIN ERROR is returned if the argument is not character
RANK ERROR is returned if the argument is not a matrix.

R← ⎕*LC* **Line Counter Stack**

⎕*LC* returns a vector containing the line numbers in the)*SI* stack. In immediate execution, →⎕*LC* will continue the execution of the last suspended function at the line of suspension.

⎕*LOAD Y* **Load a Workspace**

The monadic system function ⎕*LOAD* replaces the active workspace by the workspace designated in the right argument. The argument is a character scalar or vector containing the workspace name, optionally preceded by a library number and optionally followed by a colon (:) and a password. The *SAVED* ... message is printed at the terminal upon successful completion of the load.

⎕*LOAD* imposes a delay period sufficient to ensure that at least thirty seconds have elapsed since the last successful execution of ⎕*LOAD* or ⎕*QLOAD*. If neither of these has been called during the thirty seconds prior to a call to ⎕*LOAD*, then execution begins without delay. The waiting period serves to protect a user from repeatedly performing ⎕*LOAD* operations (for example, by attempting to load a workspace that executes ⎕*LOAD* on itself). It also avoids system response degradation by preventing an excessive frequency of ⎕*LOAD*'s.

R← ⎕*NC Y* **Name Classification**

⎕*NC* returns the name classification of the objects named by its matrix argument. Each row of the argument represents one name. Scalar and vector arguments are treated as one row matrices. The result values are:

```
0 - name is available
1 - label
2 - variable
3 - function
4 - other    - group
             - distinguished names (⎕ names)
             - invalid names
```

DOMAIN ERROR is reported if the argument is not character
RANK ERROR is reported if the argument rank is greater than matrix.

R← ⎕*NL Y* **Name List**
R←*X* ⎕*NL Y*

Monadic ⎕*NL* returns a character matrix of the names of objects with name classifications members of the argument, e.g. ⎕*NL* 1 3 will return the names of all labels or functions.

Dyadic ⎕*NL* restricts those names returned to those with (1↑*NAME*)∈*X* e.g. '*ASDF*' ⎕*NL* 2 will return all variables beginning with one of '*ASDF*'.

⎕*NL C* is equivalent to '*ABC*.....*Z*Δ' ⎕*NL C*

DOMAIN ERROR is returned if ~*Y*∈1 2 3 ⍝ Name classification
or ~*X*∈'*ABC*...*Z*Δ' ⍝ Alphabetics.
RANK ERROR is reported if the rank of either argument is greater than vector.

□*QLOAD Y* **Quietly Load a Workspace**

The monadic system function □*QLOAD* replaces the active workspace by the workspace designated in the right argument. □*QLOAD* is identical to □*LOAD*, except that the *SAVED ...* message is **not** printed at the terminal upon successful completion of the load.

R← □*TS* **Current Time Stamp**

□*TS* returns a 7-element integer vector containing the current year, month, day, hour, minute, second and millisecond. See SATN 29.

R← □*UL* **User Load**

□*UL* returns, as a scalar integer, the number of tasks currently signed on the system.

R← □*VI Y* **Verify Input**

□*VI* returns a Boolean vector with 1's for numeric substrings and 0's for substrings containing non-numerics.
For example:

```
      □VI '1 2 1.234 7.1E¯3'
1 1 1 1

       '123 QWE 123QWE 12.12 1□23'
1 0 0 1 0

      □VI 'THIS IS A NUMBER 129856'
0 0 0 0 1
```

□*VI* can be used to remove invalid data from the result of □*FI*. For example:

```
      S←'123 QWE 123QWE 12.12 1□23' ◇ (□VI S)/□FI S
123 12.12

      S←'THIS IS A NUMBER 129856' ◇ (□VI S)/□FI S
129856
```

R← □*WA* **Working Area Available**

□*WA* returns, as a scalar integer, the number of unused bytes in the active workspace.

R← □*NAMES*	Files currently tied
R← □*NC Y*	Name classification of object named in *Y*
R← □*NL Y*	Name list of objects of class∈*Y*
R←*X* □*NL Y*	Name list of objects of class∈*Y* with 1st character∈*X*
R← □*NUMS*	File tie numbers of tied files
R← □*OUT Y*	Output control - see SATN 3
R← □*PACK Y*	Package assembly - see SATN 14
□*PDEF Y*	Package definition - see SATN 14
X □*PDEF Y*	Package definition of selected objects - see SATN 14
R←*X* □*PEX Y*	Package expunge - see SATN 14
R←*X* □*PINS Y*	Package insert - see SATN 14
R← □*PLOCK Y*	Package lock - see SATN 14
R←*X* □*PLOCK Y*	Package lock of selected objects - see SATN 14
R← □*PNAMES Y*	Package names - see SATN 14
R← □*PPDEF Y*	Package protective definition - see SATN 14
R←*X* □*PPDEF Y*	Package protective definition of selected objects - see SATN 14
R←*X* □*PSEL Y*	Package selection - see SATN 14
R←*X* □*PVAL Y*	Package value - see SATN 14
□*QLOAD Y*	Quietly load workspace *Y*
R← □*RDAC Y*	File read of access matrix
R← □*RDCI Y*	File read of component information
R← □*READ Y*	File read
X □*RENAME Y*	File rename
X □*REPLACE Y*	File replace of a component
X □*RESIZE Y*	File reservation change
R← □*RUN Y*	Initiate an N-task - see SATN 4
R← □*RUNS*	Inquire about active tasks - see SATN 4 and 5

	□SIGNAL Y	Signal an error - see SATN 33
X	*□SIGNAL Y*	Signal an error - see SATN 33
R←	*□SIZE Y*	File size
X	*□STAC Y*	File set of access matrix
X	*□STIE Y*	File share tie
R←	*□SVC Y*	Report control vector - see SATN 32
R←X	*□SVC Y*	Set control vector - see SATN 32
R←	*□SVN Y*	Set clone ID - see SATN 32
R←	*□SVO Y*	Shared-variable coupling - see SATN 32
R←X	*□SVO Y*	Shared-variable offer - see SATN 32
R←	*□SVQ Y*	Shared-variable query - see SATN 32
R←	*□SVR Y*	Retract share-offer - see SATN 32
R←	*□SVS Y*	Report state of variables - see SATN 32
X	*□TIE Y*	File exclusive tie
R←	*□TS*	Current time stamp - see SATN 29
R←	*□UL*	User Load
	□UNTIE Y	File untie
R←	*□VI Y*	Numeric validation of character string
R←	*□WA*	Working area remaining
R←X	*□WS Y*	Workspace state information - see SATN 21

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

SATN 21
1 JUN 78
Rev. 1

SHARP APL TECHNICAL NOTES

TITLE: *⎕WS* and *⎕FD*

ABSTRACT: *⎕WS* provides the capabilities of many system commands plus additions and extensions. *⎕FD* provides function display, definition and locking, as well as *)ERASE*.

KEYWORDS: *)FNS*
)VARS
)GRPS
labels
)WSID
)SI
)ORIGIN
)DIGITS
)WIDTH
)TABS
⎕CT
⎕RL
)SYMBOLS
⌶29
PORT
TASKID
∇
⍫
)ERASE

SHARP TECHNICAL NOTES

Definitions:

NAMELIST - NL A character vector of names, where each name is delimited by at least one blank; or a character matrix with one name per row.

N A namelist containing a single name.

```
R←1 ⎕WS B
```

returns a character matrix of object names in the workspace, in alphabetical order. The contents of the result are determined by *B* as follows:

B	**Result**	**Equivalent**
1	Functions	*)FNS*
2	Variables	*)VARS*
4	Groups	*)GRPS*
8	Labels	

Sums of *B* return the expected result. For values of *B* such that *B*<0 or *B*>15, the result returned is equivalent to using 16|*B* as the right argument. For *B*=0 the result is `0 0 ⍴''`.

Note: The result returned is the "most local" definition of the object.

R←2 ⎕*WS B*

The contents of the result are determined by *B* as follows:

B	**Result**	**Equivalent**
1	Character vector containing the workspace ID in the following format: Column 1-10 Lib. no. (right justified) 11 Blank 12-22 Workspace name (left justified)	)*WSID*
2	Character matrix of the state indicator stack	)*SI*
3	Numeric vector containing workspace environment information. New elements may be added in the future.	

Element	**Contents**	
1	Origin of workspace	)*ORIGIN*
2	Significant digits	)*DIGITS*
3	Carriage width	)*WIDTH*
4	Tab stop setting	)*TABS*
5	Floating fuzz	⎕*CT*
6	Random link	⎕*RL*
7	Size of symbol table	)*SYMBOLS*
8	Symbols in use	
9	User number signed on	1↑⎕*AI*
10	Port signed on	
11	TASKID	
12	Translate table	**Values of element 12** 0 - N-task or B-task 1 - 2741 (typeball 1167987) 2 - TS2741 (typeball 1167988) 3 - TY33 (64 char. ASCII pairing) 4 - ASCII keyboard pairing 5 - ASCII typewriter pairing

Note: Elements 1 thru 6 are affected by SATN 13. On or after October 1, 1978 they will contain ¯1.

4 Numeric vector as follows:

Element	**Contents**
1	Workspace size
2	User number of the owner of the workspace (0 in a clear workspace)
3	Time and date that this workspace was last saved (0 in a clear workspace)

```
R←3 ⎕WS N
```

The result is a character matrix of the names in the group *N*. If *N* is not a group, the result is `0 0 ⍴''`.

```
R←4 ⎕WS NL
```

The result is a numeric vector with each element containing the number of bytes used by the corresponding object in *NL*. If *NL* contains undefined objects then the result contains `0` for those objects. The result for a group is (`+/4 ⎕WS 3 ⎕WS GPNAME`) added to the overhead for a group.

```
R←5 ⎕WS NL
```

The result is a numeric matrix with one row for each object in *NL*. The number of columns in the matrix is `1+⍴⎕LC` (i.e. one column for each row in *)SI*). Each column corresponds to the definition of the object at that level of the *SI* stack, with the most local definition in the first column and the most global in the last. Possible values in each row of the matrix are:

Value	**Meaning**
¯1	Not localized at this level
0	Localized but not defined at this level
1	Function
2	Variable
4	Group
8	Label

```
R←6 ⎕WS N
```

The result is the value of the variable named by *N*.

```
R←1 ⎕FD N
```

The result is a character vector representation of the most local definition of the function *N*, identical in appearance to ∇*X*[⎕]∇. Embedded carriage returns are used as end of line delimiters. Carriage returns in the result corresponding to those in literal strings in the function are followed by six blanks. If *N* is a locked function, the result is ''.

```
R←2 ⎕FD
```

The result is a character matrix representation of the most local definition of the function *N*, without line numbers, the brackets enclosing them, and without the opening and closing ∇. The result is left justified on the first non blank character in each line. The width of the matrix is that of the longest function line. Other lines are padded, on the right, with blanks. If *N* is a locked function, the result is 0 0 ⍴''.

```
R←3 ⎕FD X
```

Creates a function from the character array *X*. The rank of the argument determines which of two different sets of rules are used to create the resulting function.

1. If the argument is a vector, then it must be similar to the result of 1 ⎕*FD*. Carriage returns not contained within literal constants are used to delimit lines of the function. The characters up to the first carriage return make up the function header. The header must begin with a ∇ or a ⍢. If the latter, then the function is created as a locked function. Like the result of 1 ⎕*FD*, the lines of the argument (characters between carriage returns not in literal constants) must be numbered with consecutive numbers beginning with 1 and they must be contained within brackets. Also, carriage returns contained within literal constants must be followed by six blanks. These six blanks will not become part of the literal constant, but they are required for consistency with function display and the result of 1 ⎕*FD*. A trailing ∇ *OR* ⍢ is required and it may appear on or after the last numbered line.

2. If the argument is a matrix, then it must be similar to the result of 2 ⎕*FD*. The first row of the argument will be converted to the header of the function and subsequent rows will make up the lines of the function. Like the result of 2 ⎕*FD*, the rows of the matrix should not contain line numbers and they should not contain a ∇ or ⍢ other than within quoted strings or in comments.

With either a vector or matrix argument, blanks which would be superfluous in keyboard function definition are ignored. If the function definition was successful, the name of the function is returned as a character vector result. If the name of the function created corresponds to a local identifier in a currently pendent or executing function, the newly created function will be local to that function and will be erased when the function in which it is localized completes execution. A local function can only be displayed via 1 or 2 ⎕*FD*.

For improperly formed arguments, the result is a two-element integer vector containing information about the error. The first element indicates the type of error; the second element indicates the row of the matrix argument or element of the vector argument which is the beginning of the line containing the error. The meaning of the error numbers is as follows:

1 *WSFULL*

2 *DEFN ERROR*: improperly formed header; function name in use; vector argument does not contain a leading and trailing ∇ or ⍛, contains an extra one, is missing a line number in brackets, or contains non-consecutive line numbers; a matrix argument contains a carriage return not in a literal constant

3 *CHARACTER ERROR*: the argument contains a character which is not valid within the definition of a function (e.g. a backspace character); a vector argument contains a carriage return in a literal constant which is not followed by six blanks; a comment in a matrix argument contains a carriage return

4 *SYMBOL TABLE FULL*: creating the function would require more symbol table entries than are available

5 *OPEN QUOTE*: the argument contains an odd number of quotes outside of comments

6 *SYSTEM ERROR*: please report it to the APL operator

7 *EMPTY LINE*: in a matrix argument, an entire row is blank or in a vector argument, the line-delimiting carriage returns have only blanks between them.

R←6 ⎕*FD NL*

The "most local" definition of each object in *NL* is erased, if possible. The result is a character matrix with each row containing the name of an object not erased. If all specified objects are erased, the result is 0 0 ⍴''.

R←7 ⎕*FD NL*

The "most local" definition of each function in *NL* is locked. The result is a character matrix of the names in *NL* that were not functions.

General Notes Concerning Errors

For 1 ⎕*WS* and 2 ⎕*WS*, arguments other than integer scalars or 1-element vectors will cause a *DOMAIN ERROR*, *RANK ERROR*, or ⎕*FD*/⎕*WS ARGUMENT ERROR* report.

For 3 ⎕*WS*, 4 ⎕*WS*, and 5 ⎕*WS*, ill-formed names (e.g. character strings with embedded carriage returns) will cause a ⎕*FD*/⎕*WS ARGUMENT ERROR* report.

For 6 ⎕*WS*, an ill-formed name will cause a ⎕*FD*/⎕*WS ARGUMENT ERROR* report. A name which is not that of a variable (e.g. an undefined symbol) will cause a ⎕*FD*/⎕*WS VALUE ERROR* report.

For 1 ⎕*FD* 2 ⎕*FD* and 6 ⎕*FD*, an improperly formed right argument will cause a ⎕*FD*/⎕*WS ARGUMENT ERROR* report.

For 1 ⎕*FD* and 2 ⎕*FD*, a right argument which is a valid symbol (whether defined or not) but which is not the name of a function will cause a ⎕*FD*/⎕*WS DEFN ERROR* report.

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

SATN 22
15 JUL 81
Rev. 4

SHARP APL TECHNICAL NOTES

TITLE: APL Workspace Transfer.

AUTHOR: R. Bernecky and M. Symes. (Rev.4) by Greg Bezoff.

ABSTRACT: Historically the transfer of workspaces between installations with different implementations of APL (or even different release levels of the same implementation) was a difficult, haphazard affair that generally required the careful ministrations of an assembly language system programmer.

This SATN details how to transfer workspaces to and from a SHARP APL system in a way which adheres to the proposed "Workspace Interchange Convention (version 0)" - WSIS0 - formulated by a committee of the STAPL executive.

This convention should greatly simplify the transfer of workspaces. In particular it allows all transfer-related problems to be resolved by an APL programmer.

An extension of the published convention is included which allows SHARP APL files, enclosed arrays, and package datatypes to be transferred.

KEYWORDS:
Workspace
File
Convention
Transfer
WS SELDUMP
WSIS0

Since the dark ages, people have transferred workspaces from one site to another in "dump" format. The only way to transfer workspaces (other than manual display and entry), was to use these mysterious, arcane, internal format dump tapes. This worked fine until someone else wrote an APL system. Now Burroughs, IPSA, IBM and others have APL implementations, all with their own workspace structures, and all different.

For a time it was possible to write complex, customized utilities to convert one flavour of workspace to another, but it rapidly became a losing battle: there are always new formats, or changed old ones. Clearly, another approach was needed.

Since a workspace can be transferred by displaying all functions and variables on a terminal and then typing them back in, it is clear that a source form of transfer is possible. There are exceptions to this (e.g. loss of precision on conversion from internal floating point to decimal and back, inability to transport locked functions, etc.), but the technique is suitable for most purposes.

A standard representation of APL functions and variables in source form on external media allows transfer of workspaces without the delays, system errors and hassles that otherwise result.

Such an APL source level workspace transfer convention was proposed at the APL workshop held at Queen's University in May, 1976. The workshop developed a standard scheme for representing functions and variables as character vectors. The convention did not concern itself with the details of transfer such as recording medium, character set, recording density, block size, media labelling conventions, etc. These were to be mutually decided upon by the sending and receiving sites. The basic assumption was that each sending or receiving site had some technique for placing an APL character variable as a physical block on an external medium, and for getting an APL character variable from a physical block on an external medium. Each site could then write relatively straightforward APL functions for sending or receiving workspaces.

This convention, extended to allow the sending site to specify its APL character set in terms of an already accepted standard (the APL-ASCII typewriter-pairing character set), was implemented in SHARP APL.

Later, a committee set up by the STAPL executive amended the Queen's convention slightly, and generalized Sharp's method for including the specification of the sending site's character set. (Appendix A gives the full WSIS0 convention.)

The procedure allows sites to transfer workspaces without regard for the internal workspace structure at either site. The convention treats the medium logically as the catenation of several character vectors, (or in some cases simply as a bit stream), physically broken into blocks of length appropriate to the transmission medium.

The usefulness of *'WS SELDUMPS'* (internal format selective workspace dumps) as a workspace transfer medium (between a SHARP APL site and a non-SHARP APL site) has steadily declined, to the point that they are now considered to be obsolete for this purpose.

I.P. Sharp Associates **strongly** advises against using *SELDUMP* tapes as a medium for importing

workspaces to, or exporting them from, a site in such circumstances. Users are instead advised to use the workspace transfer format described here.

The workspace transfer system does not pretend to cope with the general problem of language incompatibility between sites (e.g. ⎕*READ* versus ⎕*FREAD*). These conflicts must be resolved at either the receiving or sending site by an APL programmer.

TRANSMITTING WORKSPACES

Example: Transmit two workspaces, 6180339 *PRODIGAL* and 20 *SCAN*, on a single tape:

```
'HOLDMYWS' ⎕CREATE 1
(1 3⍴1000 ¯1 0) ⎕STAC 1
```

(create an output file)
(give batch access to file, so it can be transferred to tape)

First Workspace

```
)LOAD 6180339 PRODIGAL

)COPY 6 WSTRANSFER WWSOUT

5000 WWSOUT 1
```

(put workspace to file 1 in blocks of 5000 characters)

```
WSID:
6180339 PRODIGAL  OK? (Y/N):  Y
```

(workspace name is acceptable)

```
NOTE FOR THIS WORKSPACE:  USES SHARP APL FILE SYSTEM PRIMITIVES

LAST WORKSPACE?  (Y/N):  N
```

(not the last for this file)

```
DONE
```

Second Workspace

```
      )LOAD 20 SCAN

      )COPY 6 WSTRANSFER WWSOUT

      5000 WWSOUT 1

WSID:  20 SCAN    OK? (Y/N):  N

NEW NAME:  SCAN                          (change workspace name before exporting)

NOTE FOR THIS WORKSPACE:  USES I-BEAMS BUT NO FILE PRIMITIVES

LAST WORKSPACE? (Y/N):  Y                (last for this file)

DONE

      ⎕UNTIE 1
```

At this point, the user should request that the file be copied to the desired external medium (usually tape) in the following manner:

a) One physical record per file component, specifying the maximum physical record size used; e.g. 5000 in the example above.

b) No character translation: note that if any translation is performed at either site, information will be lost if $0 \in \square AV \in TAB$ (where *TAB* represents the translate table used).

c) Other parameters are left to the discretion of the sites involved, but typically include
LABELLING: STANDARD LABELS
TAPE DENSITY: 1600/6250 BPI

RECEIVING WORKSPACES

The user should request that the incoming file (tape or otherwise) be placed in an APL file with the following characteristics:

a) One file component per physical record
b) No character translation

Labelling and density depend on the whims of the sending installation. Then, to create the workspaces, type:

```
'NEWSTUFF' ⎕STIE 1                (incoming file)

)CLEAR

)SYMB 300                         (some guess as to what will be needed)

)COPY 6 WSTRANSFER WWSIN

WWSIN 1 ◇ ⎕EX 'WWSIN'             (process the file; erase WWSIN when done)


WSIS VERSION:  0

TRANSLATE TABLE PROCESSING

WSID:  6180339 PRODIGAL   OK? (Y/N):  N

NEW NAME:  PRODIGAL               (workspace name is changed)

NOTE:

USES SHARP APL FILE SYSTEM PRIMITIVES
                                  (sending site's note is printed)

SAVING:  PRODIGAL                 (workspace is being saved by N-tasks)

WSID:  SCAN  OK? (Y/N):  Y        (name is acceptable)

NOTE:

USES I-BEAMS BUT NO FILE PRIMITIVES

END OF TRANSFER

SAVING:  SCAN

1                                 (result of ⎕EX)
```

CANONICAL REPRESENTATION OF PACKAGES

The canonical representation of the SHARP APL datatype "**package**" has been defined (as a non-standard object; i.e. type 'X') as follows:

<type name>	:	PACKAGE
<identifier>	:	name of package, followed by names of its elements
<separator>	:	colon
<values>	:	the canonical representation vectors of the elements of the package in the order they appear in the **<identifier>** field.

Example:

```
55XPACKAGE PP      A1  FOO:9NA1  0 4419FFOO 2  2  3 FOOX←1
```

is a representation of a package *PP* containing a variable *A1* and a function *FOO*.

Notice that the definition of the representation of the package is implicitly recursive, since the definition of a package is recursive.

Such representations will be generated when executing *WWSOUT* for a workspace which contains packages. Conversely *WWSIN* will define packages from such representations.

CANONICAL REPRESENTATION OF ENCLOSED ARRAYS

The canonical representation of the SHARP APL datatype "enclosed array" has been defined so as to remain consistent with the standard canonical representation vector:

<pad> <length> <type> <identifier> <blank> <rank> <blank> <shape vector> <elements> <pad>

(see Appendix A, section 1.1)

All of the fields of the canonical representation vector retain their former definitions except the <type> and <element> fields. It was necessary to extend the definitions of these two fields in order to accomodate enclosed arrays, and they are explained below.

<type> : *E*

<elements> : A sequence of the canonical representation vectors of the elements of the enclosed array (in ravelled order), where the <identifier> field of each element is the single character ∘ .

Notes:

1. Since the elements of the enclosed arrays must be arrays, (numeric, character, or enclosed), any canonical representation vector that is an element of an enclosed array must be of <type> *C*, *N*, or *E*.

2. Note that the "enclosed array' *CRV* has a potentially recursive nature, since the <elements> of a <type> *E CRV* may include <type> *E CRV*s whose <elements> may include <type> *E CRV*s and so on .....

Examples:

1) Let `X←1 2 , Y←'AB' , Z←(<X),<Y`

Then `Z` is represented as:

`29EZ 1 2 10N∘ 1 2 1 29C∘ 1 2 AB`

Characters	**Explanation**
`29`	Length of the *CRV* (excluding `'29'`)
E	Enclosed array
`Z`	Name of enclosed array is `Z`

<blank>1<blank>	Rank of *Z* is 1
2<blank>	Shape of *Z* is 2
10	Length of the representation of the first element of *Z* is 10 (excluding '10')
N	First element of *Z* is a numeric array
∘	Indicates that this is an element of an enclosed array

<blank>1<blank>	Rank of the 1st element of *Z* is 1
2<blank>	Shape of the first element of *Z* is 2
1 2	Ravelled display form of the first element of *Z*
9	Length of the representation of the 2nd element is 9 (excluding '9')
C	2nd element is a character array
∘	Element of sub-element of enclosed array

<blank>1<blank>	Rank of 2nd element is 1
2<blank>	Shape of 2nd element is 2
AB	Ravelled display form of 2nd element.

2) This example demonstrates the recursive nature of the enclosed array.

Let `X←1 2 , Y←'AB' , Z←(<X),<Y ,`
`W←2 2⍴(<Z),(<Y),(<Y),<X`

Then *W* is represented as:

```
72EW 2 2 2 29E∘ 1 2 10N∘ 1 2 1 2
9C∘ 1 2 AB9C∘ 1 2 AB9C∘ 1 2 AB10
N∘ 1 2 1 2
```

FILE TRANSFER

A representation of a SHARP APL file can be constructed for export by adopting the convention that components of the file are assigned to variables *COMP*1, *COMP*2, *COMP*3, etc., respectively, and the standard representations of these variables constructed; this stream of representations being preceded by a non-standard (i.e. type 'X') representation of the file name. This latter representation is constructed as if it were a pseudo-variable '*FILEID*':

Example:

```
      35XPFILEID 1 22        666 SYSFILE
```

Conversely, an input stream adopting these conventions can be processed to build a SHARP APL file.

Transfer of large data files by this method is not recommended, because the representation of numeric variables is a very extravagant one. It is mainly intended as an attempt to extend the convention to allow transfer of files associated with workspaces, containing such things as parameters, tables and overlay functions.

To export files along with a workspace:

```
)LOAD 1234567 SAMPLEWS

)COPY 6 WSTRANSFER WWSOUT

'1234567 SAMPLEFILE1' ⎕STIE 33
'1234567 SAMPLEFILE2' ⎕STIE 34

2000 WWSOUT 1, 33 34          (output file is no. 1; files to be transferred as well as
                               the workspace are numbers 33 and 34)


WSID:  1234567 SAMPLEWS   OK? (Y/N):  Y

NOTE FOR THIS WORKSPACE:  ........

FILE NAME:  1234567 SAMPLEFILE1    OK? (Y/N):  Y

NOTE FOR THIS FILE:  ......

FILE NAME:  1234567 SAMPLEFILE2    OK? (Y/N):  Y

NOTE FOR THIS FILE:  .....

LAST WORKSPACE? (Y/N):  Y

DONE
```

To import the workspace and file on such a transfer tape, use *WWSIN* just as previously indicated.

NAME CONFLICTS

In a system such as this, there is always the problem of name conflicts between names used by the system transfer system itself and those of the APL objects in the user's workspaces. To help avoid those problems, all names used by the transfer system begin with the character "*W*". If the user also uses such names, there is a possibility of breakdown because of this, (or, more likely, incomplete transfer). To avoid these conflicts, the user should probably rename such objects prior to transfer, and include a note indicating which names have been altered.

ERRORS ENCOUNTERED TRYING TO EXPORT

A locked function in a workspace (or file) will cause the message:

```
LOCKED FN:
```

when encountered, and will result in the inclusion of a note in the output file.

ERRORS ENCOUNTERED TRYING TO IMPORT

There are a number of possible errors which may be encountered in trying to receive a workspace (or file): either if the incoming file is not constructed according to the interchange convention, or if there is a character set problem; i.e. the sending site uses graphic combinations that are not in the receiving site's ⎕AV. Some of these errors will cause the program to encounter a deliberate *DOMAIN ERROR*, thus:

```
**SUSPENSION
DOMAIN ERROR etc.
```

This suspension gives the receiver an opportunity to diagnose (and perhaps fix manually) the problem. Depending on the source of the error, the values of the following variables are relevant at this point:

⍙TYPE	-	type of object being defined (*P*,*C*,*N*,*E*,*F*,*X*)
⍙NAME	-	name of object
⍙RANK	-	rank of object
⍙SHAPE	-	shape of object
⍙CRV	-	the unparsed residue of the (translated) canonical representation vector
⍙COMP	-	number of last component of input file read

The values (or lack of them) may indicate the source of the trouble.

Note: Due to the recursive nature of enclosed arrays, the variables *⍙TYPE*, *⍙NAME*, *⍙RANK*, and *⍙SHAPE* will pertain only to an enclosed array as a whole, never any of the elements of an enclosed array. *⍙CRV*, however, will be 'stripped down' to the point at which the error occurred.

To proceed beyond the suspension, type:

```
→0
```

The possible errors are:

1. *ILLEGAL CODES*:
(followed by a list of offending codes and their graphic combinations). During translation table processing, the existence of incoming codes whose graphic combinations are not present in the receiving site's □*AV* has been detected.

2. *ILLEGAL CODES IN STREAM*:
(followed by the offending codes). During the translation of incoming characters, elements listed in error(1) have been encountered.

3. *MISSING LENGTH FIELD*
During an attempt to extract a single canonical representation vector, no preceding length field was found. This is a difficult error from which to recover, and an attempt to proceed will cause termination of the program.

4. During the analysis of a representation vector, the following errors can occur:

4.1 *INVALID CRV TYPE* : type character is not one of *P,C,N,E,F,X*

4.2 *NO IDENTIFIER* : identifier field is missing

4.3 *INVALID IDENTIFIER* : the identifier field contains an illegal name

4.4 *INVALID RANK* : rank field is not a non-negative integer

4.5 *INVALID SHAPE* : shape field contains other than non-negative integers

4.6 *INVALID CHARACTERS IN NUMERIC ARRAY*
: elements field of a supposedly numeric array contains other than numeric elements.

4.7 *MISSING LENGTH FIELD WITHIN ENCLOSED ARRAY*
: the canonical representation of a sub-element of an enclosed array has no length field

4.8 *INVALID VARIABLE TYPE WITHIN ENCLOSED ARRAY*
: the type of an element of an enclosed array is not one of *C,E,N*

4.9 *MISSING OR INVALID IDENTIFIER OF SUBELEMENT OF ENCLOSED ARRAY*
: identifier of a sub-element of an enclosed array is not ∘

4.10 *NUMBER OF ELEMENTS INCONSISTENT WITH SHAPE VECTOR*
: there are more or less elements for an array than the shape vector specifies

4.11 *FN NOT DEFINABLE* : failed attempt to execute ⎕*FX* to define a function (this will normally be due to a name conflict, or a locked function in the transferred workspace)

4.12 *FN IDENTIFIER DOES NOT MATCH FN DEFINITION*
: identifier in representation vector does not match the result of ⎕*FX*

4.13 *INVALID PSEUDO-VARIABLE IDENTIFIER*
: identifier of a pseudo-variable is not one of:
WSIS
TRANSLATE
END
NOTE
WSID

4.14 *INVALID NON-STANDARD OBJECT TYPE*
: the type of a non-standard representation (type 'X') is not one recognized by SHARP APL:
PACKAGE
UNDEF
PFILEID

Proceeding beyond these errors will cause the representation vector in question to be discarded.

5. □*RUN ERROR*:
(followed by the error code). The auto-save of an incoming workspace is achieved by spawning a split N-task. The attempted □*RUN* failed.

TRANSFERRING THE INTERCHANGE SYSTEM ITSELF

For a non-SHARP APL site to be a party to a workspace interchange, they must of course, have a version of the interchange system. If it is desired to help them establish this by sending them the code (written in SHARP APL) of Sharp's version of the interchange system, a procedure has been established for shipping the code in a way that requires only a small amount of typing at the receiving site to bootstrap the system.

Consult the variable *HOWWSISBOOT* in the workspace 6 *WSTRANSFER* for information about how to achieve this.

APPENDIX A

PROPOSED WORKSPACE INTERCHANGE CONVENTION

To the APL community:

A draft standard for the interchange of APL workspaces between different APL implementations follows on the next few pages. It is the work of an ad hoc committeee formed at the recent Minnowbrook Workshop of APL Implementers. The committee comprises the following people:

Dana E. Cartwright, Syracuse University
Patrick E. Hagerty, Sperry Univac (Chairman)
Eric B. Iverson, I.P. Sharp Associates
Roger Lipsett, Digital Equipment Corporation
John W. Myrna, Scientific Time Sharing Corporation
William Phelps, IBM San Jose
James Ryan, Burroughs Corporation
James Triplett, University of Massachusetts
Michael Wheatley, IBM Canada

The ideas of the standard stem from discussions among many people over several years. The most recent ancestors of this proposal are a document by R. Bernecky of I.P. Sharp Associates and some modifications to Bernecky's work by M. David of Scientific Time Sharing Corporation. Both the Bernecky and David schemes have been used successfully for numerous workspace transfers by IPSA and STSC. The final editing and preparation of the text of the draft standard was done by Dana Cartwright of Syracuse University.

Several areas were deliberately avoided by the committee and left to future standards efforts. The exchange of APL data files is one of these. Since files can be represented as ordered collections of APL variables, a standard for these can build on the concept of the Canonical Representation Vector (CRV).

Another area that was avoided is data compaction. Expressions such as

```
A←3+5×⍳1E9
```

can be handled by some APL implementations but are impractical to evaluate and display in ravel order. The CRV designator *X* has been reserved for local agreement between the two parties to a workspace transfer; it can be used for data of this type if the need arises.

This draft standard is being submitted for comment from the APL user community. Comments should be addressed to the Committee Chairman:

Patrick E. Hagerty, M9-116
Sperry Univac
P.O. Box 500
Blue Bell, PA 19424

Please submit comments by 15 April 1978. A final version of the standard incorporating relevent comments will be submitted for publication in "APL Quote Quad" as soon as possible thereafter.

Sincerely,
Patrick E. Hagerty

THE PROPOSED STANDARD

Prepared by Dana E. Cartwright

Purpose of the Standard

This standard is intended to facilitate the exchange of APL functions and variables between APL installations in a manner that is independent of specific APL implementations and operating systems.

Provision is made for installations to cooperate in defining local extensions to the standard, so that exchanges need not be limited to functions and variables.

This standard does not define the APL language itself, and it does not address site-to-site variations in the language which could preclude meaningful interchange of APL functions and variables.

Organization of the Standard

The basis for an exchange is the specification of an algorithm for converting functions and variables to **bit streams** (sequence of binary digits). A **workspace** (a collection of APL objects) is represented by a catenation of such bit streams. Streams representing workspaces can be catenated to form larger streams representing multiple workspaces.

The standard is presented hierarchically, beginning with the representation of individual functions and variables as character vectors, and ending with details of mapping a bit stream onto a physical recording medium. Each level within the hierarchy is designed to be independent of "deeper" (more detailed) levels.

At the first level of desription, a scheme is introduced whereby any APL function or variable (character or numeric) can be represented as an APL character vector. Such representation is called a "canonical representation vector".

The second level of description discusses the construction of an aggregate stream, representing collections of functions and variables. In this construction, more types of canonical representation vectors are included to provide control information to the receiving installation.

In the definitions of the first two levels, it is assumed that the notion of an APL "character" is well understood. However, since installations differ in their representation of character data, the third level of description is concerned with the precise definition of a "character". As a result of this definition, a character vector is reinterpreted as a stream of bits, with a mapping from bits to characters provided by a "translate table" based upon the keyboard-pairing APL-ASCII terminal transmission codes (see Appendix B).

The fourth level of discussion is solely concerned with describing a method of mapping a bit stream onto a physical recording medium.

Level 1 Description

1.1 Canonical Representation Vectors

A canonical representation vector is a character vector which describes a single APL function or variable and has the form:

<pad><length><type><identifier><blank><rank><blank><shape vector><elements><pad>

The delimited elements have the following values:

<pad> Zero or more blank characters, used as padding to increase the length of the vector if so desired.

<length> One or more digits representing the number of characters in the canonical representation vector; <length> includes character counts for <type>, <identifier>, <rank>, <blank>, and <shape vector>, but not for any leading or trailing <pad> nor the length digits themselves.

<type> A letter denoting the type of object being transmitted. The following types are defined:

C - character array
N - numeric array
F - function (actually the canonical representation thereof)
P - pseudovariable (see Level 2 Description)
X - installation-dependent canonical representation vector.

<identifier> The name of the variable or function being described.

<blank> A single blank character.

<rank> One or more digits representing the rank of the object being transmitted.

<shape vector> Zero or more digits representing zero or more integers that define the shape of the object; each digit is followed by one blank.

<elements> The display form of the raveled variable, or the raveled canonical representation of the function being described.

There is no upper bound on the number of digits that can appear in the <length>, <rank>, and <shape vector> values.

The **value** of a canonical representation vector is defined to be the APL array denoted by the <type>, <rank>, <shape vector>, and <elements> portions thereof.

1.2 Representation of Variables, Examples

Each variable is represented as a separate canonical representation vector. For example,

```
      A←2 3⍴'RATFAT'
```

is represented as

```
15CA 2 2 3 RATFAT
```

Characters	Explanation
15	Length of this canonical representation vector, excluding the length digits
C	*C* for character
A	Name of the variable
<blank>	Separator
2	Rank
<blank>	Separator
2<blank>3<blank>	Shape vector
RATFAT	Display form of raveled value

In most systems a canonical representation of a variable can be generated using the function *VREP*, shown below. The right argument is a character vector naming the variable to be represented.

```
    ∇ ΔR←VREP ΔA
[1]   ⍝ ΔA IS THE NAME OF THE VARIABLE.
[2]   ⍝ GET INTERNAL VALUE.
[3]    ΔR←⍎ΔA
[4]   ⍝ CATENATE NAME, RANK, SHAPE VECTOR, AND VALUE.
[5]    ΔR←ΔA,' ',(⍕(⍴⍴ΔR),⍴ΔR),' ',⍕,ΔR
[6]   ⍝ APPEND DATA TYPE.
[7]    ΔR←'NC'[(⍳1)+' '=1↑0⍴⍎ΔA],ΔR
[8]   ⍝ APPEND VECTOR LENGTH.
[9]    ΔR←(⍕⍴ΔR),ΔR
    ∇
```

The names *ΔA*, *ΔR*, and *VREP* must be changed if there are global variables or a function with the same name in the workspace being transferred.

The use of *VREP* to convert three variables to canonical representation vectors is shown in the example below.

```
      S←(⍳0)⍴'?'
      S
?
      VREP 'S'
6CS 0 ?
      NIL←0 0 0 0⍴0
      NIL
            [empty line]
      ⍴NIL
0 0 0 0
      VREP 'NIL'
```

```
15NNIL 4 0 0 0 0

      CAC←0 ¯1 ¯2
      ⍴CAC
3
      VREP 'CAC'
16NCAC 1 3 0 ¯1 ¯2
```

The function *VREP* is sensitive to the digits setting in the active workspace when numeric data items are being processed. Unnecessary loss of precision can be prevented by specifying the maximum digits setting before *VREP* is executed. If the sending site supports an internal precision greater than that supported at the receiving site, loss of precision may be unavoidable.

1.3 Representation of Functions, Examples

Functions are represented in canonical form (⎕*CR*) in a manner similar to character variables. For example, if the function *ADD* is defined as

```
    ∇ R←A ADD B
[1]   R←A+B
    ∇
```

it is represented as the following vector:

```
29FADD 2 2 9 R←A ADD BR←A+B
```

Characters	Explanation
29	Length of vector, excluding the length digits
F	*F* for function
ADD	Name of the function
<blank>	Separator
2	Rank
<blank>	Separator
2<blank>9<blank>	Shape vector
R←A ADD BR←A+B	Function image returned by ⎕*CR*, raveled

The function is reconstructed from its representation by first generating the value of the representation as though it were a character matrix, and then applying ⎕*FX*. The name returned by ⎕*FX* should match the <identifier> of the representation.

A locked function cannot be represented, but it can be detected under program control and treated as an empty array. For example, if the funciton *ADD* were locked, it would be represented as

```
11FADD 2 0 0
```

In most systems, the function *FREP* will generate the required representation of a function (whether or not it is locked).

```
    ∇ ΔR←FREP ΔA
[1]  ⍝ ΔA IS THE NAME OF THE FUNCTION.
[2]  ⍝ GET CANONICAL REPRESENTATION.
[3]   ΔR←⎕CR ΔA
[4]  ⍝ APPEND DATA TYPE, RANK AND SHAPE VECTOR.
[5]   ΔR←'F',ΔA,' ',(⍕(ρρΔR),ρΔR),' ',,ΔR
[6]  ⍝ APPEND VECTOR LENGTH.
[7]   ΔR←(⍕ρΔR),ΔR
    ∇
```

Note: It is assumed that the system function ⎕*CR* returns a value with shape `0 0` if its argument is not an unlocked, defined function.

The names Δ*A*, Δ*R*, and *FREP* must be changed if there are global variables or a function with the same name in the workspace being transferred.

The next example shows how *FREP* represents two sample functions.

```
    ∇ TEST
[1]  ONE
[2]  TWO
[3]  THREE
   ∇

      FREP 'TEST'
32FTEST 2 4 5 TEST ONE  TWO  THREE

     ∇ R←A ADD B
[1]    R←A+B
     ∇

      FREP 'ADD'
29FADD 2 2 9 R←A ADD BR←A+B
```

Level 2 Description

2.1 Multiple Canonical Representation Vectors

Canonical representation vectors can be catenated without including a separator character. For example, consider

```
      (VREP 'NIL'),VREP 'CAC'
15NNIL 4 0 0 0 0 16NCAC 1 3 0 ¯1 ¯2
```

Since *VREP* inserts a blank after the shape vector but *NIL* is an empty variable, the representation of *NIL* ends with a blank. The blank character is not needed to separate the two canonical representation vectors, as shown in the following example.

```
      (VREP 'CAC'),VREP 'NIL'
16NCAC 1 3 0 ¯1 ¯215NNIL 4 0 0 0 0
```

No blank is needed to separate the last character in the representation of *CAC* from the first character in the representation of *NIL*, because the first vector length of 16 is sufficient to indicate the end of the first canonical representation vector.

2.2 **Workspace Environment**

Comparison tolerance, index origin, latent expression, printing precision, printing width, and random link can all be transferred if the system variables ⎕*CT*, ⎕*IO*, ⎕*LX*, ⎕*PP*, ⎕*PW*, and ⎕*RL* are included among the variables being transferred.

If the receiving site does not support certain system variables, they can be displayed and bypassed when encountered in processing. If the sending site does not support system variables at all, environmental information can still be transferred by generating counterfeit system variables. The function *QREP*, defined below, produces a canonical representation vector from arbitrary character data and can be used to generate counterfeit system variables.

```
     ∇ ΔR←ΔV QREP ΔA
[1]   ⍝ ΔV IS THE SYSTEM VARIABLE SETTING.
[2]   ⍝ ΔA IS THE NAME OF THE SYSTEM VARIABLE
[3]   ⍝ CATENATE NAME, RANK, SHAPE VECTOR, AND EXTERNAL VALUE.
[4]    ΔR←ΔA,' ',(⍕(⍴⍴ΔV),⍴ΔV),' ',⍕,ΔV
[5]   ⍝ APPEND DATA TYPE.
[6]    ΔR←'NC'[(⍳1)+' '=1↑0⍴ΔV],ΔR
[7]   ⍝ APPEND VECTOR LENGTH.
[8]    ΔR←(⍕⍴ΔR),ΔR
     ∇
```

In the following example, an environment is described in which the index origin is 1, the printing precision is 16 digits, and the latent expression is a call to the function *AUTO*.

```
      (1 QREP '⎕IO'),16 QREP '⎕PP'
8N⎕IO 0 19N⎕PP 0 16
      'AUTO' QREP '⎕LX'
13C⎕LX 1 4 AUTO
```

The decision as to which system variables (if any) to include in a transfer is made by the sending installation. The six system variables named above are examples of those that may commonly be chosen for inclusion.

Some environmental information, such as the state indicator, is not normally captured in an interchange. Section 2.4 contains details on how such information can be represented.

2.3 **Pseudovariables**

A pseudovariable is a character array whose canonical representation vector has type *P* rather than *C*. Such variables are included in streams of canonical representation vectors to provide control information and delimit workspaces. The pseudovariables defined by this standard are detailed below.

A stream constructed to conform with this standard should not include pseudovariables with names other than those defined herein.

2.3.1 **Interchange Standard Identifier**

A pseudovariable canonical representation vector named *WSIS* (workspace interchange standard) having as its value the character scalar 0 (zero) must be the first canonical representation vector of a stream. It identifies the stream as having been created in accordance with this standard.

The "value" of *WSIS* is the name of the particular workspace interchange standard being used. It is anticipated that there will be future new standards based upon this one. Each new standard, including revisions to this one, will have a different name to allow the receiving site to determine the basis of the exchange.

Thus any stream created under this standard should begin as follows (leading blanks can be inserted freely):

```
9PWSIS 0 0
```

2.3.2 **Translate**

A pseudovariable canonical representation vector name *TRANSLATE* must appear as the second canonical representation vector of a stream. Its value is a matrix which defines a translation table as discussed in the Level 3 Description. See Section 3.2 for details.

2.3.3 **Note**

A pseudovariable canonical representation vector named *NOTE* can be inserted freely into a stream between canonical representation vectors. Its value is treated as a comment. No restriction is placed upon a *NOTE* except that its presence or absence must not affect the outcome of the interpretation of the remainder of the stream. For example, it is not within the spirit of this standard to send, say, a group by having a special *NOTE* with the value *'GROUP'*, followed by a canonical representation vector containing the group (see Section 2.4 for information about how to include extra-standard material).

An appropriate use for *NOTE* is as documentation for a nonstandard system feature. Suppose the sending site uses a system function *⎕FOO*. In the process of building a stream of functions, one could include a special *NOTE* if *⎕FOO* were used in any of the functions. *NOTE* might explain how *⎕FOO* works and suggest how the receiving system could simulate it.

NOTE can also be used to include the name and address of the sender.

2.3.4 **Workspace Identifier**

An optional pseudovariable canonical representation vector named *WSID* identifies the workspace being transferred, and serves as a workspace delimiter when more than one workspace is contained in one stream. If this pseudovariable is included, it must be the first canonical representation vector for each workspace being transferred. Variables and functions following a *WSID* pseudovariable are associated with the same workspace until another *WSID* is encountered.

There is no fixed format for the value of the *WSID* pseudovariable, because the workspace identifier is not likely to be transferable between systems. The *WSID* used at the sending site is probably the best value to use, for it associates the workspace with its original identifier.

A typical example of *WSID* is '1234567 *WORK*'. The canonical representation vector for this *WSID* is

```
23PWSID 1 12 1234567 WORK
```

2.3.5 End of the Stream

A pseudovariable named *END* must always be the last canonical representation vector of a stream. It can be followed by other material but the presence or absence of such material must not affect the interpretation of the stream.

The *END* pseudovariable is used only to delimit the physical end of the stream. Its value has no specific meaning.

A possible *END* pseudovariable is

```
8PEND 0 0
```

2.4 Inclusion of Nonstandard Material

Canonical representation vectors of type *X* can be used to represent APL objects other than functions or variables. The <length> and <type> fields of such a canonical representation vector must conform to the definition in Section 1.1. However, the material following the type character can be in any format agreed upon by the sending and receiving sites, consistent with the <length> specification. Receiving installations must be prepared to recognize canonical representation vectors of type *X* and bypass them if necessary.

2.5 Use of Other Types

A stream constructed to conform with this standard will not include canonical representation vectors with a type other than *C*, *F*, *N*, *P*, or *X*.

Level 3 Description

3.1 Definition of the Term "Character"

In the preceding two levels of discussion, the term "character" was used without inquiring into its precise meaning. We now define "character" to mean "an index into ⎕*AV*". Thus, the number **n** will be considered to represent the character ⎕*AV*[**n**]. Such substitutions will consistently be done with ⎕*IO*←→0.

When the term "APL-ASCII character" is used, the equivalence between indices and characters is as defined in Appendix B.

3.2 "Translate" Pseudovariable

The definition of a "character" as given above is both implementation and installation dependent, since it depends upon the order in which characters appear in ⎕*AV*.

To permit the receiving installation to properly interpret a stream of characters represented as ⎕*AV* indices, the pseudovariable *TRANSLATE* is defined. The value of this pseudovariable is an array of size ((2*⌈2⊛⍴⎕*AV*),**q**), where **q**≥2, the elements of which are chosen from the APL-ASCII transmission codes.

A given row **n** of the matrix contains the APL-ASCII characters used to represent the character ⎕*AV*[**n**] at the sending site. (See Appendix B for a definition of the APL-ASCII transmission codes.) The codes can appear in any order within the row.

For a non-overstruck character, one element contains the corresponding APL-ASCII character, and the other element(s) contains the APL-ASCII character SPACE.

For an overstruck character, the row contains the two (or more) APL-ASCII characters used to form the character.

For a character (that is, an index into ⎕*AV*) that has no meaning at the sending site, each element of the row contains the APL-ASCII character NUL.

The number of columns in the matrix depends upon the maximum number of characters needed to form an overstrike. One-column matrices are disallowed.

Only one *TRANSLATE* pseudovariable may appear in a stream. It must follow the *WSIS* pseudovariable.

For example, consider a ⎕*AV* with the following value:

+⎕⍰⍲*A*<*NL*>⍬

where <*NL*> represents the character "new line", a character that does not exist in the APL-ASCII character set but can be represented as the sequence

"carriage return", "linefeed"

and ⍰ represents a character that "has no meaning" at the sending site.

As shown below, pairs of APL-ASCII characters can be used to represent the characters in this ⎕*AV*.

⎕*AV* **Index**	**Character**	APL-ASCII **Characters**		ASCII **Indices**
0	+	+	<blank>	45 32
1	⎕	⎕	<blank>	76 32
2	⍰	<NUL>	<NUL>	0 0
3	⍲	~	∧	84 41
4	*A*	*A*	<blank>	97 32
5	<*NL*>	<*CR*>	<*LF*>	13 10
6	⍬	|	o	77 79

3.3 **Mapping of Characters to Bits**

Since ⍴⎕*AV* is finite, it follows that an index into ⎕*AV* can be encoded as a finite number of bits. Specifically, the number of bits is given by the expression

```
⌈2⍟⍴⎕AV
```

This value is called the "framesize".

Streams of characters can be represented as streams of bits, where each character is encoded in "framesize" number of bits, using ⎕*AV* and ⊤ to define the mapping between characters and bits. The character ⎕*AV*[**n**] is represented as

(framesize⍴2**)**⊤**n**

The receiving installation calculates "framesize" by inspecting the *TRANSLATE* pseudovariable. This value applies to the first bit of the stream following the *TRANSLATE* pseudovariable. Starting with the first bit of the stream, up to and including the last bit of the last element of the *TRANSLATE* pseudovariable, a default framesize of 8 and the APL-ASCII translation table applies. Thus, every stream begins with the *WSIS* and *TRANSLATE* pseudovariables in 8-bit APL-ASCII characters, regardless of the "framesize" of the rest of the stream.

Continuing with the example from Section 3.2, once can map the pseudovariable *TRANSLATE* to a bit stream in the following manner:

- Form a Canonical Representation Vector

  ```
  31⍕TRANSLATE 2 7 2 + ⎕ <NUL><NUL>~∧A <CR><LF>|⍳
  ```

- Represent the Canonical Representation Vector as APL-ASCII Characters

  ```
  51 49 112 ...
  ```

  where `51, 49`, and so on are indices into the APL-ASCII table in Appendix B.

- Represent the APL-ASCII Characters as 8-bit Frames

  ```
  00110011 00110001 ...
  ```

Level 4 Description

4.1 **Physical Media**

A stream of bits representing a collection of canonical representation vectors is normally written onto a portable recording medium for transmission to another site. Many media are appropriate for such use, but this standard suggests a format only for 9-track magnetic tape.

4.2 9-Track Magnetic Tape

The bit stream is arbitrarily segmented into 8-bit **bytes**, each byte written in parallel onto the tape. The recording density, number of bytes per physical tape record, record format, presence or absence of labels, number of files per tape, and presence of trailing tape marks are all subject to negotiation between the sending and receiving sites.

However, in the absence of any other agreement, the following defaults have been set:

- ANSI standard label
- 1600 bytes per inch (BPI), 9-track
- Fixed-length blocks of 1890 bytes each, no control words
- No truncated last block -- any desired fill character(s) used to pad after the *END* pseudovariable.

Extended Example

Consider a workspace containing the following function and variable. Let the workspace name be *FOOWS*.

```
     ∇ FN
[1]   ⍕'A+B'
     ∇

      VAR←0 100
```

The canonical representation vectors for these are as follows:

```
      FREP 'FN'
22FFN 2 2 6 FN   ⍕'A+B'

      VREP 'VAR'
14NVAR 1 2 0 100
```

For this example, only one system variable is included.

```
      ⎕IO←1
      VREP '⎕IO'
8N⎕IO 0 1
```

The entire workspace is now represented as follows:

```
15PWSID 1 5 FOOWS8N⎕IO 0 122FFN 2 2 6 FN   ⍕'A+B'14NVAR 1 2 0 100
```

(Note that the first canonical representation vector is the pseudovariable *WSID*.)

If multiple workspaces are to be included in the stream, additional *WSID* pseudovariables, followed by the canonical representations of the functions and variables in those workspace, must appear.

To bring such a stream into conformance with this standard, it must be enclosed in *WSIS*, *TRANSLATE*, and *END* pseudovariables.

The *TRANSLATE* pseudovariable is considered first.

Suppose that ⎕*AV* for this system is as follows:

n	⎕*AV*[**n**]
0	+
1	⎕
2	⌽
3	*A*
4	*B*
5	*D*
6	*E*
7	*F*
8	*I*
9	*N*
10	*O*
11	*P*
12	*R*
13	*S*
14	*V*
15	*W*
16	0
17	1
18	2
19	3
20	4
21	5
22	6
23	8
24	<blank>
25	'

The *TRANSLATE* pseudovariable defines a matrix of shape `32 2`, since `32 ↔ 2*⌈2⊛26` and the only overstrike in ⎕*AV* requires just two APL-ASCII characters to represent it. The value of the *TRANSLATE* pseudovariable is as follows:

Row	Column 1	Column 2
0	+	<blank>
1	⍞	<blank>
2	|	○
3	*A*	<blank>
4	*B*	<blank>
5	*D*	<blank>
6	*E*	<blank>
7	*F*	<blank>
8	*I*	<blank>
9	*N*	<blank>
10	*O*	<blank>
11	*P*	<blank>
12	*R*	<blank>
13	*S*	<blank>
14	*V*	<blank>
15	*W*	<blank>
16	0	<blank>
17	1	<blank>
18	2	<blank>
19	3	<blank>
20	4	<blank>
21	5	<blank>
22	6	<blank>
23	8	<blank>
24	<blank>	<blank>
25	'	<blank>
26	<NUL>	<NUL>
27	<NUL>	<NUL>
28	<NUL>	<NUL>
29	<NUL>	<NUL>
30	<NUL>	<NUL>
31	<NUL>	<NUL>

The first row of the pseudovariable consists of the APL-ASCII characters plus (+) and blank, indicating that the first position of this installation's ⍞*AV* is "+".

Since a valid representation of a workspace must begin with a *WSIS* pseudovariable to identify it as being created under this standard, a complete representation of workspace *FOOWS* would appear as follows:

```
9PWSIS 0 082PTRANSLATE 2 32 2 + ⍞ |○A B D E F I N O P R S V W 0 1 2 3 4 5 6 8
  ' <NUL><NUL><NUL><NUL><NUL><NUL><NUL><NUL><NUL><NUL><NUL>
 <NUL>15PWSID 1 5 FOOWS8N⍞IO 0 122FFN 2 2 6 FN    ⍵'A+B'14NVAR
 1 2 0 1008PEND 0 0
```

The above stream must be represented as a bit stream before it can be written onto magnetic tape. This is considered in two steps. Shown below is the bit stream that represents the *WSIS* and *TRANSLATE* pseudovariables in APL-ASCII keyboard-pairing codes.

```
9       P       W       S       I       S          ...
00111001011100000111011101110011011010001011100 11 ...
```

Above each group of bits, the proper character interpretation of those bits has been indicated. For example, since the stream begins with the default interpretation (see Section 3.3), the first character is represented as the bit stream `00111001` (decimal 57), which is the APL-ASCII character "9" (see Appendix B). Similarly, the second character is represented as `01110000` (decimal `112`), which is the APL-ASCII character "*P*", and so forth.

The rest of the stream must be constructed in accordance with the *TRANSLATE* pseudovariable. Since the number of rows in *TRANSLATE* is `32`, the framesize is `5` since `5←→2⊛32`. The rest of the stream is as follows:

```
1    5    P    W    S    I    D    BLANK1    BLANK    ...
10001101010101101111011010100001011100010001110000  ...
```

Again, the appropriate interpretation of each frame of bits has been placed above that frame. For example, the first frame is `10001` (decimal `17`). Examining row 17 of the *TRANSLATE* pseudovariable indicates that the graphic for this frame is "1". The remaining frames are treated in a similar manner.

Reprinted from:
"APL Quote Quad"
December 1977

APPENDIX B
APL/ASCII CHARACTER SET

APL/ASCII typewriter-pairing transmission codes; *ARBOUT* value is row+16×col and is also shown in lower portion of each box.

	0	1	2	3	4	5	6	7
0	*NUL* 0	*DLE* 16	*SP* 32	0 48	¯ 64	⋆ 80	◇ 96	*P* 112
1	*SOH* 1	*DC1* 17	¨ 33	1 49	⍺ 65	? 81	*A* 97	*Q* 113
2	*STX* 2	*DC2* 18	) 34	2 50	⊥ 66	⍴ 82	*B* 98	*R* 114
3	*ETX* 3	*DC3* 19	< 35	3 51	∩ 67	⌈ 83	*C* 99	*S* 115
4	*EOT* 4	*DC4* 20	≤ 36	4 52	⌊ 68	~ 84	*D* 100	*T* 116
5	*ENQ* 5	*NAK* 21	= 37	5 53	∈ 69	⍋ 85	*E* 101	*U* 117
6	*ACK* 6	*SYN* 22	> 38	6 54	_ 70	∪ 86	*F* 102	*V* 118
7	*BEL* 7	*ETB* 23	] 39	7 55	∇ 71	⍵ 87	*G* 103	*W* 119
8	*BS* 8	*CAN* 24	∨ 40	8 56	∆ 72	⊃ 88	*H* 104	*X* 120
9	*HT* 9	*EM* 25	∧ 41	9 57	⍳ 73	⍎ 89	*I* 105	*Y* 121
10	*LF* 10	*SUB* 26	≠ 42	(58	∘ 74	⊂ 90	*J* 106	*Z* 122
11	*VT* 11	*ESC* 27	÷ 43	[59	' 75	← 91	*K* 107	{ 123
12	*FF* 12	*FS* 28	, 44	; 60	⍬ 76	⊢ 92	*L* 108	⊣ 124
13	*CR* 13	*GS* 29	+ 45	× 61	| 77	→ 93	*M* 109	} 125
14	*SO* 14	*RS* 30	. 46	: 62	⍑ 78	≥ 94	*N* 110	$ 126
15	*SI* 15	*US* 31	/ 47	\ 63	○ 79	- 95	*O* 111	*DEL* 127

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

SATN 23
15 JUL 78
Rev. 1

SHARP APL TECHNICAL NOTES

TITLE: Comparison Tolerance

ABSTRACT: The definition of tolerant comparison used in SHARP APL has been changed. This SATN describes the justification for and effects of that change.

KEYWORDS: Tolerance
Comparison
$\square CT$
Fuzz
Floating point
Error
Fraction

Error is inevitable in the computation and representation of numbers in any computer system. This is particularly true for large numbers and numbers containing a non-zero fraction.

APL facilitates the comparison of such numbers by providing the concept of a comparison tolerance, whereby two numbers are considered equal if they fall within this tolerance. The tolerance is controlled by the value of the system variable $\square CT$.

$\square CT$ defines a relational tolerance dependent on the magnitude of the larger of the numbers being compared. The definition used in the past, however, was tailored too much to the idiosyncracies of the computer, and too little to the needs of the APL user. It was difficult to model in APL, and yielded anomalous results.

R.H. Lathwell proposed a more straightforward definition of tolerant comparison (Appendix 1) that does not differ greatly, in practical terms, from the old one, but is easier to comprehend.

This definition has been implemented in SHARP APL, with the exception of Lathwell's definitions of floor and ceiling. The definitions of floor and ceiling used in SHARP APL were developed by D.L. Forkes and R. Bernecky (I.P. Sharp Associates, Toronto) and represent a slight variant of an algorithm proposed by L.M. Breed (IBM) - reference: IBM Technical Report TR03.023.

SHARP APL utilizes tolerant comparison in the following primitive functions, because they all imply some form of comparison between two numbers.

less than	$A<B$
less than or equal	$A\leq B$
equal	$A=B$
greater than or equal	$A\geq B$
greater than	$A>B$
not equal	$A\neq B$
floor	$\lfloor A$
ceiling	$\lceil A$
membership	$A\in B$
index of	$A\iota B$

It is emphasized that no other primitive functions in SHARP APL use tolerant comparison.

Tolerant comparison considers two numbers to be equal if they are within some neighborhood. The neighborhood has a radius of $\square CT$ times the larger of the two in absolute value. In the following, all primitive functions are exact (i.e. tolerant comparison is not used).

The formal definition for tolerant equality is now:

```
      ∇ R←A TEQ B
[1]    R←(|A-B)≤⎕CT×(|A)⌈|B
      ∇
```

Similarly,

TNE	$A\neq B$	`~A TEQ B`
TLT	$A<B$	$(A<B)\wedge$`A TNE B`
TLE	$A\leq B$	$(A\leq B)\vee$`A TEQ B`
TGE	$A\geq B$	$(A\geq B)\vee$`A TEQ B`
TGT	$A>B$	$(A>B)\wedge$`A TNE B`
TEPS	$A\in B$	`∨/A∘.TEQ,B`
TIOTA	$A\iota B$	`⎕IO++/∧\B∘.TNE A`
FLOOR	$\lfloor A$	`(⌊.5+A)-(⌊.5+A) TGT A`
CEILING	$\lceil A$	`(⌊.5+A)+(⌊.5+A) TLT A`

FLOOR AND CEILING

Choice of a suitable definition for floor and ceiling is more subtle than one might expect. Anomalies tend to creep in when $\square CT$ and the argument are large enough so that both integers adjacent to the argument are tolerantly equal to the argument.

Lathwell's definition of floor as printed in the APL/76 proceedings results in absolute fuzzing for arguments >1:

```
     ∇R←FO A
[1]   R←⌊A+⎕CT×1⌊|A              ⍝ wrong
     ∇
```

Various decorations or changes to Lathwell's definitions don't help:

```
     ∇R←F1 A
[1]    R←⌊A+⎕CT×1⌈|A                ⍝ can produce results many integers away from A
     ∇


     ∇R←F2 A
[1]    R←⌊A+1⌊⎕CT×1⌈|A              ⍝ fails for large integral values of A
     ∇
```

We could alter the definition to:

```
     ∇R←F3 A
[1]    R←⌊A+ALMOST1⌊⎕CT×1⌈|A
```

but the result is non-intuitive, and the value of *ALMOST*1 is machine-dependent.

Forkes and Bernecky produced the following definition:

```
     ∇R←F4 A
[1]    R←⌊A
[2]    R←R+(A TEQ R+1)∧A TNE R
```

The result is the smaller integer, unless the argument is tolerantly equal only to the larger integer.

Breed pointed out an anomaly in this definition, for large arguments, and increasing values of ⎕*CT*:

```
      A←123456789123.8

      ⎕CT←1E¯15 ◇ F4 A
123456789123

      ⎕CT←5E¯12 ◇ F4 A
123456789124

      ⎕CT←1E¯10 ◇ F4 A
123456789123
```

This oscillation of the result is undesirable.

Breed proposed the following algorithm:

```
    ∇R←F5 A
[1]   R←(×A)×⌊ALMOSTPOINT5+A   ⍝ nearest integer to A.
[2]   R←R-(R-A)>⎕CT×1⌈|A       ⍝ TGT, except near zero.
    ∇
```

This algorithm doesn't oscillate, but still contains an arcane, machine-dependent constant.

Forkes then suggested a machine-independent variant that yields results only slightly different from *F5*:

```
    ∇R←FLOOR A
[1]   R←(×A)×⌊.5+|A            ⍝ nearest integer to A.
[2]   R←R-(R-A)>⎕CT×1⌈|A
    ∇
```

FLOOR differs functionally from *F5* only in the treatment of large values of *A* such that

```
(.5=1|A)∧.5≤⎕CT×|A.
```

In such cases, *F5* chooses the integer closer to zero, whereas *FLOOR* chooses the integer farther away from zero. Since the choice is arbitrary, and since it is common practice to round *N*.5 up to *N*+1, this seems reasonable.

FLOOR is the algorithm now implemented in SHARP APL. The algorithm for ceiling is:

```
    ∇R←CEILING A
[1]   R←-FLOOR-A
    ∇
```

Breed has recently published "Definitions for Fuzzy Floor and Ceiling", an IBM technical report TR03.024 - dated March 1977. The report is an analysis of several algorithms for floor and ceiling, including the *FLOOR* function presented here.

SIGNIFICANCE OF THE NEW DEFINITION

1) Setting ⎕*CT* to a given value now implies that, to be considered equal, two numbers must be identical to approximately `10⊛÷⎕CT` decimal digits.

   Note: SHARP APL restricts the maximum value of ⎕*CT* to be `<16*¯8`.

2) It follows directly from the definition on page 2 that comparison of numbers of unlike signs, or comparison with zero, is exact.

3) The current implementation of SHARP APL should produce results identical to the models presented here.

4) The definition is independent of the machine on which it is implemented. It is intended to facilitate interchange of applications.

5) Certain anomalies in the old definition disappear. Previously, comparisons of numbers very close to powers of 16 tended to produce strange results:

   Figure 1 (Page 6) shows the regions of tolerant equality for both definitions, using an absurdly large value of ⎕*CT* (`.1`).

   Consider the point *X*=*UGLY*, and the monotonically increasing points *Y*=*EQ*1, *NE*1, *EQ*2, *NE*2.

   Note that `(`*UGLY*`=`*EQ*`1)∧(`*UGLY*`≠`*NE*`1)` but *UGLY*=*EQ*2 as well, even though *EQ*2 is further away from *UGLY* than *NE*1.

   Prior to the change in definition, the following could be observed:

```
     ∇FOO;⎕CT;⎕IO;X
[1]   ⎕CT←0 ◇ ⎕IO←0
[2]   X←1+(⍎1↓(¯2×⍳10)×16*¯14),(2×⍳10)×16*¯13 ⍝ admittedly strange.
[3]   ⎕←⊖'.○'[X∘.=X]
     ∇
```

   See figure 4 - page 6.

6) It should be stressed that only comparisons between values that are on the brink of equality are affected. Essentially, a rather fuzzy border has been altered slightly so that, while still fuzzy, it is at least straight.

   Compare figures 1, 2 and 3 to view the effect of changing ⎕*CT*. Notice that the affected regions rapidly become quite small.

FIGURE 1

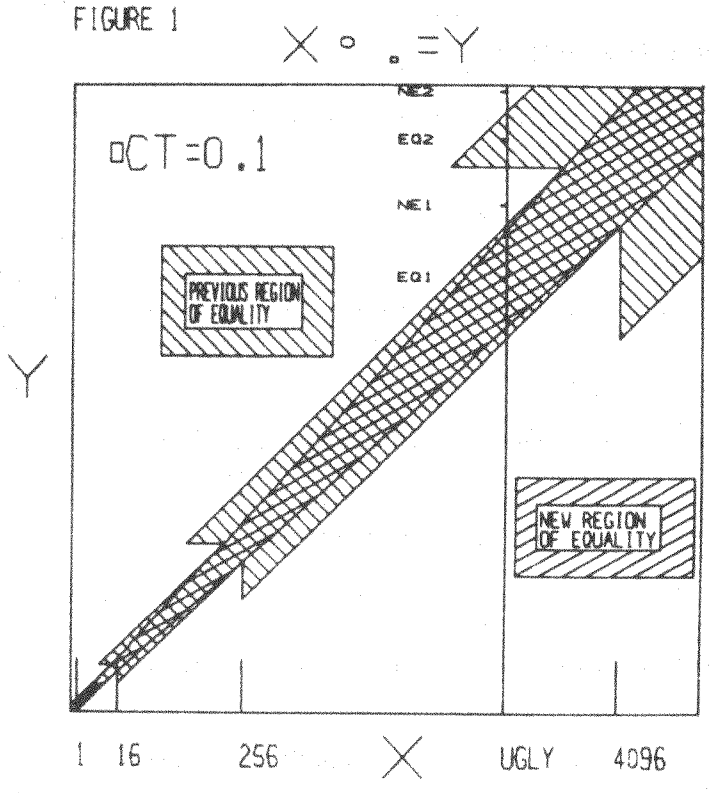


FIGURE 2

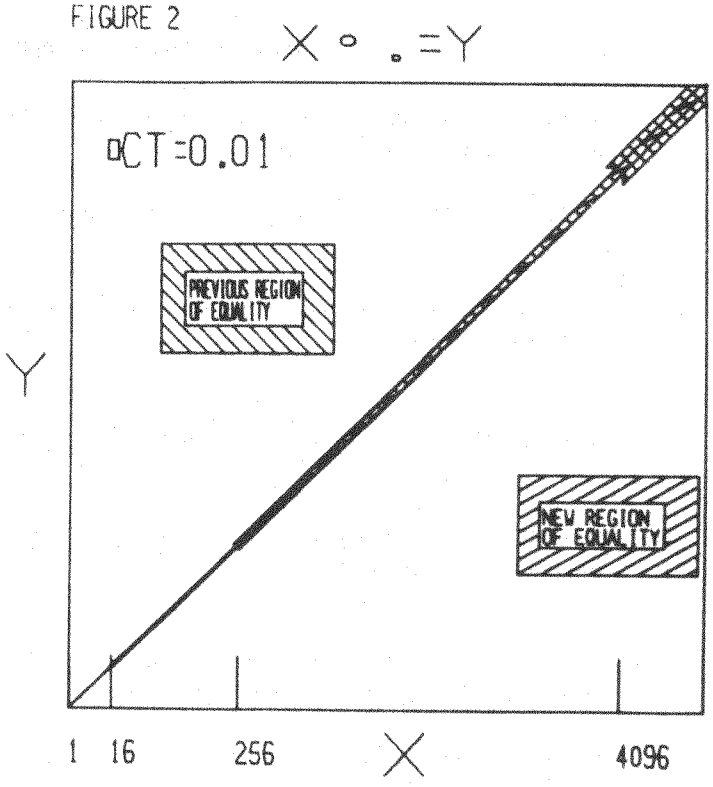
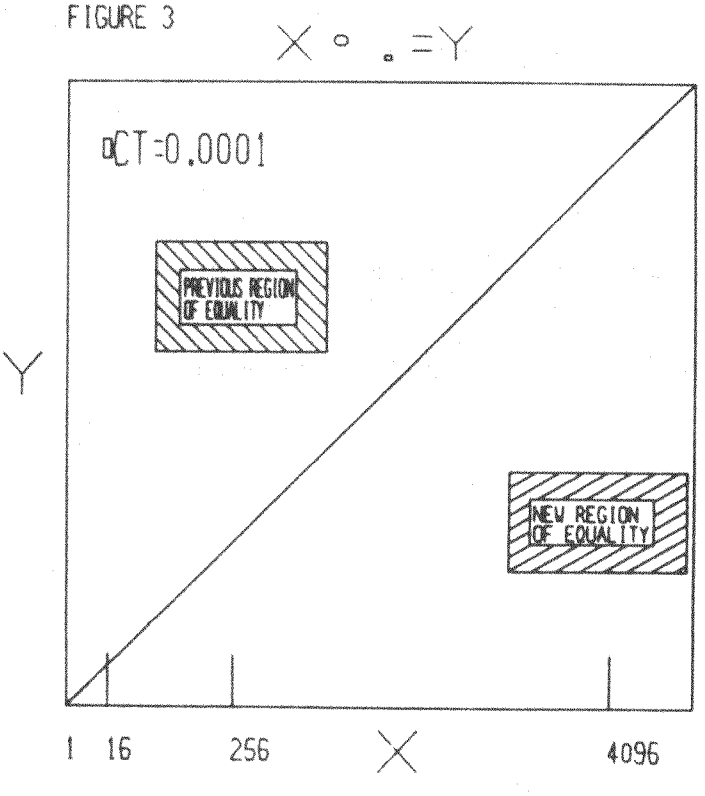


FIGURE 3



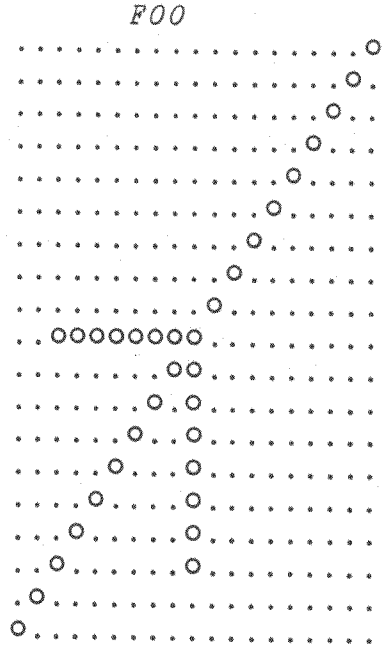


FIGURE 4

IMPLICATIONS OF $\square CT\leftarrow 0$

1) It is obvious from the new definition of tolerant comparison that setting $\square CT\leftarrow 0$ yields exact mathematical relational functions.

```
(|A-B)≤□CT×(|A)⌈|B
(|A-B)≤0×(|A)⌈|B
(|A-B)≤0                equal if and only if A=B
```

2) APL interpreter performance is improved, due to the utilization of more efficient algorithms. For most affected functions the improvement is measurable but not significant. For dyadic epsilon, certain inner products (`∧.=` , `∨.≠`) and dyadic iota, the improvement can be considerable.

3) User functions that attempt to be exact (rounding functions are a common example) can be written in a more obvious manner:

Rounding Functions

```
     ∇ R←RND1 A
[1]    R←⌊.5+A
     ∇
```

This function fails because `⌊` is fuzzed and `⌊.5+3.4999999` might produce `4` as a result.

```
     ∇ R←RND2 A
[1]    R←⌊(.5+A)-1|.5+A
     ∇
```

This works, regardless of the setting of $\square CT$, but is a bit obscure. (Function, courtesy of Doug Forkes)

$\square CT\leftarrow 0$ allows use of the obvious rounding function again.

```
     ∇ R←RND3 A;□CT
[1]    □CT←0 ◇ R←⌊.5+A
     ∇
```

4) **Searching**

Users frequently encode search arguments as floating point numbers, and use dyadic iota to search a catalog. In such a search exact comparison is absolutely necessary. The following function not only achieves exact comparison, but runs (assuming both arguments have many elements) considerably faster than with $\square CT\neq 0$.

```
     ∇ R←A EXACTIOTA B;□CT
[1]    □CT←0 ◇ R←A⍳B
     ∇
```

APPENDIX 1

APL Comparison Tolerance

R.H. Lathwell
IBM APL Design, Philadelphia

INTRODUCTION

An important aspect of the original implementation of APL/360 was the treatment of arithmetic functions as abstract functions defined on the continuous set of real numbers [1,2]. This had various consequences, including automatic conversion between different machine representations of numbers so that storage and other aspects of system 360 architecture could be used efficiently while suppressing its details. [2]

"Fuzzy" comparisons were introduced so that the actual discrete, fixed-precision, hexadecimal representation could be partly disguised: for example, a comparison between 7 and +/10ρ0.7 should regard the two values as equal. (0.7 cannot be exactly represented as a hexadecimal fraction, and the difficulty can be observed on an IBM System/370 APL system by displaying 7-+/10ρ0.7).

The technique used in the original implementation as to regard numbers whose difference was zero in the first twelve (of fourteen) hexadecimal digits to be equal. The definitions in [3] were developed by M.A. Jenkins and R.H. Lathwell in 1968 when it was found that, for some values, implications of relational functions such as $B>A \leftrightarrow \sim B \leq A$ were violated. It was later discovered that these definitions did not completely correct the difficulty, and when APL system variables were introduced [4], the tolerant functions were rederived, resulting in the definitions given here.

Approximate Functions

Computer arithmetic functions (addition, subtraction, multiplication and division) are approximations: the set of real numbers is represented by a set of discrete numbers and the functions map arguments chosen from this set to range members which are closest to the abstract results. Because the values produced by two expressions related by an identity might not be identical, a primitive form of **tolerant comparison** is required to make common identities hold.

The notion of comparing numbers in mathematics arises in several guises, as in the idea that a sequence of numbers $B[1]$, $B[2]$, ... may approach some limiting value A. This is normally defined in terms of whether B belongs to some **neighborhood** of A. The neighborhood is in turn defined in terms of a **radius** about A.

In order to make comparison symmetric in the two arguments for equality and inequality, the function which determines the radius must depend on both arguments A and B; to allow control of the tolerance, it must also depend on a **comparison tolerance** value.

The implemented relational functions ($<, \leq, =, \geq, >, \neq$) are approximations and use the value of the system variable $\square CT$ (for comparison tolerance) as a parameter, as do inverse indexing and membership (dyadic ⍳ and ∈), because of the implied comparison, and floor and ceiling because of the implied comparison to integers.

In the following discussion, APL primitive symbols such as ≠ denote exact algebraic functions, while names such as *TNE* are used to denote tolerant functions. Function definitions are given in the form *F*:α+ω; the colon separates the name of the function from the expression which defines it. The special symbols α and ω represent the left and right arguments, respectively, of a dyadic function. Either symbol can be used as the argument of a monadic function. [5]

The dependence on □*CT* of the radius of the neighborhood may as well be linear, and to make the notion of tolerant comparison the practical one commonly used, for example, in engineering, it should also be a function of the magnitudes of the comparands. The following definition is therefore adopted for the radius function *RCT* (relative comparison tolerance):

```
RCT:□CT×α F ω
```

Where *F* is a non-negative valued function of the magnitudes of its arguments.

Tolerance Relations

Using the radius function *RCT*, the relational functions are defined to take into consideration the neighborhood of equality.

Consider the following identity:

```
A≠B↔0≠A-B
   ↔0<|A-B
```

Tolerant inequality must exclude the neighborhood of equality, and hence can be defined as:

```
TNE:(α RCT ω)<|α-ω
```

Similarly, tolerant greater must exclude the neighborhood, that is, *A*>*B*↔0>*B*-*A*, and therefore

```
TGT:(α RCT ω)<α-ω.
```

Substituting for *RCT*:

```
A TNE B  ↔(A RCT B)<|A-B
         ↔0<(|A-B)-A RCT B
         ↔0<(|A-B)-□CT×A F B
         ↔(A F B)<(|B-A)÷□CT        (□CT≠0)
```

and *A F B* is bounded by (|*A*)⌈|*B*.

This bound is, in fact, a satisfactory function and:

$$F:(|\alpha)\lceil|\omega$$

and therefore $RCT:\square CT\times(|\alpha)\lceil|\omega$, and symmetry is assured by the commutativity of $\lceil$. Tolerant inequality then becomes

$$TNE:0<(|\alpha-\omega)-\square CT\times(|\alpha)\lceil|\omega$$

A neighborhood of radius $\square CT\times(|A)\lceil|B$ is defined, and A and B are unequal if one is not contained within this neighborhood about the other.

Tolerant inequality can be expressed in terms of functions which are independent of F as follows:

$$\begin{aligned} A\ TNE\ B &\leftrightarrow 0<(|A-B)-\square CT\times(|A)\lceil|B \\ &\leftrightarrow 0\lceil\times(|A-B)-\square CT\times(|A)\lceil|B \end{aligned}$$

and

$$\begin{aligned} A\ TGT\ B &\leftrightarrow 0<(A-B)-\square CT\times(|A)\lceil|B \\ &\leftrightarrow 0\lceil\times(A-B)-\square CT\times(|A)\lceil|B \end{aligned}$$

In order to ensure that the relational functions satisfy their fundamental identities, the remaining four functions are defined in terms of TNE and TGT.

$TEQ:\sim\alpha\ TNE\ \omega$	(tolerant equality)
$TLE:\sim\alpha\ TGT\ \omega$	(tolerant not greater)
$TLT:(\alpha\ TNE\ \omega)\wedge\sim\alpha\ TGT\ \omega$	(tolerant less than)
$TGE:(\alpha\ TGT\ \omega)\wedge\sim\alpha\ TNE\ \omega$	(tolerant not less)

Tolerant floor and ceiling

The fundamental identities which floor and ceiling must satisfy are:

$$(\lfloor A)\leq A$$
$$(\lceil A)\geq A$$
$$(\lfloor A)=-\lceil -A$$

Substituting tolerant relational functions, these become:

$$(TFL\ A)TLE\ A$$
$$(TCL\ A)TGE\ A$$
$$(TFL\ A)TEQ\text{-}TCL\text{-}A$$

If tolerant floor (TFL) is expressed in terms of the abstract function as $TFL\ A \leftrightarrow \lfloor A+G\ A$, and tolerant ceiling as $TCL\ A \leftrightarrow \lceil A-G\ A$, where G is a measure of the error and depends on $\square CT$, and the appropriate substitutions made, then

$$(G\ A) \leq \square CT \times |A$$

and, using the upper bound,

$$TFL : \lfloor \omega + \square CT \times |\omega$$
$$TCL : \lceil \omega - \square CT \times |\omega$$

In the above, for sufficiently large values of $\square CT$ and $|A$, there are several integers which will satisfy the tolerant relations. If tolerant floor and ceiling are constrained so that the result is an integer adjacent to A, then

$$((TFL\ A) = \lfloor A) \vee (TFL\ A) = \lceil A$$

Applying this additional constraint results in

$$TFL : \lfloor \omega + \square CT \times 1 \rfloor |\omega$$
$$TCL : \lceil \omega - \square CT \times 1 \rfloor |\omega$$

and $\square CT < 1$

Note that, in effect, an upper bound for $\square CT$ is established.

Discussion

The functions TNE, TGT, TEQ, TGE, TLT, TLE, TFL, and TCL are approximations to the mathematical functions $\neq$, $>, \geq, <, <, \lfloor$ and $\lceil$. They use the value of $\square CT$ to determine a neighborhood of equality. When $\square CT = 0$, the approximations become the exact functions.

Absolute comparison, independent of $\square CT$, can also be obtained by comparisons with zero:

$$\begin{aligned} A\ TNE\ 0 &\leftrightarrow 0 < (|A-0) - \square CT \times (|A) \lceil 0 \\ &\leftrightarrow 0 < (|A) \times 1 - \square CT \\ &\leftrightarrow A \neq 0 \end{aligned}$$

When $\square CT > 0$, the tolerant functions lose the property of transitivity, and algebraic relations which depend on this property cannot be used. For example,

$A\ TNE\ B \leftrightarrow (A+C) \nabla TNE \nabla B+C$ does not hold (consider $C=-A$).

The intransitivity of equality is well-known in practical situations and can be easily demonstrated by sawing several pieces of wood of equal length. In one case, use the first piece to measure subsequent lengths; in the second case, use the last piece cut to measure the next. Compare the lengths of the two final pieces.

Large values of $\square CT$ cause the tolerant functions to yield results which are mathematically consistent but which are, to some extent, counter-intuitive. This generally happens when the neighborhood of equality about an integer includes an adjacent integer:

```
A TEQ A+1←→~A TNE A+1
        ←→(⎕CT×|A+1)≥1
        ←→(|A+1)≥÷⎕CT          (⎕CT≠0)
```

So that, for any particular non-zero value of $\square CT$, adjacent integers greater in magnitude than $\div\square CT$ will be equal. For small values of $\square CT$, tolerant functions yield intuitively correct results in cases where the exact result is not `(7=+/10⍴0.7)`.

$\square CT$ can, as was the "fuzz" in APL/360, be regarded as a measure of the number of significant digits (approximately $\div\square CT$ decimal digits) to which numbers must agree in order to be considered equal. In general, $\square CT$ should be chosen to be the smallest value which is large enough to mask common arithmetic errors.

Acknowledgement

D.L. Orth provided many useful suggestions in the course of discussion and careful review of the derivations.

References

1. L.M. Breed and R.H. Lathwell, "Implementation of APL 360", symposium on interactive systems for experimental applied mathematics, eds., M. Klerer and J. Reinfelds, Academic Press, New York, 1968.

2. A.D. Falkoff and K.E. Iverson, "The APL 360 Terminal System", eds., M. Klerer and J. Reinfelds, Academic Press, New York, 1968.

3. R.H. Lathwell and J.E. Mezei, "A Formal Description of APL", Colloque APL, Institute de Recherche D'Informatique, Rocquencourt, France, 1971.

4. A.D. Falkoff and K.E. Iverson, "The Design of APL", IBM J. Res. Develop. 17, 353, (July 1973).

5. K.E. Iverson, "Elementary Analysis", APL Press, Swarthmore, 1976.

From: Proceedings of the conference APL76 by ACM-STAPL.

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

SATN 24
23 MAR 77

SHARP APL TECHNICAL NOTES

TITLE: Symbol Table

ABSTRACT:)*SYMBOLS* can increase and decrease the workspace symbol table size and discard unreferenced symbol table entries.
The symbol table is garbage collected during 3 ⎕*FD* and ⍎, and package functions such as ⎕*PDEF*, if required.

KEYWORDS:

SHARP AR TECHNICAL NOTES

)SYMBOLS N attempts to change the size of the workspace symbol table to approximately *N* . In fact, due to the hashing algorithm used in symbol look-up, *N* will be increased to:

```
N1←¯1+21×⌊1+N÷21
```

That is, the requested size may be exceeded by as much as 20 entries.

If the number of referenced symbols in the workspace is greater than *N1* , *SYMBOL TABLE FULL* is reported, and no other action is taken.

If there is insufficient room in the workspace for the requested size of symbol table, *WS FULL* is reported, and no other action is taken.

Otherwise all unreferenced symbols are discarded and the symbol table size is set to *N1* symbols.

Note that even if the size is unchanged, unreferenced symbols are eliminated.

A symbol is not removed from the symbol table if it is a system variable, a defined variable, or if it is referenced, that is, if it appears:

a) In any line (including the header) of any function.

b) In any statement in the *)SI* stack.

c) In the immediate execution statement (the "last line" which may be recalled and edited via the *)N* edit feature). This statement may be discarded, along with the *)SI* stack, by *)RESET*.

Symbol Table Garbage Collection

The functions 3 ⎕*FD* and ⍎ and package functions such as ⎕*PDEF* now discard unreferenced names from the symbol table if it is necessary to do so to avoid *SYMBOL TABLE FULL*. They will not increase the size of the symbol table.

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

SATN 25
15 MAY 1977

SHARP APL TECHNICAL NOTES

TITLE: Extensions To Argument Passing

ABSTRACT: a) A user-defined dyadic function can now be used monadically.

b) If the name of the result of a function matches the name of an argument, the result will be initialized to the value of the argument.

KEYWORDS: Argument
Dyadic
Function
Monadic
Result

A) Users are undoubtedly aware that primitive APL functions may often be used both monadically and dyadically.
For example:

```
A-B
```

and

```
-B
```

are both permitted, the latter being equivalent to:

```
0-B
```

This feature has now been extended to user-defined functions.

If a user-defined function is dyadic it may be called monadically, in which case its left argument will be undefined.

This case may be detected by `4 ⎕WS 'NAME'` which returns the workspace consumed by *NAME* and will be `0` if and only if *NAME* is undefined.

Consider the following example:

```
    ∇ R←N ROUND X;⎕CT
[1]   ⍝ ROUNDS <X> TO <N> DECIMAL PLACES
[2]   ⎕CT←0 ⍝ SO ⌊ IS EXACT

[3]   →(0≠4 ⎕WS 'N')/OK ⍝ IF <N> PROVIDED
[4]   N←0 ⍝ DEFAULT - SIMPLE ROUNDING
[5]  OK:X←X×10*N ⍝ SCALE UP
[6]   R←⌊X+0.5 ⍝ ROUND
[7]   R←R÷10*N ⍝ SCALE DOWN
    ∇
```

Notice how omission of *N* results in the common case of rounding to the nearest integer.

To use the function:

```
      2 ROUND 1.234  1.236
1.23  1.24

      ROUND 1.234  1.236
1  1
```

The same function might achieve its effect by completely avoiding *N* when omitted:

```
    ∇ R←N ROUND X;⎕CT
[1]  ⍝ ROUNDS <X> TO <N> DECIMAL PLACES
[2]   ⎕CT←0 ⍝ SO ⌊ IS EXACT

[3]   →(0≠4 ⎕NC 'N')/OK ⍝ IF <N> PROVIDED
[4]   R←⌊X+0.5 ⍝ SIMPLE ROUNDING
5]    →0 ⍝ RETURN
[6]   OK:X←X×10*N ⍝ SCALE UP
[7]   R←⌊X+0.5 ⍝ ROUND
[8]   R←R÷10*N ⍝ SCALE DOWN
    ∇
```

Three natural applications of the optionally omitted argument are as follows:

1) Omitting a frequently occurring value, and adopting it as a default. This is illustrated in the first example. It is also the rationale behind monadic - ÷ * ⊗ ⌹ .

2) Using the omitted argument to indicate *ALL VALUES*. In the past this has required passing a value inappropriate to the function as data, as a flag to say *ALL VALUES* or perhaps *NO VALUES*. Now it is possible to use omission of the left argument for this purpose. It is this use which is seen in monadic ⎕*PDEF*.

3) Permitting a function to perform two related functions - one monadic and one dyadic. Primitive functions that exhibit this trait are:

   ? ! ⍎ .

   This third application is subject to abuse, in that users may choose to perform totally unrelated functions. APL is probably guilty of this in the case of | but has the valid excuse of a limited set of primitive symbols (dictated by the number of keys on the typewriter). However, when used sensibly, the technique is a valuable mnemonic.

B) The name of the result of a function may now match one of its arguments. This is primarily an aid to readability and style, although in some cases it can reduce slightly the size of a function and avoid *WSFULL*.

Consider the rounding function listed above. At some arbitrary point the author had to stop using the argument *X*, and begin using the result *R*. Usually this is done on the first or last line of a function, but in the example the author selected the second-last line to illustrate how irrelevant, and even distracting, the matter is. In this function, *X AND R* are really the same object, undergoing a transformation.

```
    ∇ R←ROUND R
[1]   R←⌊R+0.5
      ∇
```

```
    ∇ R←ROUND X
[1]   R←⌊X+0.5
      ∇
```

The examples above illustrate both the new relaxed definition and the older form which naturally is still permitted.

A common opportunity to employ this technique occurs when the result is merely an indexed modification of the argument.

```
    ∇ X←I INJECT X
[1]   X[I]←0
      ∇
```

```
    ∇ R←I INJECT X
[1]   R←X
[2]   R[I]←0
      ∇
```

A second opportunity occurs when the result is the argument, in certain cases.

```
    ∇ R←SCAN R
[1]   →(1≥¯1↑⍴R)/0
[2]   R←(SCAN((-⍴⍴R)↑¯1)↓R),+/R
      ∇
```

In all the examples the result name has matched the righthand argument. It may of course match the left argument if the application so requires.

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

SHARP APL TECHNICAL NOTES

TITLE: ENHANCEMENTS TO THE FILE SUBSYSTEM

EFFECTIVE: 24 SEP 77

ABSTRACT: This SATN describes the following modifications to the *SHARP APL* File Subsystem:

1. Separate permission codes for the □*RDAC* and □*STAC* system functions.

2. A new file function, □*FHOLD* , which permits file interlocks to be secured with a passnumber.

3. A new file function, □*APPENDR* , which behaves similar to □*APPEND* but returns an explicit result of the component number just appended if the operation was successful. Also, both □*APPENDR* and □*APPEND* may now produce a *FILE INDEX ERROR* if the number of the component they are trying to append is too large.

4. □*CREATE* now results in a shared rather than an exclusive tie. In addition, □*CREATE* now provides a default storage reservation in a new file of approximately 100,000 bytes.

5. □*ERASE* no longer requires a file to be exclusively tied.

6. Some changes in the formal rules for scanning access matrices.

The appendix to the SATN also provides an up-to-date list of the permission codes associated with each file function.

KEYWORDS:
□*RDAC*
□*STAC*
□*HOLD*
□*FHOLD*
□*APPEND*
□*APPENDR*
□*CREATE*
□*ERASE*
File functions
Permission codes
Access
Access matrices

1. □*RDAC* AND □*STAC* - ACCESS MATRIX CONTROL

The permission codes for □*RDAC* and □*STAC* have been augmented to permit the user to provide more detailed control over the access specifications of the two functions.

The □*RDAC* system function permits the user to read the access matrix of a file, and returns the value of the matrix as its result. The system function □*STAC* sets the access matrix for a file, and immediately imposes the new access restrictions upon all users who have the file tied at the time the □*STAC* is performed.

Previously, a permission code of 256 granted both □*RDAC* and □*STAC* access to a file. It was not possible to provide a user with □*RDAC* access without also enabling him to set the access matrix via □*STAC* .

A permission code of 256 will continue to provide □*RDAC* and □*STAC* access. In addition, permission code 4096 will allow □*RDAC* alone, and 8192 will allow □*STAC* . Permission of 4096+8192 (=12288) is thus equivalent to providing access of 256 . Providing permission of 256 and 4096 or 8192 is permitted, but is redundant and has no practical use.

Syntax

The syntax of □*RDAC* and □*STAC* remains the same as before. A complete description of the operations may be found in the publication **The** *SHARP APL* **File Subsystem Instruction Manual** (Copyright (C) 1977).

Access

The access restrictions for both □*RDAC* and □*STAC* are that the file must be tied; the passnumber must match the one in effect; and the user must have appropriate access to the file.

□*RDAC* requires a permission code of 256 or 4096 .

□*STAC* requires a permission code of 256 or 8192 .

Examples

In these and subsequent examples, origin-dependent expressions are assumed to be evaluated in origin-1.

Display current access matrix:

```
      ⎕RDAC 8
 1234567      256        0
```

Provide only ⎕*RDAC* permission with a passnumber:

```
      (1 3 ⍴⎕AI[1], 4096 1123) ⎕STAC 8
```

Display new access matrix:

```
      ⎕RDAC 8 1123
 1234567     4096     1123
```

Attempt to alter it:

```
      (0 3 ⍴0) ⎕STAC 8 1123
FILE ACCESS ERROR
      (0 3 ⍴0) ⎕STAC 8 1123
               ^
```

2. □*FHOLD* - FILE HOLD

A new file function, □*FHOLD* , has been added to allow synchronization of shared file operations with the optional use of a passnumber. Both □*FHOLD* and □*HOLD* set and release the same interlocks.

The monadic function □*FHOLD* sets an interlock on each of the files designated in its argument, after releasing all interlocks that were previously set. Interlocks will not be set while any other user has an interlock set on any of the designated files; □*FHOLD* will wait until all such other interlocks have been released before permitting execution to continue. A □*FHOLD* only delays execution of other users' □*FHOLD*'s (or □*HOLD*'s). Thus, while an interlock is set, other users are delayed in turn from completing their □*FHOLD*'s , but not from executing other file operations. The □*FHOLD* proceeds without delay if all of the files in the argument are available.

The interlocks set by □*FHOLD* are released when the user who set them executes another □*FHOLD* ; signs off (or is dropped); or enters immediate-execution mode (whether by an execution error, program termination, or by signalling *ATTN* one or more times to achieve a stop in execution or an interrupt). In particular, entering just a □*FHOLD* in immediate execution, rather than from within a program, has no effect since any interlocks that are set are immediately released before the next keyboard entry is accepted. Note that file holds remain active over □ and □ , and are not released when the keyboard unlocks for input in these cases.

□*FHOLD* '' has the effect of releasing all interlocks and continuing execution without delay. The interlock set on an individual file may be released without affecting any other active interlocks simply by untying the file.

Syntax

□*FHOLD* is a monadic system function and does not return an explicit result. Its syntax is:

```
□FHOLD A
```

where the argument *A* is an integer array of file tie numbers and passnumbers. *A* may be either a scalar, vector, or one-row matrix of file tie numbers; or a two-row matrix whose first row contains file tie numbers and whose second row contains the corresponding passnumbers. File tie numbers must be distinct, and must designate currently-tied files. If a passnumber is elided, it is assumed to be zero. Conversely, zero may be used to fill passnumber positions in a two-row matrix argument if some of the files for which interlocks are being set do not have passnumbers. An empty vector or a one- or two-row, zero-column matrix releases any interlocks and does not set any.

Access

The access restrictions for □*FHOLD* are that the designated files must be tied; the passnumbers must match the ones in effect; and the user must have □*FHOLD* access to each of the files.

The permission code for □*FHOLD* is 2048 .

Examples

Release any active interlocks:

```
⎕FHOLD ''
```

Set interlocks on files 2 and 3:

```
⎕FHOLD 2 3
```

File 3 is passnumbered; file 2 is not:

```
⎕FHOLD 2 2 ⍴ 2 3 0 ,PASSNO
```

Set interlocks on files 1 to 5 with the same passnumber:

```
⎕FHOLD (⍳5),[0.5] PASSNO
```

COMPARISON OF ⎕FHOLD WITH ⎕HOLD

The capabilities of ⎕*FHOLD* and ⎕*HOLD* are very similar. There are, however, some important differences:

1. ⎕*FHOLD* can take a matrix argument; the argument to ⎕*HOLD* must be a scalar or a vector.

2. ⎕*FHOLD* requires a passnumber match; ⎕*HOLD* does not use passnumbers.

3. ⎕*FHOLD* uses permission code `2048` ; ⎕*HOLD* uses permission code `64` .

Error messages

Incorrect use of ⎕*FHOLD* can result in the following error messages:

DOMAIN ERROR	The argument to ⎕*FHOLD* is not integer-valued; or contains a zero or negative tie number.
FILE ACCESS ERROR	The user is not permitted to ⎕*FHOLD* one or more of the designated files; or a passnumber is incorrect.
FILE SUBSYSTEM NOT AVAILABLE	The File Subsystem is not available for use. This is a temporary situation when the *SHARP APL* system is just starting up or shutting down. At other times, consult the APL Operator.
FILE TIE ERROR	Some file number appears more than once in the argument; or some element of the argument is not tied to a file.
LENGTH ERROR	The argument is a matrix with an incorrect number of rows.
RANK ERROR	The argument is not a scalar, vector, or matrix.

3. ⎕*APPENDR* - FILE APPEND WITH RESULT

A new file function, ⎕*APPENDR* , has been added to permit the user to append data to a file and secure the number of the component just appended in a single, race-free operation.

The dyadic function ⎕*APPENDR* appends a value to the end of a designated file. It creates a new component in the file with a value equal to the left argument of the function and a component number equal to one more than the largest previous component number. The new component number is returned as the explicit result.

Previously, to determine the component number created by a ⎕*APPEND* operation, it was necessary to additionally execute a ⎕*SIZE* on the file. In a shared-file environment in which more than one user may be accessing the file, these two operations create a race condition which further requires the use of ⎕*HOLD* to avoid having the ⎕*SIZE* return possibly out-of-date information. ⎕*APPENDR* alleviates these problems by automatically providing the component number to the user.

Both ⎕*APPENDR* and ⎕*APPEND* may now produce a *FILE INDEX ERROR* if the number of the component they are trying to append is not less than the highest internally representable integer (`¯1+2*31`). If this happens, the file should be broken up into smaller segments, or else copied over into a new file if the first component number in the file is also quite large.

Syntax

⎕*APPENDR* is an explicit dyadic system function. Its syntax is:

```
COMPNO←A ⎕APPENDR FILENO[,PASSNO]
```

where *A* is any APL object or expression; *FILENO* is a positive integer designating a file tie number; *PASSNO* is an optional integer passnumber; and *COMPNO* is an integer scalar representing the number of the component just appended. It is equal to:

```
(⎕SIZE FILENO,PASSNO)[2]
```

immediately before the append operation.

Access

The access restrictions for ⎕*APPENDR* are that the designated file must be tied; the passnumber must match the one in effect; and the user must have ⎕*APPENDR* access.

The permission code for ⎕*APPENDR* is `16384` .

Examples

Current end of file:

```
      (⎕SIZE 2)[2]
10
```

Append a package:

```
      (⎕PACK 'DIR INDEX') ⎕APPENDR 2
10
```

Append a vector:

```
      CN←(⍳3) ⎕APPENDR 2
       CN
11
```

COMPARISON OF ⎕*APPENDR* WITH ⎕*APPEND*

⎕*APPENDR* and ⎕*APPEND* provide similar capabilities. The differences are:

1. ⎕*APPENDR* returns a component number as its explicit result; ⎕*APPEND* does not return a result.

2. ⎕*APPENDR* uses permission code `16384` ; ⎕*APPEND* uses permission code `8` .

Error Messages

Incorrect use of ⎕*APPENDR* can result in the following error messages:

DOMAIN ERROR	See description under ⎕*FHOLD* .
FILE ACCESS ERROR	See description under ⎕*FHOLD* .
FILE FULL	The specified data could not be appended to the file because the space occupied by the file already exceeded its storage limit.
FILE INDEX ERROR	The number of the component to be appended was too large to be expressed as an integer value.
FILE SUBSYSTEM NO SPACE	The operation cannot be done with the available system storage. Use ⎕*ERASE* and ⎕*DROP* to eliminate any unnecessary storage, and retry the operation. If the problem persists, consult the APL Operator.
FILE SUBSYSTEM NOT AVAILABLE	See description under ⎕*FHOLD* .
FILE TIE ERROR	The file number is not tied to a file.
LENGTH ERROR	The right argument has an incorrect number of elements.
RANK ERROR	The right argument is not a scalar or a vector.
WS FULL	The explicit result requires more workspace storage than is available.

4. □*CREATE* - FILE CREATE

The dyadic system function □*CREATE* creates a new, empty file. It establishes a new name in the desired library and share-ties the file to the specified tie number.

Previously, □*CREATE* exclusively-tied the file after creating it. Even with a share-tie, other users cannot access the file until the owner has appropriately set the access matrix using □*STAC* . However, this change could be significant for an application which creates a file, performs a □*STAC* , and then proceeds (under the previously correct assumption) to work with the file as if other users could not access it. This assumption is no longer valid, and programs affected in this manner should be changed accordingly as soon as possible.

In addition, the default storage reservation for a □*CREATE* operation will be approximately 100,000 bytes, increased from 50,000 bytes. This does not increase or otherwise affect any file storage charges, as these are based on the storage actually consumed by a file, and not the space reserved for it.

Syntax

The syntax of □*CREATE* remains the same as before.

Access

The access restrictions for □*CREATE* remain the same as before.

5. □*ERASE* - FILE ERASE

The dyadic system function □*ERASE* erases an entire file. It drops all components from the file, unties it, and deletes the file name from the appropriate library. The space occupied by the file components is made available for use by other files.

Previously, □*ERASE* required that the file be exclusively-tied. An exclusive-tie is no longer necessary; however, if the file is share-tied, then the user executing the □*ERASE* must be the only user tied to the file at the time.

Syntax

The syntax of □*ERASE* remains the same as before.

Access

The access restrictions for □*ERASE* are that the user must have the file exclusively-tied, or must be the only user share-tied to the file at the time; the passnumber must match the one in effect; and the user must have □*ERASE* access.

Error Messages

Incorrect use of □*ERASE* can result in certain error messages. The following list amends the complete description provided in **The** *SHARP APL* **File Subsystem Instruction Manual**:

FILE ACCESS ERROR	The user is not permitted to erase the file; or the passnumber is incorrect.
FILE TIED	The file is currently share-tied by some other user.

6. ACCESS MATRIX SCANNING RULES

Some changes have been made to the rules governing the manner in which file access matrices are scanned by the system.

When a $\square TIE$ or $\square STIE$ operation is performed, the rows of the access matrix of the appropriate file are scanned in an attempt to select a row matching both the user number and the passnumber. To do this, each access matrix may be considered to have on the bottom of it an implicit row giving the owner of the file complete permission without a passnumber. The owner is guaranteed this level of permission unless it is specifically removed by another row in the matrix. The search of an access matrix then proceeds according to the following rules:

1. Any row with a user number other than zero or the user attempting to do the tie is not considered.

2. Any row with a user number of zero is taken to include the user attempting to do the tie only if the passnumber matches the one given.

3. A search of the access matrix terminates with the first row having an exact match on both the user number and the passnumber. If none is found, then the search will traverse the entire access matrix (including the implicit owner row).

4. As the search proceeds, the first instance of a row having a user number of zero and a matching passnumber is noted. This row is used only if the entire access matrix is scanned without finding an exact user number and passnumber match (including the implicit owner row).

These rules may be summarized by the program *GETPERM* shown below:

```
        ∇ PERMNOW←P GETPERM M;OWNER;PASSNO
[1]     OWNER←1↑P ◇ PASSNO←1↑1↓P
[2]     M←M,[1] OWNER, ¯1 0 ◇ M←((M[;1]∈2↑⌶29)∧M[;3]=PASSNO)⌿M
[3]     PERMNOW← 1 3 ↑ M[⍒M[;1];]
        ∇
```

The left argument P is a one- or two-element vector containing the user number of the owner of the file being tied, and the passnumber being used. The passnumber is taken to be zero if none is provided. The right argument M is the three-column access matrix of the file. The program returns as its explicit result a single-row matrix containing the row of the right argument that best matches the information in the left argument and the user number of the person doing the tie, according to the above rules. If no such row could be found, then zeroes are returned.

Effects

The new way in which access matrices are scanned may cause the system to select a different row from a matrix than it would have previously. In particular, access matrices with at least one non-zero passnumber **and** at least one row specifying user number zero or the owner may be affected; the changes are transparent to any other access matrix. Any access matrix which now behaves differently does so **only** for the owner of the file; the access restrictions for any other user are not altered.

Below is a summary of the types of access matrices which are directly affected by the new scanning rules. In each case, the owner's ability to access his file is increased by the fact that a scan may now select a row of the matrix that was previously rejected for him. In the description, *OWNER* is the user number of the owner of the file. *PERM1* and *PERM2* represent non-zero permission levels for a row, and may or may not be equal. *PASSNO1* and *PASSNO2* represent non-zero file passnumbers, and likewise may or may not be the same.

1. An access matrix containing a row of the form *OWNER*,*PERM1*,*PASSNO1* and no row of the form *OWNER*,*PERM2*,0 is affected. The owner may now tie his file with *PASSNO1* or without a passnumber, whereas previously he could tie it only with *PASSNO1* .

2. An access matrix containing a row of the form 0,*PERM1*,*PASSNO1* and no row of the form *OWNER*,*PERM2*,*PASSNO1* is affected. The owner may now tie his file with *PASSNO1* , whereas previously he could not.

3. An access matrix containing a row of the form 0,*PERM1*,*PASSNO1* and of the form *OWNER*,*PERM2*,*PASSNO2* in that order is affected. The owner may now tie his file with *PASSNO1* or *PASSNO2* , whereas previously he could tie it only with *PASSNO2* .

In addition, the ordering of the rows in an access matrix is no longer important, unless two rows contain the same user number and passnumber but specify different levels of permission. In this case, the first matching row found will be selected, and the other similar rows ignored.

The changes described affect a limited number of access matrices. If, in a particular situation, the extra permission granted to the owner of a file is undesirable, it can easily be removed simply by adding an overriding row to the matrix giving the owner no (zero) permission with the appropriate passnumber. In most cases, however, the effects are positive and can even lead to the removal of now-redundant rows in access matrices.

APPENDIX

The access matrix for a file, set by the ⎕*STAC* function and returned by the ⎕*RDAC* function, is a three-column integer-valued matrix. The first element in a row represents the account number for which that row applies, or 0 if it applies to all users. The last element in a row is the passnumber that must be used. The second element in a row of the access matrix is the sum of the power-of-two values corresponding to the particular operations that are to be allowed. These coded values are as follows:

File Operation (s)	Permission Code
⎕*READ*, ⎕*SIZE*	1
⎕*TIE*	2
⎕*ERASE*	4
⎕*APPEND*	8
⎕*REPLACE*	16
⎕*DROP*	32
⎕*HOLD*	64
⎕*RENAME*	128
⎕*RDAC*, ⎕*STAC*	256
⎕*RDCI*	512
⎕*RESIZE*	1024
⎕*FHOLD*	2048
⎕*RDAC*	4096
⎕*STAC*	8192
⎕*APPENDR*	16384

Thus a value of `2+16 (=18)` indicates authorization to execute only ⎕*TIE* and ⎕*REPLACE* operations on the file. A permission code of `¯1` indicates authorization for all operations. Any non-zero value permits the use of ⎕*STIE* and ⎕*UNTIE* .

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

SATN 28
11 JUL 77

SHARP APL TECHNICAL NOTES

TITLE: TERMINAL CONTROL

ABSTRACT: Facilities are described which extend the range of terminals and terminal features available to users of *SHARP APL*.

KEYWORDS: *)TERM*
Translate Set
⎕ARBIN
⎕ARBOUT

1.

)*TERM* Command

This system command is used to alter the parameters used by the system in formatting and interpreting terminal input and output. Parameters that vary include idle calculations, type of idle, response to line turnaround (most ASCII terminals are sent at least a linefeed), and allowable control characters (not all terminals can tab, or backspace, for example).

When a user initially signs on, he is assigned a terminal type of *IBM2741*, *TY33*, or *ASCII*.

A user may list valid arguments to)*TERM* by loading workspace `5 TERM`, and typing *TERMINALS*.

To determine what characteristics a given terminal is assumed to have, load the workspace `5 TERM` and use the function *TERM*. For example, *TERM* `TEK4013` will describe that device as the system sees it.

2.

Translate Set

A user may now determine which translate set has been assigned to him by the system. `(2 ⎕WS 3)[⎕IO+11]` returns values with the following interpretation:

0	*NTASK* or *BTASK* (no Translate Set)
1	Correspondence 2741 (Typeball 1167987)
2	BCD 2741 :Typeball 1167988)
3	TY33 (64 character ASCII terminal)
4	Typewriter Pairing ASCII terminal
5	Bit Pairing ASCII terminal

The character variables *ARBBIT* and *ARBTYPE* in workspace `1 ARBOUT` can be used in conjunction with `(2 ⎕WS 3)[⎕IO+11]` to translate between internal and external representations of (non-overstruck) *APL* characters for most ASCII terminals. For example, if `⎕IO←→0`, and `(2 ⎕WS 3)[11]←→4`, then

```
T←ARBTYPE[1↑⎕ARBIN ARBTYPE⍳ 'ENTER ONE CHARACTER:']
```

will prompt for, accept, and translate to internal form, a single non-overstruck *APL* graphic.

3.

□*ARBOUT*

□*ARBOUT* accepts a right hand argument of numbers or characters, and transmits it to the attached terminal subject to the following transformations:

3.1 Numeric Arguments.

3.1.1 ASCII Terminals

Range	Processing
0-255	EVEN Parity (bit 8) forced.
256-383	Parity bit set to 0 (space)
384-511	Parity bit set to 1 (mark)
512+	*DOMAIN ERROR*

Thus, 65 and 193 are both transmitted as *X* '41' (α or ASCII uppercase 'A')

3.1.2 BCD Terminals

Range	Processing
0- 63	Lowercase characters (BA8421)
64-127	Uppercase; (BA8421) ↔ 128 | argument
128	One character time of delay; a BCD idle may or may not be transmitted.
128+	*DOMAIN ERROR.*

Thus 49 is correspondence *G* or BCD *A*. Odd parity is forced.

3.2 Character arguments.

Character arguments are converted to numeric arguments by the expression (-□*IO*)+□*AV*⍳argument. If the numbers so generated exceed 127, a *DOMAIN ERROR* is reported, otherwise they are treated as in 3.1 above.

□*OUT* considerations

If a □*OUT* file is in effect, □*ARBOUT* generates the *CONTROL MESSAGE*:
□0014 0001 *ARBOUT*
followed by a component containing the argument.

⍞ considerations

If there is any data 'resting' in the buffer used for ⍞ input/output, it is forced out prior to beginning the □*ARBOUT*.

4.

□*ARBIN*

□*ARBIN* takes a righthand argument suitable for □*ARBOUT*, and issues it as a prompt. The system is then immediately conditioned to accept input, and the input is returned as a numeric vector whose elements are in the range 0-127. The result will be the codes sent by the user, with no translation and no editing. Currently all characters are accepted up to the first CR (ASCII) or EOT (BCD), or a BREAK or a transmission error occurs.

NOTES:
1) A transmission of more than about five hundred characters is considered a transmission error.
2) No echo (turnaround) characters are sent after receipt of the CR. For ASCII terminals, this means that the LF and idles usually sent automatically must be explicitly sent (via □*ARBOUT*) if desired. For BCD devices, this places the responsibility of providing correct terminal protocol with the user.

BREAK is currently handled by restarting the operation (including reissuing the prompt).

Transmission errors cause the message *RESEND* to be typed, followed by the message *INTERRUPT*, and execution is suspended.

A user may signal an interrupt by responding *O*, backspace, *U*, backspace, *T* to the □*ARBIN* prompt. This will cause the message *INTERRUPT* to print, and execution will be suspended.

□*OUT* Considerations

If a □*OUT* file is in effect, □*ARBIN* generates the *CONTROL MESSAGE*:
□0015 0002 *ARBIN*
followed by a component containing the prompt, followed by a component containing the result.

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

SATN 29
15 JUNE 78

SHARP APL TECHNICAL NOTES

TITLE: System Time and Timestamps

EFFECTIVE: 7 OCT 1978

ABSTRACT: This SATN describes the following modifications in the treatment of time and timestamps in SHARP APL.

1) □*TS* will indicate century.

2) System timestamps will be reported in UTC.

3) Functions will be provided to aid SHARP APL users in converting UTC timestamps to local time.

In addition, an interim system command will make a subset of these modifications available immediately.

KEYWORDS: □*TS*
Time
Node
Workspace 1 *TS*
UTC
)UTC
Workspace 1 *NEWS*

U.S. Sharp Corporation
Sharp Plaza
Mahwah, NJ 07430-2135
(201) 529-8200

SHARP APL TECHNICAL NOTES

1) ⎕TS - Current Time Stamp

The definition of ⎕TS will be expanded to provide the complete Gregorian year, including the current century.

⎕TS is a system function with the syntax:

```
R←⎕TS
```

where the result R is a seven-element numeric vector containing the current year, month, day, hour, minute, second, and millisecond. Previously, the year reported included only two digits of information. The following examples illustrate the old and new definitions:

```
    ⎕TS      ⍝ old definition
78 7 15 20 49 11 271
    ⎕TS      ⍝ new definition
1978 7 15 20 49 12 94
```

It will be necessary to modify current applications to reflect this change. The expressions

```
 100|⎕TS[⎕IO]   ⍝ when selecting the year, or
1900|⎕TS        ⍝ when using the entire timestamp
```

will return results using either definition consistent with the old definition.

2) System Timestamps

In order to provide SHARP APL users with an invariant time standard, all system timestamps will be based on Coordinated Universal Time (UTC). This change is being made to free users from the idiosyncrasies of local time adjustments. UTC is the international time standard disseminated by national time services, and is maintained by the interval of the atomic second. It is similar to the more familiar standard of Greenwich Mean Time (GMT), the mean solar time for the meridian at Greenwich, England.

This change will affect the following timestamps:

- ⎕TS - Current time and date
- ⎕RDCI - File component information
- ⌶20 - Time of day relative to date of signon
- ⌶24 - Signon time
- ⌶25 - Date of signon
- Workspace timestamps (including 2 ⎕WS 4)
- Signon/signoff timestamps

Additionally, the Computer Center clock and the operating schedule (see *SCHEDULE* in workspace 1 *NEWS*) will be maintained in UTC.

3) Local Timestamps

Functions will be provided in workspace 1 *TS* which will aid users in calculating their local time. To facilitate the conversion to local time, a data base will be maintained containing local offsets from UTC for the majority of locations served by SHARP APL.

Currently, each user signed onto SHARP APL is assigned a port number and a unique task-ID (see SATN 1). A port number has the format NNNLLL, where NNN indicates node number, and LLL specifies line number within that node. Node numbers are assigned to each intelligent communications controller within the Sharp Communications Network. The node number assigned to a task indicates the most local controller to the user. N-tasks and B-tasks are assigned node numbers equal to the task which originated them.

The overwhelming majority of SHARP APL users access a node locally and directly. For these users, local time may be calculated automatically, using the node number as an implicit argument to the function *LTS* (see following description). For users accessing SHARP APL through a node not in their time zone, facilities are provided to specify the local offset from UTC explicitly.

Following is a description of the major functions in workspace 1 *TS*. This documentation also is available in the variable *DESCRIBE*, in that workspace.

a) **result**← *LTS* **time**
result← **offset** *LTS* **time**

Returns as its explicit result the local timestamp from an argument in UTC. **Time** may be either a seven-element numeric vector in ⎕*TS* format, or a single number in ⎕*RDCI* format (see "The SHARP APL File System Manual" for details on the format of ⎕*RDCI*). The local offset from UTC is either calculated based on the node to which the user is connected, or may be specified explicitly in the optional left argument.

Offset if provided, must be a single integer specifying offset from UTC in seconds. Permissible values are in the range ¯43200 ≤ **offset** ≤ 43200 (43200 seconds = 12 hours).

The result is either an adjusted timestamp in the same format as **time**, or a character vector containing an error report. Possible error reports are:

```
TIMEFRAME NOT AVAILABLE
NODE NOT AVAILABLE
```

Note that, when used monadically, *LTS* calculates the offset on the basis of the timestamp provided; if the user's node is located in a time zone in which time is seasonally adjusted, the resulting offset will reflect the adjustment in effect at the specified time. A value for **time** not in the data base will result in a *TIMEFRAME NOT AVAILABLE* error report.

b) **result**← *UTC* **time**
result← **offset** *UTC* **time**

Returns a UTC timestamp from an argument in local time. Argument and result format are identical to *LTS*.

c) **offset←***GETOFFSET* **node**
offset←time *GETOFFSET* **node**

Returns a local offset from UTC in seconds for the node number specified by the argument. **Node** must be a single integer specifying a node number in the Sharp Communications Network. The offset is calculated based on either the current date and time, or the timestamp specified explicitly in the optional left argument. This may be in □*TS* or □*RDCI* format.

The result is a numeric scalar offset from UTC or a character error report. Possible errors include:

TIMEFRAME NOT AVAILABLE
NODE NOT AVAILABLE

d) **node←***CURNODE*

Returns as a scalar integer the node number to which the user is connected.

e) **result←***NODE* **argument**

Returns node numbers and locations from an argument in either format described below. The result is a character matrix of 40 columns, the first four columns being node numbers right-justified, the fifth a space, and the remaining columns containing location left-justified within the field. Node location will be specified by city, an optional state or province, and a country name, delimited by commas as necessary.

The permissible formats the argument may take are:

1. A numeric array of node numbers, or
2. A character array of node locations.

Any other format will result in *SPECIFICATION ERROR*.

If a character argument is given, it may be either a scalar, a vector, or a one-row matrix containing a single node location; or a matrix containing one location per row. The result will contain one row for each location found in the data base. Matches will be performed on as many characters as are provided in the argument. An empty character vector argument will return the entire table.

If a numeric argument is supplied, the *NODE* function will return one row per node number specified, unless any of the given nodes do not exist, in which case *SPECIFICATION ERROR* will result.

Note that, given a character argument, the number of rows in the result may not necessarily be identical to the number of locations provided, since a location may be supported by more than one node. Conversely, if no match for a given location is found, there is no resulting row generated; an argument for which no matches are found will result in an empty array.

f) **offset←*OFFSET* time**

Returns an offset from UTC in seconds from an argument in local time. This function is designed to aid those users not locally accessing a system node. The argument must be an integer vector in □*TS* format; the user may, however, omit any elements after the hour specification, in which case the function will substitute current values for subsequent fields in its calculations. Any other input format will result in

IMPROPERLY FORMED TIMESTAMP.

g) **result← *TOUTC* time**

Converts a pre-UTC (Toronto local time) EST/EDT system timestamp to UTC. This function is designed to aid users in maintaining continuity of time relationships within their data bases upon the change to UTC. The argument must be a single timestamp in □*TS* or □*RDCI* format.

The resulting timestamp will be converted to UTC if the specified time and date are prior to the system UTC conversion; if the timestamp provided is prior to 1 January 1974, then a scalar ¯1 will be returned; otherwise the argument will be returned directly. *TOUTC* therefore will allow an argument in either time standard maintained by SHARP APL since 1 January 1974, and will return the correct equivalent UTC timestamp.

4))*UTC*

Because the nature of the changes described in this SATN will likely necessitate modification of user programs, a temporary system command,)*UTC*, is being introduced for the interim between 1 May 1978 and the effective date of this SATN. In addition to allowing modification of current programs, this system command will make it possible to develop new systems which will be unaffected by the forthcoming changes.

)*UTC ON* will cause adjustment of the results returned by the system functions □*TS*, □*RDCI*, and □*WS*; and the ⌶ beams ⌶20, *I24* and ⌶25. The resulting information will be UTC-adjusted, and □*TS* will return the current century as previously documented. Timestamps printed by system commands ()*LOAD*,)*COPY*,)*SAVE*, and)*DROP*) will continue to report EDT.

)*UTC OFF* will cause the above functions to return EDT as before. This is the default setting in a workspace.

)*UTC ON* and)*UTC OFF* will print the previous setting (i.e. *ON* or *OFF*), while)*UTC* (with no argument) will print the current setting. Any argument other than *ON* or *OFF* will result in an *INCORRECT COMMAND* error message. In addition, any use of)*UTC* within a)*SEAL*'d workspace will result in *INCORRECT COMMAND*.

The setting for)*UTC* is only applicable within the workspace in which it was specified; that is, its effects are not propagated over)*LOAD*'s. The current setting will, however, be maintained when the workspace is)*SAVE*'d and)*LOAD*'d.

Although)*UTC* affects the results returned by □*RDCI* and 2 □*WS* 4, it is very important to note that this does not alter the actual file component or workspace save times, which will continue to be stored in EDT.

This system command will be discontinued on or after 7 October 1978.

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

SATN-30
1 JAN 79

SHARP APL TECHNICAL NOTES

TITLE: Numeric Display

ABSTRACT: The precision of numeric display has been extended to 18 digits. ⎕*FMT* and ⍕ have received several enhancements.

EFFECTIVE: 3 February 1979

KEYWORDS: ⎕*PP*
⎕*FMT*
⍕

Sharp Associates
145 King Street West
Toronto, Ontario M5H 1J8
(416) 364-5551

SAT[illegible]
[illegible]JAN 79

SHARP APL TECHNICAL NOTES

TITLE: Numeric Display

ABSTRACT: The precision of numeric display has been extended to 18 digits. [illegible] and 3 have received several enhancements.

KEYWORDS: [illegible]
[illegible]

The Problem

Large numbers, and numbers with a fractional part, are not stored internally by SHARP APL in the decimal format you might expect. They are stored in a format known as hexadecimal floating point. This is done for a simple reason: the types of computers that support SHARP APL have special instructions to manipulate numbers in this format, hundreds of times faster than the same numbers could be manipulated in decimal. The hexadecimal format does have a disadvantage; it does not always represent the original decimal value exactly.

For example, `2.3` is represented in hexadecimal format as 4124CCCCCCCCCCCD which, in turn, has an exact decimal value of

```
2.3000000000000004440892098500626161694526672363281 25
```

In fact, all numbers between `2.29999999999999934` and `2.30000000000000155` will be represented as 4124CCCCCCCCCCCD.

The problem of numeric display, then, is: "Given a number in hexadecimal format (like 4124CCCCCCCCCCCD), which value from the range of possible decimal values should we display?"

The solution in the past has been to display the exact value, rounded to ⎕*PP* digits. In our example of `2.3` the exact value is

```
2.3000000000000004440 89.....
```

Rounded to 16 digits (the maximum value previously permitted for ⎕*PP*), that is

```
2.300000000000000
```

Suppressing the trailing zeroes, we get

```
2.3
```

So far, so good. However, this approach has two shortcomings:

(a) Two different 16-digit decimal numbers may be converted to the same internal value.
(b) Two different internal values may display the same 16-digit decimal number.

These two anomalies are demonstrated in the following:

```
      ⎕PP←16      ⍝ FORMER MAXIMUM

      .07         ⍝ ILLUSTRATE PROBLEM A
0.07000000000000001

      ⌊/⍳0
7.237005577332262E75

      7.237005577332262E75-⌊/⍳0    ⍝ ILLUSTRATE PROBLEM B
¯1.004336277661869E59
```

The Solution

Problem (b) may be solved easily by increasing the upper limit on ⎕*PP* until the problem disappears. This occurs when ⎕*PP* is permitted to equal 18. No two internal values have the same 18-digit decimal display.

This change has been made to SHARP APL. Numbers that actually require 18 digits are rather uncommon (an example is `0.000000056000000000000088`). Seventeen-digit examples are very common, for example `⌊/⍳0` from an earlier example.

In solving problem (b) however, we have actually aggravated problem (a). If two different 16-digit numbers convert to the same internal value, then hundreds of 18-digit numbers will do so. This problem cannot really be solved without retracting our solution to problem (b), but it can be disguised. Rather than choosing as our 18-digit display the 18-digit decimal number closest to the internal value, we choose an 18-digit decimal number which

a) lies between the upper and lower bounds of values which convert to that internal value, and
b) has the least number of significant digits.

To illustrate with an example:

Before the change

```
      ⎕PP←16
      .07   .07000000000000001
0.07000000000000001 0.07000000000000001
```

After the change

```
      ⎕PP←16
      .07   .07000000000000001
0.07 0.07
```

In summary, then

a) When we display a number we use the smallest possible number of digits.
b) Up to 18 digits can be displayed. ⎕*PP* may take values from 1 to 18 inclusive.

NOTE: We are indebted to Larry M. Breed of IBM for his analysis of the problem and for proposing the above solution.

Additional Enhancements

A) ⎕*FMT*

1) *E* format phrases may use all qualifiers and decorators.

2) *G* format may use *B* qualifier.

3) There is no interaction between ⎕*FMT* and ⍞ input/output if the result of ⎕*FMT* is not displayed.

4) The "underscore" feature is no longer used. Trailing zeroes are always displayed as zero.

5) The *S* qualifier now permits substitution for the *E* and ¯ associated with the exponent of *E* format numbers.
e.g.

```
      'M<->S<ED¯->BE30.20,BG<Z999>' ⎕FMT ((⌈/⍳10),0,÷⌈/⍳10;5 0 6)
-7.2370055773322621000D75  005

-1.381786968815111000D-76 006
```

B) ⍕ THORN

Dyadic ⍕ has been re-written. The majority of changes should be transparent.

1) *E* format numbers will always display the specified number of digits, independent of the value of the exponent.

2) *F* format numbers will always have at least one digit (zero if necessary) to the left of the decimal point.

3) The "underscore" feature is no longer used. Trailing zeroes are always displayed as zero.

Before:

```
     12 ¯5 10 5 30 ¯20 ⍕⍉3 2⍴÷3 0.3
3.33333E¯1     .33333      3.3333333333333333____E¯1
3.333333E0    3.33333      3.3333333333333333_____E0
```

After:

```
     12 ¯5 10 5 30 ¯20 ⍕⍉3 2⍴÷3 0.3
3.3333E¯1     0.33333      3.33333333333333333000E¯1
3.3333E0      3.33333      3.33333333333333333000E0
```

Monadic thorn exactly reflects the changes in numeric display discussed in C.

C) NUMERIC DISPLAY

1) ⎕*PP* has been extended to 18 digits.

2) When displaying a vector *E* format will be used when
   a) *E* format is shorter, and
   b) *F* format would require more than 10 digits.

3) When displaying a matrix, *F* format will be used when the result can fit in a column width of ⎕*PP*+6 or less. Otherwise *E* format will be used. In either case the actual column width will be constant for the entire matrix and the minimum value possible to permit the decimal point to occupy the same relative location in each column.

Before:

```
      ⎕PP←10
      2 2 ⍴ .5 .33 .123
  0.5          0.33
0.123          0.5
```

After:

```
      ⎕PP←10
      2 2 ⍴ .5 .33 .123
0.5   0.33
0.123 0.5
```

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

SATN-31
1 FEB 79

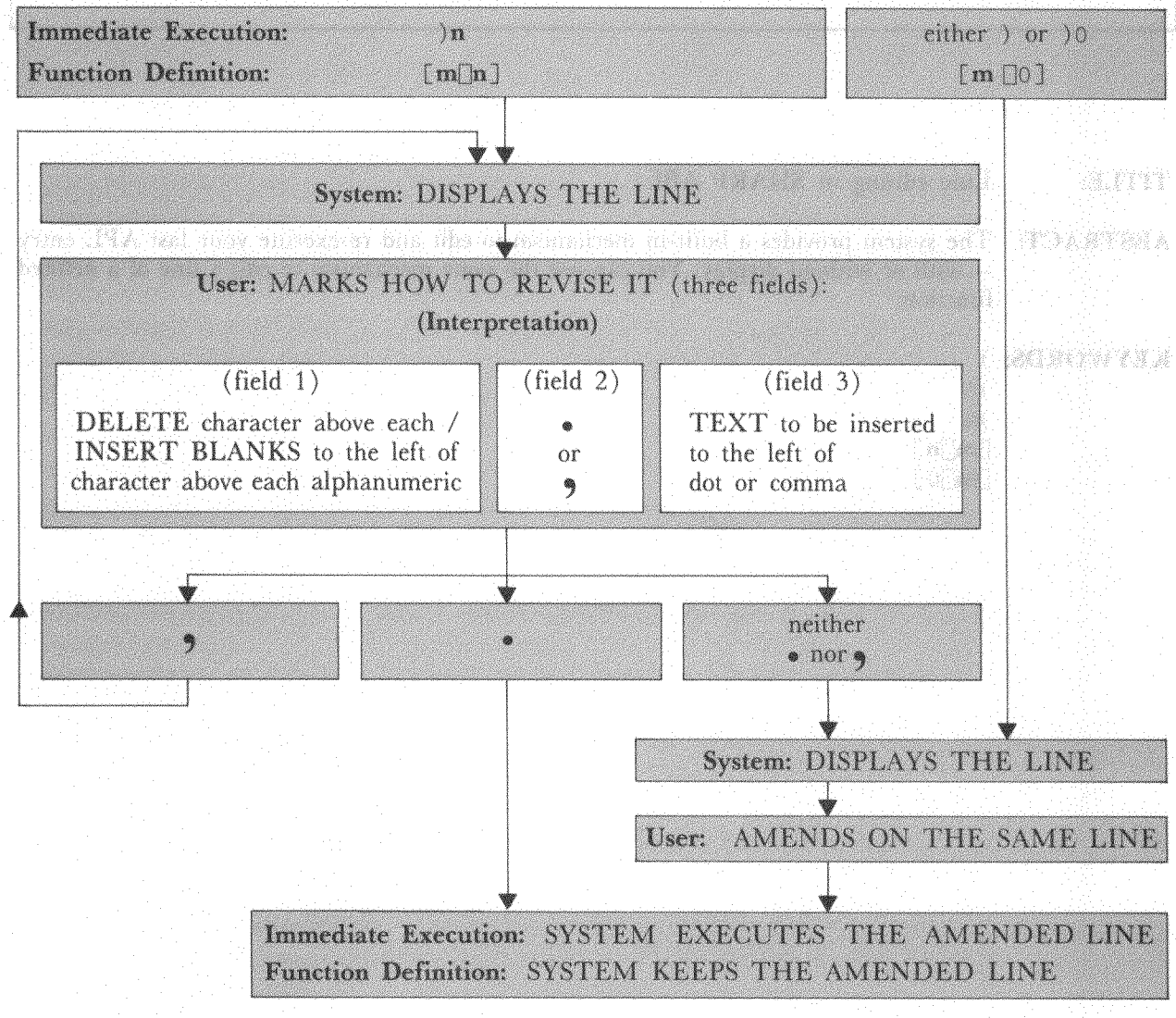
SHARP APL TECHNICAL NOTES

TITLE: Line editing in SHARP APL

ABSTRACT: The system provides a built-in mechanism to edit and re-execute your last APL entry -- with or without display. You can use the same mechanism to edit a line of a defined function.

KEYWORDS:)
)0
)**n**
[**m**□**n**]
[**m**□0]

LINE EDITING IN SHARP APL



INTRODUCTION

SHARP APL provides a primitive editing facility. You can use it to edit the line of APL you've just entered for execution, or while you're building or modifying the definition of a function. Because it's a built-in facility, it's always available, and doesn't require a particular program or workspace.

Correcting a line before you've pressed RETURN

Before you've pressed RETURN, the LINEFEED key deletes the character at the present cursor position and everything to the right of it. (Position the cursor with SPACE or BACK-SPACE.)

While you're entering a line of APL from the keyboard, and before you've pressed the RETURN key to request the system to act on it, you may notice you've made a typographical error. You can correct that mistake by backspacing until the cursor is at the error. Pressing the LINEFEED key (ATTENTION on IBM terminals) deletes the unwanted character **as well as everything typed to the right of it.**

The cursor moves down one line so that it is directly below the point from which you erased. Then you can retype the balance of the line. Use LINEFEED (or ATTN) again, as often as you need, to make further changes. The system doesn't attempt to evaluate your line until you press RETURN.

```
      Z←'THE TOTALIS: 'BS LF
                @IS: 'RETURN
      @
```

The example above illustrates what you see at the terminal when you use the BACKSPACE LINEFEED (BS LF) combination. The @ character indicates the position of the cursor.

Editing after you've pressed RETURN

You can bring back a line that's already entered and edit it. There are two situations in which you can do that:

1. While you're in **immediate execution mode**, recall a line by entering) either alone or followed by a number. That lets you edit the **one** preceding line of APL you entered from the keyboard.

2. While you're in **function definition mode**, recall a line of the function's definition by entering [**m**□**n**], where **m** is the number of the line. The number that follows the □ serves the same purpose as the number that follows the) while you're in execution mode.

When you've finished editing the line, the system executes it (if you're in execution mode) or makes it part of the function's definition (if you're in definition mode).

Note that you cannot bring back a line of entry which started with a) or a ∇. This means that system commands and entries requesting function definition cannot be recalled.

ONE-PASS EDITING

) or)0 requests one-pass editing of the previous APL line. [m⎕0] requests one-pass editing of line m. The system displays the line and leaves the cursor at the end of it.

After the system displays the line, it turns the keyboard over to you and waits for you to modify the line. When you've entered your changes and again press RETURN, the system acts on the line, including your alterations.

The request [m⎕0] or)0 leaves the cursor waiting on the **same** line at the next available position to the right. A right parenthesis standing alone has the same effect as)0.

Writing some more on the same line

Suppose the change that you want to make is an addition to the end of the line. The quickest way to do this is to request editing with)0 or [m⎕0]. The system displays the line and leaves the cursor on the **same** line, in the next available position to the right.

```
      )0
Z←'THE TOTAL IS: '@
```

The system now awaits additional characters on the same line. When you've typed the additions, pressing RETURN causes the system to act upon your revised line without further display.

```
Z←'THE TOTAL IS: ''ZIP'''RETURN
        @
```

TWO-PASS EDITING

Positioning the cursor

)n requests two-pass editing of the last APL line. [m⎕n] requests two-pass editing of line m. The system displays your line and puts the cursor on the following line at the position designated by n.

After **[m⎕n]** or **)n**
causes display of the line you've requested, the system waits with the cursor on the **following** line at the position you indicated. You can use any positive integer as long as it is not greater than the value of the system variable ⎕*PW* (Print Width).

You don't have to calculate **n** exactly; you can always use SPACE, BACKSPACE, or TAB to adjust the position of the cursor.

Deletions

During two-pass editing, each character marked with a slash is deleted. (However, all characters are treated as text when they are to the right of a dot or a comma.)

After you request two-pass editing with **[m⎕n]** or **)n**, enter a / under each character to be deleted.

```
      )17
Z←'THE TOTAL IS: ''ZIP'''
                /RETURN
```

After you press RETURN, the system displays the line with the deletions. The cursor waits on the same line at the next position to the right.

```
Z←'THE TOTAL IS:''ZIP'''@
```

If after you've made deletions, you wish to add additional characters to the right of the line, you can enter them directly. Pressing the RETURN key tells the system to take the line, as it appears at the terminal -- that is, after your deletions and additions.

Notice that you can delete any number of characters. The deleted characters don't have to be consecutive. However, the slash marks that indicate deletion **must** be to the left of a dot or comma (if any).

```
      )21
Z←'THE TOTAL IS: ''ZIP'''
                   /////RETURN
Z←'THE TOTAL IS: ''Z@
```

You can append one or more characters to the right of the line.

```
Z←'THE TOTAL IS: ''ZERO'''RETURN
      @
      Z
THE TOTAL IS: 'ZERO'
      @
```

Insertions

There are two ways to do insertions:

1. First insert enough blanks to accommodate the insertions. The blanks don't have to be consecutive. In another pass, overstrike the inserted blanks with the characters you want.

2. Insert a single block of consecutive text.

Watch out: Don't insert so many blanks that the line would be more than □*PW* positions long. See "Common errors" for details.

Scattered insertions

To the left of each position that you mark by an alphanumeric character (0-9 or A-Z) the system inserts blanks. (However, all characters are treated as text when they are to the right of a dot or a comma.) The digits one to nine insert the corresponding number of blanks. The letters A to Z insert blanks according to the code: A equals five, B equals ten, C equals fifteen, etc.

```
      )17
Z←'THE TOTAL IS: ''ZERO'''
                @
                2RETURN
Z←'THE TOTAL IS:@  ''ZERO'''
```

The system displays the line with two blanks inserted. The cursor is on the same line waiting at the **leftmost** of the blanks you inserted. The keyboard is turned over to you. You may then overstrike the blanks with the characters you want to insert. (In the example above, the 2 is on the following line for illustrative purposes only.) When you press RETURN, the system acts on the line as modified.

```
Z←'THE TOTAL IS:***''ZERO'''RETURN
      @
```

Block insertions

If the line you enter contains a dot or a comma, any characters to the right of the first dot or comma are a block of text to be inserted. There are two ways to do this:

Terse editing (marked by .)

Serial editing (marked by ,)

The presence of a dot or a comma divides your editing entry into three fields:

1. To the left of the first dot or comma (or the whole line, if there isn't any dot or comma) scattered deletions and insertion of blanks.

2. The dot or comma indicates where a block insertion is to go, and where you want the cursor to be afterwards.

3. To the right of the first dot or comma, all characters are text to be inserted as a consecutive block.

TERSE EDITING

If you want to edit in one pass and without display, use the dot form of editing. The system acts on the revised statement immediately, as soon as you press RETURN. Entering a dot alone after you've requested line editing causes the system to act on the line immediately.

Example of insertion using .

```
      )26
Z←'THE TOTAL IS:***''ZERO'''
                        .!RETURN
```

The system makes the insertion you requested and acts on the revised statement **without displaying it**. (In this example, you could verify that your insertion was accepted by asking the system to display the altered value of the variable Z.)

```
      Z
THE TOTAL IS:***'ZERO!'
```

You may insert any characters (including blanks), punctuation, and APL symbols (including those formed by overstriking one character with another). Note that BLANK is a character. (You can't see it but it is.) BACKSPACE is **not** a character. It serves **only** to position the cursor. To insert a dot with the dot form of editing, you will need two dots. The first dot locates the character which you want to appear immediately to the **right** of the dot you insert.

Example of simultaneous deletions and insertion

With the dot form of editing, a combination of slashes, alphanumerics, and a block of text to the right of the dot causes the simultaneous deletion and insertion of characters. Pressing RETURN tells the system to act on the revised line without display.

Using / and . together, you can combine an insertion and several deletions. (However, the deletions have to be to the **left** of the insertion.) Each slash you enter to the **left** of the first dot indicates a character to be deleted. All characters which you enter to the **right** of the first dot are to be inserted. Put the dot where you want the inserted characters to go. The character above the dot will follow the insertion.

```
      )4
Z←'THE TOTAL IS:***''ZERO!'''
   ////          //     //....RETURN
       @
```

The system accepts the simultaneous deletions and insertion, and immediately acts on the modified line **without displaying it**. Note that you **can** insert dots in your line. **Anything** after the first dot or comma is inserted.

```
      Z
TOTAL IS:***ZERO!...
```

SERIAL EDITING

If you want to edit in successive passes with display, use the comma form of editing. The system displays your line -- with current revisions -- and waits for you to make further alterations. When you've finished making changes, entering a dot and pressing RETURN tells the system to act upon the line without further display.

Successive alterations

Comma editing follows the same convention as dot editing. That is, to the **left** of the first comma, each slash or alphanumeric character deletes or inserts blanks. The block of characters to the **right** of the first comma is text to be inserted. The comma locates the character which you want to appear immediately to the **right** of the added characters. (That is, you want your insertion to appear to the **left** of the character under which you type the comma.)

When the system has made the deletions and the insertion, it displays the line as it now appears, and awaits further instructions. The new instruction may contain a dot, a comma, or neither.

```
      )12
Z←'TOTAL IS:***ZERO!...'
          /,
Z←'TOTAL IS***ZERO!...'
            @
```

Notice that the cursor appears on the following line under the character where you initially entered the comma. The keyboard is turned over to you and you can now make further changes according to any of the conventions allowed during two-pass editing.

```
Z←'TOTAL IS***ZERO!...'
            //////////, (0)RETURN
Z←'TOTAL IS (0)'
                  @BS
   .THERETURN
```

Re-invoking line editing

Once you've used a dot, the system acts on the line. If it turns out that your modified line still isn't satisfactory, you can always invoke line editing again. For example, the preceding sample showed a dot used to insert *THE*. In fact, that wasn't correct. (This insertion in the example above is shown on the following line for illustrative purposes only.) If you display *Z* you'd see

```
      Z
THETOTAL IS (0)
```

You could make a new correction by invoking)4 and using a dot followed by one blank. The system would make the change, but since you used dot rather than comma, wouldn't show it to you.

Changing your mind -- aborting editing

The consecutive keystrokes *O* BACKSPACE *U* BACKSPACE *T* request an exit from editing. The line is saved unchanged even if characters have already been typed.

It is possible to abort the line editing process -- whether or not you've already typed in changes. To exit from editing all you do is enter *O* BACKSPACE *U* BACKSPACE *T*. You must enter this sequence of characters and backspaces consecutively. Even if you space over to the right and accidentally, or intentionally, type in more characters, the system will recognize the *O* BS *U* BS *T* sequence after you've entered a RETURN.

After line editing has been aborted, the system returns to what it was doing before you started. If you were in immediate execution, the cursor returns to the usual APL indent position. The keyboard is turned over to you. Your previous line of APL will be saved unaltered regardless of what else you typed on the line before entering *O* BS *U* BS *T*. If you are in definition mode, the system displays the next line number (in brackets) and awaits a function editing instruction.

Line editing with suspended functions

A line of entry which ultimately invokes a suspended function is not directly available for editing. It remains intact and unaltered, and you cannot recall it with) or)n until you clear the appropriate levels of the state indicator. Entering a naked branch → clears a suspended function from the top of the state indicator, but doesn't affect the line of input which invoked it. Enter as many → as necessary to get to an appropriate level of the state indicator. Then) or)n recalls the line which invoked the suspended function you last cleared.

If, while in immediate execution mode, you enter a line of APL which results in a suspended function, you can't retrieve that line with a) or)n until you've cleared the appropriate level of state indicator. After a function becomes suspended, if you enter a) to recall your previous APL line, it appears as if there is none. The cursor simply returns to the standard APL indent position.

You can use the system command *)SI* to request display of the state indicator. Then you can enter → successively until you reach the desired level of the state indicator. Note that if you enter additional statements while in the suspended state, you do not affect the ability to recall the line which resulted in the suspension.

This feature of the system can be useful in retrieving a problem line without retyping it. Note that the system command *)RESET* clears the entire state indicator stack **including** the last line of immediate input.

Common errors

Exceeding *⎕PW* If you request editing using a right parenthesis and a number which exceeds the value of *⎕PW*, the system responds with the message *ALREADY SIGNED ON SHARP APL SYSTEM.*

If you attempt to make insertions that would make your line longer than permitted by *⎕PW* **and** you're using a form of editing which requires display, the system rejects your request with the message *CHAR ERROR.* Then it displays the last valid version of the line. The cursor waits on the same line in the first available position to the right. You can continue by appending characters to the right or overstriking blanks. If you press RETURN without making any additional changes, your line remains as it was before you started the editing sequence that elicited the *CHAR ERROR.*

Inserting unintentional blanks Entering either a dot or a comma and spacing further to the right (with or without backspacing) results in the insertion of the number of blanks you spaced to the right as well as any characters you subsequently entered before pressing RETURN.

To avoid the inclusion of unintentional blanks after spacing to the right of a dot or comma, use a combination of SPACE or BACKSPACE to position the cursor where you want it. Then press LINEFEED. Now you can type the desired characters and press RETURN.

Illegal overstrikes Illegal overstrikes during line editing are treated in the same way as in any other situation. That is, when you accidentally overstrike one character with another to form a combination that is meaningless to APL, the system rejects your entry with the message *CHAR ERROR*. It displays your entry up to that point, and waits for you to complete it correctly.

SUMMARY OF LINE EDITING

Command	Description
LF	**Before** pressing RETURN, LINEFEED deletes characters above and to the right of the cursor. Position the cursor with SPACE or BACKSPACE.
) or)0	In immediate execution mode, requests one-pass editing of the previous APL line. The system displays the last line of APL and leaves the cursor at the end of it.
[**m**□0]	In function definition mode, requests one-pass editing of the line numbered **m**. The system displays the line and leaves the cursor at the end of it.
)**n**	In immediate execution mode, requests two-pass editing of the previous APL line. The system displays the last APL line and puts the cursor on the following line at the position designated by **n.**
[**m**□**n**]	In function definition mode, requests two-pass editing of the line numbered **m**. The system displays the line and puts the cursor on the following line at the position designated by **n**.
/ or **ALPHANUMERICS**	During two-pass editing, each slash or alphanumeric serves to show where a character is to be deleted or one or more blanks is to be inserted. **But** those slashes and blanks that are to the right of the first dot or comma do **not** have this effect; they're text to be inserted. Each slash deletes the one character above it. Each alphanumeric inserts blanks to the left of the character above it. etc. The digits one to nine insert the corresponding number of blanks. The letters A to Z insert blanks according to the code: *A* equals five, *B* equals ten, *C* equals fifteen, etc.

. or ,	During two-pass editing, the presence of a dot or a comma causes the system to treat the modification in three fields. To the **left** of the first dot or comma, slashes or alphanumerics cause deletions and blanks. To the **right**, all characters are inserted as a consecutive block. The first dot or comma shows **where** the insertion is to go. The entire block of text to the right is inserted to the left of the character marked by a dot or comma.
[/ **ALPHANUMERICS]** . **[TEXT]**	**Terse editing:** Deletes or inserts as explained. The system acts on the modified line **without** displaying it.
[/ **ALPHANUMERICS]** , **[TEXT]**	**Serial editing:** Deletes or inserts as explained. The system displays the line as modified, and then waits with the cursor on the following line at the position marked by the comma on the first pass. Further changes can be made with comma, dot, or neither.
O BS *U* BS *T*	*O* BACKSPACE *U* BACKSPACE *T*: Consecutive keystrokes requesting an exit from editing. The line is saved unchanged even if characters have already been typed.

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

SATN-32
30 MAR 79

SHARP APL TECHNICAL NOTES

TITLE: SHARED VARIABLES

ABSTRACT: Shared Variables are APL variables which are simultaneously defined in two active SHARP APL workspaces, and thus provide communication between concurrent autonomous tasks.

KEYWORDS: Shared Variables
□*SC*
□*SVC*
□*SVN*
□*SVO*
□*SVQ*
□*SVR*
□*SVS*

U.S. Sharp Laboratories
Camas, Washington 98607
(360) 834-8741

SA-TN-02
30 JAN 79

SHARP APL TECHNICAL NOTES

System functions and variables for communication between workspaces

Your active workspace can communicate with another active workspace when it **shares** a variable with it. A **shared variable** is one that is common to your active workspace **and** the active workspace of another task.

Sharing is always pairwise. It takes place only by mutual agreement between two workspaces. While it's possible to set up a network with many parties, that network is made up of links each of which has only two end-points.

Sharing with whom? Workspaces and auxiliary processors: It's possible to share a variable (or several variables) with any other workspace that's active at the same time yours is. In principle, you could use shared variables to communicate with the active workspace of any other task running at the same time, whether it be an N-task, B-task, or T-task; whether that task be another task of your own, one belonging to another user, or one run by the steward of a public workspace. But sharing is always voluntary, and never takes place until **both** of the participating workspaces have agreed to share.

It's also possible to share variables with a program that is completely outside APL. That raises the possibility that an APL user, working as usual within an APL environment, may make use of facilities that are outside the APL system itself, and which perhaps provide quite different services, or operate in a quite different language. This is possible **provided** the non-APL program is running at the same computer facility at the same time **and** follows the rules for communicating with the APL system's shared variable processor.

From the point of view of the APL user, such a program, running outside APL but communicating by shared variables, is called an **auxiliary processor**. An auxiliary processor might be a file access mechanism, or a utility such as file-sort or print, or a language compiler, or almost any special purpose program of sufficient interest and benefit to warrant an interconnection with APL. Just which auxiliary processors are available—or whether any are at all—depends on the system's managers.

It doesn't matter with whom or what you share a variable: the functions used for sharing do not differ. In particular, they're the same regardless of whether you're sharing a variable with an auxiliary processor or with the active workspace of another APL task. To cover both those possibilities, this manual uses the word "processor". That term is deliberately chosen to refer equally to an APL active workspace and an auxiliary processor that uses shared variables. The processor with whom you share a particular variable is referred to as your **partner**.

Applications of shared variables in SHARP APL: Many applications are possible. The following are among the most common:

1. To communicate between your T-task's active workspace and the active workspace of an N-task you've started, either to receive back from it results it has calculated, or to give it new instructions after it has started. Such an N-task is useful for calculations that require more space or more computing time than would be convenient to include in your T-task.

2. To permit several different T-tasks to communicate with a common processor. For example, a data base administrator may run an N-task which shares variables with individuals enrolled in the data base, receives and coordinates their various requests, and does the actual reading and writing of files. Shared variables permit a single N-task both to manage contention between different users, and to validate or edit what individuals request. Following such a scheme, only the N-task may actually tie the files involved in the data base; the individual users have no need to tie them, to know what files are used, or even to know that they exist.

3. To model the interaction of autonomous but communicating processes. Shared variables are extremely useful for modelling the behaviour of a system made of components which work independently but which on occasion communicate amongst themselves.

How a shared variable permits communication

A shared variable is a single variable which is visible both from your active workspace and from your partner's active workspace. A variable that two workspaces share is thus what in engineering is called an **interface**.

When you give the shared variable a new value, your partner may then refer to it, and thereby receive information from you. Similarly, when your partner gives the shared variable a new value, you may refer to it, and thereby receive information from your partner. (Notice that when a variable is shared, you may find that it has a different value each time you refer to it, even though you never set it at all.)

EXAMPLE: EVENTS IN YOUR WORKSPACE AND IN YOUR PARTNER'S WORKSPACE BEFORE AND AFTER YOU SHARE A VARIABLE *X*

Time	Your Active Workspace	Your Partner's Active Workspace
1	`X←⍳5`	

2	`X`
	`1 2 3 4 5`
		`X`
		`VALUE ERROR`
		`X`
		`^`

3	Sharing established between you and your partner	
4	`X←'ABC'`	
5		`X` `ABC`
6		`X←2×⍳3`
7	`X` `2 4 6`	`X` `2 4 6`

No special function for "send" or "receive": When either you or your partner uses the shared variable, what each of you sees is its current value. When either of you sets the shared variable, that changes the current value.

When you use this property to communicate with another processor, there's no need for a special function for an action you might call "put", "send", or "write", nor for a corresponding action you might call "get", "read", or "receive". The act of setting a new value for the shared variable has the effect of transmitting it to your partner, and the act of referring to the shared variable has the effect of receiving from your partner.

For effective communication in most applications, it's important that the two workspaces keep in step. For example, you may want to ensure that each time you read the shared variable, your partner has given it a new value exactly once. If the two of you don't keep in step, the one that's receiving may miss something, or may inadvertently receive the same message more than once. Communication can be appropriately **interlocked** by using **access control**, described in the sections entitled "state vector" and "access control".

How sharing is initiated

Sharing between you and your partner starts when each of you has made an offer to the other. It doesn't matter who offers first. Your offer identifies both the processor to whom it's directed and the name of the variable you propose to share. You do that by using the dyadic function ⎕*SVO*. Its left argument identifies the processor with whom you propose to share, and its right argument contains the name (or names) you propose to share.

For example, suppose there is an active workspace belonging to a processor which can be identified simply by the number `7844919`, and you wish to share with it a variable named *X*. You could extend to it an offer to share *X* by the expression

```
      7844919 ⎕SVO 'X'
```

Under some conditions, it takes two numbers to identify a processor. If you wanted to share *X* with the processor identified as `1618033 3`, you'd need to enter

```
      (1 2⍴1618033 3) ⎕SVO 'X'
```

Before you can make such an offer, you need to know the processor's ID, and what names to share with it. You may have that information by pre-arrangement. Or, using the function ⎕*SVQ*, you can inquire what processors (if any) are offering to share with you, and what names they propose to share.

General offer: If in the left argument of ⎕*SVO* you use a processor ID of `0` (or a 2-element ID of `1 2⍴0 0`), you have made a **general offer**. That is, you've offered that particular variable to anyone. Sharing can be established by any processor which replies with a specific offer (that is, one whose left argument uses your specific processor ID).

You can't use a general offer to accept a general offer made by someone else. Conversely, if you make a general offer, it can't be accepted by someone else's general offer, but requires a specific response.

Degree of coupling: Each time you use ⎕*SVO* to offer to share a variable with another processor, the explicit result is the **degree of coupling** of the name that you offer. You can also use ⎕*SVO* monadically to inquire about the current degree of coupling of a name in your workspace. For example, if you make the offer illustrated in the preceding paragraph, you might get the result `1`:

```
      (1 2⍴1618033 3) ⎕SVO 'X'
1
```

Or you might simply inquire about the degree of coupling of the names *X*, *Y*, and *OK*, and discover that they are respectively `1`, `2`, and `0`:

```
      ⎕SVO 3 2⍴'X Y OK'
1 2 0
```

A degree of coupling may be `0`, `1`, or `2`. The value indicates the number of processors which have offered to share each of the names you mentioned.

When the degree of coupling becomes 2, you have offered the name and another processor has made a corresponding offer. Sharing commences. The shared variable is then visible both in your active workspace and in the active workspace of your partner. A value set after the degree of coupling becomes 2 is visible to either partner. Until one of you sets a new value for the variable you have just begun to share, each workspace continues to contain the former (unshared) value.

When the degree of coupling for a name (in your workspace) is 1, you have made an offer but your partner has not. Sharing is not complete, and no communication can take place. (When a processor has made you an offer that you haven't accepted, that name will have degree 1 for that processor. However, in **your** workspace, that name is still quite independent of what anyone else has done, and so for you, that name's degree of coupling remains 0.)

When the degree of coupling is 0, the corresponding argument is not a name or is not a shared variable.

Note about the terms "use" and "set:" Since communication depends upon one partner making use of a value that the other partner has set, it's important to maintain a clear distinction between the terms used for these contrasting actions. The terms "set", "use", and "reference" are defined as follows:

1. Each time you use the name of a variable immediately to the left of a specification arrow, you **set** the variable. ("Set" thus corresponds to "write".)

2. Each time you use the name of a variable in a position other than immediately to the left of a specification arrow, you **use** the variable. ("Use" thus corresponds to "read".)

3. Any occurrence of the name of a variable is a **reference** to the variable, and may be either a set or a use. An expression such as *A←X←B* contains only one reference to *X*, in which *X* is set (and *A* receives the result of the specification *X←B*, which is *B*.)

4. There are three functions which (while in some sense they "use" the value of a shared variable) do **not** count as a use for the purposes of access control. They are ⎕*PACK* (which may incorporate the name and value of a variable into a package), 6 ⎕*WS* (which reports the current value of a variable), and 4 ⎕*WS* (which reports the number of bytes occupied by the current value of a variable). See also the section on ⎕*PACK* and ⎕*WS* with a shared variable.

Initial value of the shared variable: Once sharing starts, the shared variable exists in both workspaces, and so has the same value regardless of the workspace from which you look at it. But suppose either or both of you had already established a value for a variable of that name before sharing started? That situations is covered by the following rules.

1. If at the time sharing is established, neither of you has assigned a value, then the shared variable has no value.

2. If at the time sharing is established a value has been assigned by one of you but not the other, that value becomes the value of the shared variable.

3. If at the time sharing is established, each of you had already assigned a value to the variable that you now share, the shared value is the value set by whichever of you first offered to share the variable. (Watch out: that's not necessarily the same as saying "whichever of you first assigned a value to the variable".)

Regardless of how the shared name comes by its initial value, its initial state (as described in the section "The state vector") is always 0 0 1 1.

Duration of coupling: Once a share has been established (that is, when the degree of coupling of a variable is 2), sharing continues until any of the following happens:

1. Either of you retracts the offer to share, using the system function ⎕*SVR*.

2. Either of you signs off (or is bounced, or terminated in any other way).

3. Either of you frees the name in the active workspace. Either of you could do that by any of the following:

   a. By expunging the name in your active workspace. You might do that by using the commands)*ERASE* or)*COPY*, or the functions ⎕*EX* or 6 ⎕*FD*, or by using ⎕*PDEF* to redefine that name (regardless of what new meaning ⎕*PDEF* may give it).

Note that when the shared variable is global, if you use the command)*COPY* to replace it by another object of the same name, that has the effect of expunging the old use of the name, even though you thereby replace it by another use of the same name.

   b. By terminating execution of a function to which the shared name is local.

   c. By replacing the entire contents of the active workspace, by a command such as)*LOAD* or)*CLEAR*, or by one of the system functions ⎕*LOAD* or ⎕*QLOAD*.

When you expunge the name of the shared variable, sharing ceases. You no longer have a way of referring to that variable's value. However, the variable continues to be visible in your partner's active workspace, and your partner continues to have an offer to share that variable with you. If you renew your offer to share that name with the same partner, you'll again be able to see its value.

Watch out: You may use shared variables for some applications for which you also use file ties. Notice that the maximum life of a shared variable is the duration of the active workspace, whereas the maximum life of a file-tie is the entire duration of the task. An offer to share never survives)*LOAD* or)*CLEAR*, whereas a file-tie does.

Processor ID

To share a variable, each of the participating workspaces must make an offer to the other. To do that, each must be able to identify the other. You identify an active workspace by its **processor ID**.

You need to know your partner's processor ID, and you need to establish a processor ID of your own. When you inquire about offers directed to you, or make an offer to another processor, you don't explicitly mention your own processor ID. However, at that time the system must be able to refer to an explicit ID that identifies you during the life of the active workspace you're using. To do that, the system uses a two-element vector. The first element is simply your account number (usually a 7-digit positive integer). An auxiliary processor is assigned a number in a similar way, but by convention is given a number between 1 and 1000.

Clone ID: The second element of the processor ID serves to distinguish between the active workspaces of two tasks running concurrently under the same account number. It is called the **clone ID**. The clone ID is a number that an active workspace may arbitrarily assign itself.

If you never run an N-task or a B-task, the clone ID won't matter, since you won't need to distinguish your active workspace from their active workspaces. You can leave unchanged the default clone ID of 0, which the system otherwise assumes for every new active workspace.

Even if you run several tasks at the same time, the clone ID doesn't matter **until** you either inquire about share offers addressed to you (using the function ⎕*SVQ*) or make some share offers yourself (using the dyadic form of ⎕*SVO*). When you use either of those functions, you must have established a clone ID that distinguishes your present active workspace from the present active workspace of any other task using shared variables.

When a workspace is first loaded (and also in the workspace you get at sign-on), the clone ID is `0`. Thus workspaces belonging to N-tasks or T-tasks owned by a single account are not distinguishable until they assign themselves distinct clone IDs. One of them may retain the clone ID `0`. If none of them explicitly sets a clone ID, whichever is first to make a share offer or inquiry (that is, to use ⎕*SVQ* or dyadic ⎕*SVO*) gets `0` as its clone ID. Once one of the workspaces has laid claim to `0`, none of the others will be able to use ⎕*SVQ* or dyadic ⎕*SVO* (and hence won't be able to use shared variables) until it establishes a different clone ID.

Setting a new clone ID: The function ⎕*SVN* is used to set the clone ID for your active workspace. The argument of ⎕*SVN* is the single integer you propose. For example, to propose the clone ID `99`, you execute:

```
⎕SVN 99
```

The result is the clone ID in effect following your use of ⎕*SVN*. If you've succeeded in setting a new clone ID, the result is the new value. However, once you've started to make use of your clone ID, you can't revise it unless you retract all your shared variables. During use, your clone ID is said to be "frozen".

If you propose a clone ID that doesn't distinguish your workspace, the function returns the result `¯1` (which couldn't be a clone ID, since they're restricted to non-negative numbers).

Freezing the clone ID: As long as you have a shared variable, the system freezes your clone ID. You can't set it to anything else. When you reduce to zero the number of shared variables in your workspace (including variables that may at the moment be shadowed), you are again free to change your clone ID. (During the execution of a function containing a local name, the local meaning is said to be "visible", and is said to "shadow" other uses of the name.) Sharing a variable without first setting a clone ID has the effect of freezing your clone ID at `0`.

If you attempt to use ⎕*SVQ* or dyadic ⎕*SVO* when your clone ID is the same as one frozen by another of your processors, the system rejects your statement with the error message *IDENTIFICATION IN USE*. That's an implicit error, since your statement does not explicitly contain the processor ID.

Picking a name for a shared variable

When you make an offer to share, you specify a name by which each variable will be known in your workspace. Each name is formed in the same way as any other name in the workspace. It can't be a distinguished name (that is, it can't be one of the system names which start with ⎕). It may be a name local to a function now in execution. In that case, the share will terminate as soon as execution of that function terminates.

A name you propose for use within the workspace must have no visible use as a label or as the name of a function. It's all right to offer a name which doesn't yet have any visible use; that is, it doesn't matter whether the variable exists at the time you offer its name. The act of offering to share a name establishes its nameclass as "variable".

Surrogate names: You and your partner may refer to the same variable by different names. When you state the name by which the variable will be known in your workspace, you may also state a different name by which your partner may refer to it. The second name is called a **surrogate name**. For example, you might propose to share a variable which in your workspace is called *COMMAND* but which your partner calls *WISH*.

(In the same fashion your partner may also use a surrogate name, so that within its active workspace it may use yet another name to refer to the variable whose surrogate name is *WISH*. But the surrogate you offer must match the surrogate offered by your partner.)

The surrogate name (if you use one) appears to the right of the name to be used within your workspace, separated from it by at least one blank. For example, to offer to `7844919 0` a name which you will call *COMMAND* but which it will see as *WISH*, your entry and its result might appear as follows:

```
      (1 2⍴7844919 0) ⎕SVO 'COMMAND WISH'
1
```

If your processor ID is `1436713 99`, your partner could make a matching offer to you. For example:

```
      (1 2⍴1436713 99) ⎕SVO 'WISH'
2
```

Or your partner could elect to use some quite different name internally (for example, *X*) by an offer such as

```
      (1 2⍴1436713 99) ⎕SVO 'X WISH'
2
```

The system requires that the **last** name on each row in your offer match the **last** name in the corresponding offer from your partner. That remains true whether the last name on each row is the only name or is the surrogate name.

Inquiring about incoming offers

The function ⎕*SVQ* permits you to inquire about shared-variable offers being made to you.

When the right argument of ⎕*SVQ* is an empty vector, ⎕*SVQ* returns a matrix containing the processor ID of each processor that has an outstanding offer to you. For example, if you execute ⎕*SVQ* `⍳0` and find the following result:

```
      ⎕SVQ ⍳0
7844919       0
1618033       3
```

this means that offers to share with your processor ID have been made by processors `7844919 0` and `1618033 3`, and are still outstanding. That is, you've not accepted them and they've not been retracted.

When you you use ⎕*SVQ* with a single processor ID as its argument, it returns a character array of the name (or names) which that processor is offering to share with you, but which you have not yet accepted. The argument of ⎕*SVQ* can only refer to one processor. If the argument is a scalar or a one-element vector, the system assumes a clone ID of `0`.

If an active workspace with the processor ID `1618033 3` has an outstanding offer to you, you might execute ⎕*SVQ* `1618033 3` and get the following result:

```
      □SVQ 1618033 3
X
WINDOW.
```

This means that the processor has made offers (thus far unrequited) to share with you the names *X* and *WINDOW*.

Ambiguous use of □*SVQ* or dyadic □*SVO*: When you use □*SVQ* you are in effect asking, "What offers are being made **to me**?" Similarly, when you offer to share a variable with dyadic □*SVO*, you are in effect saying "**I** make this offer". Either of those actions is ambiguous if at the time your workspace lacks a unique clone ID. The system rejects such a use with the error message *IDENTIFICATION IN USE*.

Point of view in the state vector and access controls

When two workspaces share a variable, sharing is symmetric. Neither workspace takes precedence over the other. It doesn't matter which made its offer first and which second. There are two functions which set or report controls for the shared variable. Each of them takes as its argument or returns as its result a 4-element vector. In each vector, the first two elements always concern **setting** the shared variable, while the last two always concern **using** it. Which processor the elements refer to is a matter of point of view. The first and third elements always refer to the workspace that's making the inquiry, while the second and fourth always refer to the other workspace.

Set		**Use**	
□	□	□	□
You	**Your Partner**	**You**	**Your Partner**

Each workspace sees itself in the first and third positions, and its partner in the second and fourth positions.

This convention follows ordinary speech: the pronoun "you" does not refer to a particular person, but depends on who is speaking. The first and third elements of the state vector or of the access vector are in this SATN labelled "you", while the second and fourth are labelled "your partner".

Keeping track of who's set what: In many applications, to make sure that parts of the communication are not lost, it is essential to set an **access control** to prevent one processor from setting values that the other can't use, or from using when the other hasn't set.

It may also be useful to examine the shared variable's **state vector**. The state vector indicates which of you has set a value of which the other is still ignorant. It also shows which of you is aware of the shared variable's current value.

The state vector

The function □*SVS* gives you the state of each of the variables named in its argument.

In general, the result is a 4-column matrix, containing on each row the state vector for the corresponding name in the argument. When you inquire about a single variable and use a scalar or vector argument to do so, the result is a 4-element vector.

The first two elements: who has set a value that the other hasn't used? The first two elements of the state vector indicate which processor has set a value of the shared variable that has not been used by the other. The first element refers to "you", and the second to "your partner".

The partner who has set a value that remains unused is indicated by a 1. A 1 can occur as one of those elements when that partner was the last to set the shared variable **and** the other partner hasn't used it. There's no way the first two elements could both be 1. However, they could both be 0. When both partners are aware of the shared variable's value, neither of them can have set a value of which the other is unaware, and so the first two elements are both 0. The first two elements are also 0 when sharing has just been established, and neither processor has yet set the shared variable.

The last two elements: who's aware? The last two elements of the state vector indicate which of the partners is aware of the shared variable's current value. The third element refers to "you", and the fourth to "your partner".

The partner who last set a value for the shared variable must be aware of it (having just set it. The other partner can be aware of the shared variable's value only if it has used it since the time it was set. Thus the last two elements may be either one 1 or two 1s.

When sharing has just been established, it is presumed that neither processor has yet set a value for the shared variable, but that each knew what value the variable had in their respective workspaces before it was shared.

Possible states: During active sharing, only three distinct states of □*SVS* are possible. (The fourth state is all zeros, indicating that the name is not shared.)

POSSIBLE VALUES OF □*SVS* AND THEIR INTERPRETATION

Four possible states

Who has set a value that the other hasn't used?		Who knows the current value?		
You	Your partner	You	Your partner	**Interpretation**
0	0	1	1	You and your partner are both aware of the current value, and neither has since set (and initial value when sharing first established).
1	0	1	0	You've set the shared variable, but your partner hasn't used it.
0	1	0	1	Your partner has set the shared variable, but you haven't used it.
0	0	0	0	The name isn't the name of a shared variable.

You might think that a 4-element vector is a rather extravagant way to represent a system with only four possible states. That representation was chosen because it corresponds to the one used for the access control vector.

Access control

Either of the partners may assure effective communication between them by setting the appropriate elements of the **access control** vector.

The goal in synchronizing communication is to **inhibit** the partner who is about to set a new value from doing so too soon (before the other has had a chance to use the former value), and to inhibit the partner who is using from doing so too soon (before the other has had a chance to provide the data to be used). The access control vector allows the system to enforce the appropriate delays.

Like its state vector, a shared variable's access control is a 4-element Boolean vector. By making the appropriate elements 1, either partner may specify that certain actions will be inhibited. You set the access control vector by using the dyadic function □*SVC*. Before either partner has set it, its value is 0 0 0 0.

You can't set a name's access control until after you have offered to share it. If you attempt to do so, the result is the access control as it was, unchanged.

The right argument of □*SVC* is the name of a shared variable, and the left argument is the 4-element vector indicating which actions you propose to inhibit. Alternatively, the right argument may be a matrix of names, in which case the left argument is either a corresponding matrix of 4-element vectors, or a single vector which applies to all of them.

A shared variable's access control vector has four elements with significance as follows:

Inhibit Set		**Inhibit Use**	
□	□	□	□
You	**Your Partner**	**You**	**Your Partner**

Wherever a variable's access control vector contains a 1, the corresponding action is inhibited. The effect of a 1 in each of the four positions is as follows:

1 You are inhibited from setting a new value for the shared variable until after an intervening use (or set) by your partner.

2 Your partner is inhibited from setting a new value for the shared variable until you've done an intervening use (or set).

3 You are inhibited from using the value of the shared variable until after an intervening set by your partner.

4 Your partner is inhibited from using the value of the shared variable until after an intervening set by you.

Effective access control: The system keeps track of the access control vector proposed by each partner for each variable. The result of each use of □*SVC* is the **effective** access vector. It is calculated by OR-ing together the access control vectors proposed by each of the partners.

Because the effective access control is obtained by the logical function OR, either partner is free to insert a 1 in any of the four positions (that is, to inhibit any of the four possible actions). However, one partner cannot change to 0 any element that the other set to 1. That is, you can always increase the level of inhibition for yourself or your partner, but you can remove only those inhibitions which you set.

Effect of access control on the three possible states: In most circumstances, communication is considered effective only if each partner receives everything sent by the other exactly once (neither missing anything nor receiving spurious duplicates). An effective transmission is thus one that changes the state vector from one of its three possible settings to another.

There are two ways an action could leave the state vector unchanged:

1. If one of the partners were to set the shared variable twice in a row (before the other partner could use the earlier value).

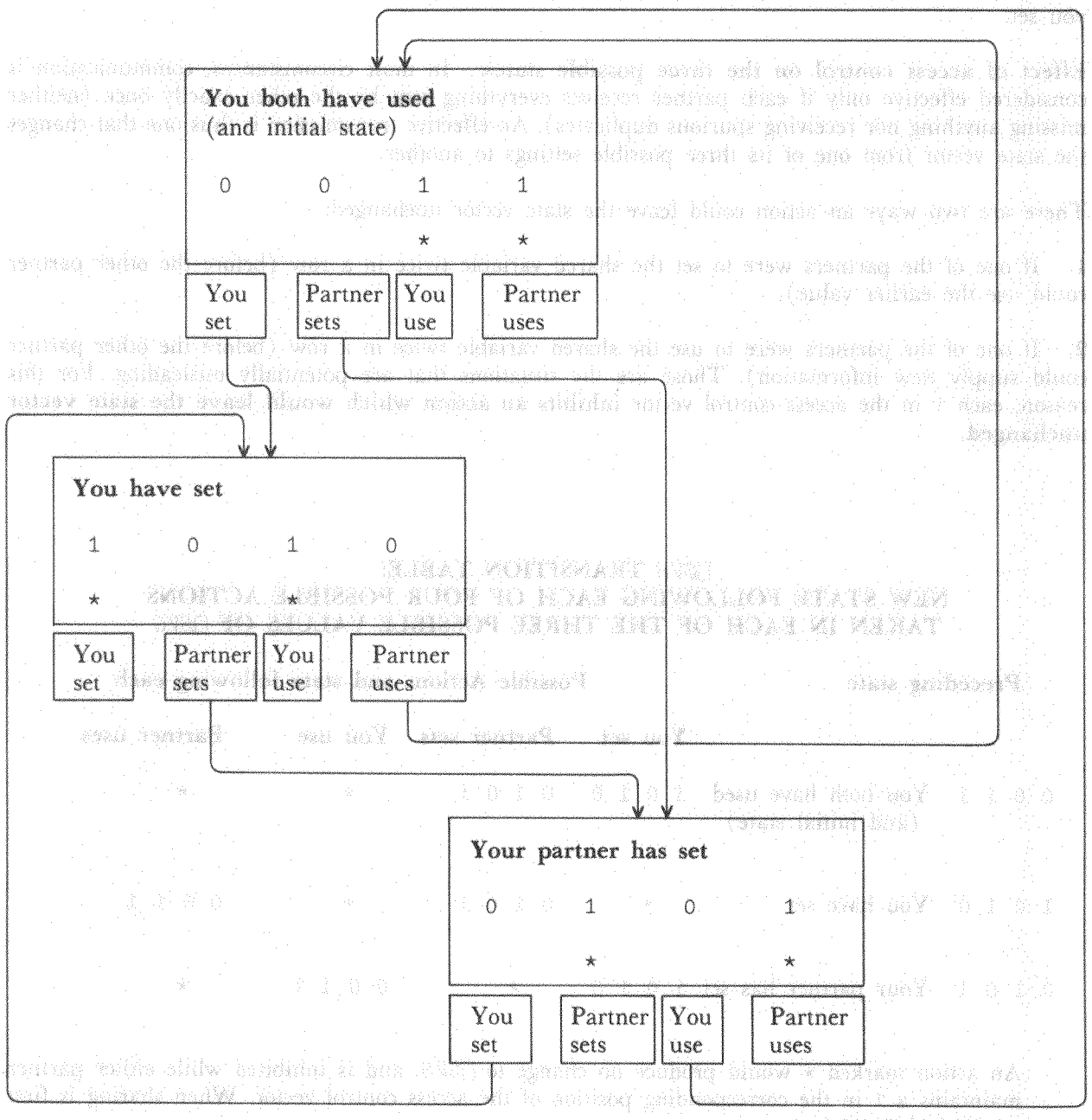
2. If one of the partners were to use the shared variable twice in a row (before the other partner could supply new information). Those are the situations that are potentially misleading. For this reason, each 1 in the access control vector **inhibits an action which would leave the state vector unchanged.**

□*SVS* TRANSITION TABLE:
NEW STATE FOLLOWING EACH OF FOUR POSSIBLE ACTIONS
TAKEN IN EACH OF THE THREE POSSIBLE VALUES OF □*SVS*

Preceding state		**Possible Actions and state following each**			
		You set	**Partner sets**	**You use**	**Partner uses**
0 0 1 1	You both have used (and initial state)	1 0 1 0	0 1 0 1	*	*
1 0 1 0	You have set	*	0 1 0 1	*	0 0 1 1
0 1 0 1	Your partner has set	1 0 1 0	*	0 0 1 1	*

An action marked * would produce no change to □*SVS*, and is inhibited while either partner maintains a 1 in the corresponding position of the access control vector. When sharing is first established □*SVS* is 0 0 1 1.

□SVS TRANSITION DIAGRAM:
PATH TO NEW STATE FOLLOWING EACH OF FOUR POSSIBLE ACTIONS FROM EACH OF THE THREE POSSIBLE VALUES OF □SVS



An action marked by * would produce no change in □SVS, and is inhibited while either partner maintains a 1 in the corresponding position of the access control vector.

The information in the preceding figures is represented graphically in Figure 16 of **APL Language**, IBM publication GC26-3847-4. In that publication, the state vector and the access control vector are depicted as 2-by-2 matrices.

Attempting to carry out an action that is inhibited: Suppose you have a variable whose state vector at present contains `1 0 1 0` ("You've set") and whose control vector has a `1` as its first element ("Inhibit me from setting until after my partner has used"). You attempt to give it a new value anyway. What happens? You experience a delay. There is no interrupt or error message, but the system waits to make your specification until your partner uses (or sets) the variable. The delay lasts indefinitely—as long as it takes. You can't do other work, but the CPU costs for the delay are negligible.

From the terminal of a T-task, you can signal a strong interrupt (two successive uses of the ATTN or BREAK key). This aborts your attempt to use the shared variable. The system displays the error message *SV INTERRUPT*, and displays the statement it was attempting to execute.

A similar delay will occur if a variable's state vector has a `1` in the third position (meaning that you know what the variable's value is), and there's a `1` in the third element of the variable's access control (meaning "Inhibit me from using until after my partner has set"). There's an indefinite delay until your partner has again set a value for the shared variable. As before, you can't do other work, but the delay has negligible CPU cost. You can interrupt the attempt to use the interlocked variable by signalling a strong interrupt.

Watch out: While you're trying to discover what went wrong with a program that has been interrupted by an error, it may seem natural to attempt to display a shared variable. But such a display counts as a use of the variable, and might be inhibited.

If you wish to be able to make repeated references to the value of an interlocked shared variable, the best technique is to immediately assign its value to another (non-shared) variable, so that you can refer to it as often as need be, without making a new reference to the shared variable.

Note on `⎕PACK`, `4 ⎕WS` and `6 ⎕WS` applied to a shared variable: When you use the name of a shared variable in the argument of any of those three functions, as far as the shared variable processor is concerned that does **not** constitute a use of the shared variable. With `⎕PACK` and `6 ⎕WS` you get the last value visible in your workspace, and with `4 ⎕WS` you get the size of the last value visible in your workspace. Thus you only get a value that you already know (either because you've already made some other use of it, or because you're the one who set it). Using any of those three functions has no effect on the shared variable state.

Calculating in advance which actions will not be delayed: By examining a variable's state vector and its access control vector, you can tell which actions will **not** be subject to delay. Recall the four possible actions: You set, your partner sets, you use, and your partner uses. For a variable *X*, an action you're free to take immediately (that is, with no delay) is any of those indicated by a `1` in the expression

```
(⎕SVS 'X')∧⎕SVC 'X'
```

Notice that if your partner has set a value for the shared variable, you aren't obliged to use it even when the variable is fully interlocked. You could go ahead and set a new value yourself. In most circumstances, that would be foolish, but you might want deliberately to signal your partner that you no longer want to use what the partner is transmitting. This is symmetrical: when you've set the shared variable your partner isn't obliged to use the value you've set either, but always has the option to set the variable again.

Meaning of some possible settings of a variable's control vector: The following examples illustrate the effect of some possible settings of a shared variable's control vector.

`0 0 0 0` No interlock. Such a setting is reasonable only when you know that there are other constraints that will prevent the two processors from getting out of step. This might happen if the programs which use this shared variable without interlocks also make use of another variable that is interlocked.

`1 1 0 0` Neither of us can set a new value until the other has used it. (Sometimes called half duplex.)

`0 0 1 1` Neither of us can use the shared variable until the other has set it. (Sometimes called half duplex.)

`1 0 0 1` You can't set until your partner has used, and your partner can't use until you've set. (Sometimes called simplex transmission from you to your partner.)

`0 1 1 0` You can't use until your partner has set, and your partner can't set until you've used. (Sometimes called simplex transmission from your partner to you.)

`1 1 1 1` Full interlock: reversing half-duplex.

State-change signal

Suppose you have a function that uses several shared variables (for example, an N-task that serves several users at once). The function may on occasion reach a point where it has nothing to do immediately, but is waiting for a reply to come in from any of its several shared variables.

⎕SC is a variable that is automatically shared with the system. Whenever you use *⎕SC*, you get the value that the system has set. That value is either `0` or `1`. The interesting thing about *⎕SC* is usually not **what** its value is, but **when** you get it. While you are using shared variables, between any two successive uses of *⎕SC*, the system must have detected a **state change**. If there hasn't been a state change, execution is delayed until there is one.

State change: A state change is any change in the result of *⎕SVS* for any variable you're now sharing, or any new offer to share, or any retraction of a variable already shared.

Once your active workspace has established a valid clone ID, each time a state change occurs, the system sets the value of *⎕SC* to `1`. You cannot use that value more than once; when you next refer to *⎕SC*, the value you get will be a value that the system set **after** it set the value you used previously, and therefore indicating that there has been at least one state change since you last used *⎕SC*. Thus the value of *⎕SC*—when you succeed in referring to it—is always `1`.

Whenever your workspace lacks a valid clone ID, the value of *⎕SC* is `0`, and is delivered immediately.

If there has been a state change since you started sharing or since the last time you referred to *⎕SC*, you get that value `1`. But if there hasn't been a state change, the system doesn't return the value of *⎕SC* **until there's been a state change.** Hence, referring to *⎕SC* amounts to waiting indefinitely (at no CPU cost) until some state change takes place. However, if someone retracts an offer that was made to you but which you have **not** accepted, that doesn't count as a state change.

A reasonable sequence for using *⎕SC* is:

1. Using *⎕SVS*, verify that there's nothing you can do at present with the relevant shared variables.

2. Use *⎕SC* in an expression such as

```
→(⎕SC/LABEL),ERROR
```

Provided you have a valid clone ID, that will delay the branch to *LABEL* until some state change occurs. But if you lack a valid clone ID (and so can't possibly be making effective use of shared variables) control will pass immediately to *ERROR*.

3. When you've been able to use ⎕*SC*, use ⎕*SVS* or ⎕*SVQ* to review the situation and decide what action is possible or appropriate.

Retraction

The function ⎕*SVR* retracts your offer to share each of the variables named in its right argument. This cancels sharing of any variables for which sharing was in effect, or withdraws the offer to share those that the intended partner hasn't yet accepted.

The result is a vector indicating the level of coupling of each of the variables before the offer was withdrawn.

Illustrative programs for an N-task with multiple users

The following definition is intended to illustrate how you might set up an N-task or B-task capable of accepting shared variable offers, and handling work in rotation from each of the various sharers, each running at its own pace.

```
          ∇ MONITOR CLONEID; SHVARS; IDS
[1]         SHVARS← 0 0 ⍴IDS← 0 2 ⍴0
[2]         CLONEID←⎕SVN CLONEID
[3] CHECK:  WORK
[4]         OLDOFFERS
[5]         NEWOFFERS
[6] DWELL: →⎕SC/CHECK
[7]         'CLONE ID NOT ESTABLISHED' ⎕SIGNAL 555
          ∇
```

The subroutines called by *MONITOR* are:

```
    ∇ OLDOFFERS; OK; J
[1]   →(∧/OK←2=⎕SVO SHVARS)↑0
[2]   J←⎕SVR(~OK)⌿SHVARS
[3]   SHVARS←OK⌿SHVARS
[4]   IDS←OK⌿IDS
    ∇

    ∇ NEWOFFERS; J; ID; NEWIDS; NEWNAMES; SURROG
[1]   NEWIDS←⎕SVQ ''
[2]  TEST: →(0=1↑⍴NEWIDS)↑0
[3]   NEWNAMES←ID NAMEFN SURROG←⎕SVQ ID←NEWIDS[⎕IO;]
[4]   J←(((1↑⍴NEWNAMES),2)⍴ID) ⎕SVO NEWNAMES,' ',SURROG
[5]   J← 1 1 1 0 ⎕SVC NEWNAMES
[6]   SHVARS←SHVARS ON NEWNAMES
[7]   IDS←IDS,[⎕IO]((1↑⍴NEWNAMES),2)⍴ID
[8]   NEWIDS← 1 0 ↓NEWIDS
[9]   →TEST
[10] ⍝APPENDS NEW IDS AND NAMES TO GLOBALS <IDS> AND <SHVARS>
    ∇
```

```
    ∇ WORK; CHANGED
[1]   CHANGED←((⎕SVS SHVARS)[;1+⎕IO])/⍳1↑⍴SHVARS
[2]  TEST: →(0∈⍴CHANGED)↑0
[3]   ⍎SHVAR,'← PROCESS ',SHVAR←SHVARS[''⍴CHANGED;]
[4]   CHANGED←1↓CHANGED
[5]   →TEST
    ∇
```

In an application such as this, it is common for each of the T-task workspaces which shares with the "monitor" to offer the same name. Suppose each offers to share *X*. In order to distinguish the various names, for each *X* offered to it the monitor must construct a unique internal name. One scheme for constructing names is to include in the internal name the processor ID of the workspace making the offer. For example, when the monitor receives from processor `7844919 5` an offer to share *X*, the reply uses *⎕SVO* with a right argument which includes *X* as the surrogate name paired with an internal name, perhaps as follows:

```
(1 2⍴7844919 5) ⎕SVO 'X7844919005  X'
```

A function *NAMEFN* could generate unique names in such a fashion.

Storage space for the value of a shared variable

A value that your partner has set isn't visible in your active workspace until you refer to it. It's possible for your partner to give the shared variable a value that requires so much storage space that (although it fits in your partner's workspace) there isn't room in your workspace to receive it. If that happens, at the moment that you attempt to refer to its value, the system interrupts work and displays the error message *WS FULL*.

Saving a workspace that contains shared variables: When a workspace is saved, the value of each of its variables must be physically present within it. When you give the command *)SAVE* or *)CONTINUE*, the system first moves into the workspace the current value of each of the shared variables which you haven't yet used.

If it proves impossible to transfer the value of a shared variable because there isn't space, the system displays the error message *WS FULL*, and does **not** save the workspace.

Following a bounce or line-drop, the system executes *)CONTINUE* on your behalf. If there isn't sufficient space to bring in the values of all of your shared variables, the system saves the workspace anyway, with as many of the values as it can accommodate.

DEFINITIONS OF SHARED-VARIABLE FUNCTIONS

R←⎕*SVQ Y* **Shared-variable query**

Argument: Either an empty vector, or a processor ID. When the argument is a processor ID, it may be a two-element numeric vector, or else a single number—either a scalar or a one-element vector. A single number is understood to imply a clone ID of 0. The values in the right argument must be non-negative integers.

Effect: Any use of ⎕*SVQ* implies the existence of a unique processor ID. Where there are active workspaces belonging to other tasks running concurrently under the same account number and also making use of shared variables, your workspace must have established a unique clone ID. If this has not been done, the system rejects your use of ⎕*SVQ* with the error message *IDENTIFICATION IN USE*.

Result when the argument is empty: When the argument of ⎕*SVQ* is an empty vector, the result is a numeric matrix containing the processor IDs of all processors who have outstanding (that is, unaccepted) offers to share with you. The result matrix has as many rows as there are distinct processors making offers, and two columns. The first column is the account number of the processor making the offer, and the second is its clone ID.

Result when the argument is not empty: A character matrix of names offered by the processor identified in the right argument. The matrix has a row for each name the processor has offered but you've not accepted. There as many columns as necessary to accommodate the longest name. If the processor identified in the argument has no outstanding offers, or if the argument (although a plausible processor ID) does not, in fact, identify an offering processor, the result is an empty matrix.

R←⎕*SVN Y* **Set clone ID**

Argument: A non-negative integer scalar. The proposed clone ID must not conflict with the clone ID of another workspace running under the same account number.

Effect: Provided the proposed clone ID does not conflict with another clone ID currently established for another workspace running on the same account number, and provided this workspace's clone ID has not been frozen, ⎕*SVN* sets the active workspace's clone ID to *Y*. (The clone ID is frozen once the active workspace makes any use of ⎕*SVQ* or dyadic ⎕*SVO*.)

Watch out: There's no way to survey the clone ID's adopted by other active workspaces. Applications which require an appropriate clone ID are presumed to adopt in advance a convention governing what clone ID they will use.

Result: If the clone ID was set successfully, the result is the new clone ID, *Y*. If the clone ID had already been frozen, the result is the frozen clone ID. If the proposed clone ID conflicts with a clone ID currently in use by another workspace running on the same account number, the result is ¯1.

R← ⎕*SVO Y* **Shared-variable coupling**
R←*X* ⎕*SVO Y* **Shared-variable offer**

Right argument: A character array of names. The array is generally a matrix, with one row for each name. Where the argument contains a single name, it may be a vector, or, if the name contains only a single character, a scalar.

Each name may be stated alone (on a single row of the matrix), or paired with its surrogate. Where two names occur on a single row (or where the argument is a vector of two names), they must be separated by at least one blank. The first name is the name used within the task's workspace, and the second name is a surrogate. A surrogate name is the name which the system uses to match your offer with the offer made by another processor.

Each of the names you propose must be well-formed. Either it must be a name that has no visible use, or it may already be in use as the name of a variable. Thus, it can't be a label or the name of a function. A surrogate name may be any well-formed name, regardless of what significance it may have within your workspace. When you use ⎕*SVO* monadically, it isn't necessary to repeat a name's surrogate. However, if you do include the surrogate, you must spell it correctly. The system won't recognize a name even when it's spelled correctly if you pair it with an incorrect surrogate.

Left argument: When the function is used dyadically, its left argument is a numeric array of the processor IDs indicating to whom the names in the right argument are offered. If the left argument is a scalar or a one-element vector, the system assumes a clone ID of 0. In general, the left argument is a 2-column matrix, with one row for each processor ID. The first column contains the processor's account number, and the second column its clone ID. Each element must be a non-negative integer.

The left argument must contain one processor ID for each name (or pair of names) in the right argument. However, if the left argument consists of a single processor ID (as a scalar or single-element vector with an implicit clone ID of 0), it is presumed to apply to each of the names in the right argument.

Watch out: If you use a two-element vector, the system interprets that as **two** distinct processors, each with a clone ID of 0, and not as a single processor ID.

Effect of monadic ⎕*SVO*: None.

Effect of dyadic ⎕*SVO*: Each of the names (or surrogate names) in the right argument is offered to the processor (or processors) identified in the left argument. Reiterating an offer of the same name to the same partner is regarded as an inquiry similar to monadic ⎕*SVO*, and has no new effect. The system does not accept a new offer (that is, of the same name to a different partner) until you retract the first offer.

Making a shared variable offer implies that the active workspace can be uniquely identified. It must have a clone ID distinct from the clone ID in use by any other processor which has the same account number, is now running, and is making use of shared variables. If you need a distinct clone ID but haven't set one, the system rejects your use of dyadic ⎕*SVO* with the error message *IDENTIFICATION IN USE*.

While you have at least one name whose degree of coupling is at least 1, your clone ID is frozen. If you haven't explicitly set a clone ID, sharing a variable has the effect of freezing your clone ID at 0.

Result of *⎕SVO*: A numeric vector indicating the degree of coupling of each of the names (or name pairs) in the right argument. For dyadic *⎕SVO*, this is the degree of coupling produced by the offer. Coupling is reported according to the following code:

0 No coupling (or not a well-formed name)

1 Offer extended by one partner but not accepted

2 Reciprocal offers by both partners; sharing in effect

R←⎕SVS Y **State of variables named in** *Y*

Argument: A character array of names. In general, the argument is a matrix, with one row for each name. Where the argument contains a single name, it may be a vector, or, if the name contains only a single character, a scalar.

Each name may be stated alone (on a single row of the matrix), or paired with its surrogate. Where a name and its surrogate occur on a single row (or where the argument is a vector of two names), they must be separated by at least one blank. The first name is the name used within the workspace; the last name is the surrogate. A surrogate name is the name which your partner must offer instead of the name you use within your workspace. You don't need to include the surrogate name, but if you do, it must be correct. The system won't recognize a name even when it's spelled correctly if you pair it with an incorrect surrogate.

Result: A Boolean array. Where the argument was a matrix, the result is a 4-column matrix having as many rows as the argument. Where the argument was a vector or a scalar, the result is a 4-element vector. The result indicates the state of each of the names (or name pairs) in the right argument. A row of the result may have any of the following four possible values:

0 0 1 1 Both processors are aware of the current value of the shared variable.

1 0 1 0 You have set a new value of the shared variable, but your partner has not used it.

0 1 0 1 Your partner has set a new value for the shared variable, but you have not used it.

0 0 0 0 The corresponding name in the right argument does not identify a shared variable.

R←X ⎕SVC Y **Set control vector for names** *Y* **to** *X*
R← ⎕SVC Y **Report control vector for names** *Y*

Right argument: Same as for *⎕SVS*.

Left argument: When the function is used dyadically, the left argument is a Boolean array containing the proposed new values for control vectors matching each of the names appearing in the right argument.

In general, the left argument is a four-column matrix, containing as many rows as there are rows in the right argument. However, the left argument may contain only one access control vector, in which case it may appear either as a 1-by-4 matrix or as a 4-element vector. When the left argument contains only one vector, but the right argument contains several rows, the same control vector is applied to each of the names on the various rows. The left argument may also be a scalar or a 1-element vector. In that case, the system treats the argument as if it were a 4-element vector having the same value throughout.

Effect of monadic $\square SVC$**:** None.

Effect of dyadic $\square SVC$**:** For each of the names in the right argument that represents a currently visible shared variable, the control vector is reset. A 1 in each of the four positions of the control vector inhibits action as follows:

Set		**Use**	
□	□	□	□
You	**Your Partner**	**You**	**Your Partner**

The new setting is the value stated in the corresponding row of the left argument OR-ed with the value most recently set by the other partner. (The effect of this rule is that you cannot reset elements of the control vector which your partner has set to 1.)

Result: The current effective value of the control vector for each of the variables named. For dyadic use of $\square SVC$, that is the value following execution of the function. Where a name in the right argument is not a shared variable, or is not a well-formed name, the corresponding row of the result is 0 0 0 0.

$R\leftarrow\square SVR\ Y$ **Retract share-offer of names in** Y

Argument: Same as for $\square SVS$.

Effect: Where the visible referent of a name in Y is a shared variable, sharing is discontinued. This has no effect on the value of the shared variable. Retraction is recorded as a state change, and causes the system to set $\square SC$. This may be detected by the other partner. Following discontinuation of sharing, the variable's control vector and state vector are both reported as 0 0 0 0.

Where the visible use of the name is an offer which has not been accepted, the offer is withdrawn. Withdrawal of an unaccepted offer does not count as a state change, and has no effect on the name's control vector or state vector, which remain 0 0 0 0.

Where there is no visible use of the name, or it is not a well-formed name, there is no effect.

Result: The degree of coupling of the variable before the offer was withdrawn. This is recorded as 2 for a variable that was shared; 1 for a name which you had offered but which had not been accepted; and 0 for an item that had not been offered or was not a well-formed name.

R←□*SC* **State change variable**

This variable is automatically shared with the system. Any time you refer to it, the value you see is one set by the system. Its value may be a scalar `1` or `0`.

When your active workspace has not established a valid clone ID (by use of □*SVN*, or implicitly by use of □*SVQ* or dyadic □*SVO*) each time you use □*SC*, the system immediately assigns it the value `0`.

Once your workspace has established a valid clone ID, your use of the variable □*SC* is interlocked by the system. Thereafter, each time there is some change in the shared-variable state of your workspace, the system sets the value of □*SC* to `1`. **But if there has been no change since the last time you used** □*SC* **(or since you started using shared variables) the system delays supplying that value `1` until a state change occurs.** You'll experience an indefinite delay until the state change.

A change in the shared-variable state arises from any of the following:

1. When your partner retracts a variable you were sharing.

2. When a processor specifically directs a new offer to share to your active workspace.

3. When any of your shared variable partners takes an action that alters the result of □*SVS* or □*SVC* with respect to any variable you are now sharing. (Notice that a state change might be produced by something your partner does to a variable which is shadowed in your workspace, so that the variable is not now visible to you.)

You can interrupt the delay imposed during an attempt to use □*SC* by signalling an interrupt (two consecutive uses of BREAK or ATTN).

If you use □*SC* when you have not yet used the shared variable functions □*SVN*, □*SVQ*, or dyadic □*SVO*, you'll receive the value of □*SC* immediately.

Value of □*SC* **when used:** The value of □*SC* is always a scalar `1` when the workspace has a valid clone ID, and `0` otherwise..

Boon change variable

This variable is automatically shared with the system. Any time you write to it, the value you are to one set by the system. Its value may be a scalar 1 or 0.

When your active workspace has not established a valid clone ID (by use of [CID], or implicitly by use of [CHK] or dyadic [CFX]) each time you use [CC], the system immediately assigns it the value 0.

Once your workspace has established a valid clone ID, your use of the variable [CC] is intercepted by the system. Temporarily, each time there is some change in the shared-variable state of your workspace, the system sets the value of [CC] to 1. But if there has been no change since the last time you used [CC] (or since you started using shared variables), the system delays supplying that value (1 until a state change occurs. Your [CC] expression is indefinitely delay until the state change.

A change in the shared-variable state arises from any of the following:

1. When your partner retracts a variable you were sharing.

2. When a partner specifically shares a new value to share in your active workspace.

3. When any of your shared variable partners takes an action that alters the result of [CRS] or [CNL] with respect to any variable you are now sharing. (Notice that a state change might be produced by unsetting your partner does to a variable which is shadowed in your workspace, so that the variable is not now visible to you.)

You can interrupt the delay imposed during an attempt to use [CC] by signalling an interrupt (two consecutive uses of [BKSPACE] or [ATTN]).

If you use [CC] when you have not yet used the shared variable functions [CRS], [CNL], or dyadic [CFX], you'll receive the value of [CC] immediately.

Value of [CC] when unset. The value of [CC] is always a scalar 1 within the workspace has a valid clone ID, and a otherwise.

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

SATN-33
26 MAR 79

SHARP APL TECHNICAL NOTES

TITLE: Event Trapping

ABSTRACT: Two system variables and a system function are described which provide for program control of errors, interrupts and other events.

KEYWORDS: □*TRAP*
□*ER*
□*SIGNAL*

SHARP AR TECHNICAL NOTES

Introduction

The **American Heritage Dictionary** defines an **event** as 'an occurrence, incident, or experience, especially one of some significance'. Trapping an event bypasses the normal system response to the event and substitutes some other, user defined, response. Obviously event trapping does not provide for the trapping of all events, but rather for a small set of events which have particular significance. The events that are currently trappable are:

- execution errors (e.g. *SYNTAX ERROR* and *VALUE ERROR*),
- user defined execution errors,
- interrupts (e.g. *ATTENTION* and *INTERRUPT*), and
- return to immediate execution mode

Event trapping allows the programmer to specify an action to be performed when an error, interrupt, or other event occurs. If an event is not trapped, the default system action is performed, which is usually to return to immediate execution mode.

The event trapping facility consists of two system variables (⎕*TRAP* and ⎕*ER*) and one system function (⎕*SIGNAL*). The system variable ⎕*TRAP* contains user defined **trap definitions**. A trap definition defines the response to take when a specific event occurs. The response usually consists of two parts: a line of APL to be executed when the event occurs, and an action code describing how the line is to be put into execution. The system variable ⎕*ER* contains an **event report** about the last event that occurred. The report in ⎕*ER* usually contains essentially the same information that would be displayed on the terminal if the event were not trapped, and can be useful in programmatically determining the appropriate response. The system function ⎕*SIGNAL* signals an error to the next lower state indicator level. It can be used to signal both system defined errors (e.g. *RANK ERROR*) and user defined errors.

Benefits of Event Trapping

The benefits of event trapping fall into four main categories:

Improved user interface: - Without event trapping, an error or other event is handled in the system default manner. The system default assumes the user is a programmer, and reports the event in programming terms and expects a response in similar terms. With event trapping, the application remains in control and can either react automatically to the event, or can report it in terms the user understands, and can allow him to respond in terms he understands. Applications can now have a better, human engineered, interface with the user.

Reliability: - Applications can now conveniently and automatically handle events that previously would have caused the application to fail.

Efficiency: - Event trapping allows considerable simplification in designing and building systems along with a corresponding increase in execution efficiency. Most of the validation code used to detect errors in advance so they can be avoided is unnecessary when using event trapping. Rather than validating every transaction, recovery code is invoked only when an error has occurred. In addition, recovery code is usually easier to write.

A classic case is ⌹, where the validation necessary to avoid *DOMAIN ERROR* for singularity essentially requires a complete APL model of the primitive. In contrast, trapping the *DOMAIN ERROR* is efficient and simple.

Security: - In secure systems, allowing the user to gain control in order to respond to an error or other event violates security. Before event trapping, secure systems had to make use of the CLEAR-OUT facility which provides security at a very high cost in convenience. Event trapping provides levels of security without sacrificing convenience for the application user.

⎕*TRAP*

The system variable ⎕*TRAP* is set by the user and is used by the system. It is a character vector or matrix and contains 0 or more trap definitions. If it is a matrix then each row is a trap definition and if it is a vector then the trap definitions are delimited by occurrences of the first character in the vector.

A trap definition consists of TYPE, ACTION, and an optional LINE.

> **TYPE** is one or more integers separated by blanks indicating which events the trap definition applies to. Appendix A lists all the currently defined events and their numbers. A 0 in TYPE indicates the trap definition applies to all error events (i.e. 1 to 999) and a 1000 in TYPE indicates the trap definition applies to all interrupt events (i.e. 1001 to 1999).
>
> **ACTION** is a single letter code from the set *C D E N S*. It indicates how the trap definition is to be interpreted and in some cases determines how the LINE of APL is put into execution.

LINE is the line of APL that is used to resume execution when a trap is taken. When a trap is taken, execution is resumed much as if the user had entered the contents of LINE. If the ACTION is *D*, then LINE is interpreted specially as described in the section on the *D* ACTION.

For example,

```
⎕TRAP←'∇ 2 3 E SV ∇ 4 C →RANKE'
```

defines a *⎕TRAP* containing two trap definitions. The first trap definition has a TYPE of 2 3, an ACTION of *E*, and a LINE of *SV*; the second trap definition has a TYPE of 4, an ACTION of *C*, and a LINE of *→RANKE*.

ACTION CODES

The *C* (cutback) ACTION indicates the trap is to be taken and that, before executing the trap LINE, the state indicator is to be cut back to the level where the *⎕TRAP* containing the trap definition was localized. Hence, a trap LINE with a *C* ACTION is executed in the environment in which the trap was localized.

The *D* (do) ACTION indicates the trap is to be taken and that the system should do the procedure defined by the contents of LINE. *D* is the only valid ACTION for event type 2001 (return to immediate execution) and is not valid for any other event type. The only valid contents of LINE with an ACTION of *D* are *CLEAR*, *EXIT*, or empty. An empty LINE has no effect and allows the return to immediate execution. *CLEAR* causes the workspace to be cleared before the return to immediate execution. *EXIT*, if the state indicator is empty, allows the return to immediate execution. If the state indicator is not empty, the top level is removed, and other levels are removed until the new top level is a function level, and then the event of return to immediate execution is signalled again. The *CLEAR* option provides a replacement for the facility provided by the functions *CLEAROUT* and *NOCLEAR* from library 1 *WSFNS*. To preserve the security provided by *CLEAROUT* in existing systems, a *CLEAROUT* setting prevents the trapping of events. The *EXIT* option can be used to clear secure functions and localized data from the state indicator until it is safe to return control to the user.

The *E* (execute) ACTION indicates the trap is to be taken and that the trap LINE is to be executed. A trap LINE with an ACTION of *E* is executed in the environment in which the event occurred.

The *N* (next *⎕TRAP*) ACTION indicates the event is not to be trapped by the current *⎕TRAP* and that the search for a valid trap definition should continue in the next lower localized *⎕TRAP*. An ACTION of *N* would normally be used in conjunction with the TYPES of 0 (all events from 1 to 999) and 1000 (all events from 1001 to 1999) to exclude certain events. For example *'∇4 N ∇ 0 E RECOVERY'* contains two trap definitions and has the effect of trapping all errors except *RANK ERROR*.

The *S* (stop search) ACTION indicates the event is not to be trapped and should be handled in the system default manner. Like an ACTION of *N*, the *S* ACTION can be used in conjunction with other trap definitions for TYPES of 0 and 1000. *N* allows the search to continue in traps localized at lower levels, whereas *S* causes the search to stop.

THE SEARCH FOR A TRAP DEFINITION

When an event occurs, the system searches for a trap definition for that event. If a trap definition is found it is followed, otherwise the default system response is followed. The search in a vector *⎕TRAP* proceeds from the leftmost trap definition to the rightmost, and in a matrix *⎕TRAP* proceeds from the first row to the last. The system first searches the most local value of *⎕TRAP* and then searches in turn each shadowed value of *⎕TRAP* in the state indicator until the value of *⎕TRAP* in the global environment is searched. It is very important to understand that the system searches down the state indicator through all of the shadowed values of *⎕TRAP* looking for a trap definition for the event that has occurred.

⎕TRAP VALIDATION

⎕TRAP is a shared variable that is shared with the system. When the user sets a value into *⎕TRAP*, it is validated by the system. If it is a valid setting, it is left unchanged; but if it is invalid, the system sets *⎕TRAP* to be `''`. The following examples illustrate this behaviour:

```
      ⎕TRAP←'∇ 2 E SE' ◇ ⍴⎕TRAP
8

      ⎕TRAP←'∇2ESE' ◇ ⍴⎕TRAP
0
```

There are two reasons for validating *⎕TRAP* and setting it to `''` if it was invalid. The first is that we want to be able to add new trap capabilities and new events to the system, but are unsure of what new syntactic requirements there will be. With strict control over the contents of *⎕TRAP* we can ensure that extensions and enhancements will not conflict with old *⎕TRAP* values. The second reason is that if the system found an error in parsing a trap definition it would be awkward and inconvenient to report it, as this would conflict with reporting the original event.

Appendix B contains a statement of the rules for validating *⎕TRAP*.

ERROR EVENTS

Errors during execution of APL statements signal events in the range `1` to `999`. An error is caused when an APL statement is ill-formed, or when necessary environmental conditions are not met. Examples of errors are: *SYNTAX ERROR*, *VALUE ERROR*, *WS FULL*, *WS NOT FOUND*, and *FILE INDEX ERROR*. A current list of system defined error events and their numbers is in Appendix A. The system function *⎕SIGNAL* causes user defined error events in the same range of `1` to `999`. A trap definition for a type of `0` traps all system and user defined error events.

INTERRUPT EVENTS

Execution within a function, a statement, or a primitive can be interrupted by a request from the user. The user requests an interrupt by setting a stop control, by pressing the attention (break) key one or more times, or by entering *O* backspace *U* backspace *T* in response to a request for input. Depending on the kind of interrupt requested and the state of the system, execution can be interrupted with one of seven different interrupt events.

STOP is event number 1001 and is caused when the system finds that the next function line to be executed has a stop control set. Trapping this event can be a debugging aid, as the environment can be checked and recorded before starting the execution of selected function lines. The function interrupted by a STOP can be safely resumed as the interruption occurs at the beginning of a line.

ATTENTION is event number 1002 and is caused when the user has pressed the attention key one or more times and the system is about to start execution of a function line. Trapping this event allows the application to continue running in order to finish a 'critical section' and to remain in control for security and integrity reasons. The function interrupted by an ATTENTION can be safely resumed as the interruption occurs at the beginning of a line. Incorrect handling of ATTENTION trapping can easily result in an application that can only be stopped by bouncing the task.

INTERRUPT is event number 1003 and is caused when the user has pressed the attention key two or more times, and more than a reasonable number of CPU units has been consumed. Normally, the system would get to the start of a new line before the CPU limit expired, and an ATTENTION event would occur. However, if the line in execution takes more than the allowed amount of CPU, then an INTERRUPT event is caused. Note that an INTERRUPT occurs in the middle of a line (whereas ATTENTION and STOP occur at the beginning) and resuming execution may require special treatment.

INPUT INTERRUPT is event number 1004 and is caused when the user enters *O* backspace *U* backspace *T* in response to an input request, or when there is a transmission error in the input to *□ARBIN*.

SV INTERRUPT is event number 1005 and is caused when the attention key is pressed more than once and execution is suspended in a reference to a shared variable.

FILE INTERRUPT is event number 1006 and is caused when the attention key is pressed a third time while execution is suspended waiting for the file system to service a file request. This event indicates that the file system is running slowly, is not running properly, or that an interlock exists for the particular function requested (e.g. a read of a file is interlocked by a large drop on the same file by another user, or a hold is interlocked by the hold of another user).

FILE BACKUP INTERRUPT is event number 1007 and is identical to the FILE INTERRUPT event except that the fact that the system file backup is running (archiving of on-line data to magnetic tape) is the most probable cause of the slow response from the file system to the request.

Considerable care must be taken in designing and programming traps for interrupts. When a user requests an interrupt from his terminal he wants to regain control, and will be very frustrated if he is apparently completely ignored. Applications should be as responsive as possible to user interrupts. If not, the user will probably assume the system is broken and either abandon the terminal, or terminate the task by breaking the connection. Interrupt trapping should normally be used only to slightly postpone, until a point convenient and appropriate to the application, returning control to the user. In general, soon after the trap is taken, the application should let the user know that his request for control has been seen and that it will eventually be honoured. With incorrect handling of interrupts it is easy to get into a loop that can only be broken by terminating the task.

Execution in a task must stop before the task can be terminated. Line disconnect, □*BOUNCE*, task connect limit expiry, task CPU limit expiry, and all other requests for a bounce of a task, request an interrupt (as if attention were pressed three times) to get execution to stop. Errors and interrupts are not trappable if a bounce of the task has been requested.

RETURN TO IMMEDIATE EXECUTION

Event number 2001 is the event caused by a return to immediate execution mode. This event, and the ability to respond automatically, is primarily provided as a security mechanism. It provides a simple and guaranteed way of ensuring that secure data and functions are cleared from the workspace or the state indicator before control is returned to the user. In many applications it is the trap of 'last resort'. That is, if trapping of another event has failed, or an untrappable error or interrupt has occurred, then before relinquishing control to the user in immediate execution mode, a check for a trap for event 2001 is made. If a trap definition exists for event number 2001, then appropriate cleanup is done. The cleanup can leave the workspace as is, can clear it entirely, or can remove levels from the state indicator that contain information that must remain secure. A detailed description of the possible trap definitions for event number 2001 is provided in the earlier section on the *D* ACTION.

□*ER* is not set for event number 2001.

ERRORS AND INTERRUPTS THAT CANNOT BE TRAPPED

An error or interrupt occurring in the execution of a trap line cannot be trapped. This makes it easier to debug traps and makes it impossible to get into an uninterruptible loop where an event in the trap line simply reinvokes the same trap line. An error or interrupt occurring in functions called by the trap line and an error in the trap line caused by ⎕*SIGNAL* from a function can be trapped.

An error or interrupt occurring when CLEAROUT is set, is not trapped.

An error or interrupt occurring when a bounce of the task has been requested, is not trapped.

An INTERRUPT (event 1003) occurring when a CPU LIMIT has been set, is not trapped.

There are currently five errors reported by APL that are not errors in execution, but rather are errors detected in preparing the line for execution. These five errors are not trappable events. They can occur both for an input line from a terminal and from a trap definition. The five errors are:

OPEN QUOTE indicates there are unbalanced quotes in the line.

CHAR ERROR indicates there is an invalid character in the line.

WS FULL ERROR indicates there is insufficient space to prepare the line for execution. Note that this is distinct from *WS FULL* which is an execution error.

SYMBOL TABLE FULL ERROR indicates there is insufficient space in the symbol table to prepare the line for execution. Note that this is distinct from *SYMBOL TABLE FULL* which is an execution error.

DEFN ERROR indicates the line contains improper use of the ∇ character.

⎕*TRAP* EXAMPLES

For the vector form, the first character specifies the delimiter. Any character can be used, but all the examples here use ∇ as a convenient and mnemonic delimiter. Examples will be given in the vector form, but the matrix form is quite simple. For example:

```
'∇ 2 E SYNT ∇ 6 E VAL'
```

is equivalent to

```
2 8⍴'2 E SYNT6 E VAL '
```

The examples are incomplete and are hardly exhaustive. In fact, they barely scratch the surface. A facility like event trapping lends itself to experimentation from the terminal and is certainly best understood through use.

```
⎕TRAP←'∇ 11 E DOMER'
```

This trap would execute the function *DOMER* in the environment in which a *DOMAIN ERROR* occurred.

```
⎕TRAP←'∇ 11 C DOMER'
```

When a *DOMAIN ERROR* occurred, the state indicator would be cut back to the level which localized this ⎕*TRAP*, and then the function *DOMER* would be executed.

```
⎕TRAP←'∇ 2 6 E ERA ∇ 4 5 E ERB'
```

SYNTAX ERROR and *VALUE ERROR* would cause *ERA* to be executed, and *RANK ERROR* and *LENGTH ERROR* would cause *ERB* to be executed.

```
⎕TRAP←'∇ 0 S'
```

This trap would prevent any errors from being trapped by a ⎕*TRAP* at a lower state indicator level, hence causing them to return control to the user in the normal manner. This could be useful in debugging.

```
⎕TRAP←'∇ 1 15 N ∇ 0 E ERRS'
```

WS FULL and *SYMBOL TABLE FULL* errors would not be trapped by this ⎕*TRAP*, but all other errors would cause the function *ERRS* to be executed. *WS FULL* and *SYMBOL TABLE FULL* could return control to the user or could be trapped by a ⎕*TRAP* lower on the state indicator.

```
⎕TRAP←'∇ 0 E ERS ∇ 1000 E INTS ∇ 2001 D CLEAR'
```

All trappable errors would result in executing *ERS*, all trappable interrupts would result in executing *INTS*, and any attempt to return control to the user with a return to immediate execution would result in the workspace being cleared.

```
⎕TRAP←'∇ 2001 D EXIT'
```

Any error or interrupt not trapped by another trap would be reported to the user, but the state indicator would be cut back before the user was given control in immediate execution.

```
    ∇ Z←L DEX R; ⎕TRAP; ⎕ER
[1]   →(' '∧.=,R)/LL
[2]   ⎕TRAP←'∇0 C ⎕TRAP←⍳0 ◇ →RE'
[3]   Z←⍎R
[4]   →0
[5]  RE: →((6=1↑⎕FI,⎕ER)∧'DEX[3] '∧.=⎕ER[1+⎕IO;⍳7])/0
[6]  LL: ⎕TRAP←'∇0 C ⎕TRAP←⍳0 ◇ →LE'
[7]   Z←⍎L
[8]   →0
[9]  LE: →((6=1↑⎕FI,⎕ER)∧'DEX[7] '∧.=⎕ER[1+⎕IO;⍳7])/0
[10]  ⎕ER
[11]  ⎕SIGNAL 11
    ∇
```

For many error trapping problems, the function *DEX* provides a simple and powerful solution. It is a somewhat simplified model of a proposed dyadic form of ⍎. If the right argument is not empty, it is executed; and if there is no error, its result is the result of *DEX*. If there is an error, then the left argument is executed, and its result is the result of *DEX*. An error in executing the left argument is reported with ⎕*SIGNAL*. If the right argument is empty, the left argument is executed. The main complication in the function, at label *RE* and again at *LE*, is to distinguish errors in executing the expression from the value error caused by the expression not having a result. Resetting ⎕*TRAP* to be empty in the trap LINE is a very useful technique as it can prevent trapping errors in the code handling the original event.

⎕*ER* - *E*vent *R*eport

The system variable ⎕*ER* is set by the system and is used by the user. It is set by the system to contain information about the event when an error or interrupt occurs. It is set as a character matrix with three rows, containing essentially the same information that is normally displayed on the terminal when an error or interrupt occurs. The first row contains the event number and the event name. The event number is right justified in four positions and is separated from the event name by a blank. The second row contains the line display and the third row contains a caret under the location of the error or interrupt in the line display. The line display of a locked function contains a ▼ and the caret points at it.

If there is insufficient space to build a complete ⎕*ER*, it is truncated to be a 1 by 4 matrix containing only the event number. The default ⎕*ER* is `3 0⍴''`.

The information in ⎕*ER* can be useful in analysing trapped events and in analysing N-task and B-task failures (its contents in the continue workspace describes the error that terminated the task).

WARNING: It may be desirable and necessary in the future to change the format of ⎕*ER*. Such change might, for example, involve the addition of new information or improved caret position. Applications which analyse and manipulate ⎕*ER* should be designed to minimize the impact of changes in the format of ⎕*ER*.

⎕*ER* EXAMPLES

```
    4 RANK ERROR
      ↑↑4
      ^


   11 DOMAIN ERROR
FOO[6] ÷0
       ^


    6 VALUE ERROR
PROP[5] ⍒
        ^


1002 ATTENTION
FOO[1] →1
       ^


1003 INTERRUPT
GOO[8] R←A⌹B
          ^
```

⎕*SIGNAL*

The system function ⎕*SIGNAL* allows a user defined function to cause an error event in a manner similar to a primitive function. Executing ⎕*SIGNAL* causes an error in the next lower state indicator level. If the state indicator is empty, then the error is caused in the global environment. A ⎕*SIGNAL* executed with ⍎ causes an error in the environment the ⍎ is in (hence a function can signal an error to itself by using ⍎).

⎕*SIGNAL* is a divalent system function. The right argument must be a numeric scalar or vector. If the right argument is an empty vector, then no error is signalled and execution continues normally. The first element of the right argument is the error event that is signalled. If the first element is not a positive integer less than 1000, then a *DOMAIN ERROR* is caused in the execution of ⎕*SIGNAL*.

The left argument, if present, must be a character scalar or vector of length less than 256. The left argument is used as the event name for display on the terminal and for ⎕*ER*. If the left argument is not provided then the default system event name is used (see Appendix A).

The system will never cause an error event with a number greater than 499. User defined error events should generally be 500 or greater.

□*SIGNAL* EXAMPLES

```
    ∇ R←A F B
      .....
      'RIGHT ARG NOT NUMERIC' □SIGNAL (0≠1↑0ρB)/500

      ...
      'LEFT ARG NOT CHARACTER' □SIGNAL (0=1↑0ρA)/505
      ...

      □SIGNAL (2≠ρρB)/4
    ∇


      3 F 'A'
RIGHT ARG NOT NUMERIC
      3 F 'A'
        ^


      3 F 3
LEFT ARG NOT CHARACTER
      3 F 3
        ^


      'A' F 3
RANK ERROR
      'A' F 3
          ^
```

The function *ERRORS* returns as its result a matrix of all error events.

```
    ∇ R←ERRORS; □TRAP; □ER; I; ERT
[1]   I←0
[2]   R← 0 0 ρ''
[3]  L: □TRAP←'∇ 500 C →0 ∇ 0 C □TRAP←⍳0◇→T'
[4]   ⍝'□SIGNAL I←I+1'
[5]  T: ERT←((' 'V.≠4↓□ER[□IO;]),1↓(ρR)⌈ρ□ER)↑□ER[,□IO;]
[6]   R←(((1↑ρR),1↓ρERT)↑R),[□IO] ERT
[7]   →L
    ∇
```

The following will display all error events:

```
    ∇ F
[1]   ((' 'V.≠4↓,□ER[□IO;]),1↓ρ□ER)ρ□ER[□IO;]
[2]   □SIGNAL 1+□FI,□ER
    ∇
```

```
      ⎕TRAP←'∇ 500 C ∇ 0 C F'

      ⍳1E9
    1 WS FULL
    2 SYNTAX ERROR
    3 INDEX ERROR
.....
```

APPENDIX A

Error Events:

```
  1 WS FULL
  2 SYNTAX ERROR
  3 INDEX ERROR
  4 RANK ERROR
  5 LENGTH ERROR
  6 VALUE ERROR
  7 FORMAT ERROR
 11 DOMAIN ERROR
 15 SYMBOL TABLE FULL
 16 NONCE ERROR
 18 FILE TIE ERROR
 19 FILE ACCESS ERROR
 20 FILE INDEX ERROR
 21 FILE FULL
 22 FILE NAME ERROR
 23 FILE DAMAGED
 24 FILE TIED
 26 FILE SYSTEM ERROR
 28 FILE SYSTEM NOT AVAILABLE
 30 FILE SYSTEM TIES USED UP
 31 FILE TIE QUOTA USED UP
 32 FILE QUOTA USED UP
 33 FILE RESERVATION ERROR
 34 FILE SYSTEM NO SPACE
 38 FILE COMPONENT DAMAGED
 45 WS LOCKED
 46 WS NOT FOUND
 72 INTERFACE QUOTA EXHAUSTED
 73 NO SHARES
 74 INTERFACE CAPACITY EXCEEDED
 75 SHARE TABLE FULL
 76 PROCESSOR TABLE FULL
 77 IDENTIFICATION IN USE
 78 □PP ERROR
 79 □PW ERROR
 80 □IO ERROR
 81 □RL ERROR
 82 □CT ERROR
 83 □HT ERROR
```

Interrupt Events:

```
1001  STOP
1002  ATTENTION
1003  INTERRUPT
1004  INPUT INTERRUPT
1005  SV INTERRUPT
1006  FILE INTERRUPT
1007  FILE BACKUP INTERRUPT
```

Other Events:

```
2001  RETURN TO IMMEDIATE EXECUTION
```

APPENDIX B

A value is invalid for ⎕*TRAP* if:

1. it is not a vector or a matrix
2. it contains non-character data
3. it contains an invalid trap definition

A trap definition is invalid if:

1. it is non-empty and contains no number in TYPE.

2. it contains an invalid number in TYPE. Valid numbers can only be formed by the digits `0` through `9` and must be followed by a blank.

3. it contains a number in TYPE that is not an element of the vector
```
2001 0, +\1999⍴1
```

4. ACTION is not *C D E N* or *S*

5. ACTION is not *D* and TYPE contains `2001`

6. ACTION is *D* and TYPE contains any number other than `2001`

7. ACTION *D* not followed by empty line, *CLEAR*, or *EXIT*

8. ACTION *N* or *S* followed by non-blanks

9. LINE contains more than 500 characters, ignoring leading and trailing blanks.

10. LINE begins with `∇`, `⍫`, `)` or `[`

A trap definition is valid if it is empty (i.e. no non-blanks between delimiters in the vector form, or in a row in the matrix form). An empty vector or matrix is a valid value for ⎕*TRAP*.

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

SATN 34
10 SEP 80

SHARP APL TECHNICAL NOTES

TITLE: **REPLICATION**

ABSTRACT: The domain of the left argument of the slash operator has been extended to non-negative integers. The derived monadic function is called replication and is a consistent extension to compression.

Author: Bob Bernecky

U.K. Sharp Corporation
Thorp Road
Newton Aycliffe
Co. Durham DL5 4EX

SHARP APL TECHNICAL NOTES

Slash (/)

Like any monadic operator, the slash takes as its argument the object just to the left of it. When the argument of slash is a function, the resulting derived function is one of the family of monadic functions called **reductions**, (such as +/ or ∧/ or ⌈/ and so on). When the left argument of slash is a boolean array, the resulting derived function is one of the family of monadic functions called **compressions**. A compression is a kind of selection. It either selects or suppresses each of the sub-arrays from the right argument.

The SHARP APL system now provides an extended version of the slash operator. All existing uses of compression work just as they used to, but some new capabilities have been added. The derived function produced by the extension is called **replication** and is said to **replicate** its argument. Replication may be defined on vectors as follows:

```
      REPLICATE: ((1↑⍺)⍴1↑⍵),(1↓⍺) REPLICATE 1↓⍵ : 0=⍴⍵ : ⍵
```

In the extended definition, the left argument of slash may contain any non-negative integers. Replication works in a manner similar to compression; for an element J of the left argument, the result contains J replicas of the corresponding sub-array of the right. For example:

```
      (⍳4)/⍳4
1 2 2 3 3 3 4 4 4 4

      3⌿1 2⍴'HO'
HO
HO
HO

      2/'ABC'
AABBCC

      1 0 2 3/12
12 12 12 12 12 12
```

Replication is a consistent extension of compression, since a boolean left argument selects either 1 or 0 replicas. Moreover, all the variant forms that apply to compression apply equally to replication, including scalar or one-element vector left and right arguments, use of the axis operator, and application to right arguments of rank greater than 1. For example:

```
      3/4 1⍴⍳4
1 1 1
2 2 2
3 3 3
4 4 4
```

SOME APPLICATIONS OF REPLICATION

Replication provides a simple and efficient solution to several common programming problems.

Add a one-column matrix *Y* to each column of a matrix *M*:

Without replication:
```
M←⍉(⍴⍉M)⍴Y
```

With replication:
```
M←(¯1↑⍴M)/Y
```

Condensed representation of data containing runs: A time series can be represented by a vector containing an element for each time period:

```
VF←3 3 5 6 6 6 6 3 7 7 3 5
```

If long unbroken runs of the same value occur in *VF*, it can be represented more compactly by the single value at the beginning of each run, and by a vector of number of occurrences:

```
VV←3 5 6 3 7 3 5
N←2 1 4 1 2 1 1
```

Replication gives the vector *VF* in terms of *N* and *VV* as follows:

```
VF ↔ N/VV
```

Variants of the index generator: Replication with a constant left argument is frequently useful in constructing variants of the index generator. The following variant of a function suggested by Joey Tuttle [1] generates the coordinates of a rectangular spiral in the X-Y plane:

```
SPIR: R,[1.5] 0,¯1↓R←0,(4/⍳⍵)×(4×⍵)⍴1 1 ¯1 ¯1
```

Avoiding Expansion: Some applications of expansion may be replaced by simpler expressions using replication. The following example replaces each occurrence of the first element of ω within the vector ω by α replicas:

```
EFC: (1,⍺)[1+⍵=1⍴⍵]/⍵


      3 EFC '∘THIS∘IS∘IT'
∘∘∘THIS∘∘∘IS∘∘∘IT
```

Typical uses include double- (or n-) spacing of text, and expanding a vector before performing indexed assign into it, as might be done in a string replacement.

Each Index Generator: This catenates the results of iota applied to each element of its argument:

```
      STEP: (⍳+/⍵)-⍵/+\0,¯1+⍵

      STEP 3 4 2
1 2 3 1 2 3 4 1 2
```

One use of *STEP* is to draw all connections among a set of points:

```
      CAP: Z,[.5](STEP⌽⍳⍵)+Z←(⌽⍳⍵)/⍵←⍵-1

      CAP 5
1 1 1 1 2 2 2 3 3 4
2 3 4 5 3 4 5 4 5 5
```

Building Expansion Masks: Replication simplifies generation of masks needed for expansion. For example, if *LL* is a vector of lengths of lines of text stored as a vector, and we wish to turn the text vector into a matrix, the appropriate expansion mask is commonly produced as follows:

```
      XMOP: ,⍵∘.≥⍳⌈/0,⍵

      XMOP LL←3 2 6
1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1
```

Replication may be used to get the same result. Jerry Cudeck's expand mask generator [2], *EXPA*, takes a left argument of lengths of the existing items, and a right argument of the number of fill items to be inserted after each of them. *EXPB* is similar, but inserts the fill before each item.

```
      EXPA: V/(⍴V←,⍺,[1.5]⍵)⍴ 1 0
      EXPB: V/(⍴V←,⍵,[1.5]⍺)⍴ 0 1


      LL EXPA (⌈/LL)-LL
1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1

      LL EXPB (⌈/LL)-LL
0 0 0 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1
```

Paul Berry [3] suggested the following generalization of *EXPA* and *EXPB*, in which the sign of each element of the right argument indicates whether the fill goes before (negative) or after (positive) each item:

```
      EXP: (,|(⍵<0)⌽⍺,[1.5]⍵)/,(⍵<0)⌽((⍴⍵),2)⍴1 0

      LL EXP 1 ¯1 1 ×(⌈/LL)-LL
1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
```

Linear Interpolation: The *LINSERT* function [2] allows the user to specify a fixed or varying number of equidistant points to be interpolated between each pair of adjacent elements of a vector. Note how the scalar left argument of replication is used to advantage in the first example:

```
      LINSERT: +\(1⍴⍵),⍺/((1↓⍵)-¯1↓⍵)÷⍺

      4 LINSERT 1 2 6 4
1 1.25 1.5 1.75 2 3 4 5 6 5.5 5 4.5 4

      2 1 4 LINSERT 1 2 6 4
1 1.5 2 6 5.5 5 4.5 4
```

Graphics: Sharp APL Graphics [4] represents a collection of disparate line drawings (objects) as a matrix. The first column is control information such as pen up, pen down, pen colour, new object; trailing columns are the cartesian coordinates of the endpoints of each line, (XY for 2-dimensions, XYZ for 3-D, and so on for higher spaces). Graphic translation (moving an object from where it is to where you want it) involves adding the distance to be moved along each cartesian axis to each point in the object. Scaling (making an object larger or smaller) is similar in that it involves multiplying each point by some scale factor.

Replication may be used to advantage for these primitive graphics functions. Assume that *PIX* is a collection of three objects containing 4, 3, and 5 points respectively. Delta is the distance to move each object:

```
      DELTA←3 2⍴5 4  3 2  6 1
```

The distances to move each point are given by:

```
      4 3 5⌿DELTA
5 4
5 4
5 4
5 4
3 2
3 2
3 2
6 1
6 1
6 1
6 1
6 1
```

The translation is performed by *PIX*+0, 4 3 5⌿*DELTA*, and scaling by the same amount would be performed by *PIX*×1, 4 3 5⌿*DELTA*.

Bibliography:

[1] J.K. Tuttle, I.P. Sharp Associates Limited. Private communication.
[2] Jerry Cudeck, I.P. Sharp Associates Limited. Private communication.
[3] Paul Berry, I.P. Sharp Associates Limited. Private communication.
[4] Bernecky, B., Moore, R.D., Wooster P.K.; **Sharp APL Graphics**, I.P. Sharp Associates, 1977.

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

SATN-35
15 SEP 80

SHARP APL TECHNICAL NOTES

TITLE: **EXTENDED UPGRADE AND DOWNGRADE**

ABSTRACT: Monadic Upgrade and Downgrade have been extended to arrays of rank greater than 1, producing a permutation vector which may be used to sort the array along the first axis.

Dyadic Upgrade and Downgrade solve the general problem of sorting alphabetic lists involving multiple type fonts as well as upper- and lower-case alphabets. This implementation is based upon the definition proposed by Howard J. Smith, Jr., in the Proceedings of APL79, pp.123-127.

Author: Peter Wooster

U.S. Sharp Association
345 King Street West
Toronto, Ontario M6K 1Z9
(416) 924-5281

DATE: 03
FILE: 327-70

SHARP API TECHNICAL NOTES

TITLE: CAT SEMEN: CHROMATIC and POLYMORPHISM

ABSTRACT: Sharp's Diagonal and Honeycomb Blue have created a need to assess if and where there is predatory competition in color which may be used to aid the Sharp Blue Hex axis.

Diagonal, Diagonal and Honeycomb solve the general problem of sorting alphabetic text including multiple sizes. Data is used to assess Cat Semen Chromatic Polymorphism is based upon the definition proposed by Howard J. Smith, Jr. in the Proceedings of APLIC 79, pp 123-132.

Author: Peter Watson

MONADIC GRADE

The domain of the grade functions (⍋ and ⍒), previously restricted to numeric vectors [2], has been extended to all numeric arrays of rank greater than 0; the result of ⍋*A* is a permutation vector *P* of shape *K*←''⍴⍴*A* which, applied in the leading index position of *A* (as in *A*[*P*;;;]) will bring the *K* sub-arrays of *A* into "ascending" order.

For a matrix *A*, "ascending" order means that the rows of *A*[⍋*A*;] are in ascending order on their base values in some sufficiently large base, such as *B*←1+⌈/,|*A*. Thus the values of the vector *B*⊥⍉*A*[⍋*A*;] are in ascending order. In other words, the ordering is "lexical", assigning greatest weight to the leading columns.

An argument *A* of rank greater than 2 is treated as if ravelled along all axes following the first, that is, ⍋*A* ↔ ⍋((1↑⍴*A*),×/1↓⍴*A*)⍴*A*. For a rank 3 array, this is equivalent to assigning greatest weight to the leading rows, and, within them, to the leading columns. Downgrade is extended analogously.

For example, in a clear workspace:

```
      ⎕←M←?4 5⍴9
2 7 5 5 2
1 7 7 9 4
5 8 1 1 5
7 1 4 1 4

      ⍋M
2 1 3 4

      ⎕←M[⍋M;]
1 7 7 9 4
2 7 5 5 2
5 8 1 1 5
7 1 4 1 4
```

```
      ⎕←T←?4 2 5⍴9
7 6 9 8 5
1 6 4 7 9

7 3 1 7 3
6 7 9 4 3

9 7 7 6 1
6 8 3 4 7

5 3 3 4 2
5 9 9 1 9

      ⍋T
4 2 1 3
```

In business applications it is often useful to sort using multiple keys. For example, the matrix *OWED* has columns for account number, amount owed, and number of days past due. To sort by amount owed within number of days, use ⍒*OWED*[;3 2] as follows:

```
   OWED                          OWED[⍒OWED[;3 2];]
 315628    1500      30          5000726     422     120
3959214    1282      60          1509725   13133      90
 271627   10075      30            13322    4523      90
1509725   13133      90          3959214    1282      60
5000726     422     120           812331    1200      60
  61526      68      30           271627   10075      30
  13322    4523      90           315628    1500      30
 812331    1200      60            61526      68      30
```

The extended monadic grade may be modelled by the following functions [3]:

```
MG: CS(⍳1↑⍴⍵),((1↑⍴⍵),×/1↓⍴⍵)⍴⍵

CS: CS 0 ¯1↓⍵[⍋⍵[;(''⍴⍨⍴⍵)-~⎕IO];]:1=1↓⍴⍵:,⍵
```

DYADIC GRADE

The dyadic upgrade function ⍋ is based on the definition proposed by Howard Smith [1]. The extension of the downgrade function ⍒ is based upon the definition of upgrade. Both apply to character arguments only.

SYNTAX

X⍋*Y* alphabet *X* upgrade of *Y*

X⍒*Y* alphabet *X* downgrade of *Y*

Left argument:	Any character array describing a collection of alphabets to be used in comparing characters in *Y*. Each axis of *X* corresponds to a level of comparison, with the last axis major.
Right argument:	Any non-scalar character array.
Result:	A permutation vector of length `1↑⍴Y`.

A vector left argument specifies the "alphabet" or "collating sequence" in the obvious way. Formally:

A⍋*B* ↔ ⍋ *A*⍳*B*

For example:

```
      A←' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
      NAMES                NAMES[A⍋NAMES;]
JONES                ABEL
BAKER                BAKER
JACOBS               JACOBS
JAMES                JAMES
ABEL                 JONES
```

The foregoing definition applies to right arguments of rank other than two, according to the extended definition of monadic ⍋.

If *A* is a matrix where two rows represent two fonts, then *A*⍋*B* grades *B* using the ordering of the fonts (as specified by the columns) as a secondary ordering **within** the ordering specified by the rows. We will illustrate this by the example used by Smith [1], but with uppercase italic substituted for the lower-case Roman, and with underscored italic substituted for capitals:

```
      AL←2 27⍴A,' ABCDEFGHIJKLMNOPQRSTUVWXYZ'

      AL
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ

      ⎕←A1←' ',,0 1↓AL
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ

      ⎕←A2←' ',,⍉0 1↓AL
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
```

T	T[AL⍋T;]	T[A1⍋T;]	T[A2⍋T;]
AMA	ACID	ACID	ACID
YMCA	ALUMINUM	ALUMINUM	ALUMINUM
TRUDGEN	AMA	AMA	AMA
TEKTITE	AMA	AMMONIA	AMMONIA
PI	AMA	EMBOLISM	AMA
STROKE	AMMONIA	PAVILION	AMA
PAVILION	DPD	PHOSPHATE	DPD
PIPING	EMBOLISM	PHOTOSYNTHESIS	EMBOLISM
RESPIRATION	NSPF	PI	NSPF
PUMP	PAVILION	PIPING	PAVILION
PHOTOSYNTHESIS	PH	PLUG	PHOSPHATE
UNDERWATER	PHILODENDRON	POOL	PHOTOSYNTHESIS
TSUNAMI	PHOSPHATE	POROSITY	PH
POOL	PHOTOSYNTHESIS	PUMP	PI
NSPF	PI	PH	PIPING
RECOVERY	PIPING	RECOVERY	PLUG
ALUMINUM	PLUG	RESPIRATION	POOL
EMBOLISM	POOL	STROKE	POROSITY
PLUG	POROSITY	TRUDGEN	PUMP
TSUNAMI	PUMP	TSUNAMI	PHILODENDRON
TRUDGEN	RECOVERY	UNDERWATER	RECOVERY
PH	RESPIRATION	AMA	RESPIRATION
POROSITY	STROKE	AMA	STROKE
PHOSPHATE	TEKTITE	DPD	TRUDGEN
DPD	TRUDGEN	NSPF	TSUNAMI
AMMONIA	TRUDGEN	PHILODENDRON	TEKTITE
AMA	TSUNAMI	TEKTITE	TRUDGEN
PHILODENDRON	TSUNAMI	TRUDGEN	TSUNAMI
ACID	UNDERWATER	TSUNAMI	UNDERWATER
AMA	YMCA	YMCA	YMCA

Comparison of the second column above with the sorting produced by vector left arguments in columns 3 and 4, makes the point (emphasized by Smith) concerning the useful versatility of the extended grade applied to left arguments of rank greater than 1.

For a vector left argument *A*, numeric sorting is applied to the numeric encoding *A*⍳*B*; for a matrix left argument *M* the encoding of each element of *B* is ⎕*IO*+(⍴*M*)⊤((,*M*)⍳*B*)-⎕*IO*. For example:

```
      ⎕IO+(⍴AL)⊤((,AL)⍳' ABCABC')-⎕IO
1 1 1 1 2 2 2
1 2 3 4 2 3 4
```

Grading is then applied to the highest order index **within** the next index (and so on if the rank of the left argument is greater than two).

Formally, the encoding *E*, and its use in defining a dyadic grade function *G*, may be stated as follows [3]:

```
G: MG(⍒0 1,1↓0×⍳⍴⍴ω)⌷(⊖(Eα),⍴α)[;(,α)⍳ω]

E: D⊤(⍳×/D←⍴ω)-⎕IO
```

In the function *G*, the vector ⍴α is appended to the encoding produced by *E* in order to take care of the case where the text contains characters not in the alphabet.

The foregoing definition does not cover the case of repeated occurrences of a character in the left argument; in that case, according to Smith, the encoding assigned is "... the minimum index of that character along each axis ..." The following redefinition of *E* covers this case:

```
E: ¯1+÷(÷1+D⊤(⍳×/D←⍴ω)-⎕IO)⌈.×A∘.=A←,ω
```

With this revised definition of *E*, the function *G* models the dyadic grade function. In particular, a scalar left argument produces an identity permutation.

SAMPLE ALPHABETS

Vector Alphabets

A vector alphabet specifies a collating sequence. Any characters not in the alphabet are treated as equal and after those in the alphabet.

Some common vector alphabets are:

1. Alphabetic order.

```
      A←' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      T←'JONES MET SMITH IN LONDON'

      T[A⍋T]
 DEEHIIMMNNNNOOOSTTJSL

      T[A⍒T]
JSLTTSOOONNNNMMIIHEED
```

Note the positions of the underbarred characters not included in the alphabet.

2. ⎕*AV*. ⎕*AV* contains all available characters and is useful when an arbitrary but specific order is required, as in comparing two tables expected to contain the same data or when finding duplicates in a single table. ⎕*AV* is system dependent.

```
   TAB          T←TAB[⎕AV⍋TAB;]      (1,1↓∨/T≠¯1⊖T)⌿T
ONE                  ALL                  ALL
RING                 ALL                  AND
TO                   AND                  BIND
RULE                 BIND                 BRING
THEM                 BRING                DARKNESS
ALL                  DARKNESS             FIND
ONE                  FIND                 IN
RING                 IN                   ONE
TO                   ONE                  RING
FIND                 ONE                  RULE
THEM                 ONE                  THEM
ONE                  RING                 THE
RING                 RING                 TO
TO                   RING
BRING                RULE
THEM                 THEM
ALL                  THEM
AND                  THEM
IN                   THEM
THE                  THE
DARKNESS             TO
BIND                 TO
THEM                 TO
```

Note that in Sharp APL **space** appears after the alphabet in ⎕*AV*.

3. Alphabetic order with underbarred letters after the corresponding non-underbarred letter. See alphabet *A2* in the section on dyadic grade.

4. Alphanumeric order with underbarred letters after non-underbarred letters. This is the order used by)*FNS*,)*VARS*,)*LIB*. It is similar to *A*1 above but includes Δ, ⍙, and the digits.

```
      CSQ4
 ABCDEFGHIJKLMNOPQRSTUVWXYZΔABCDEFGHIJKLMNOPQRSTUVWXYZ⍙0123456789

FNS←⎕NL 3                    FNS[CSQ4⍋FNS;]

BOOLSORT          BOOLSORT
TUG               BWFS
TYPE              FOO
FOO               FOO1
VTOM              FOO2
VTOM              GRADE
MONITOR           MONITOR
ΔDRAW             RCAT
GRADE             TEST
RCAT              TIMER
TIMER             TUG
BWFS              TYPE
MONITOR           VTOM
FOO2              ΔDRAW
TEST              MONITOR
⍙⍙                VTOM
FOO1              ⍙⍙
```

5. Hexadecimal order:

```
      CSQ5←'0123456789ABCDEF'
      HEX←CSQ5[⎕IO+⍉(5⍴16)⊤10* 3 1 2 0 5 1 4 3]

     HEX                HEX[CSQ5⍋HEX;]
003E8                00001
0000A                0000A
00064                0000A
00001                00064
186A0                003E8
0000A                003E8
02710                02710
003E8                186A0
```

Higher Rank Alphabets

Some common higher rank alphabets are:

6. Underbars within spelling. This is alphabet *AL* in the section on dyadic grade. Note the position of *TRUDGEN*.

7. Alphabetic order with underbars and non-underbars equal:

```
 ABCDEFGHIJKLMNOPQRTSUVWXYZ∆ABCDEFGHIJKLMNOPQRSTUVWXYZ∆
 ABCDEFGHIJKLMNOPQRSTUVWXYZ∆

DATA[CSQ7⍙DATA;]

BAR
BAR
BAR
BAR
BAR
BAT
BAT
BY
BY
CAB
CAB
CAB
CAR
CAR
CAR
CART
CART
CART
CAT
CAT
CAT
```

The following table shows how this equality is achieved:

```
    -----------
0|   A A B B |
1|   A   B   |
    -----------
     0 1 2 3 4
```

A is in position `0 1`, and *A* is in positions `1 1` and `0 2` so that its effective position is also `0 1`, since that is the result of `1 1⌊0 2`.

8. Numeric order with space equal to zero.

```
      CSQ8
 0123456789
0

NUMS     NUMS[CSQ8⍋NUMS;]

013              7
666            007
  7              7
015              8
007             12
 12            013
  7             13
  8            015
 13            666
```

9. If several fonts have been encoded as elements of ⎕*AV*, the user may wish to sort by case within font within spelling. This can be achieved by using a rank 3 alphabet, with the planes representing case, the rows representing font, and the columns spelling.

ABCDEFG.....Z	upper case,	bold ASCII
ABCDEFG.....Z		ASCII
ABCDEFG.....Z		APL (underbarred)
abcdefg.....z	lower case,	bold ASCII
abcdefg.....z		ASCII
ABCDEFG.....Z		APL

References:

[1] Smith, Howard J. Jr., Sorting,- A New/Old Problem: Proceedings, APL79 pp.123-127, [1979 ACM].

[2] Berry, Paul C., Sharp APL Reference Manual 1980, pp.158-159.

[3] Iverson, K.E., private communication.

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

SATN 36
October 1980

SHARP APL TECHNICAL NOTES

TITLE: Direct Definition

ABSTRACT: This note discusses the **direct definition** method of defining functions used in two earlier technical notes, SATN 34 and SATN 35.

Author: Kenneth E. Iverson

SHARP AR TECHNICAL NOTES

Direct Definition is a method of defining functions which is coming into wide use in APL programming and in documentation, as, for example, in two earlier technical notes, SATN 34 and SATN 35. The present technical note defines the method, provides functions to translate from direct to canonical form, and suggests further sources of information on the use of direct definition.

We will begin with some examples of the use of the translator:

```
      )LOAD 1 DIRECTDEF
SAVED  10.59.31 09/25/80

      DEFINE
PLUS:⍺+⍵

      3 PLUS 4
7
      ⎕CR'PLUS'
R←A PLUS W
R←A+W
⍝PLUS:⍺+⍵

      DEFINE
ID:X∘.=X←⍳⍵

      ID 3
1 0 0
0 1 0
0 0 1
      X
VALUE ERROR
      X
      ∧
      ⎕CR'ID'
R←ID W;X
R←X∘.=X←⍳W
⍝ID:X∘.=X←⍳⍵

      DEFINE
EQUALSHAPE:∧/(⍴⍺)=⍴⍵ : (⍴⍴⍺)≠⍴⍴⍵ : 0

      2 3 EQUALSHAPE 4 5 6
0
      ⎕CR'EQUALSHAPE'
R←A EQUALSHAPE W
→(0=1↑,(⍴⍴A)≠⍴⍴W)/3
→0,0⍴R←0
R←∧/(⍴A)=⍴W
⍝EQUALSHAPE:∧/(⍴⍺)=⍴⍵ : (⍴⍴⍺)≠⍴⍴⍵ : 0
```

In the definition of *EQUALSHAPE*, the second expression (following the second colon) is a **proposition**, or condition, that determines which of the other two is to be executed.

The function *SHOW* can be used to display the direct definitions of functions as follows:

```
      SHOW'ID'
ID:X∘.=X←⍳⍵
      SHOW'SHOW'
SHOW:M:'⍝'=,1 1↑⊖M←⎕CR⍵:1↓(⊖M)[⎕IO;]
      SHOW'DEFINE'
DEFINE:(∧/0=⎕FX CFD X)/'NOT DONE',0⍴⎕EX (∧\':'≠X)/X←⍞
```

The Function *CFD* (Canonical From Direct) used in the function *DEFINE* above is the heart of the translator. Because of the particular scheme used to separate the segments demarked by the colon, the canonical representation produced is somewhat dispersed. The behaviour of *CFD* is illustrated by the following case of a function for generating Fibonacci numbers:

```
      C←CFD 'FIB:Z,+/¯2↑Z←FIB ⍵-1:⍵=1:1'
      C
R←    FIB                              W          ;Z
→(0=1↑,                          W =1   )/3
→0,0⍴R←                             1
R←          Z,+/¯2↑Z←FIB  W-1
      ⍝FIB:Z,+/¯2↑Z←FIB ⍵-1:⍵=1:1

      ⎕FX C
FIB
      FIB 12
1 1 2 3 5 8 13 21 34 55 89 144

      ⎕CR 'FIB'
R←FIB W;Z
→(0=1↑,W=1)/3
→0,0⍴R←1
R←Z,+/¯2↑Z←FIB W-1
⍝FIB:Z,+/¯2↑Z←FIB ⍵-1:⍵=1:1
```

Formal Definition

The major aspects of direct definition have been illustrated in the foregoing examples, and they will now be summarized formally:

1) A function is represented by a character vector of two or four parts delimited by colons.

2) The first part is the name of the function to be defined.

3) The remaining parts are expressions in which ⍺ denotes the left argument of the function and ⍵ denotes the right. The valence of the function is therefore 2 if ⍺ occurs in any of the expressions, and is otherwise 1 or 0 according to whether ⍵ occurs or not.

4) The result of the function is the result of the last expression executed.

5) If there is more than one expression, the second is executed first, giving a result which determines which of the other two is to be executed -- the first if the result is zero, the last if the result is not zero.

6) Any name immediately to the left of an assignment arrow within any one of the expressions is localized. "Side-effects" are therefore prevented.

7) The symbols α, ω and : occurring within quotes do not act as formal arguments or separators.

Editing Functions

The following function provides the "slash and comma editing" familiar in the ∇ form of editing:

```
EDIT:EDIT(('/'≠K↑A)/K↑ω),(1↓K↓A),(K←+/∧\A≠',')↓ω:0=ρA←⍞,0ρ□←ω:ω
```

For example, one may use *EDIT* to produce a function *DEF* as a revision of the function *DEFINE* as follows:

```
      C←EDIT SHOW 'DEFINE'
DEFINE:(∧/0=□FX CFD X)/'NOT DONE',0ρ□EX (∧\':'≠X)/X←⍞
   ///                                                /,ω
DEF:(∧/0=□FX CFD X)/'NOT DONE',0ρ□EX (∧\':'≠X)/X←ω

      □FX CFD C
DEF
      DEF 'TIMES:α×ω'

      1 2 3 4 TIMES 4 3 2 1
4 6 6 4
```

The function *DEFINE* itself, of course, remains unaltered.

the functions *DEF*, *EDIT*, and *SHOW* can now be used to define a function convenient for revision:

```
      DEFINE
REVISE:DEF EDIT SHOW ⍞

      REVISE
ID
ID:X∘.=X←⍳ω
          ,⊥|
ID:X∘.=X←⍳⊥|ω

      ID ¯2.718
1 0
0 1
```

The function *CFD* and its auxiliary functions provide interesting examples of the use of direct definition. They appear below. Other examples may be found in the bibliography provided.

```
DEFINE:(∧/0=⎕FX CFD X)/'NOT DONE',0ρ⎕EX (∧\':'≠X)/X←⍞

 CFD:(SQZω) LCL ω FORM ω SPLIT 'αA' IN 'ωW' IN ω

  IN:(,(ωEQα[⎕IO])∘.≠4↑1)/,ω,(⍵3,ρω)ρ1⍵'  ',1↓α

  SPLIT:⍉D,(1↑ρD←(0,+/¯6,I)↓(-(3×I)++\I←ωEQ':')⍵ω,(⍵6,ρω)ρ' ')↑α

  FORM:3⍵FORMS[(2⌊2⊥(∨/αEQ'α'),∨/αEQ'ω'),(I+1),0 5;],ω[0,(I←2↓⍳F
      ),(1+⎕IO←0),F←¯1+1↑ρω;]

  LCL:ω,((1↑ρω),ρα)↑⍉0 ¯2↓(K+2×K<1⍵K←I∧K∈((1⍵αEQ'←')>αEQ'⎕')/K←+
      \~I←α∈ALPH)⍵' ',α,[⎕IO+.5]';'

  SQZ:,1 1↓⎕CR ⎕FX'Q',' ',[⎕IO-.5]ω,0ρ⎕EX'Q',0ρQ←0

    EQ:(α=ω)∧=\''''≠α
```

```
      ALPH                    FORMS
012345678                   R←
9ABCDEFGH                   WR←
IJKLMNOPQ                   WR←A
RSTUVWXYZ                )/3→(0=1↑,
ABCDEFGHI                   →0,0ρR←
JKLMNOPQR                        ⍴
STUVWXYZ⎕
```

Limitations of the Translator

One may not define functions with names which conflict with those used in the translator. These include *CFD*, and the underscored names used in the functions auxiliary to *CFD*, as well as the argument and result names *A*, *W*, and *R* used in the canonical representation produced. Briefly, one should avoid the use of underscored names.

The names used in the translator can be modified to replace these limitations by others. The alternative translator given in [4] (and expressed in canonical form) imposes only the restriction that names not terminate in the digit 9.

Bibliography

1. Iverson, K.E., **Elementary Analysis**, APL Press, 1976

2. Orth, D.L., **Calculus in a New Key**, APL Press, 1976

3. Mauldon, J.G., **Strong Solutions for the Counterfeit Coin Problem**, RC 7476, IBM Research Center, Yorktown Heights, N.Y., 1978

4. Iverson, K.E., Programming Style in APL, **An APL User's Meeting**, I.P. Sharp Associates, 1978, (pp. 200-221)

5. McIntyre, D.B., The Architectural Elegance of Crystals Made Clear by APL, **An APL User's Meeting**, I.P. Sharp Associates, 1978, (pp. 233-250)

6. McIntyre, D.B., Experience with Direct-Definition One-liners in Writing APL Applications, **An APL User's Meeting**, I.P. Sharp Associates, 1978, (pp 281-297)

7. Iverson, K.E., The Derivative Operator, **APL Quote-Quad**, Vol 9, No. 4, Part 1, ACM, 1979, (pp. 347-354)

8. McDonnell, E.E. and J.D. Shallit, Extending APL to Infinity, **APL 80**, North Holland Publishing Co., June 1980.

9. McDonnell, Articles in the following issues of **APL Quote-Quad (ACM)**: Vol. 7, No. 4; Vol. 8, Nos. 1, 2, 4; Vol. 9, Nos. 2, 4; Vol. 10, Nos. 1, 2.

I.P. Sharp Associates
2 First Canadian Place,
Suite 1900,
Toronto, Ontario M5X 1E3
(416) 364-5361
Telex 0622259

SATN-37
1 MAY 1982
Rev. 1

SHARP APL TECHNICAL NOTES

TITLE: IBM 3270 USER GUIDE

AUTHOR: John D. Burger

ABSTRACT: SHARP APL under MVS supports the IBM 3270 display stations through an auxiliary processor called SAPV (Sharp Auxiliary Processor for VTAM). This document contains a description of how the 3270 operates in both standard and full screen modes. A description of the functions (in workspace `5 IBM3270`) that provide full screen management support similar to IBM's AP124X, is included.

SHARP API TECHNICAL NOTES

INTRODUCTION

SHARP APL under MVS supports the IBM 3270 display stations through an auxiliary processor called SAPV (Sharp Auxiliary Processor for VTAM). SAPV uses the S-task facility to interface with SHARP APL and uses VTAM to support the IBM 3270's. When SAPV connects a 3270, it determines what features the device has and whether or not it has the APL character set.

The IBM 3270 device operates in two modes: standard screen and full screen. In standard screen mode, the device operates as a normal APL terminal with pre-defined characteristics. While in full screen mode, the screen is under the complete control of an APL application program.

This document provides a full description of how the device operates in both modes. Included is a description of workspace `5 IBM3270` which contains functions that provide full screen management support similar to IBM's AP124X. For an in-depth description of 3270s, see the IBM manual GA27-2749, IBM 3270 **Information Display System Component Description**.

NOTE: Origin 0 is used throughout this document.

STANDARD SCREEN MODE

Screen Format

While in standard screen mode, the IBM 3270 display is divided into three areas.

input area	-	The bottom 3 lines in which the user may input and edit
status line	-	A single line immediately above the input area. Page number and status information is displayed on this line.
output area	-	The remainder of the screen. Entered input and APL output is displayed here.

1) Input area

Input in this area may be erased by using the 'erase input' key. Null characters initially fill this area, but are replaced when a prompt is issued or as the user types (including blanks). Moving the cursor with the cursor positioning keys does not overwrite null characters, so the space bar should be used when blank characters are needed. Text in the input may be edited (see the section on special keys) or replaced by typing over it.

2) Status line

The status line separates the input and output areas. It displays the current status, a page number and perhaps the letter *N*, if no-autopaging has been selected (explained later). Page numbers are from 1-9999 and indicate which page from the page-file is being displayed (the number of pages available can be set at SAPV startup time). Page numbers are cyclical. Device status is a single letter (one of *I*, *R*, *S*, *K*) with the following meaning:

I	-	input	-	APL is awaiting input from the user.
R	-	run	-	APL is processing the last input.
S	-	select	-	The user can communicate with SAPV rather than APL. In particular, the user can examine screen images from the page-file or control autopaging.
K	-	keyboard locked	-	APL is awaiting input, but the keyboard is locked (due to *)KEYB LOCK*) so that messages from the system or other users can be displayed without delay.

3) Output area

All APL output and user's input is displayed here. The output area is filled from top to bottom until the screen is full. When full (and autopaging is set), the screen clears, the page number advances and output resumes. Previous images are saved in a page-file so that they may be recalled later (see the section on Page Recall). The screen may be cleared manually by using the clear key, or under program control (explained in the Full Screen section), which also advances the page number.

NOTE

When first connecting to SAPV, if SAPV cannot determine the character set of the device, a 'menu' is displayed in the input area, which reads:

PLEASE ENTER THE NUMBER OF ENTRY THAT BEST DESCRIBES YOUR TERMINAL
1. UPPER CASE ONLY 2. LOWER AND UPPER CASE
3. APL3277 4. APL3278 5. APL3229

Entering a number at this time informs SAPV of the character set to use and allows the connection to continue.

Standard Screen Input and Editing

1) Protected Areas

Input and editing always take place in the input area, so the output area can never be modified by the user. When APL is executing (*R* on the status line), the entire screen is protected. During selection (*S* on the status line), the input and output areas are protected while information on the status line may be altered (see the section on Page Recall). Typing in a protected area causes the keyboard to lock. Pressing the reset key unlocks the keyboard to allow the user to relocate the cursor into a non-protected area.

2) Character sets

APL characters may be entered by the user only from a device with the APL option and keyboard. A terminal with the APL option can be in either APL or non-APL mode. Normally it should be used in APL mode. Some APL characters, available by overstriking on some terminals, are available directly on the IBM 3270 APL keyboard. These characters (on the key fronts) are typed by holding the ALT key and then pressing the key containing the desired character.

An IBM 3270 without APL characters may also be used. Use the right parenthesis found in the standard character set to sign on and for system commands. In addition, any available character found in the APL character set may be used. On a non-APL 3270, the 'UPPER CASE ONLY' character set maps both lower and upper case alphabetics to APL *A* through *Z*, and the 'LOWER AND UPPER CASE' character set maps lower case alphabetics to APL *A* through *Z*, and the upper case alphabetics to APL *A* through *Z*. Unrecognized characters result in 'entry error'.

SHARP APL has introduced four new symbols that are not available directly on the 3270 keyboard. These symbols have been translated to available 3270 characters. On an APL terminal it is necessary to temporarily switch to non-APL mode to enter these special characters.

SHARP APL	IBM 3270	Name
≡	%	match
⍤	@	on
⍥	&	over
◇	#	diamond

Illegal characters appear as a solid block on an APL 3270, and as double quotes on a non-APL 3270.

3) Special keys

At sign-on, the PF (program function) keys are assigned an initial definition as set by the installation. These definitions may be read or altered using the local system command)*DPFK* (see the section on local system commands) or with the functions in workspace `5 IBM3270` (see the description in the Full Screen Management section). Typing)*DPFK* displays the current PFK definitions. Input area management is facilitated by the cursor positioning keys, forward and back tab key, new line key, and edit keys. Cursor positioning keys move the cursor up, down, backward or forward (on certain terminals, backward or forward rapidly if the ALT key is held). Tabbing backward or forward moves the cursor to the first or next position of an input field respectively. The new line key moves the cursor to the first position of the first input field on the next line. Text may be inserted or deleted at the cursor position by using the insert or delete key. Attention is signalled using the PA1 key (non-SNA terminals), or the ATTN key (SNA terminals), and PA2 puts the device into selection mode (*S* on the status line). At the end of this chapter is a summary of each special key and its function in each mode.

4) Line recall

Input area data may have been entered by a user, or placed in the input area as a prompt. Any line of any available page's output area may be recalled by positioning the cursor at that line and pressing the enter key. Recalling a line in this way results in the replacement of any existing input area data with the recalled data. The recalled data then becomes the new prompt and may be edited as usual.

5) Page Recall and Control

PA2 causes the device to enter selection mode. While in *S* mode, the device may be paged backward using PF1 or forward using PF2. To recall a page by page number, type that page number over the existing page number, then press ENTER. If the entered page number is less than the lowest available page number, the first available page is displayed. Pressing the break key (PA1 for non-SNA and ATTN for SNA terminals), or entering too high a page number, or pressing ENTER without changing the page number, causes the device to return to the previous mode.

When output fills the screen, the screen clears for more output, which is called autopaging. To avoid automatic paging, the user may enter the letter *N* in place of the page number while in *S* mode. While non-autopaging is in effect, the letter *N* is displayed beside the page number. Without autopaging, the device shifts into *S* mode when the screen fills, so that the user may then select a page number or press enter to display the next screen's output. Optionally, while in *S* mode, the user may re-enable

autopaging by entering the letter *A* in place of the page number and pressing ENTER. The screen will then refresh when filled.

6) Program Function Keys

When a program function key is pressed, its text is inserted at the current cursor position and the resulting text becomes the new prompt (if type *N*), or is entered (if type *E*). Note that, in standard screen mode, PF keys are usable only while the cursor is in the input area. Text is assigned to the program function keys by the local system command *)DPFK* (see the section on Local System Commands) or by the function *WRITEPFK* in workspace `5 IBM3270` (described in the Full Screen section).

7) Local System Commands

Several local system commands are available for the IBM 3270 (these system commands are processed by SAPV rather than APL). The PFK group allow the user to program, read and execute program function keys. There are up to 24 logical PFK's which can be referenced even if no key is physically present. Below is a summary of the PFK local system commands.

)DPFK - Display Program Function Keys

Display the definition of all program function keys.

)DPFK K - Display Program Function Key

Display the definition of program function key number *K*.

)DPFK K T D - Define Program Function Key

K is the key number, *T* is the type and *D* is the definition.

T is one of *E* (ENTER) or *N* (NO ENTER). When *E* is chosen, the text is entered as if the user had pressed the enter key. *N* causes the PFK's text to remain in the input area where it can be edited.

D is any expression or text. The text is inserted at the cursor position and the resulting new prompt is entered (type *E*) or not entered (type *N*).

)PFK K - Execute Program Function Key

This command is equivalent to the user pressing program function key *K* except that the text is not inserted into the existing prompt. Any of the 24 PFK's may be executed with this command.

Special Key Summary for Each Standard Screen Mode

Mode	Key	Result
I	*PA1*	
I	*PA2*	Puts the device into *S* mode. The prompt is reissued upon returning to the previous mode.
I	*CLEAR*	Clears the screen and advances the page number.
I	*ENTER*	Moves input from the input area to the output area and passes it to APL for processing. When the cursor is in the output area, a line from the screen at the cursor position is displayed in the input area (see Input Editing).
I	*PFK*	Contents are inserted at the cursor location in the input area. If the definition is of type *N*, the device remains in *I* mode and editing is allowed otherwise the resulting prompt is handled as if the enter key had been pressed.
I	ATTN	Signals *O BS U BS T* and issues a new prompt (SNA terminals only).
R	*PA1*	Signals attention (non-SNA only).
R	*PA2*	Puts the device into *S* mode and halts output (pending output is not lost). Returning to *R* mode allows output to continue (non-SNA terminals only).
R	*CLEAR*	Clears the screen and advances the page number (non-SNA terminals only).
R	*ENTER*	Ignored.
R	*PFK*	Ignored.
R	ATTN	Signals attention, (SNA terminals only).
S	*PA1*	Returns to previous mode and signals attention.
S	*PA2*	Ignored.
S	*CLEAR*	Clears the screen and advances the page number.
S	*ENTER*	Returns the device to the previous mode unless the page number was altered by the user, in which case that page is displayed (if possible). If *N* or *A* was entered in place of the page number, then that paging status is established and the device returns to the previous mode.
S	*PF1*	Displays the previous page (if it exists).
S	*PF2*	Displays the next page (if it exists).
S	ATTN	Returns to previous mode and signals attention (SNA terminals only).
K		As for *R* mode.

FULL SCREEN MANAGEMENT

APL FUNCTIONS FOR FULL SCREEN MANAGEMENT

In fullscreen mode an application program may define rectangular areas called *fields*. Each *field* has a defined area and attributes. Area is defined by specifying a starting row and column, marking the upper lefthand corner, depth and width. Attributes include write protection, intensity, and color (for units so equipped). Specifications are made in the form of a format matrix. The function *FORMAT* in workspace 5 *IBM*3270 passes the specifications to the fullscreen manager.

Workspace 5 *IBM*3270 contains other functions which simplify management of the 3270 device in full screen mode. This section contains a summary of these functions, then a detailed description of each. These functions are intended to be functionally equivalent to the cover functions outlined in the IBM manual SH20-2341 (VSAPL Extended Editor and Full Screen Manager) thus providing all the capabilities of IBM's AP124X. In addition, 5 *IBM*3270 provides functions that control hardware features not covered by APL124X (at publication time) such as color. The three areas addressed by these functions are:

Field definition

Writing to or reading from the screen

Miscellaneous

Most of the functions described below use variables *CTLS* and *DATS*, which are shared with I.P. Sharp Associates' AP124. See the end of this section for a list of the shared variable operations.

FUNCTION SUMMARY

1) **Defining Screen Fields**

a) *FORMAT* - defines the location and attributes of screen fields.

b) *REFORMAT* - redefines selected screen fields.

c) *READFORMAT* - returns the format matrix defining current screen fields.

d) *FIELDTYPE* - assign field attributes such as protection.

e) *INTENSITY* - sets a display attribute such as normal, invisible or intense.

f) *COLOR* - sets a field's color (for devices so equipped).

g) *PSS* - sets a field's program symbol set (currently unsupported).

h) *HIGHLIGHT* - sets field highlighting attributes such as reverse video, blinking and underscore (for devices so equipped).

i) *FIELDATTR* - controls autoskip and trailing blank processing.

2) **Writing To or Reading From the Screen**

a) *WRITE* - puts data into the output buffer which is later written to the screen.

b) *IMMWR* - as for *WRITE* but data (including buffered data) is written immediately.

c) *READSCREEN* - performs a physical write of any buffered data, waits for the user to cause input, then returns a vector of input information, cursor position and field numbers.

d) *GETFIELDS* - returns data associated with specified field numbers.

e) *WRITEPFK* - defines program function keys.

f) *READPFK* - reads the program function key definitions.

3) **Miscellaneous**

a) *ERASESCREEN* - clears the screen.

b) *DELAYCLEAR* - causes a screen clear before the next physical write.

c) *FMTCHK*3270 - validates a format matrix.

d) *SETCURSOR* - moves the cursor to a field, row, column combination after the next physical write.

e) *ALERT* - causes the audible alarm to sound after the next physical write.

f) *STATE* - reports the current device status.

g) *CLEARSTD* - clears the standard screen.

FUNCTION DETAILS

1) Formatting functions

FORMAT

FORMAT FMTMATRIX

FMTMATRIX is a numeric matrix with one row describing each field and from 4 to 11 columns. The columns represent the starting row, starting column, depth, width, type, intensity, color, program symbol set, highlights, skip, and trailing blank processing. Default attributes are assumed when any attribute is not specified.

The default format matrix is:

`1,1,`*DEVICE ROWS*`,`*DEVICE COLS*`,2,1,0,0,0,0,1`

Type is one of the following:

`0` - character input/output allowed

`1` - numeric character input/output allowed

`2` - character output only (default)

A fieldtype of 2 causes that field to be 'protected', that is, data cannot be entered there by the user.

Intensity is one of:

`0` - do not display data

`1` - normal intensity (default)

`2` - high intensity

Color must be one of:
(There are 7 colors plus a default color available on the IBM 3279.)

`0` - green (default)

`1` - blue

`2` - red

`3` - pink

`4` - green

`5` - turquoise

6 - yellow

7 - white

Program Symbol Set must be 0 (default).

Highlighting is one of:

0 - no highlighting (default)

1 - blink

2 - reverse video

4 - underscore

Skip is one of:

0 - skip to next unprotected field when the cursor is advanced past the end of the current unprotected field (default)

1 - allow the cursor to move into the next screen area regardless of its definition

Blank Processing is one of:

0 - blanks contained in written data are unchanged

1 - blanks contained in written data are printed as null characters, allowing insertion (default)

On the screen, following each row of a defined field is another attribute character which 'undoes' the characteristics of the field. Remember that an attribute character occupies a physical space on either side of each field row (not included in the width specification) so allow for this when designing a screen. When the format matrix skip column is 0 (the default), undefined areas are 'skipped' over when the cursor moves there during input. When the cursor encounters a skip area, it moves to the next unprotected field row. If blank processing is specified, (format matrix blank processing column is 1, the default) trailing blanks in the data are changed to nulls to allow insertion.

REFORMAT

FLD REFORMAT FMTMATRIX

FLD is a scalar or a vector of field numbers, *FMTMATRIX* is a formatting matrix as described for the *FORMAT* function. *REFORMAT* applies one row of *FMTMATRIX* to each element of *FLD*. Field data is not erased during the reformat operation.

READFORMAT

R←READFORMAT

The result is the current format matrix.

FIELDTYPE

FLD FIELDTYPE TYP

TYP is one of the field types specified in the *FORMAT* section above. *FLD* is a scalar or a vector of field numbers.

INTENSITY

FLD INTENSITY INT

INT is one of the intensities listed in the *FORMAT* section. *FLD* is a scalar or a vector of field numbers.

PSS

FLD PSS PS (currently unsupported)

PS is the program symbol set number whose characters are to be used in the specified fields *FLD* (a scalar or vector of field numbers).

COLOR

FLD COLOR COL

COL is one of the color numbers listed in the *FORMAT* section. *FLD* is a scalar or a vector of field numbers.

HIGHLIGHT

FLD HIGHLIGHT HI

HI is one of the highlight options listed in the *FORMAT* section. *FLD* is a scalar or a vector of field numbers.

FIELDATTR

FLD FIELDATTR ATT

ATT is an integer scalar or a vector from 0 to 3 indicating the attribute type.

0 - no autoskip, no trailing blank processing
1 - autoskip, no trailing blank processing
2 - no autoskip, trailing blank processing
3 - autoskip, trailing blank processing (default)

FLD is a scalar or a vector of field numbers.

2) Functions to Write and Read

In full screen mode some special characters can be displayed. These are, in 0 origin, as follows:

□*AV*	**Meaning**
244	tee down
245	tee
246	big minus
247	big plus
248	bottom right corner
249	bottom left corner
250	top left corner
251	top right corner
252	field mark
253	dup
254	null

WRITE

FLD WRITE DATA

WRITE places data into the screen's buffer which is written to the screen during the next physical write. *FLD* is a scalar or a vector of field numbers and *DATA* is a character data matrix. Each row of data will be written to the field number in the corresponding position of *FLD*.

IMMWR

FLD IMMWR DATA

As for the function *WRITE*, except that the physical write is performed immediately.

READSCREEN

R←READSCREEN

Reading data from the screen is a 2-part operation using the functions *READSCREEN* and *GETFIELDS* (described below). *READSCREEN* waits for the user to cause input, then returns completion information. *GETFIELDS* returns the data. The following chart describes the completion information resulting from *READSCREEN*:

R's elements

	1	2	3	4	5	6 ...
USER ACTION	*COMPLETION CODE*	*MODIFIER*	*CURSOR POSITION FIELD*	*ROW*	*COLUMN*	*FIELDS*
ENTER KEY	0	0	*FLDNUM*	*ROW*	*COL*	*FLDNUMS*
PF KEYS	1	1-24	*FLDNUM*	*ROW*	*COL*	*FLDNUMS*
PA KEYS	4	1-3	-	-	-	-
CLEAR KEY	5	-	-	-	-	-
NO INPUT	6	-	-	-	-	-

From the result, an application program can ascertain if and how full screen input was caused and what fields were modified.

GETFIELDS

R←GETFIELDS FLD

FLD is a scalar or a vector of field numbers. The result is a character matrix where each row contains the data for the specified field number. This is the second part of a read operation.

WRITEPFK

WRITEPFK DEFN

The right argument contains the definitions of specified program function keys. Definitions are identical to the local system command *)DPFK* (described earlier under standard screen local system commands). Several definitions may be sent together by separating them with a single carriage return.

READPFK

R←READPFK

The result is the definition of each program function key seperated by a single carriage return.

3) **Miscellaneous Functions**

ERASESCREEN

ERASESCREEN

The full screen display is cleared.

DELAYCLEAR

DELAYCLEAR

The screen is cleared before the next physical write.

FMTCHK3270

*R←FMTCHK*3270 *FMTMATRIX*

The argument is a format matrix as described for the *FORMAT* function. This function performs a validation of that matrix checking for invalid fields (i.e. won't fit on screen, etc.) and returns an error message if the matrix is invalid.

SETCURSOR

FLD SETCURSOR ROW,COL

After the next physical write, the cursor is placed at the row and column of the specified field. If the field number is 0, the row and columns are from the upper left hand corner of the screen. The cursor position is automatically updated when a field is modified by the user.

ALERT

ALERT

The audible alarm (for devices so equipped) sounds after the next physical write.

STATE

STATE

STATE reports current device status (using *FSMIQ* described in the □*ARBIN*/□*ARBOUT* section) including the number of standard screen pages, number of logical screens, first and last page numbers, and screen dimensions.

CLEARSTD

CLEARSTD

The standard screen is cleared and the page number is advanced, unless the page is already clear.

CTL/DAT FULL SCREEN MANAGEMENT

An application requests full screen management by assignment to a control variable which is shared with I.P. Sharp Associates' Auxiliary Processor number 124 (AP124). This control variable is assigned a numeric scalar or vector that specifies an action code and (in some cases) field numbers. AP124 responds by assigning a return code to the control variable indicating whether or not the action was successful. Any data is transferred between the application and the AP through a shared data variable. Below is a list of operation codes and a description of each.

CTL	DAT	Description, or Full Screen Function Name using the Operation
0	*	*DELAYCLEAR*
1	*FMTMATRIX*	*FORMAT*
1,*FLD*	*FMTMATRIX*	*REFORMAT*
2,*FLD*	*DATA*	*IMMWR*
3	*	*READSCREEN*
4,*FLD*	*DATA*	*WRITE*
5,*FLD*	*	*GETFIELDS*
6,*FLD*	*TYP*	*FIELDTYPE*
7,*FLD*	*INT*	*INTENSITY*
9	*	*READFORMAT*
11	*	*ALERT*
12	*POS*	*SETCURSOR*
16	*ATT*	*FIELDATTR*
20	*	*ERASESCREEN*
28	*	Returns a raw data stream representation of the full screen image

Several pairs of variables may be shared simultaneously with AP124 thus allowing several independent logical screens. To accomplish this, fullscreen shared variable naming conventions have been adopted. The first three letters of the control variable must be CTL and similarly, DAT for the data variable. These variables may be up to 11 characters long and are paired for a logical screen on all but the first three characters. (Workspace `5 IBM3270` uses *CTL_S* and *DAT_S*.)

In cases where both a data variable and a control variable are required, the data variable is assigned before the control variable. A non-zero return code in the control variable indicates that an error occurred during the fullscreen operation. Below is a list of the return codes and a description of each.

Code	Description
0	NORMAL RETURN - OPERATION SUCCESSFUL
11	CONTROL VARIABLE RANK ERROR
12	CONTROL VARIABLE LENGTH ERROR
13	CONTROL VARIABLE DOMAIN ERROR
14	INVALID COMMAND
15	REQUEST TO POSITION CURSOR IN AN UNDEFINED FIELD
21	DATA VARIABLE RANK ERROR
22	DATA VARIABLE LENGTH ERROR
23	DATA VARIABLE DOMAIN ERROR
24	DATA VARIABLE NOT SHARED
30	INVALID FIELD NUMBER
32	DEFINED FIELD EXTENDS BEYOND THE SCREEN
33	REFERENCE OUTSIDE FIELD DEFINITION
35	LIGHT PEN FIELD STARTS IN COLUMN 1
36	LIGHT PEN FIELD (HEIGHT 1) NOT CONTAINED IN 1 PHYSICAL SCREEN LINE
37	INVALID FIELD TYPE
38	INVALID FIELD INTENSITY
41	DATA VARIABLE NOT SPECIFIED IN CORRECT SEQUENCE
42	DATA VARIABLE NOT REFERENCED IN CORRECT SEQUENCE
43	INVALID TRANSLATE TABLE CODE
52	DEVICE IS NOT A 3270
53	REQUIRED SHARED STORAGE UNAVAILABLE
54	PRINTER NOT AVAILABLE
89	UNKNOWN SHARED VARIABLE RETURN CODE
91	PHYSICAL FIELD TABLE OVERFLOW
92	PHYSICAL FIELD TABLE ERROR, INTERRUPT
94	DEVICE NOT AVAILABLE
95	UNEXPECTED I/O ERROR
96	CHAINED CCW STRING NOT COMPLETE
97	BAD 3270 ORDERS IN OUTPUT DATA
98	FULLSCREEN SUPPORT NOT AVAILABLE
99	UNKNOW 3270 DEVICE ERROR
201	INVALID COLOR
202	INVALID HIGHLIGHT
203	UNUSED
204	INVALID PROGRAM SYMBOL SET
205	INTERNAL BUFFER OVERFLOW
206	INVALID END COLUMN IN FORMAT MATRIX
207	INVALID NULLS COLUMN IN FORMAT MATRIX

Since I.P. Sharp Associates' AP124 does not have outstanding offers, the degree of coupling will be:

1 - after the application's initial shared variable

2 - as AP124 accepts the offer.

ARBIN/ARBOUT FULL SCREEN MANAGEMENT

This section describes how an application can make complete use of the IBM 3270 device's capabilities through the primitives ⎕*ARBIN* and ⎕*ARBOUT*. The standard screen uses only a small subset of the device's capabilities. A user can construct a raw data stream as outlined in the IBM manual GA27-2749 (IBM 3270 component description) and transmit it using either of the primitives. In this way an application program has control over screen format, highlighting, color, intensity and graphics (if the device is so equipped). The IBM data stream, exactly as described in the IBM manuals, can be used, but I.P. Sharp has simplified the data stream by introducing Sharp Order (SO) codes. In addition to attribute control, SOs allow an application program to:

Define a logical screen

Define logical screen fields on any logical screen

Write to any logical screen or logical screen field

Read from any logical screen or field

Write to the program function keys

Read from the program function keys

Constructing a Data Stream

This section outlines the formation of a valid IBM 3270 data stream with emphasis on the Sharp Order extensions.

1) The Header

A data stream consists of a character vector containing IBM data stream commands and data (for write and read commands only). To the front of this data stream is added 24 bytes of control information which tell the device controlling software (SAPV) what area of terminal control should be addressed. This header currently includes a SAPV command and a logical screen number (the remaining bytes are reserved). When formed, the header and data stream are transmitted as a character vector using ⎕*ARBIN* or ⎕*ARBOUT*. A typical transmission might look like:

```
R←⎕ARBIN HDR,DS
```

where *HDR* is the 24 byte character vector and *DS* is the data stream. The header tells SAPV how to apply the data stream. The first byte of the header, the SAPV command, must be one of:

Command	□AV	Meaning
FSMINQ	0	Device inquiry.

Position	Meaning
0	SAPV version
1	reserved
2	graphic escape character
3	number of rows
4	number of columns
5	number of standard screen pages
6	number of logical screen pages
7	extended capabilities (0 or 1)
8-11	reserved
12-15	first standard screen page number (256 base)
16-19	last standard screen page number (256 base)
20-27	VTAM device ID (Z-code)

The result may be extended in the future.

These values are decoded by the function *STATE* (see Miscellaneous Full Screen Functions, earlier).

FSMWRT 1 The data stream contains information to be written to the screen.

FSMRD 2 If the data stream is present, it contains information to be written to the screen. After the write, wait for the user to cause input (ENTER key, PA1, PA2, CLEAR or any PFK) and return the input to the application program. This information includes termination type (ENTER, PFK, etc.), cursor position (row and column) and data. Data is returned in IBM EBCDIC device codes.

FSMPWR 3 Program Function Key Write. The data stream is in the form *)DPFK K T D* followed by a carriage return. Several PFK definitions may be joined together.

FSMPRD 4 Program Function Key Read. Returns the current definition of each program function key. This command takes an empty data stream.

FSMSWCH 5 Switch logical screens. *FSMSWCH* is followed by □*AV*[**logical screen number**]. The logical screen image is copied from the current screen to that logical screen number.

FSMRDT 6 Read and Translate (same as *FSMRD* except the data is translated). Data that is read from the screen is translated into Z-codes using the default device translate table. The advantage of *FSMRDT* over *FSMRD* is that the application does not have to do the translation from device codes to Z-codes after a read.

FSMWRS 7 Write to Screen file. The full screen image currently displayed is written to a screen file and may be recalled with a read command.

FSMCOL 8 Read or modify standard screen color and highlight attributes. To modify these settings, the data stream must consist of 0 or more triples of field number, color,

and highlight. The result (⎕*ARBIN* only) is a vector of pairs of the previous settings for each field. Below is a chart of the possible settings.

Field	⎕*AV*	**Meaning**
OUTF	0	Output area field attributes.
OUT	1	Output character attributes.
INOUT	2	Character attributes of input when it is in the output area.
PROUT	3	Protected prompts in the output area.
MOD	4	The mode letter.
SEL	5	Page number when protected.
SELI	6	Page number when not protected.
AUTO	7	Autopage field.
PRM	8	Protected prompt in the input area.
INF	9	Input area attributes.
INFP	10	Input area when protected.

Color/⎕*AV*	**Highlight**/⎕*AV*
blue/241	blink/241
red/242	reverse/242
pink/243	underscore/243
green/244	
turquoise/245	
yellow/246	
white/247	

For example, to have yellow, reverse video input when it is in the output area, the triple is

```
⎕AV[2 246 242]
```

FSMCLR	9	Clear the standard screen by going to the top of the next page. *FSMCLR* has no effect if the standard screen is already clear.

Following the header's SAPV command is the logical screen number. Its range is from ⎕*AV*[0-255] for screen numbers 0-255. Element number 7 of the *FSMINQ* result is the number of logical screens available.

2) The Data Stream

Following the 24 byte SAPV header is the data stream, which is valid for write and read commands only. Refer to IBM manuals GA27-2749 (3270 Component Description) and GL27-6999 (3270 Programming) for a more detailed data stream description. The commands in the data stream perform screen writes and reads as well as assignment of cursor position and attribute characters. A writing data stream must begin with a write command followed by a write control character. After that, the data stream contains any combination of orders and data. Although a screen read can be done by transmitting a SAPV read command (*FSMRD* or *FSMRDT*) with an empty data stream, data can be transmitted simultaneously. In this way information can be displayed and a response solicited.

SAPV decides whether the data stream is all IBM codes or all Sharp Order codes by checking the value of the first byte. Below is a list of valid Sharp Order codes:

a) First Byte (the command)

Command	⎕*AV*	Meaning
SOWRT	13	Writes the following orders and data to the screen. This command adds to any existing screen image.
SOEWC	11	The screen is cleared and reset to its primary size before processing the rest of the data stream.
SOEWA	12	As for *SOEWC* except the alternate (larger) screen size is available for writing.

b) Second Byte

The data stream's second byte is a write control character (*WCC*). The *WCC* character controls, among other things, keyboard resetting and alarm sounding (for devices so equipped). A keyboard reset clears the input inhibited condition. The most common *WCC*'s are listed below but refer to the IBM manual for a complete breakdown of the *WCC* byte (each one resets the modified data tag bit).

Command	⎕*AV*	Meaning
WCR	3	Reset the keyboard after the write and do not sound the alarm.
WCN	1	Do not reset the keyboard and do not sound the alarm.
WCRA	7	Reset the keyboard and sound the alarm.
WCNA	5	Do not reset the keyboard but do sound the alarm.

c) The Remainder

The remainder of the data stream consists of Sharp Order (SO) codes and data. Sharp Orders are used to position, define and format data, erase parts of the screen or insert the cursor. Below is a list of Sharp Orders with a brief description of any bytes that must follow, then a descriptive paragraph of each order.

Order	⎕AV	Bytes	Meaning
SOALLE	0		The following data is all in IBM data stream format (EBCDIC)
SOE	1	⎕AV[256 256⊤*COUNT*]	The following encoded *COUNT* bytes are in IBM data stream format (EBCDIC)
SOALLZ	2		All of the following data is in Z-codes
SOZ	3	⎕AV[256 256⊤*COUNT*]	The following encoded *COUNT* bytes are Z-codes
SOSBA	4	⎕AV[*COLUMN*]	Set the buffer address to row *ROW* and column *COLUMN* on the screen
SOEUA	5	⎕AV[*ROW*,*COL*]	Erase all unprotected areas from the current buffer address to location *ROW* and *COLUMN*
SOSF	6	*ATTRIBUTE*	Start a screen field at the current buffer location and assign it attribute *ATTRIBUTE*
SOSA	7	*TYPE*,*VALUE*	Set attributes for characters to be displayed.
SOSFE	8	⎕AV[*COUNT*],(*COUNT*×2)⍴*TYPE*,*VALUE*	Start field extended. *COUNT* is the number of type/value pairs.
SORA	9	⎕AV[*ROW*,*COL*],*CHAR*	Repeatedly write character *CHAR* from the current buffer address to address *ROW*,*COL*
SOMF	10	⎕AV[*COUNT*],(*COUNT*×2)⍴*TYPE/VALUE*	Modify field. To the attribute character starting at the current buffer address, set the attributes described in the type/value pairs that follow
SOEWC	11		The screen is cleared and reset to its primary size before processing the rest of the data stream.
SOEWA	12		As for *SOEWC* except the alternate (larger) screen size is available for writing.
SOWRT	13		Writes the following orders and data to the screen. This command adds to any existing screen image.

SOAID	14	*AID,ROW,COL*	Attention identification. Inbound data stream information including *AID* and cursor *ROW*, *COL*umn at input termination
SOLF	15	*ROW,COL,DEPTH,WIDTH*,0,0,0,*FLAGS,PAD,PREFIX ATTRIBUTE,SUFFIX ATTRIBUTE,SOZ,COUNT*	Logical field description. Includes start positions, dimensions, pad character, leading and trailing attributes. *ROW*, *COL*, *DEPTH*, *WIDTH* and *COUNT*, are 256 base, 2-byte counts.
SOIC	16	□*AV*[*ROW,COL*]	Insert cursor at the current buffer address.

SHARP ORDER CODES

DETAILED DESCRIPTION

1) **All EBCDIC** (SOALLE)

This order says that all following data and orders are in EBCDIC. If some of the orders or data following are not in EBCDIC, use SOE (see below) instead.

2) **EBCDIC** (SOE)

A 2 byte count follows. That many characters are then considered to be in EBCDIC. SOALLE and SOE are used when transmitting an order which has no Sharp Order counterpart or non Z-code data. Counts for this order are 256 base and 2 bytes long, so encode them using `⎕AV[256 256⊤⍴EBCDIC-DATA]`.

3) **All Z-codes** (SOALLZ)

As for SOALLE, the entire data stream from this point is considered to be Z-code. Use this order when writing text to the screen which requires no conversion.

4) **Z-codes** (SOZ)

As for SOE, there is a 2 byte, 256 base count immediately afterward indicating how many Z-code characters follow.

5) **Set Buffer Address** (SOSBA)

This order looks for 2 bytes indicating the row and column to which the buffer address should be set. The row and column are direct indices into `⎕AV`, so for example, `SOSBA,⎕AV[10 20]` sets the buffer address at row 10, column 20. Use this order in cases such as starting a field (see SOSF and SOSFE) which begin at the current buffer address.

6) **Erase Unprotected to Address** (SOEUA)

This order is followed by a row and column address. All unprotected data is erased from the current buffer address to the address `⎕AV[ROW,COLUMN]`.

7) **Start Field** (SOSF)

A field is started at the current buffer address. The field attribute character which must follow this order remains in effect until the next field attribute character is encountered. When the screen contains a single attribute character, the entire screen assumes the characteristics of that attribute. Attributes must be one of the following:

□*AV*	**Meaning**
76	alphanumeric input/output, non-display.
64	alphanumeric input/output, normal display.
200	alphanumeric input/output, high intensity.
92	numeric input/output, non-display.
80	numeric input/output, normal display.
216	numeric input/output, high intensity.
108	alphanumeric output, non-display.
96	alphanumeric output, normal display.
232	alphanumeric output, high intensity.
124	numeric output, non-display (note: any of these 3 cause cursor skip).
240	numeric output, normal display.
248	numeric output, high intensity.

Note: A field attribute character occupies physical space on the screen and appears as a blank character.

8) **Set Attribute** (SOSA)

This order is used to control extended features (on units with that option). Included in the extended features are color, extended highlighting and program symbols. Attributes set by the SOSA order remain in effect until another SOSA order is encountered, and override field attributes set in SOSFE orders. Subsequent data has those characteristics.

9) **Start Field Extended** (SOSFE)

SOSFE looks for a count byte (□*AV*[*COUNT*]) which indicates how many type/value pairs follow (again, refer to the IBM manual GA27-2749 for a complete breakdown of types and values). SOSFE differs from SOSF in that it is used to assign extended attributes as well as standard attributes starting at the current buffer address. SOSFE and SOSF control field attributes, while SOSA controls the attributes of single characters in the data stream.

10) **Repeat to Address** (SORA)

Writes the character following the specified buffer address from the current buffer address to the specified one.

11) **Modify Field** (SOMF)

Used to selectively change attribute characters previously set by SOSF or SOSFE at the current buffer address. The address must be that of an existing field attribute character and the data following it is identical to that of SOSFE.

12) **Erase Write Clear** (SOEWC)

The screen is cleared and reset to its primary size before processing the rest of the data stream.

13) **Erase Write Alternate** (SOEWA)

As for SOEWC except the alternate (larger) screen size is available for writing.

14) **Write** (SOWRT)

Writes the following orders and data to the screen. This command adds to any existing screen image.

15) **Attention Identification** (SOAID)

The next 3 bytes are an *AID* character and a 2 byte buffer address. From the AID character, an application knows how fullscreen input was caused. Termination codes include the enter key, clear key, a program function key or a PA key. A numeric value for the 2 byte address is achieved by *ROWCOL*←⎕*AV*⍳*ADDR*. Following the address is the data from any modified fields. SOAID is valid for an incoming data stream only (resulting from a SORD or SORDT).

AID	⎕*AV*
ENTER	125
PF1	241
PF2	242
PF3	243
PF4	244
PF5	245
PF6	246
PF7	247
PF8	248
PF9	249
PF10	122
PF11	123
PF12	124
PF13	193
PF14	194
PF15	195
PF16	196
PF17	197
PF18	198
PF19	199
PF20	200
PF21	201
PF22	74
PF23	75
PF24	76
PA1	108
PA2	110
PA3	107
CLEAR	109

16) **Logical Field** (SOLF)

Use the SOLF order to describe a logical screen field. A logical field has a defined area and attributes. SOLF is followed by 2-byte, 256 base ⎕*AV* indices for starting row, starting column, depth, width, 3 reserved bytes, flags (explained later), a pad character, two properly formed SOSF, SOSFE, or SOMF orders, SOALLZ or a properly formed SOZ order, then any text to be written in that field. Even when data is not present, the SOZ order must be present with a length count of ⎕*AV*[0 0]. Row, column, depth, and width are 256 base, 2-byte counts. A suffix field is required since the prefix field attribute remains in effect until another field attribute is encountered. Flag bits are used to inhibit display of all or part of the SOLF data. This is desirable since the application may need to replace only an attribute (such as color) or just the data without affecting the rest of the field. The first 3 bits of the flag byte control writing of the prefix attribute, suffix attribute and data respectively. When a bit is on, that write is inhibited, so, to write data only, use ⎕*AV*[2⊥1 1 0 0 0 0 0 0] as the flag byte (note: the trailing 5 bits are reserved). Any non-overstruck character may be used as a pad character, although the most common are null (⎕*AV*[254]) and blank (⎕*AV*[152]).

For example, to write *HI THERE* and attributes in a field whose start position is row 1 column 5 and whose depth is 10 and length 20, the SOLF order is:

```
SOLF,⎕AV[0 1,0 5,0 10,0 20, 0 0 0 0 152],SOSF,
     ATTA,SOSF,ATTZ,SOALLZ 'HI THERE'
```

In the above example, *ATTA* and *ATTZ* are the prefix and suffix attribute characters as described in the IBM 3270 component description manual. Several SOLF order streams may be transmitted at once allowing several simultaneous screen field writes.

17) **Insert Cursor** (SOIC)

The cursor is placed at the current buffer address. For example, to place the cursor at row 10, column 20, use *SOSBA*,⎕*AV*[10 20],*SOIC*.

Transmitting a Complete Header and Data Stream

By constructing a header for SAPV and a data stream for the device, ⎕*ARBIN* and ⎕*ARBOUT* allow an application complete control over an IBM 3270 screen (see the sections on header and data stream formation above).

Transmission is one of the following

```
R←⎕ARBIN (CS,22ρ⎕AV[0]),DS
```

where *CS* is a SAPV command and a screen number and *DS* is an IBM data stream, or

```
⎕ARBOUT (CS,22ρ⎕AV[0]),DS
```

⎕*ARBIN* differs from ⎕*ARBOUT* in that it gives a result, which is necessary in the case of a screen read. Although either ⎕*ARBIN* or ⎕*ARBOUT* can be used in most cases, ⎕*ARBOUT* is preferable in a fully debugged system as its response time is less. All data streams (incoming or outgoing) using ⎕*ARBIN* have a 24 byte header (provided by the application in the outgoing and by the system incoming). The result of ⎕*ARBIN* contains ⎕*AV*[0] at position 4 if the operation was successful and an error code otherwise. Workspace 5 *UTIL*3270 contains functions to format and transmit data streams and to decipher error codes.

Error	Meaning
0	All OK
1	Send access method failure
2	Receive access method failure
3	Output exceeds DBUF size
4	Command illegal
5	Data>DBPTR
6	Reserved fields not 0
7	Read data exceeded DBUF
8	Invalid program function key data
9	Invalid logical screen number
10	Invalid COLHI index, color, attribute triple
11	Invalid length
12	Invalid count in Sharp data stream
13	Translation of Sharp data stream too large for buffer
14	Undefined Sharp Order code in data stream
15	Sharp Order, reserved fields not 0
16	Sharp Order, logical field order invalid
17	Invalid page number
18	Unknown error code!!!

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

SATN-39
1 JUN 81

SHARP APL TECHNICAL NOTES

TITLE: THE SHARP APL S-TASK INTERFACE

AUTHOR: Richard H. Lathwell

ABSTRACT: An S-task is a *SHARP APL* task which obtains its input and delivers its output as character-vector values of a shared variable, but is otherwise identical to a T-task.

INTRODUCTION

An S-task is a *SHARP APL* task which obtains its input and delivers its output as character-vector values of a shared variable, but is otherwise identical to a T-task. The S-task interface permits any program which is capable of communicating with the Sharp Shared-Variable Processor (i.e., any program which can share variables) to communicate with *SHARP APL* in the manner of a user at a terminal. This description of the S-task interface is from the point of view of APL programs; in order to prepare non-APL programs for this purpose, refer to SIN-48 *SHARP APL* Installation-written Auxiliary Processors, and to the DSECT TYIOMODE contained in the APE APL source libraries.

SHARP APL PROCESSOR CHARACTERISTICS

In order to communicate with any processor with shared variables, a definition of its characteristics is required. To this end, the SHARP APL Processor is designed to behave as follows:

- A **line** of input or output consists of a sequence of APL characters.
- The APL Processor waits for a **single line** of input
- The line of input is processed and results in an arbitrary number of lines of output, but no fewer than one.
- The final line of output is an **input prompt** which contains an indication that the APL Processor is waiting for input.

ESTABLISHING AN S-TASK

When SHARP APL has been installed with appropriate configuration parameters, it connects itself to the shared variable processor as **processor 1**. An S-task may then be created by offering to share a variable with processor 1:

```
1 ⎕SVO 'NAME'
```

where *NAME* is a vector of 15 or fewer characters denoting the name of the variable to be used for the interface. The account which offers the variable to processor-1 is known as the **initiator** of the S-task. When resources permit, the S-task controller will allocate the necessary tables, obtain a clear workspace, assign a task identification number, and then match the offer to share. When the degree of coupling becomes 2, the task is in the same state as a T-task after a terminal has become connected to the system, but before the user has signed on.

The normal system limits on the number of tasks which can be created by a single user also apply to S-tasks: if the total number of tasks signed on with the initiator's account number has reached this limit, a new offer to processor 1 will be ignored, and the variable must be re-offered after the number of tasks has been reduced. The initiator of an S-task is analogous to the initiator of an N-task in that the S-task may be terminated at any time by a ⎕*BOUNCE*.

COMMUNICATION WITH AN S-TASK

S-task input and output is in the form of APL character vectors of four or more elements. The first four characters of each value contain control information which must be present. The first two of these characters are not used at the present time, and must be binary zeroes, i.e., `⎕AV[0 0]`. (**Note**: index origin 0 is assumed throughout this discussion.) The third element is used to denote commands to the S-task controller from the initiator and responses from the S-task controller (see below); `⎕AV[0]` in this position indicates that the value is to be passed on to APL. The fourth element is used to indicate the meaning of the value which is being passed from SHARP APL to the initiator.

For example, the first input to an S-task will normally be an attempt to sign on to SHARP APL:

```
NAME←⎕AV[0 0 0 0],')1234:KEY'
```

Assuming that the account number and key are valid, the response returned in *NAME* will be:

```
PREFIX,'2134)22.13.00 04/01/80 SUEBLUE'
```

i.e., the normal indication of a successful signon. Otherwise the response might be:

```
PREFIX,'INCORRECT SIGNON'
```

or

```
PREFIX,'NUMBER NOT IN SYSTEM'
```

The important point to note is that, for every value set in the interface variable, APL or the S-task controller will respond with one or more values in reply.

APL OUTPUT BLOCKING

While APL sends output a line at a time, the S-task controller will catenate several lines together in order to reduce the number of shared variable accesses required. Each line is preceded by two characters which define the length of the line including the two length characters. Each line must be 'deblocked' by a code segment such as the following:

```
READ:→(0<⍴BLOCK)/TAKE      ⍝ BRANCH IF BLOCK CONTAINS LINES
   BLOCK←NAME              ⍝ RECEIVE OUTPUT FROM APL
TAKE: L←256⊥⎕AV⍳2↑BLOCK    ⍝ LENGTH OF NEXT LINE
  OUTPUT←2↓L↑BLOCK         ⍝ NEXT LINE
   BLOCK←L↓BLOCK           ⍝ DROP LINE FROM BLOCK
      ...
```

QUALIFICATION OF VALUES RECEIVED FROM APL

The fourth element of prefixes received from APL indicates the meaning of the rest of the vector. Common combinations are shown in figure 1.

If

```
MODE←(8ρ2)⊤⎕AV⍳OUTPUT[3]
```

then the elements of *MODE* have the following meaning when their values are 1:

MODE[0] - The S-task workspace is in immediate execution mode, i.e., is in a state where system commands will be recognised and processed. The purpose of this signal is so that initiator programs can provide 'system commands' of their own: when this mode element is 1, input lines with a right parenthesis as the first non-blank character can safely be intercepted by the initiator.

MODE[1] - The rest of the line contains an obfuscating blot to obscure passwords, etc. The blot can consist of more than one line.

MODE[2] - The S-task is not signed on to SHARP APL.

MODE[3] - A keyboard-locked terminal is ready for input.

MODE[4] - *APL* wishes the terminal cursor or carrier reset to the left margin of a new line. This may be received following a break signal.

MODE[5] - The vector contains arbitrary output from ⎕*ARBIN* or ⎕*ARBOUT*. This will occur alone when more arbitrary output will follow, and with *MODE*[6] or *MODE*[7] for the final block of a ⎕*ARBIN* prompt or ⎕*ARBOUT* data respectively.

MODE[6] - The vector is a prompt for input: SHARP APL expects input after this output. Normally, the input will be catenated with this prompt by the initiator, and returned to APL. However, the initiator may instead process the prompt in conjunction with additional input, and return the processed information to APL.

MODE[7] - The value is output, and more output will follow.

CONTROL CHARACTERS

Certain character values have meanings of a control nature when passed to or from APL as normal input or output. Common control characters are:

CRLF ⎕*AV*[156] Carriage return/linefeed. Advance the medium to a new line with the carriage or cursor positioned at the left margin. APL regards the *CRLF* character as an 'end of input' marker, and it does not examine characters which follow it.

LF ⎕*AV*[159] Linefeed. Advance the medium to a new line without repositioning the cursor or carriage.

BS ⎕*AV*[158] Backspace. Reposition the cursor one position ot the left.

TAB ⎕*AV*[2] Tabulation. Never included in S-task output. When included with S-task input, *TAB* will be evaluated with respect to ⎕*AV*.

EOB ⎕*AV*[157] End-of-block. This character commonly occurs in S-task output for historical reasons. It has no meaning and should be ignored when it occurs in normal output.

S-TASK CONTROLLER COMMANDS

The third element of values set by the initiator is a command to the S-task controller as follows:

⎕*AV*[0] - indicates the character vector is APL input.

⎕*AV*[1] - is a request for accounting information. The response from the S-task controller is a three-element **integer** vector containing 3↑⎕*AI* of the S-task. If the S-task is not signed on to APL all three elements will be zero.

⎕*AV*[2] - is a request to signal BREAK, analogous to depressing the BREAK key on a terminal connected to a T-task.

⎕*AV*[3] - is a request for disconnect permission, which allows the initiator to retract the interface variable without immediately terminating the S-task. If permission has not been requested, retracting the variable will result in the immediate termination of the S-task in a manner similar to the termination of a T-task when the telecommunication line is disconnected. After retraction, the interface variable remains as an offer from processor 1 to the initiator; resharing resets the permission to retract. The S-task terminates if it requests input while the variable is retracted.

⎕*AV*[4] - is a request to reset the retraction permission obtained previously by ⎕*AV*[3].

S-TASK CONTROLLER RESPONSES

The S-task controller responds to each non-zero command (except for accounting information which returns an integer vector) with a four-element character vector (preceded by a two-character length) whose third element indicates the result of the command:

⎕*AV*[0] - indicates that the command was successfully executed.

⎕*AV*[1] - indicates an invalid value: an invalid command value, non-character value, non-vector value, or a character vector of fewer than four elements.

⎕*AV*[2] - indicates that APL could not accept input because of insufficient workspace.

⎕*AV*[3] - indicates that the S-task controller refused to honour a request because the S-task was not signed on to SHARP APL.

S-TASK TERMINATION

An S-task will be terminated by any of the following:

- *⎕BOUNCE*
- Sending a valid signoff command, e.g., *)OFF*, when *MODE*[0] is 1.
- Retracting the interface variable, e.g., *⎕SVR 'NAME'*, without disconnect permission.
- A request by the S-task for input while the interface variable is retracted with disconnect permission.

Figure 1

Typical Mode Combinations

	MODE INDEX							
	0 *IMMX*	1 *BLOT*	2 *NSO*	3 *KBLI*	4 *BACK*	5 *ARB*	6 *IN*	7 *OUT*
Occurs with all modes when the S-task is not signed on to APL.			X					
Normal output.								X
Normal input.							X	
Immediate execution prompt.	X						X	
KB locked terminal ready for input. Input prompt will be emitted upon receipt of a 'break' signal.				X				X
Input prompt text is a blot.	X	X					X	
Output text is a partial blot.		X						X
ARBOUT final sequence.						X		X
ARBOUT - more to follow.						X		
ARBIN input prompt.						X	X	
APL requests cursor at the left margin, new line.					X			X

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

SATN-40
20 JUN 81

SHARP APL TECHNICAL NOTES

TITLE: COMPLEX NUMBERS

AUTHOR: Eugene E. McDonnell

ABSTRACT: Complex numbers have been added to SHARP APL as an internal data type, and most of the primitive functions have been extended, where appropriate, to give complex results, and to take arrays containing complex numbers as arguments. Our implementation follows, in large measure, the design proposed by Paul Penfield, Jr., of MIT, in his 'Proposal for a Complex APL', given at the APL79 Conference.

SATN-40
20 JUN 81

COMPLEX NUMBER APL AVAILABLE

> Once we conceive of the real line as embedded in a plane of complex numbers, we have entered a whole new domain of mathematics. All our old knowledge of real algebra and analysis becomes enlarged and enriched when reinterpreted in the complex domain. In addition, we immediately see countless new problems and questions which could not even have been raised in the context of the real numbers alone.
>
> Philip J. Davis and Reuben Hersh, **The Mathematical Experience**, Birkhausen, Boston, 1980

Complex numbers have been added to SHARP APL as an internal data type, and most of the primitive functions have been extended, where appropriate, to give complex results, and to take arrays containing complex numbers as arguments.

The extension is not complete. The floor, ceiling, residue, representation, and dyadic format functions have not been extended, because there remain points of uncertainty regarding their definitions. The way in which complex numbers are displayed in this release is provisional. As we gain more experience in the use of these numbers, we may change the form.

Until this announcement, writers of APL applications which required complex numbers had to simulate them by various devices which made the use of APL's primitive functions and operators difficult. With this announcement, all the facilities of APL, except as noted in the preceding paragraph, may be used directly on complex numbers.

Generally speaking, existing applications are not affected by this change. The potential differences are local to just a few primitive functions, and are relatively minor. Those functions which would benefit from complex numbers yield complex results instead of a domain error.

Complex constants and display

If two real scalar numeric constants are connected by the letter *J*, as in `3J¯4`, this is interpreted as the rectangular representation of a complex constant, with the real part first and the imaginary part second. Each of the two real numbers can be written using either decimal or scaled representations.

In displaying a column of numbers, some of which have nonzero imaginary parts, all the numbers in the column are right justified. For example, if `Z` is the vector whose elements are the fifth roots of `¯32`, a one-column matrix `W` derived from it would be displayed as follows (with `⎕PP` at `6`):

```
      ⎕←W←5 1⍴Z
 1.61803J1.17557
¯0.61803J1.90211
               ¯2
¯0.61803J¯1.90211
 1.61803J¯1.17557
```

The extensions of the primitive functions

Many of the primitive functions need no discussion since their extension to complex arguments is well understood. Here we shall describe only those functions whose extension is not obvious.

Conjugate: $+\omega$ is the conjugate of ω, that is, the value obtained by reflecting ω on the real axis. The conjugate of a real number ω is equal to ω. Thus a test to see if a number is real is $\omega=+\omega$.

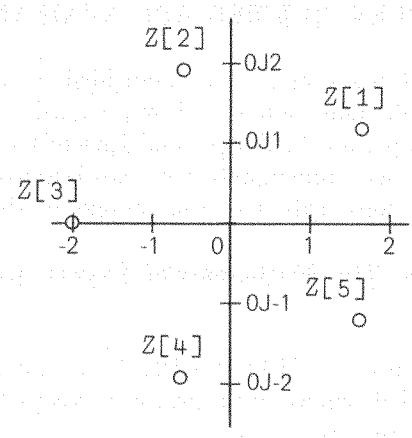


Figure 1

The elements of the vector `Z` are shown in Figure 1. The numbers `Z[1]` and `Z[5]` are conjugates, as are `Z[2]` and `Z[4]`. The number `Z[3]` is equal to its own conjugate, since it is real.

Magnitude: $|\omega$ is the magnitude, or modulus as it is often called, of ω. It is the value obtained by rotating ω about the origin onto the positive axis. It may be defined as `(ω×+ω)*0.5`. For example:

```
|3J¯4                  ↔  5
|0.6J¯0.8              ↔  1
|1.61803J¯1.17557      ↔  2
|¯1J1                  ↔  1.41421
```

Direction: $\times\omega$ is the direction of ω, and is an extension of the signum function on real numbers. The direction of `0` is `0`. For nonzero ω, $\times\omega$ is the value at the intersection with the unit circle of the ray from the origin through ω. The magnitude of $\times\omega$ for nonzero ω is `1`, and $\times\omega$ may be defined as $\omega\div|\omega$. For example:

```
×3J¯4          ↔  0.6J¯0.8
×0.03J¯0.04    ↔  0.6J¯0.8
×0J10          ↔  0J1
```

New circular function left arguments: Ten new left arguments α have been provided for $\alpha\circ\omega$, primarily for use with complex numbers.

The correspondence between left argument value and function is different from that given in [1]. The assignments described here preserve as much as possible the association between odd functions (real part, imaginary part) with odd left arguments (9 and 11), and even functions (magnitude) with even left arguments (10). They also eliminate the gap which the earlier scheme had left between 8 and 11.

$(-\alpha)\circ\omega$	α	$\alpha\circ\omega$
$-8\circ\omega$	8	`0J1×ω×(1+ω*¯2)*0.5`
ω	9	`(ω++ω)÷2` real part of ω
$+\omega$	10	`|ω` magnitude, or modulus, of ω
`0J1×`ω	11	`(ω-+ω)÷0J2` imaginary part of ω
`*0J1×`ω	12	`11○⊗ω` arc or phase angle of ω

Two of these, 8○ω and ¯8○ω, are new Pythagorean functions, based on the expression (¯1-ω*2)*0.5, but modified to allow for both signs of the square root. The value of this function on reals is never real, which is why it had not been among the APL primitives before. With it, the set of Pythagorean functions is complete.

The remaining new left arguments for α○ω are used in forming and decomposing complex numbers, using the rectangular and the polar representations of these numbers.

A number ω may be decomposed into its real and imaginary parts by 9 11○ω. For example:

```
9 11○3J¯4   ↔   3 ¯4
```

Conversely, a pair of real numbers ω representing the real and imaginary parts of a complex number may be formed into that number by ¯9 ¯11+.○ω. For example:

```
¯9 ¯11+.○3 ¯4   ↔   3J¯4
```

A number ω may be decomposed into its magnitude and arc by 10 12○ω. For example:

```
10 12○3J¯4   ↔   5 ¯0.927295
```

The arc is given in radians and is always greater than minus pi radians and less than or equal to pi radians. A positive number has an arc of 0. A negative number has an arc of pi. The arc of 0 is defined to be 0. For example:

```
      12○W
 0.628319
 1.88496
 3.14159
¯1.88496
¯0.628319
```

If *DEG*:(180×ω)÷○1 ⍝ *RADIANS TO DEGREES*, we can display the values of 12○*W* in degrees. For example:

```
      DEG 12○W
  36
 108
 180
¯108
 ¯36
```

Conversely, a pair of real numbers ω, representing the magnitude and arc of a complex number may be formed into that number by ¯10 ¯12×.○ω. For example:

```
¯10 ¯12×.○5 ¯0.927295   ↔   3J¯4
```

Equals and not equals: Two complex numbers are considered equal if the one smaller in magnitude lies on or within a circle whose center is at the one with larger magnitude, and whose radius is equal to ⌈⌷*CT* times the larger magnitude.

Greatest common divisor and least common multiple: A complex integer is one whose real and imaginary parts are integers.

If *A* and *B* are complex integers, there are four other complex integers with the property that they are the largest in magnitude of all the complex integers which divide both *A* and *B*. *A*∨*B* is that one of these which is in the first quadrant, or on the positive axis.

For example, 117J44 and ¯63J¯16 have as greatest divisors the following numbers:

```
      3J¯4  4J3  ¯3J4  ¯4J¯3
```

Of these, 4J3 is given as the value of 117J44∨¯63J¯16 since it is the one in the first quadrant.

A∨*B* is defined for noninteger complex numbers as well:

```
      1.17J0.44∨¯0.63J¯0.16    ↔    0.04J0.03
```

The least common multiple of two complex numbers, α∧ω, is defined by (α×ω)÷α∨ω. For example:

```
      ¯182J¯107∧¯7J55    ↔    ¯75J¯289
```

The least common multiple function is also defined on non-integral complex numbers.

Functions whose domain does not include complex numbers

Because the complex numbers are not ordered, those functions which depend on ordering are not extended to complex numbers. They are the dyadic functions α<ω, α≤ω, α≥ω , α>ω, α⌊ω, α⌈ω, α⍋ω, and α⍒ω, and the monadic functions ⍋ω and ⍒ω.

Functions with deferred extensions

Because we have not agreed on definitions for the functions ⌊ω, ⌈ω, α|ω, and α⍑ω, they have not been extended at this time.

The formatting functions α⎕*FMT*ω and α⍕ω will take complex arguments, but their definitions have not been extended to take note of the imaginary parts of these numbers. Thus they format only the real part. For example:

```
      'BI3'⎕FMT 3J¯4 0J2 0.2 5.2
3

5

      0⍕3J¯4 0J2 0.2 5.2
 3 0 0 5
```

Differences which may affect existing applications

Users should be aware that with this release there are differences in Sharp APL even if complex numbers are not used as arguments. This section describes these differences in detail.

There are two kinds of differences. In the first kind, a function which used to signal a domain error for certain real arguments now has a value which is not in general a real number. The functions in this category are ⍟⍵ and ⍺⍟⍵ for negative arguments, ⍺*⍵ for negative ⍺ and certain ⍵, and ⍺○⍵ for certain arguments. In the second kind, a function now has a different value for certain arguments. There are two cases of this: ⍺*⍵ for negative ⍺ and certain ⍵, and ¯4○⍵. All this is covered in detail below.

In the first case, it seems unlikely that a user program would be affected, since one doesn't ordinarily write a program to cause a domain error to be signalled. However, since Sharp introduced automatic trapping of errors in 1979, it is possible that someone could have written a program in such a way that a domain error used to be signalled by an expression which now has a value. This would, of course, cause the program behavior to be different. We judge this to be low in probability, but nonetheless feel obliged to caution our users, so that they may recognize the situation if it should occur.

We monitored the system extensively to try to judge the impact of the second of these changes (those where a different value is given). Over a one month period in 1979 there were 439 cases of ⍺*⍵ with ⍺<0 and ⍵ not an integer. On investigation, almost all of these uses were by Sharp development people, doing tests. We consulted with the owners of the few remaining workspaces and cautioned them of the impending change. There were only 176 uses of ¯4○⍵, for **all** arguments. This compares with 6,149,426 uses of 1○⍵, for example. Thus in both of these cases where a change in value occurs, we see little risk that any programs will be affected.

First case -- domain error replaced by value: There are four functions which are affected.

1. The monadic logarithm function used to signal a domain error for negative arguments. With complex numbers available we can use the compatible extended definition of logarithm for all numbers except 0:

   ⍟⍵ ↔ (⍟|⍵)+0ⅉ1×12○⍵

   This definition is compatible since for positive arguments, which have arcs of zero, the second term of the sum disappears. We can thus provide a logarithm for arbitrary nonzero complex numbers, and in particular for negative numbers. For a negative number, the imaginary part of its logarithm will be equal to pi, since the arc of a negative number is pi radians.

2. The dyadic logarithm function ⍺⍟⍵ also used to signal a domain error if either argument was negative. Since we can now give a value to the logarithm of a negative number, we can use the definition of ⍺⍟⍵ as (⍟⍵)÷⍟⍺ to give a definition for dyadic logarithms of arbitrary nonzero numbers to arbitrary nonzero base, and in particular to negative numbers and/or to negative bases.

3. The power function ⍺*⍵ used to signal a domain error for ⍺ negative and ⍵ close or equal to a rational number having an even denominator. With complex numbers, such exponents are now permitted, and the result in general is not real. For example, ¯1*0.5 is 0ⅉ1.

4a. The dyadic circle function ⍺○⍵ has had the domain of its left argument extended, as described above, to include the new values ¯12 ¯11 ¯10 ¯9 ¯8 8 9 10 11 12. An attempt to use any of these as a left argument used to give a domain error. These new left arguments are intended primarily for use in forming and decomposing nonreal complex numbers, but they are valid also for real right arguments.

4b. Several of the functions determined by particular left arguments of $\alpha\circ\omega$ have had their domains extended to include more real arguments, as well as having been extended to complex numbers in general. These are `¯7 ¯6 ¯4 ¯2 ¯1 0`$\circ\omega$.

`¯7`$\circ\omega$: Formerly ω had to be strictly between `¯1` and `1`. Now all arguments are valid except `¯1` and `1`.

`¯6`$\circ\omega$: Formerly ω had to be greater than or equal to `1`. Now all values are permitted.

`¯4`$\circ\omega$: Formerly ω could not be strictly between `¯1` and `1`. Now only `0` is prohibited.

`¯2 ¯1 0`$\circ\omega$: Formerly ω had to be between `¯1` and `1`. Now all numbers are permitted.

Second case -- changes in value: There are two functions in this category.

1. The defining expression for the power function $\alpha\star\omega$ is

   $\star\omega\times\circledast\alpha$ $\qquad\qquad$ (A)

   but this could not be used hitherto for negative α since logarithms of negative numbers were not defined. Nonetheless an attempt was made to give an answer, if ω was not close or equal to a rational number with an even denominator, by assuming that in that case the exponent had an **odd** denominator N, and thus had one of its N roots on the negative axis. Up until now we have chosen to give that particular one of the N roots as the value. For example, referring to Figure 1, the elements of Z are approximations to the fifth roots of `¯32`. One of these, `¯2`, is a real number, and is in fact the value that used to be given for the expression `¯32*0.2`. However, now that complex numbers are available we can use the defining expression (A) in every case where α is not zero. This ensures that functions such as `¯32`$\star\omega$ are (except for branch cuts) continuous over their entire domain. For example, the value of `¯32*0.2` is now given as `1.61803J1.17557`. The desired continuity can be seen by noting the closeness of adjacent values in the following example:

```
      3 1ρ¯32*0.1999 0.2 0.2001
1.61784J1.17465
1.61803J1.17557
1.61823J1.17649
```

2. The definition of the `¯4`$\circ\omega$ function has been changed from `(¯1+`ω`*2)*0.5` to $\omega\times$`(1-`ω`*¯2)*0.5`. Effectively, this doesn't change the function for positive arguments, but for negative arguments the value is now negative instead of positive. For example, the value of `¯4 0 2 ¯2` used to be `1.73205 1.73205`. With the new definition the value is `1.73205 ¯1.73205`.

General

The amount of storage required for one complex value is sixteen bytes.

$\square PP$ affects the display of each part of a complex number separately. For example, with $\square PP\leftarrow 3$, we have:

`¯9 ¯11+.○ 1234 0.1234` $\leftrightarrow$ `1.23E3J0.123`

An array whose type is complex can be used with a function which requires a Boolean, integer, or real value as argument if the complex array values are sufficiently close to Boolean, integer, or real values, respectively. The tolerance used in making this determination is not affected by $\square CT$. For example, `3J1E¯20` may be used to index the third element of a vector, or as left argument to replicate.

Acknowledgments

The complex number extension to APL has been discussed in the APL community for many years. Paul Penfield, Jr., of MIT, played a leading role in elucidating the design problems, and made a comprehensive proposal for complex APL in [1]. The Sharp complex APL extension follows this proposal in all details except for the numbering of the new left arguments to $\alpha\circ\omega$ and in output formatting. Professor Penfield also was kind enough to criticize our implementation in the course of its development. The implementation was designed and developed by Doug Forkes and Gene McDonnell.

This SATN benefitted from comments given by Arlene Azzarello, Paul Berry, Caroline Colburn, Doug Forkes, Ken Iverson, and Roland Pesch, of I.P. Sharp Associates.

Reference

[1] Penfield, Paul Jr., 'Proposal for a complex APL', **APL79 Conference Proceedings**, ACM, 1979, pp 47-53.

Bibliography

Most high school algebra texts cover the definitions of addition, subtraction, multiplication, reciprocation, and division on complex numbers. A.M. Gleason's **Fundamentals of Abstract Analysis**, Addison Wesley, 1966, in chapters 10 and 15 covers the construction of the complex number system and the definitions of the exponential, logarithm, power, and trigonometric functions on complex numbers. A more elementary discussion of much of the same material is given in chapter 8 of K.E. Iverson's **Elementary Analysis**, APL Press, Palo Alto. In Milton Abramowitz and Irene Stegun's **Handbook of Mathematical Tables**, Dover, N.Y., 1965, may be found definitions of many of the analytic functions on complex arguments. A detailed exposition of the algorithm behind the complex factorial and complex binomial functions may be found in Hirondo Kuki's 'Complex Gamma Function with Error Control', **CACM 15**, 4, April 1972. The paper by Paul Penfield, Jr., on 'Principal Values and Branch Cuts in Complex APL', to appear in the proceedings of APL81, discusses in complete detail the choices for locations of all branch cuts, direction of continuity of the branch cuts, and values at the end of the branch cuts, for all the analytic functions requiring them. We are obliged to Professor Penfield for providing us with an early draft of this valuable paper.

Those interested in the discussion regarding the extension of the floor, ceiling, residue, and representation functions may read E.E. McDonnell's 'Complex Floor', **APL Congress 73**, North Holland/ American Elsevier, 1973, for one set of definitions, and D.L. Forkes, 'Complex Floor Revisited', to appear in the proceedings of APL81, for a counter-proposal.

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

SATN-41
20 JUN 81

SHARP APL TECHNICAL NOTES

TITLE: COMPOSITION AND ENCLOSURE

AUTHOR: K. E. Iverson

ABSTRACT: This technical note treats the new functions, **enclose, disclose**, and **match** (denoted by <, >, and≡), and three new operators, **on**, **over**, and **with**. In the current release of SHARP APL the domains of these operators are rather restricted; for a discussion of their intended extensions see Bernecky and Iverson [1].

This technical note treats the new functions, **enclose, disclose**, and **match** (denoted by <, >, and ≡), and three new operators, **on**, **over**, and **with**. In the current release of SHARP APL the domains of these operators are rather restricted; for a discussion of their intended extensions see Bernecky and Iverson [1].

One important related extension not in this release is the application of ⍒ to enclosed arrays; as a consequence simple display is not defined, and entry of an expression such as ⎕←<1 2 or <1 2 will produce only the message **ARRAY**.

To facilitate the use of examples in discussion we have provided in workspace 1 *SATN*41 the function *TH* which models the proposed definition of the thorn function ⍒. It will be used freely in what follows.

NEW FUNCTIONS

The function **match** applies to any two arguments to produce a boolean scalar, 0 if the arguments differ in shape or in any element, and 1 otherwise. For example:

```
      MN←0 3 ∘.+ VN←1 2 3
      MC←2 4⍴VC←'ABC'

      MC ≡ VN
0
      VN ≡ VN*1
1
      '' ≡ ⍳0
1
```

The **enclose** function applies to any array to produce a result of rank zero; the **disclose** function is its inverse in the sense that *X*≡><*X*. The enclose function is **strict** in the sense that <*X* differs from *X*. For example:

```
      VN≡<VN
0
     B≡<B←<MC
0
```

Catenation applies to enclosed arguments, but not to one enclosed argument and one simple argument. The extension of other structural and selection functions is based on this extension of catenation. For example:

```
      VN,<VN
DOMAIN ERROR
      VN,<VN
      ∧
```

```
      ⍴P←(<MN),(<VN),(<MC),(<VC)
4
      TH Q←3 2⍴P
1 2 31 2 3
4 5 6
ABCA ABC
BCAB
1 2 31 2 3
4 5 6
      (0 1 2⌽Q)[;1] ≡ (<MN),(<VC),<MN
1
```

The extension of the 'comparison' functions $\in$ and ι is based upon the match function rather than upon equality. Thus:

```
      (<VN)∈Q
1
      P⍳<VN
2
```

The remaining primitive functions are not extended to enclosed arrays. Thus *VN*+<*VN* and (<*VN*)+<*VN* yield domain errors.

The argument rank of the disclose function is zero (i.e., it applies to each of the scalars in its argument), and >*A* therefore produces an array of shape (⍴*A*),*S* where *S* is the (necessarily common) shape of the results produced by disclosing each element of *A*. For example:

```
      ⍴R←>Q[1 3;1]
2 2 3
      R
1 2 3
4 5 6

1 2 3
4 5 6
      >Q[1;]
DOMAIN ERROR
      >Q[1;]
      ∧
      >Q[;1]
DOMAIN ERROR
      >Q[;1]
      ∧
```

The expressions >*Q*[1;] and >*Q*[;1] yield domain errors for different reasons, the first because the disclosed arrays differ in shape, and the second because they differ in type.

The disclose function is **permissive**, i.e., when applied to a simple argument (which has no enclosed elements) it behaves as an identity function and yields its argument. Thus:

```
      R ≡ >R←2 3⍴⍳6
1
```

The restriction of catenation to apply only to two numeric or to two character arguments prohibited the introduction of 'non-homogeneous' arrays; the present extension of catenation to two enclosed arguments only likewise prohibits non-homogeneous arrays. As a consequence, a function which determines whether its argument is simple may be defined rather plainly as follows:

```
      SIMPLE: X ≡ >X←1⍴⍵: 0∈⍴⍵: 1
```

In order to avoid (or at least defer) the introduction of further 'fill elements' for the functions **take** and **expand**, and the introduction of further dependence on the 'type' of an empty argument, we have introduced a *NONCE ERROR* for the overtake or expansion of enclosed arrays. For example:

```
      3↑<1 2
NONCE ERROR
      3↑< 1 2
        ∧
```

COMPOSITION OPERATORS

Three new operators have been introduced, called **on**, **over**, and **with**; they are denoted by the symbols ⍤,⍥, and ⍣, the first two of which are sometimes given the nicknames **paw** and **hoof**.

The dyadic operator⍤ applies to two functions in the form *F*⍤*G* and may be read as *F* **on** (the result of) *G*. The resulting derived function is ambivalent and may therefore be used in the dyadic form *X F*⍤*G Y* as well as the monadic form *F*⍤*G Y*.

The monadic case is similar to composition in mathematics, which is defined by the expression *F G Y*. This is the definition adopted here for the case that *Y* is a single argument of the function *G*. However, for an array of such arguments we insist that the composition is **close** in the sense that the expression *F G* is applied individually to each subarray argument of *G*, and the overall result is formed (as in the example of the disclose function given earlier) by placing the axes of the individual results last.

The significance of close composition may be best appreciated by comparing the results of *F*⍤*G X* and *F G X* in a few examples:

```
      TH FONTS←(<2 3⍴'ABCDEF'),<2 3⍴'123456'
ABC123
DEF456
      >FONTS
ABC
DEF

123
456
      ⌽>FONTS
A1
D4

B2
E5
```

```
C3
F6
      ⍉⍤⍥>FONTS
AD
BE
CF

14
25
36
      ⍴ >FONTS
2 2 3
      ⍴⍤⍥>FONTS
2 3
2 3
      , >FONTS
ABCDEF123456
      ,⍤⍥>FONTS
ABCDEF
123456
```

The dyadic case of the derived function $F⍤⍥G$ is defined similarly by the identity:

```
      A F⍤⍥G B ↔ (G A) F (G B)
```

again with the understanding that the composition is **close**, and that the identity applies strictly only if A and B are single arguments of the function G. For example:

```
      4 ⍴ + 1 2 3
1 2 3 1
      4 ⍴⍤⍥+ 1 2 3
1 1 1 1
2 2 2 2
3 3 3 3
      X←<1 2 3
      X ×⍤⍥> X
1 4 9
      X×>X
DOMAIN ERROR
      X×>X
      ∧
```

Even with the present restrictions which preclude composition with derived and user-defined functions, the user will find many interesting applications of composition. For example, ≡⍤⍥⍴ compares the shapes of its arguments; ,⍤⍥< is the **pair** function that encloses each of its arguments and catenates them; and summing and 'padding' of each of the enclosed vectors in an array may be performed as follows:

```
      M←2 2⍴(<2 3),(<5 7 11),(<⍳0),(<2 3 4 5)

      1 ⊥⍤⍥> M
5 23
0 14
```

```
      (⌈/,ρ⍤⍥> M) ↑⍤⍥> M
2  3  0  0
5  7 11  0

0  0  0  0
2  3  4  5
```

The operator ⍤⍥ provides another form of close composition in which the dyadic case is defined by the identity:

```
      A F⍤⍥G B ↔ F A G B
```

For example:

```
      N←2 4ρ⍳4,M←2 4ρ1 2 3 4 5 6 7 8

      M |⍤⍥- N
7 5 3 1
1 3 5 7

      TH R←0 <⍤⍥+ M
1234
5678
      R
**ARRAY**
      ρR
2 4
      >R
1 2 3 4
5 6 7 8
```

In the derived function $F⍤⍥G$ the function G is used dyadically, whereas in $F⍥⍤G$ it is used monadically; the association of the larger symbol (⍤) with the larger valence may prove useful as a mnemonic aid. The monadic cases of the derived functions produced by ⍥ and ⍤ are identical, that is:

```
      F⍥⍤G Y ↔ F⍤⍥G Y
```

Except that it is also based on close composition, the monadic case of the derived function $F⍨G$ (read as F **with** G) is equivalent to the mathematical notion of the dual of F **with** respect to G, that is:

```
      F⍨G Y ↔ GI F G Y
```

where GI is the inverse of G.

The importance of the basis in close composition is particularly clear in the case of $F⍨>$ (the dual with disclose), because whatever the shapes or ranks of the arguments and results of F, the function $F⍨>$ is, in effect, a scalar function. Applied to an argument A, the function $F⍨>A$ produces a result of the same shape as A in which each element is the enclose of the result of F applied to the disclose of the corresponding element of A. For example:

```
      ρQ
3 2
      TH Q
1 2 31 2 3
4 5 6
```

```
ABCA ABC
BCAB
1 2 31 2 3
4 5 6
      ρSH←ρ¨>Q
3 2

      >SH[;1]
2 3
2 4
2 3
      >SH[;2]
3
3
3

      ρW←⍳¨> 2 3ρ1 2 3 4 5 6
2 3
      TH W
1       1 2       1 2 3
1 2 3 41 2 3 4 51 2 3 4 5 6

      1 ⊥⍤1> W
 1  3  6
10 15 21

      * 1 ⊥⍤1> ⊛¨> W
 1   2   6
24 120 720
```

The definition of the dyadic derived function provided by $F\ddot{}G$ is similar to the monadic case. Thus:

```
      X F¨G Y ↔ GI (G X) F (G Y)
```

For example, since the inverses of ⋆, ~, and < are ⊛, ~, and >, we have a number of identities suggested by the following examples:

```
      V +¨⊛ V←1 2 3 4
1 4 9 16
      V×V
1 4 9 16
      V ×¨⋆ V
2 4 6 8
      V+V
2 4 6 8

      A ∨¨~ B←⍉A←2 2ρ0 0 1 1
0 0
0 1
      A ∧ B
0 0
0 1
      A ∧¨~ B
0 1
```

```
1 1
      A∨B
0 1
1 1

      A ,⍤< B
0 0
1 1


0 1
0 1
      A ,[⎕IO-.5] B
0 0
1 1


0 1
0 1
```

RESTRICTIONS ON ARGUMENTS OF OPERATORS

The utility of the three operators will be greatly enhanced when they are extended in the manner discussed in [1], and are allowed to apply to arrays and to derived and user-defined functions as well as to primitives. In the present release they apply to primitives only, and are further restricted as follows:

a) The right argument of ⍤ (with) is limited to the following functions: ⍁ < > ~ ÷ ⋆ ⊛ - +

b) The right argument of ⍥ for the monadic case, and of ⍤⍥ for either case may be any scalar primitive, or one of the following: ⍁ < > , ⍋ ⍒ ⍪ ⍴

c) The right argument for ⍥ for the dyadic case may be any dyadic scalar primitive.

d) Because the symbols / and \ denote **operators**, an expression such as `1 0 1/X` is the application to *X* of the monadic derived function `1 0 1/` rather than the application of a dyadic function denoted by /. Since the composition operators apply to functions only, compositions of the form *F*⍤⍥/ and /⍤⍥*F* cannot be used.

e) All system functions and the function ⍎ are excluded from use as arguments to operators.

Bibliography

[1] K.E. Iverson and R. Bernecky, 'Operators and Enclosed Arrays', Proceedings, I.P. Sharp Associates 1980 Users Meeting.

TITLE: **Determinant-Like Functions Produced By The Dot Operator**

AUTHOR: K.E. Iverson

ABSTRACT: The operator denoted by the dot has been extended to provide a monadic function (as in `-.×M`) as well as the established dyadic inner product function (as in `N+.×M`). The case `-.×M` (on a square matrix `M`) is the familiar determinant function, and the general case `F.G M` is defined analogously as reduction by `F` over a set of `!N←1↑⍴M` "products" produced by reduction by `G` over `!N` distinct sets of `N` elements chosen one from each row and column of the argument. The extension to non-square arguments allows the use of `-.×` on an `N+1` by `N` argument to compute the signed volume of the simplex it represents in `N`-space.

The determinant is an important function defined on square matrices -- applied to a matrix of coefficients of a set of linear equations it determines the character of the solutions, applied to the matrix of partial derivatives of a vector function on vector arguments it yields the volume transformation effected by the function, and applied to the matrix `S,1` it yields `!1↓⍴S` times the signed volume of the simplex of `N←1↑⍴S` vertices in `(N-1)`-space.

The determinant of a square matrix `M` is defined as the alternating sum (i.e., reduction by `-`) of the `!N←1↑⍴M` products over `N` elements chosen (in each of the `!N` possible ways) one from each row and column. Analogous calculations in which other function pairs are substituted for `-` and `×` lead to other useful functions; examples include the pairs `⌊⌈`, `∧∨`, and `+×`, the last (called the **permanent**) being useful in combinatorics [1].

Because of the utility of cases other than the pair `-×`, we introduce a dyadic operator instead of a specific determinant function; because the result of the dot operator is defined only for the dyadic case of the resulting function (as in `A+.×B` and `A∧.=B`), we adopt its monadic case for the "determinant" operator. Thus `-.×B` is the determinant of `B`, and `+.×B` is the permanent.

FORMAL DEFINITION OF *F.G B*

The function *DOP* in workspace 1 *SATN42* models the determinant operator. For example, if *M* is a magic square of order 3, then:

```
      M
6 7 2
1 5 9
8 3 4

      -.×M                    '-×'DOP M
360                     360

      +.×M                    '+×'DOP M
900                     900

      ∨.∧M                    '∨∧'DOP M
2                       2
```

DOP and its supporting functions are displayed below; the subsequent comments may be helpful in grasping their details. Origin 1 is assumed throughout:

```
DOP◇(1↑⍺)CR ⍠(¯1↑⍺),'/X',0⍴X←ALL ⍵

ALL◇((⍳K),K←⍴⍴Y)⍴⍵[Y←PERM⍴⍵;]

PERM◇(S,⍴S)⍴⍴MF 1+S⊤¯1+⍳×/S←(⌊/⍵)↑⌽⍳1↑⍵

MF◇0≠1↑⍴⍵◇⍵◇⍵[,1;],[1]X+⍵[(1↑⍴X)⍴1;]≤X←MF 1 0↓⍵

CR◇0≠⍴⍴X←⍵◇⍵◇⍺CR⍠⍺,'/X'
```

The diagonal section `1 1 ⍉M` contains one element from each row and column, and is therefore one of the sets of elements over which the reduction *C/* is to be applied in evaluating *F.G M* . All such sets can be obtained from the expression `1 1⍉M[P;]`, where *P* is one of the *!N* permutations of order *N*←`1↑⍴M`. Consequently, if *AP* is an array of all permutations of order *N*, then a suitable section of *M[AP;]* provides all the required sets of elements, and *C/* over this section produces an array of all "products" to which reduction by *F* is to be applied. For example:

```
      □←AP←PERM ⍴M
1 2 3
1 3 2

2 1 3
2 3 1

3 1 2
3 2 1

      ⍴AP
3 2 1 3

      □←PR←×/R
120
162

 28
  6

504
 80
```

```
      □←R←ALL M
6 5 4
6 3 9

1 7 4
1 3 2

8 7 9
8 5 2
```

```
      -/PR
120 162
 28   6
504  80

      -/-/PR
¯42 22 424

      -/-/-/PR
360
```

The permutations *AP* are lexical in order, but the outer shape of the array is *⍳!N* rather than *!N* , and the shape of the array of products is therefore *⍳!N* as well. Reduction by *F* is therefore carried out over a succession of axes (beginning with the last) as provided by the recursively-defined function *CR*. The reader may wish to verify that the pattern of reduction provided by this argument of rank *N* ensures that the function *F* applies over the products in an order such that the permutations which produced them alternate in parity. This ordering is significant for any non-associative function *F*; for the case -.× it ensures agreement with the established definition of the determinant.

Except for one important case, the extension of determinant functions to non-square arguments may not greatly increase their utility. However, the extension is straightforward; if $>/\rho M$, then $\times/S\leftarrow(1\downarrow\rho M)\uparrow\phi\iota 1\uparrow\rho M$ distinct sets of $1\downarrow\rho M$ elements can be selected without repetition of either column or row, and these are arranged in lexical order in an array of shape S. For example:

```
      □←M←?4 2⍴9
6 4
5 5
4 3
1 4

      □←AP←PERM ⍴M
1 2
1 3
1 4

2 1
2 3
2 4

3 1
3 2
3 4

4 1
4 2
4 3

      □←R←ALL M
6 5
6 3
6 4

5 4
5 3
5 4

4 4
4 5
4 4

1 4
1 5
1 3

      □←PR←×/R
30 18 24
20 15 20
16 20 16
 4  5  3
      -/-/R
0 0 1 ¯1
      -/-/PR
21
      -.×M
21
```

For a wide matrix M, the result of $F.G\ M$ is equivalent to $F.G\ (\lfloor\backslash\rho M)\uparrow M$.

The case of -.× applied to an $N+1$ by N matrix is of special interest because it is equivalent to $-.\times M,1$, and can therefore be used to compute the volume of the simplex defined by M without explicit catenation of a final column of ones.

EVALUATION METHODS

Because of the special properties of the ordinary determinant, the function -.× on a square matrix is treated specially and is evaluated by the process used in matrix inversion; the time taken therefore varies as the cube of the order of the matrix.

All other cases are treated by explicit evaluation of each of the ÷/!|-\ρ*M* sets of ¯1↑ρ*M* elements; the time therefore varies as the product of the foregoing quantities.

Although the model shows the computation of an aray *A* of "products" of the form *G/V* followed by repeated reductions *F/...F/A*, the processes are merged so that the reductions are effected as early as possible. Consequently the number of intermediate results recorded in the reductions does not exceed ρρ*A*. The scheme can be illustrated by evaluating -/-/*A* for a matrix *A*, beginning at the right of the last row, keeping the result of its reduction while reducing the penultimate row, and then applying reduction (the one applied to the result of -/*A*) to these two results before proceeding with the reduction of another row.

COMMENTS ON USES OF *F.G M*

Applications of the ordinary determinant (-.×) on square matrices will not be discussed except to remark that it may be used as a test for invertibility before applying the function ⌹.

If *A*, *B*, and *C* are two-element vectors, then -.×*M*←3 2ρ*A*,*B*,*C* yields twice the signed area of the triangle with vertices *A*, *B*, and *C*, **signed** because the sign of the result depends on the order of the vertices, positive if they are in counterclockwise order, and negative if in clockwise order. For example, if *A*←1 1 and *B*←1 0 and *C*←0 1 then:

```
      -.×3 2ρA,B,C
¯1

      -.×3 2ρB,A,C
1
```

Questions concerning the position of point *A* in relation to the line *BC* are more easily posed and answered in terms of the ordering of the points (i.e., in terms of the signed area) than in terms such as "above *BC*" (or should it be "left of *BC*" if *BC* is nearly vertical), or "*A* lies on the line *BC*" (for example, does it lie on the line if *B* and *C* are coincident?).

The signed result of -.× is significant in higher dimensions as well; if *A*, *B*, *C*, and *D* are points in three-space (in a dextral coordinate system), then -.× 4 3ρ*A*,*B*,*C*,*D* gives (!3 times) the signed volume of the tetrahedron, the results being zero if the

points are coplanar, and positive if the points *B*, *C* and *D* are in counterclockwise order when viewed from *A*. For example:

```
      A←1 1 1
      B←1 0 0
      C←0 1 0
      D←0 0 1

      -.×4 3⍴A,B,C,D
2

      -.×4 3⍴A,C,B,D
¯2
```

We will illustrate the use of functions other than - and × in the context of the so-called assignment problem: one of *N* persons (or machines) is to be assigned to each one of *N* tasks in some optimal way, the cost associated with assigning person *I* to task *J* being given by element *M*[*I*;*J*] of the square matrix *M*.

If we wish to minimize the sum of the costs in the assignment, then the optimum value is given by ⌊.+*M*, since the summation (+/) is applied to each of the !1↑⍴*M* possible assignments, and the minimum (⌊/) is applied over them. Other useful function pairs in assignment type problems include ⌈+, ⌊×, ⌈×, ⌊⌈, and ⌈⌊.

A square boolean matrix *B* is said to be a **Latin square** if it contains exactly 1↑⍴*B* ones, with one in each row and one in each column; one boolean matrix *C* is said to cover another, *D*, if ∧/,*C*≥*D*. The expression ∨.∧*C* yields 1 if *C* covers a Latin square, and the expression +.∧*C* tells how many distinct Latin squares are covered by *C*.

ACKNOWLEDGEMENTS

The system programming for this extension to the *APL* system was done by D.B. Allen. I am grateful to Mr. Allen and to Mr. E.E. McDonnell for comments on a draft of this paper.

REFERENCES

[1] Ryser, H.J., **Combinatorial Mathematics**, Carus Mathematical Monograph Number 14, Mathematical Association of America, distributed by John Wiley and Sons, Inc., Providence, R.I., 1963.

5 November 1982

TITLE: IMPROVED DISPLAY FOR ENCLOSED ARRAYS AND A NEW SYSTEM VARIABLE ⎕*PS*

AUTHOR: Peter Wooster

Monadic thorn (⍕) and the system default display (⎕←) now format enclosed arrays. The definition is based on a proposal by Jenkins and Michel [3]. It has been extended to include an option to draw boxes around enclosed subarrays, as used by Ghandour and Mezei [2].

⍕ converts each subarray to a character representation. The character arrays created at each level are catenated to form a rectangular array of rank 2 or greater, preserving all but the last two elements of the shape vector of the subarray, i.e., `(¯2↓⍴⍕⍵) ≡ ¯2↓⍴⍵` .

The system variable ⎕*PS* is a four-element vector that controls the positioning and spacing of enclosed arrays for both default display and monadic ⍕:

Positioning		**Spacing**	
⎕*PS*[0]	⎕*PS*[1]	⎕*PS*[2]	⎕*PS*[3]
TOP ¯1	*LEFT* ¯1	*NO. OF BLK ROWS BETWEEN SUBARRAYS*	*NO. OF BLK COLUMNS BETWEEN SUBARRAYS*
CENTRE 0	*CENTRE* 0		
BOTTOM 1	*RIGHT* 1		

		IF ≤ ¯2, *BOXES ARE DRAWN*	

For example:

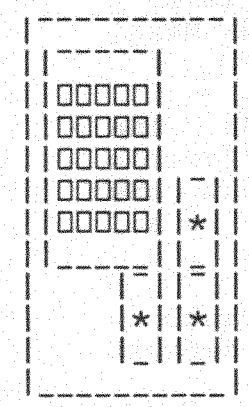
```
      A←<2 2⍴(<5 5⍴'⎕'),3⍴<'*'

      ⎕PS←¯1 ¯1 0 1 ⍝ DEFAULT ⎕PS

      A
```

```
⎕⎕⎕⎕⎕ *        ⍝ NOTE:   TOP LEFT POSITION
⎕⎕⎕⎕⎕                    NO BLANK ROWS INSERTED
⎕⎕⎕⎕⎕                    ONE BLANK COLUMN INSERTED
⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕
*     *
```

```
      ⎕PS←0

      A

⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕*
⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕
  *  *
```

A *NOTE:* *CENTERED ROWS AND COLUMNS*
NO BLANK ROWS OR COLUMNS

```
      ⎕PS←1 1 ¯2 ¯2

      A
```



A *NOTE:* *LOWER RIGHT POSITION*
NO BLANK ROWS OR COLUMNS
BOXES ARE DRAWN IN THE TWO SPACES BETWEEN ARRAYS

As shown above, each subarray will be placed in one of nine positions within its "print box".

Any number of spaces may be inserted between the subarrays of the result, and boxes showing the structure may be drawn around subarrays. If they are drawn, the boxes enclose non-simple scalars, and the lines are drawn in the space between subarrays. By convention, when ⎕*PS*[2 3] is negative, the magnitude specifies the amount of spacing. The negative sign causes non-simple scalars to be boxed, but only when ⎕*PS*[2 3]≤¯2, since two spaces are needed for the box. ¯1 does not do boxes.

Model Of Enclosed Array Thorn

The following model of ⍣ uses the variable *QPS* for ⎕*PS*.

```
    ∇ Z←THORN D;R;C;A;⎕IO;W;F;E;I;J;K;G;B;T;NPS;R;B1;BXS
[1]   ⎕IO←1
[2]   NPS←¯1¯1 0 1
[3]   →(0=⎕NC 'QPS')⍴P0
[4]   NPS←4⍴QPS
[5]   P0:BXS←¯2≥¯2↑NPS
[6]   NPS[3 4]←|NPS[3 4]+2×BXS
[7]   →(A≡>A←1↑,D)/LS
[8]   I←''⍴⍴F←''
[9]   A←⍴D←((-2⌈⍴⍴D)↑ 1 1 ,⍴D)⍴D
[10]  A←(×/W←¯2↓A),¯2↑A
[11]  D←,D
[12] L3:→((I←I+1)>⍴D)⍴L3Z
[13]  B←((-2⌈⍴⍴B)↑ 1 1 ,⍴B)⍴B←THORN>D[I]
[14]  E←((⍳¯1↑T)∈T←+\1+¯1↓0,+⌿0=T∘.|⍳J)⍀
            ((J←¯1↑T←×\1↓T),1↑T←⍬⍴B)⍴B
[15]  E←(B1⍴'¯'),[1] E,[1](B1←BXS[1],¯1↑⍴E)⍴'_'
[16]  E←(B1⍴'|'),E,(B1←(1↑⍴E),BXS[2])⍴'|'
[17]  F←F,<E
[18]  →L3
[19] L3Z:Z←(J←0)⍴D←((B←×/2↑A),¯1↑A)⍴F
[20]  C←NPS[4]+0⌈⌈≠⌈≠A⍴(F←⍴⍤>F)[;2]
[21]  R←NPS[3]+0⌈⌈/⌈≠A⍴F[;1]
[22] L1:→(B<J←J+1)⍴L2Z
[23]  G←((K←R[1+A[2]|¯1+J]),I←0)⍴''
[24] L0:→(A[3]<I←I+1)⍴L0Z
[25]  E←>D[J;I]
[26]  G←G,(K,C[I])↑(-(⍴E)+⌊((0≤NPS[1 2])×
            (K,C[I])-NPS[3 4]+⍴E)÷2⋆0=NPS[1 2])↑E
[27]  →L0
[28] L0Z:Z←Z,,G
[29]  →L1
[30] L2Z:Z←(W,(+/R),+/C)⍴Z
[31]  Z←((-⍴⍴Z)↑-¯2↑NPS)↓Z
[32]  →0
[33] LS:Z←⍣D
    ∇
```

Folding Function

Output often exceeds ⎕*PW* in width. L.M. Breed has proposed a matrix oriented folding scheme suitable for "cut and paste" assembly. The following function provides this folded display:

```
    ∇ E←FOLD B;T;J
[1]   B←((-2⌈⍴⍴B)↑ 1 1 ,⍴B)⍴B
[2]   E←((⍳¯1↑T)∈T←+\1+¯1↓0,+⌿0=T∘.|⍳J)⍀
            ((J←¯1↑T←×\1↓T),1↑T←⍬⍴B)⍴B
[3]   E←(⎕IO+ 1 0 2)⍉((1↑⍴E),T,⎕PW)⍴((1↑⍴E),⎕PW×T←⌈(¯1↑⍴E)÷⎕PW)↑E
    ∇
```

For example:

```
      ⎕PW←60
      ⎕←⍳¨>⍳5
┌─┬──┬─────┬───────┬─────────┐
│1│1 2│1 2 3│1 2 3 4│1 2 3 4 5│
└─┴──┴─────┴───────┴─────────┘

      ⎕PW←30
      ⎕←⍳¨>⍳5
┌─┬──┬─────┬───────┬─────
│1│1 2│1 2 3│1 2 3 4│1 2 3
└─┴──┴─────┴───────┴─────

      ⎕←FOLD ⍣⍳¨>⍳5
┌─┬──┬─────┬───────┬─────
│1│1 2│1 2 3│1 2 3 4│1 2 3
└─┴──┴─────┴───────┴─────

┌────┐
│4 5│
└────┘
```

References

[1] Breed, L.M. IBM Corporation, Palo Alto, Ca., personal communication.

[2] Ghandour, Z. and J. Mezei. "General Arrays, Operators, and Functions", **IBM Journal of Research and Development**, IBM Corporation, July 1973, pp. 335-352.

[3] Jenkins, M.A. Queen's University, Kingston, Ontario, and J. Michel, Universite Paris XI (Orsay). **Operators in APL with Nested Arrays: A System 1 View.**

2 April 1983

TITLE: Enhancements to Event Handling

AUTHOR: Robert C. Metzger

ABSTRACT: Numerous enhancements have been made to the event handling facility in the May 1982 and November 1982 releases of SHARP APL. Some make existing features easier to use and others provide new capabilities. The following topics are covered.

- Omitting event numbers
- Using event ranges
- Extensions of the *D* action code
- The action code qualifier *O*
- Changes to error messages
- The action code *I*
- The system variable ⎕*EC*

I. OMITTING EVENT NUMBERS

You may now write trap definitions which do not include any event numbers.

Definition:

If you write a trap definition using action codes <*C*>, <*E*>, <*N*>, <*S*>, or <*D*>, you may omit the list of event numbers. If you do this, the trap definition will apply to all events covered by that action code. The classes currently covered by the available action codes are listed below.

Action Code	Event Classes	Note
E, *C*	0, 1000	
N, *S*	0, 1000, 2000	
D	2001, 0, 1000	(see item III below)
I	6	(see item VI below)

(Currently, you may **not** omit event number 6 when using the *I* action code.)

Application:

A) The events covered by an action code may change in the future. In particular, new classes of events may be added. If you use this new shorthand notation when writing universal traps, new event classes will automatically be covered. For example, ⎕*TRAP*←'∇ 0 1000 2000 *S*' will **not** cover events which might be put in a 3000 class, but ⎕*TRAP*←'∇ *S*' will, if *S* is a relevant action for 3000-type events.

B) Since many traps are written to cover all possible events, this enhancement provides a useful shorthand notation. A trap like ⎕*TRAP*←'∇ 0 1000 *E* →*TRAP*' can now be written as ⎕*TRAP*←'∇ *E* →*TRAP*'. Since a trap must be checked for proper syntax every time it is assigned, the shorter the trap is, the less CPU time is consumed in analyzing it.

II. USING EVENT RANGES

You may now write trap definitions which include ranges of events. Ranges work similarly to event classes, but are written in a different way.

Definition:

A range of event numbers is a group of events which is treated as a unit. All the events belonging to a range do not have to be written explicitly, because they are determined by the lowest and highest event numbers in the range.

A range of event numbers is written as follows: an integer, followed by a hyphen, followed by another integer which is greater than or equal to the first.

An event range may be included in the event list of a trap definition as if it were a single event number or class. The range specifies that all event numbers, from the lower bound to the upper bound inclusive, should be covered by that trap definition. Thus, *⎕TRAP←'∇ 1 15 21 31-34 72-76 E →RESOURCE'* is a valid trap.

Note that ranges may not cross classes (i.e. 0-1001 is not valid). Also, ranges which include a class number (such as 0-3) are interpreted as if the class number were the first event number in that class. This means that range 0-3 is equivalent to 1-3

The best way to avoid confusion is not to use class numbers as the end points of ranges.

Application:

A) Ranges can be quite useful in handling user-defined events. For example, perhaps you are using an application package which uses *⎕SIGNAL* to report user errors back to the global environment. If you want to intercept all of these events, you can specify a range, perhaps 601-699, instead of having to list all of the possible event numbers (which may change as the application does).

B) Once again, this shorthand notation makes it possible to write simpler traps, which cost less CPU time to analyze.

III. EXTENSIONS OF THE *D* ACTION CODE

You can now use action code *D* with event classes 0 and 1000.

Definition:

In the past, the action code *D* was limited to use only with event 2001. It can now be used with event classes 0 and 1000, and/or any individual events within these classes.

This action code still takes one of two keyword parameters (*CLEAR* or *EXIT*), or an empty trap line, rather than a line of APL code. If the keyword *CLEAR* is used, when an error or interrupt is intercepted, the workspace will immediately be cleared. If the keyword *EXIT* is used, when an error or interrupt is intercepted, the event is propagated to the next level in the State Indicator stack beyond the level where the trap was localized. After this

happens, the event may be trapped by another trap expression. If this occurs, the process will be repeated.

Application:

If you are writing an application whose inner workings should not be available for inspection by the user, you may already be using a trap like *⎕TRAP←'∇ 2001 D CLEAR'*. This will cause the workspace to be cleared whenever APL is about to return to immediate execution mode. Of course, such a return can be caused by such diverse events as an error, an interrupt, or successful completion of a function.

Return to immediate execution is really an effect of one of the events mentioned above occurring. By allowing you to clear the workspace when the original event occurs, you have more flexibility. You might write a trap like the following, which logs errors and programmed stops, but clears out users who hit attention.

```
⎕TRAP←'∇ 0 1001 E ERRLOG ∇ 1000 D CLEAR '
```

IV. THE ACTION CODE QUALIFIER *O*

You can now specify that a trap definition is applicable only in its own environment on the State Indicator stack.

Definition:

The action code qualifier *O* must be used with another action code like *E*. Its purpose is to modify the way the trap it is contained in will be applied. The letter *O* was chosen as the action code for this qualifier because it makes the trap applicable *ONLY* in the environment in which the trap is set.

The own environment always includes the SI level which correponds to the function in which the trap was localized. It may also include other levels of the SI stack, which were created by that defined function. These would be labelled ⍎, ⎕, or *⎕TRAP*. The own environment will never include more than one SI level corresponding to a defined function.

The letter *O* must be written in the action code field, that is, after the event number list, and before the trap line or keyword. Place the *O* qualifier before the action code, and separate them with a blank.

Application:

Sometimes you want to set a trap in a function that will not be used by its subfunctions. In the past, to accomplish this, you had to make sure that each subfunction explicitly overrode the trap. Now you simply include the *O* qualifier in the trap set by the calling function.

If you need a trap which covers *WS FULL* for the function which set it and all of its subfunctions, but covers all other errors only for itself, you can write the following.

```
⎕TRAP←'∇ 1 C →WSFULL ∇ 2-499 O E →ERROR'
```

V. CHANGES TO ERROR MESSAGES

Error messages related to names and values are now more consistent.

Definition:

The error message and event number reported when a reference is made to a name which has no related value is now always *VALUE ERROR*. In the past, the message produced was *VALUE ERROR* if the name was used as if it referred to a variable or niladic function. But if it was used as if it referred to a monadic or dyadic function, the message was *SYNTAX ERROR*.

VALUE ERROR (event 6) now means that the name referred to is not related to any object accessible from the SI stack level where the reference was made.

SYNTAX ERROR (event 2) will still be reported when a statement does not conform to the rules of APL syntax.

RESULT ERROR (event 8) is reported when an expression which does not return a result is used in a context which requires a result. In the past, if you used a defined function or a primitive function which does not return a result anywhere except as the left-most function in a statement (like 1+'*ABC*' ⎕*APPEND* 1), you would get a *VALUE ERROR*. Now you get a *RESULT ERROR*.

Application:

A) Debugging APL programs will be easier now that the error messages more accurately reflect the error made by the programmer.

B) Generalized workspace management software will be easier to build now that a reference to a non-existent object always gives the same error message (see item below).

VI. THE ACTION CODE *I*

You can now execute event handling code in the mid-line environment of an error and return to the exact point in a line where the error occurred.

Definition:

A new action code *I* has been added to the ⎕*TRAP* facility. This code makes it possible to execute recovery statements and return to the same position on the line where the error occurred. For now, the *I* code only applies to event 6 (*VALUE ERROR*). In the future, it may be extended to apply to other events.

This new action code complements the existing *C* and *E* action codes. The relationship is as follows.

Action Code	Environment in which the trap line is executed	How to return to the line on which the event occurred
C	SI level at which trap is localized	impossible
E	SI level at which the event occurred	→⎕*LC*
I	SI level at which the event occurred	automatic

The difference in environments between the *E* and *I* codes involves what the system does with information related to the evaluation of the line that failed. If you use the *E* code, this information is discarded. If you use the *I* code, it is retained, so that the line may be restarted at the point where the error occurred. You can think of the *E* trap statements as executing 'between lines', whereas the *I* trap statements execute 'in mid-line'.

If a trap definition containing the *I* action code is activated, the trap line is executed at the precise point of error. The APL system adds a new level to the State Indicator stack. This new level is labelled ⎕*TRAP*, and can be seen by doing the following.

```
      ∇ FOO;⎕TRAP;UNKNOWN
[1]   ⎕TRAP←'∇ 6 I ⎕←2 ⎕WS 2 ⋄ UNKNOWN←2'
[2]   1+UNKNOWN+3 ∇
      FOO
⎕TRAP
FOO[2]
6
```

This new SI stack level is treated like a level caused by using the execute function. The difference is that if a value is created when executing the trap line, it is not passed back to the original environment. It is simply printed and discarded. Values created by execute **are** passed back to the original environment. As with execute, if the trap line contains a branch statement, it will branch in the SI environment one level down -- in the function in which the event occurred. If a trap line associated with an *I* action code executes a branch, the system **will not** return to the mid-line point of error. The *I* action code will end up causing behavior identical to the *E* action code in this case. The following example shows how this works.

```
      ∇FOO;⎕TRAP;UNKNOWN
[1]   ⎕TRAP←'∇ 6 I UNKNOWN←2 ⋄ →LABEL'
[2]   ⎕←1+UNKNOWN+3 ⋄ ⎕←'HERE'
[3] LABEL:⎕←'DONE' ∇
      FOO
DONE
```

When a trap definition containing an *I* action code is activated because of a *VALUE ERROR*, an additional row is appended to ⎕*ER*. This fourth row contains the name which has no object associated with it. This name is inserted even if the function in which the event occurred is locked.

If the trap line has not remedied the problem (i.e. the name still does not refer to an object), when the function resumes on the same line, the event will be signalled again. This time no trap for the same event will be activated.

Application:

SHARP APL currently runs in fixed-size workspaces. While this maximizes system performance, it means that large applications sometimes need to store their functions on a file, in order to provide adequate space for computations. This technique has been used on SHARP APL for many years, but the *I* action code makes it much easier to implement. The reason is that in the past, functions had to be explicitly retrieved by the routines that called them. Now retrieval can be done implicitly, by simply assigning a global ⎕*TRAP*.

The global ⎕*TRAP* would look like the following.

```
⎕TRAP←'∇ 6 I GETOBJ ⎕ER[3+⎕IO;]'
```

Now, you need a function to bring in the object which caused the value error in the first place.

```
     ∇ GETOBJ NAME
[1]  NAME FETCHOBJ FINDOBJ NAME ∇
```

This function calls two subfunctions to do its work, rather than doing it directly, because the way you locate and retrieve objects will vary with the application. *FINDOBJ* returns the component number where the object was stored on file. *FETCHOBJ* reads the component and defines the object into the active workspace.

You might use a directory in the first component of the function file to locate the object. A different approach is to compute the location of the object based on its name, rather than look it up in a table. Such transformations are often called **hashing**. This approach can be quicker than the first for lookup. Since it could potentially store many functions in a single component, it could also be slower for retrieval, since many unwanted functions would also be read in. It may also be infeasible in some cases, if a large number of names in the application map into a few components.

Using the calculated location approach, the fetching mechanism could simply bring in a single function from a package stored in the file. If the directory lookup approach is used, subfunctions and variables related to a function could be stored with it and retrieved at the same time.

Finally, the retrieval mechanism could keep track of what had been brought in. This would allow a trap on *WS FULL* to remove the least recently fetched objects so that processing could resume.

VII. THE SYSTEM VARIABLE ⎕*EC* (ENVIRONMENT CONDITION)

You can now ensure that a defined function will not be suspended if an error or interrupt occurs while it is executing.

Definition:

Since event trapping was first introduced on SHARP APL, it has been possible to trap situations in which the user interrupts processing or input. This capability can be used to provide security to programs, data, or both. The weak link in this chain has always been the fact that it was possible (with quick fingers) to signal attention before the protective ⎕*TRAP* was assigned. This was true even if the assignment was made on the first line of the sensitive function. The implementation of ⎕*EC* makes it possible to protect against this and other related problems.

⎕*EC* is a special kind of system variable, called a system shared variable. Two other variables in this class are ⎕*TRAP* and ⎕*SC*. The two differences between these two system variables and all the rest are the following:

1) They always have a value, even if they are localized and not assigned;

2) The APL system always validates values assigned to them when the assignment is made, and if the value is invalid, resets them to a default value.

⎕*EC* can only be assigned the values 0 or 1. The global default value is 0. When it is localized, the APL system assigns a default value of 1. If you assign any value other than 1 or 0 to ⎕*EC*, it will be reset to 0.

If you don't ever assign a value to ⎕*EC*, the default global value of 0 will cause the APL system to behave just as it did in the past. And even if you do assign ⎕*EC*, whenever every value of ⎕*EC* on the State Indicator stack is 0, the behavior is still the same as before. Note that with ⎕*EC*, like ⎕*TRAP*, the APL system may base its behavior on occurrences of ⎕*EC* other than the most local one, which is the only one visible to you.

How does ⎕*EC* behave when there are no event traps to complicate the situation? You can think of ⎕*EC* as setting a fence between the user and your programs. Only those programs which are below your fence on the SI stack can be suspended, either by interrupt or by error. A possible SI stack might look like the following.

```
FNI[19]
FNH[3] ⎕EC←1
FNG[7]
FNF[5] ⎕EC←0
FNE[17]
FND[11] ⎕EC←1
FNC[2]
FNB[13]
FNA[23]
{GLOBAL} ⎕EC←0
```

If the stack is situated as above, only functions *FNA*, *FNB*, and *FNC* could possibly be suspended. Even though ⎕*EC* has a value of 0 in *FNF* and *FNG*, the more global value set in *FND* takes precedence.

The specific action taken by the APL system when an event occurs in a 'fenced-off' level of the SI stack depends on the type of event that occurs.

Errors cause the SI stack to be cut back one level. The event is then signalled again in this new environment. If this level of the stack is also 'fenced-off' (that is, some value of ⎕*EC* on the SI stack is 1), the process will be repeated.

Attention (event 1002) is ignored, but is left pending. It will be responded to as soon as ⎕*EC* is everywhere 0.

Interrupt (event 1003), which is signalled with a double ATTN (or BREAK), and the other interrupts (events 1004-1007) are treated as errors are.

Stop (event 1001) and **Return to immediate execution** (event 2001) will cause the same behavior they always have.

Using ⎕*EC* really simplifies the writing of secure systems. If you simply localize ⎕*EC* in the main function, and never set it anywhere, none of the functions will ever be able to be suspended. You can compare this behavior to that of primitive functions. In both cases, you can stop the execution of the function by signalling double ATTN, but you cannot **suspend** their execution.

Stop controls set with *S*Δ **will** cause a function to be suspended, even if there is a ⎕*EC* value set to 1. This makes it easier for programmers to debug systems which use ⎕*EC*. In order to prevent a someone from circumventing your ⎕*EC* settings with stop controls, you must lock your functions. Since you can't change the stop or trace settings on a locked function, no one will be to interrupt your functions using this method. Of course, if the sensitive functions are never in the workspace when the user is in immediate execution mode, locking is unnecessary. You can avoid having to lock your application programs by having locked cover functions in the workspace which set ⎕*EC* to 1, and then bring in unlocked subordinate functions from file to do the actual work.

Normally, a single ATTN is sufficient to terminate the execution of ⎕*DL*. But since a single ATTN is ignored when ⎕*EC* is 1, you will have to signal ATTN twice in order to terminate a programmed delay.

How does ⎕*EC* work when there are event traps set? When an event occurs, each level of the SI stack is examined, starting from the most local level. The following tests are made, and if one of them is satisfied, the corresponding action is taken. If neither of the tests is satisfied at any level, the default system action for handling events is taken.

1) Does the value of ⎕*TRAP* visible at this SI level contain a trap definition which covers this event? If so, **and** if the visible trap is localized at this level, execute the corresponding trap line.

2) Is ⎕*EC* localized at this level? If so, **and** if its value is 1, handle the event according to its type as described above. If its value is 0, continue the search at the next level.

These rules are defined so as to ensure that existing applications which use ⎕*TRAP* but do not use ⎕*EC* behave exactly as before.

Application:

A) You can ensure that system variables which control the workspace environment are set before a suspension can occur.

```
     ∇SECURE;⎕EC;⎕TRAP
[1]  ⎕TRAP←'∇ 0 E HANDLEERR ∇ 1000 E HANDLEINT'
[2]  ⎕EC←0 ...
```

B) You can allow specified users to be able to suspend your functions, or to look at them if errors occur.

```
[1]  ⎕EC←~(1⍴⎕AI)∈5467270 4546320
```

C) You can protect critical sections of programs from interruption. In applications in which more than one user can simultaneously read and replace a file component, it is necessary to use file holds to ensure that the read and replace operations by one user are done without intervening reads or replaces by other users. Since file holds terminate upon return to immediate execution mode, and since a user could branch around the file holds in a suspended function, it's a good idea to prevent interruption during such update operations. This can be done with ⎕TRAP, or ⎕EC.

```
[⍬]  ⎕EC←1 ⋄ ⎕FHOLD TIE
[⍬]  DATA←⎕READ TIE,COMPONENT
[⍬]  DATA←DATA,NEWDATA
[⍬]  DATA ⎕REPLACE TIE,COMPONENT
[⍬]  ⎕EC←0 ⋄ ⎕FHOLD ⍳0
```

ACKNOWLEDGEMENTS: L.H. Goldsmith did the system programming which implemented these enhancements to the event trapping facility. L.H. Goldsmith, J.H. Schueler, and D.G. Smith reviewed a draft of this note.

2 May 1983

TITLE: Language Extensions Of May 1983

A number of language extensions are available in the May 1983 release of SHARP APL:

- Extension of the operator ⍤ to allow specification of the ranks of the cells to which a function applies (in the form *F*⍤(*M*,*L*,*R*)).
- Extension of the right domains of the compositions (*F*⍤*G* and *F*∘*G*) and dual (*F*⍨*G*).
- New functions: Lev (⊣), Dex (⊢), and Link (⊃).
- Extensions of matrix inverse and division (⌹) to arrays of rank greater than 2.

THE RANK OPERATOR

Definition

The operator On (⍤) previously only applied to two functions to provide Composition. It now accepts an integer right argument (as in ,⍤2) to specify the rank, or ranks, of the subarrays to which the function applies. For example:

```
      A←2 3 4⍴⍳24

      ,⍤2 A
 1  2  3  4  5  6  7  8  9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24

      ,⍤¯1 A
 1  2  3  4  5  6  7  8  9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24
```

As illustrated above, a non-negative argument specifies the number of **final** axes to which the function applies. A negative argument specifies the **complementary** rank, i.e., the number of **leading** axes to be **excluded**. The magnitude of the argument serves only to limit the number of axes; for the *A* given above, *F*⍤4 *A* ↔ *F*⍤3 *A*, and *F*⍤¯4 *A* ↔ *F*⍤¯3 *A*.

In the dyadic case two ranks (or complementary ranks) are specified. For example:

```
      E←2 2⍴'ABCD'
      F←2 2⍴'1234'
```

```
      E,⍤1 2 F
A12
B34

C12
D34

      E,⍤0 0 F
A1
B2

C3
D4
```

The most general right argument of ⍤ is a three-element vector whose successive elements specify the monadic, left, and right ranks. A shorter argument *R* is extended so that *F*⍤*R* ↔ *F*⍤(⌽3↑⌽*R*). If the right argument of On is a scalar, and the resulting derived function is used dyadically, that number will specify the rank of both arguments.

An expression such as *F*⍤*R A* introduces the possibility of juxtaposing constants (*R* and *A*). If *R* is a constant, and if the expression defining *A* begins with a constant, then the division between the two constants must be made clear to prevent them from being treated as a single vector. For example, the sequence

```
      A←2 3 4⍴⍳24
      R←2
      ,⍤R A
```

cannot be written in a single line as

```
      ,⍤2 2 3 4⍴⍳24
```

but must be written in a form which separates the vector constants. For example:

```
      ,⍤(2) 2 3 4⍴⍳24

      ,⍤2 (2 3 4)⍴⍳24

      ,⍤2 A←2 3 4⍴⍳24

      ,⍤2⊢ 2 3 4⍴⍳24
```

If a function has rank *R*, then the subarrays along the last *R* axes of its arguments are the **cells** of the array. The remaining axes are the **frame** of the array. Consider the expression ⌸*A*, where *A* has the shape 8 7 6 5 4.

⌸ has argument rank 2

The cells are arrays of shape 5 4.

The frame has shape 8 7 6.

In contrast, consider the expression ⌊*A*, using the same array *A*.

⌊ has argument rank 0.

The cells are scalars, that is, arrays of shape ⍬.

The frame is an array of shape 8 7 6 5 4.

A function applies independently to each cell (or pair of cells, if dyadic) to produce result cells, which must be of common shape. The result is formed by assembling the frame of result cells into an array.

If a function *F* has rank *R*, then the result of *F* ⍵ is determined by:

1) applying *F* to each of the **cells** of shape (-*R*)↑⍴⍵,
2) producing results with a **common shape** *SIR* for each,
3) assembling the whole into a result of shape ((-*R*)↓⍴⍵),*SIR*.

The frames of the arguments of a dyadic function must match in shape, or one of them must be a scalar. If the frame is a scalar, it is extended by reshaping it to match the shape of the other frame. This is a generalization of scalar extension.

By convention, if the frame is empty (i.e. 0∈(-*R*)↓⍴⍵), then the empty vector is used as the result cell shape.

Examples

Common uses for the Rank operator include:

1) Applying functions of infinite rank (such as Enclose and Ravel) to sub-arrays, within multi-dimensional arrays, and

2) Applying scalar functions (such as Addition) in ways that would not otherwise be possible under the rules of ordinary scalar extension.

The following examples illustrate uses of the Rank operator.

```
      V←⍳5 ⋄ X←10×⍳4 ⋄ Y←100×⍳3 ⋄ W←?V ⋄ M←3 4⍴⍳12 ⋄ N←?M ⋄ B←3 4
5⍴⍳60

      Q←2 5⍴'ABCDEEDBCA' ⋄ ⎕PS←-1 1 3 3
```

Examples with ENCLOSE

```
      M
 1  2  3  4
 5  6  7  8
 9 10 11 12

      <⍤0 M                          Enclose each (scalar) element
┌─┐ ┌─┐ ┌─┐ ┌─┐
│1│ │2│ │3│ │4│
└─┘ └─┘ └─┘ └─┘

┌─┐ ┌─┐ ┌─┐ ┌─┐
│5│ │6│ │7│ │8│
└─┘ └─┘ └─┘ └─┘

┌─┐ ┌──┐ ┌──┐ ┌──┐
│9│ │10│ │11│ │12│
└─┘ └──┘ └──┘ └──┘
```

```
      <⍤1 M                                  Enclose rows
┌─────────┐ ┌─────────┐ ┌─────────────┐
│1 2 3 4  │ │5 6 7 8  │ │9 10 11 12   │
└─────────┘ └─────────┘ └─────────────┘

GP      <⍤2 B                                Enclose matrices
┌──────────────────┐ ┌──────────────────┐ ┌──────────────────┐
│ 1  2  3  4  5    │ │21 22 23 24 25    │ │41 42 43 44 45    │
│ 6  7  8  9 10    │ │26 27 28 29 30    │ │46 47 48 49 50    │
│11 12 13 14 15    │ │31 32 33 34 35    │ │51 52 53 54 55    │
│16 17 18 19 20    │ │36 37 38 39 40    │ │56 57 58 59 60    │
└──────────────────┘ └──────────────────┘ └──────────────────┘
```

Examples with LINK

```
      V
1 2 3 4 5
      W
1 2 2 3 2

      V ⊃⍤0 W                                Similar to (<⍤>V),[1.5] <⍤>W
┌─┐ ┌─┐
│1│ │1│
└─┘ └─┘

┌─┐ ┌─┐
│2│ │2│
└─┘ └─┘

┌─┐ ┌─┐
│3│ │2│
└─┘ └─┘

┌─┐ ┌─┐
│4│ │3│
└─┘ └─┘

┌─┐ ┌─┐
│5│ │2│
└─┘ └─┘
```

```
      M ⊃⍤1 N                                Similar to (<⍤1 M),[1.5] <⍤1 N
┌─────────┐ ┌─────────┐
│1 2 3 4  │ │1 2 3 4  │
└─────────┘ └─────────┘

┌─────────┐ ┌─────────┐
│5 6 7 8  │ │2 4 6 1  │
└─────────┘ └─────────┘

┌─────────────┐ ┌─────────┐
│9 10 11 12   │ │1 6 8 1  │
└─────────────┘ └─────────┘
```

Examples with GRADE

```
      N
1 2 3 4
2 4 6 1
1 6 8 1
```

```
      ⍋⍤1 N
1 2 3 4
4 1 2 3
1 4 2 3
```

Upgrade each row independently

```
      ⍉ ⍋⍤1 ⍉N
1 1 1 2
3 2 2 3
2 3 3 1
```

Independent upgrade of each column

```
     Q
ABCDE
EDCBA
```

```
      Q ⍋⍤1 'ABCDEF'
1 2 3 4 5 6
5 4 3 2 1 6
```

Upgrade with multiple collating sequences

An example with RAVEL

```
      ,⍤2 B
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
```

Ravel each matrix

An example with ROTATE

```
      (1-⍳4)⌽⍤1 2 B
 1  2  3  4  5
10  6  7  8  9
14 15 11 12 13
18 19 20 16 17

21 22 23 24 25
30 26 27 28 29
34 35 31 32 33
38 39 40 36 37

41 42 43 44 45
50 46 47 48 49
54 55 51 52 53
58 59 60 56 57
```

Examples with CATENATE

```
     Y
100 200 300
```

```
      Y ,⍤0 2  B                          Catenate scalars to matrices
100   1   2   3   4   5
100   6   7   8   9  10
100  11  12  13  14  15
100  16  17  18  19  20

200  21  22  23  24  25
200  26  27  28  29  30
200  31  32  33  34  35
200  36  37  38  39  40

300  41  42  43  44  45
300  46  47  48  49  50
300  51  52  53  54  55
300  56  57  58  59  60
```

```
      M ,⍤0 N                             Same as M,[2.5] N
 1  1
 2  2
 3  3
 4  4

 5  2
 6  4
 7  6
 8  1

 9  1
10  6
11  8
12  1
```

Examples with RESHAPE

```
      5 4 ⍴⍤1 2 B                         Reshape each plane
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20

21 22 23 24
25 26 27 28
29 30 31 32
33 34 35 36
37 38 39 40

41 42 43 44
45 46 47 48
49 50 51 52
53 54 55 56
57 58 59 60
```

```
      (5×¯1↑⍴M) ⍴⍤1 M                     Five side-by-side copies
 1  2  3  4  1  2  3  4  1  2  3  4  1  2  3  4  1  2  3  4
 5  6  7  8  5  6  7  8  5  6  7  8  5  6  7  8  5  6  7  8
 9 10 11 12  9 10 11 12  9 10 11 12  9 10 11 12  9 10 11 12
```

Examples with TAKE

```
      7 ↑⍤1 M
 1  2  3  4  0  0  0
 5  6  7  8  0  0  0
 9 10 11 12  0  0  0
```

Take on all rows;
the same as ((¯1↓⍴M),7)↑M

```
      ⍉ 7 ↑⍤1 ⍉M
 1  2  3  4
 5  6  7  8
 9 10 11 12
 0  0  0  0
 0  0  0  0
 0  0  0  0
 0  0  0  0
```

Take on all columns;
the same as (7,1↓⍴M)↑M

```
      2 3 ↑⍤1 2 B
 1  2  3
 6  7  8

21 22 23
26 27 28

41 42 43
46 47 48
```

Take on each matrix

Examples with MATCH

```
      B ≡⍤1 ⍬B
0 0 0 0
1 1 1 1
0 0 0 0
```

Compare vectors

```
      B ≡⍤2 ⍬B
0 1 0
```

Compare matrices

Examples with MEMBERSHIP

```
      (⍳3) ∈⍤0 2 B
1 0 0
```

First scalar member of first plane, 2nd of 2nd plane , etc.

```
      (⍳7) ∈⍤1 2 B
1 1 1 1 1 1 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

Usual membership of vector, repeated for each matrix

Examples with IOTA

```
      M
 1  2  3  4
 5  6  7  8
 9 10 11 12

      N
1 2 3 4
2 4 6 1
1 6 8 1
```

```
      M ⍳⍤1 N
1 2 3 4
5 5 2 5
5 5 5 5
```

Independent search on rows; as in *M*[1;]⍳*N*[1;], etc.

```
      C←3 3⍴'KEIRBEEEM'
      D←2 3⍴'RBERCM'
      (D∧.=⍉C)⍳⍤(1) 1
2 4
```

Table lookup, as in ⎕*IO*++/∧*D*∨.≠⍉*C*

Examples with ADDITION

```
      X
10 20 30 40
```

```
      X +⍤1 M
11 22 33 44
15 26 37 48
19 30 41 52
```

Add scalars to rows, as in, *M*+(⍴*M*)⍴*X*

```
      Y
100 200 300
```

```
      ⍉ Y +⍤1 ⍉M
101 102 103 104
205 206 207 208
309 310 311 312
```

Add scalars to columns, as in,*M*+⍉(⌽⍴*M*)⍴*Y*

```
      Y +⍤ 0 2 B
101 102 103 104 105
106 107 108 109 110
111 112 113 114 115
116 117 118 119 120

221 222 223 224 225
226 227 228 229 230
231 232 233 234 235
236 237 238 239 240

341 342 343 344 345
346 347 348 349 350
351 352 353 354 355
356 357 358 359 360
```

Add scalars to matrices

An example with REPRESENTATION

```
      10 10⊤V
0 0 0 0 0
1 2 3 4 5
```

```
     10 10 ⊤⍤1 0 V
0 1
0 2
0 3
0 4
0 5
```

A different representation

DOMAINS OF OPERATORS ⍤, ⍥, ⍣

The left argument of the operators ⍤, ⍥, and ⍣ may be any primitive function except execute (⍎). The valid right arguments of ⍤ include any scalar or vector of integers containing at most three elements.

A primitive function *G* is a valid right argument of ⍤ and ⍥ (and of ⍣ if it has an assigned inverse) if its relevant ranks (left and right in the case of *α F⍥G ω*, and monadic in all other cases) have been assigned. The following table shows function ranks, denoting infinite rank by ¯, and unassigned rank by ⍬:

⌹	<	>	⍴	,	⍒	∈	⍋	⍳	?	⍉	⌽	≡	⊢	⊣	⊃	Function
2	¯	0	¯	¯	¯	⍬	¯	¯	0	⍬	¯	⍬	¯	¯	¯	Monadic
¯	0	0	⍬	¯	⍬	0	¯	¯	⍬	¯	⍬	¯	¯	¯	¯	Left
2	0	0	⍬	¯	⍬	¯	¯	¯	⍬	¯	⍬	¯	¯	¯	¯	Right

The functions ⊥ ⍳ ⌽ ⊖ ↑ ↓ ⍎ are not included in this table because their ranks have not been assigned.

All scalar primitive functions have rank 0 and are therefore in the right domains of ⍤ and ⍥. A monadic function *G* is in the right domain of ⍣ if it is in the right domain of ⍤ and also occurs in the following table of assigned inverses:

+	-	÷	*	~	<	⍒	⊢	⌹	>	⍟	Function
+	-	÷	⍟	~	>	⍒	⊢	⌹	<	*	Inverse

LEV ⊣ and DEX ⊢

Definition:

The dyadic functions Lev ⊣ and Dex ⊢ return their left and right arguments, respectively. Each symbol points toward the argument it returns. The names are derived form the Latin words meaning 'on the left side' (laevus) and 'on the right side' (dexter).

The monadic case of ⊢ is an identity function. It returns its argument unchanged. The monadic case of ⊣ returns no result, and is thus equivalent to ⍎''. ⊣*A* will work, but *Z*←⊣*A* will give a *RESULT ERROR*. Both cases of each function have infinite rank, and thus can be used with the operators On, Over, and With.

Application:

A) Many system functions both cause side effects (like defining functions) and return explicit results. If the result is not needed, it can be discarded using Lev. For example, the statement

```
[⍬]  JUNK←⎕EX NAMELIST
```

can also be written as

```
[⍬]  ⊣⎕EX NAMELIST
```

B) Lev, when used dyadically, provides a statement connection facility. Unlike diamond, the statement on the right side of the Lev function will execute first.

LINK ⊃

Definition:

The monadic function ⊃ is a conditional enclose. If the argument is simple (not enclosed), then it is enclosed. If the argument is non-simple, it is returned unchanged.

The dyadic function ⊃ is called Link. It encloses its left argument, applies the conditional enclose to the right argument, and catenates the results. These rules can be summarized by the following.

```
⊃ω ↔ <ω if ω is simple
   ↔ ω if ω is non-simple

α⊃ω ↔ (<α),⊃ω
```

Both cases of ⊃ have infinite rank, enabling it to be used as the right function argument in composition. Note that in the expression α⊃ω, the symbol ⊃ (like the symbol < in <ω) opens toward the argument to be enclosed.

If the right argument (ω) is simple, the shape of the result will always be 2. If the right argument (ω) is not simple, the shape of the result will be ((-ρρω)↑1)+ρω. This is due to the scalar extension of the left argument when the catenation occurs. The following example illustrates this point.

```
      ⎕PS←-1 1 3 3
      G←2 5⍴'0123456789'
      H←<⍤0 3 4⍴'ABCDEFGHIJKL'
      ⍴H⊃G
2
      H⊃G
┌─────────────────┐ ┌─────┐
│ ┌─┐ ┌─┐ ┌─┐ ┌─┐ │ │     │
│ │A│ │B│ │C│ │D│ │ │01234│
│ └─┘ └─┘ └─┘ └─┘ │ │56789│
│                 │ └─────┘
│ ┌─┐ ┌─┐ ┌─┐ ┌─┐ │
│ │E│ │F│ │G│ │H│ │
│ └─┘ └─┘ └─┘ └─┘ │
│                 │
│ ┌─┐ ┌─┐ ┌─┐ ┌─┐ │
│ │I│ │J│ │K│ │L│ │
│ └─┘ └─┘ └─┘ └─┘ │
└─────────────────┘
```

```
      ⍴G⊃H
3 5
      G⊃H
┌─────┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐
│01234│ │A│ │B│ │C│ │D│
│56789│ └─┘ └─┘ └─┘ └─┘
└─────┘

┌─────┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐
│01234│ │E│ │F│ │G│ │H│
│56789│ └─┘ └─┘ └─┘ └─┘
└─────┘

┌─────┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐
│01234│ │I│ │J│ │K│ │L│
│56789│ └─┘ └─┘ └─┘ └─┘
└─────┘
```

Application:

A) Monadic ⊃ can be used to test whether an array is simple.

```
SIMPLE◇ ~⍵≡⊃⍵
```

B) Dyadic ⊃ can be used to create enclosed arrays without having to explicitly enclose and catenate the elements.

```
OLD WAY                                  NEW WAY
(<1 2),(<'DEF'),<6 7 8 9                 1 2⊃'DEF'⊃6 7 8 9
```

Note that the new way is more readable and uses fewer functions.

MATRIX INVERSE and MATRIX DIVISION ⌹

Definition:

In the past, the functions denoted by ⌹ were limited to working with arguments whose rank was less than or equal to 2. The definition of Matrix Inverse is still based on inverting rank 2 arrays. But now, it can be applied to multiple matrices in a single operation, rather than by looping.

Old Way **New Way**

```
Z[1;;]←⌹X[1;;]      Z←⌹X
Z[2;;]←⌹X[2;;]
Z[3;;]←⌹X[3;;] ....
```

This example shows that the new Matrix Inverse function inverts the matrices along the last two axes of the argument.

Since Matrix Division is defined in terms of Matrix Inverse (⍺⌹⍵ ≡ (⌹⍵)+.×⍺) its definition is extended in a similar manner.

The rules for determining the argument and result shapes are given below. (The degenerate cases where 2>⍴⍴⍵ work as before and are not included in this table.)

Function	ρω	ρα	ρZ
Z←⌹ω	(1↑¯2↑ρω)≥¯1↑ρω	-	(¯2↓ρω),⌽¯2↑ρω
Z←α⌹ω	(1↑¯2↑ρω)≥¯1↑ρω	(1↑ρα)=1↑¯2↑ρω	(¯2↓ρω),(¯1↑ρω),1↓ρα

Application:

To solve several systems of equations which have different coefficients, but the same constants, try an expression like the following.

CONSTANTVECTOR⌹*COEFFMATRIX*1,[0.5] *COEFFMATRIX*2

To solve several systems of equations which have the same coefficients, but different constants, try an expression like the following.

(*CONSTANTVECTOR*1,[1.5] *CONSTANTVECTOR*2)⌹*COEFFMATRIX*

Examples:

```
      A←2 2 2ρ0 1 1 2 3 5 8 13
      ⌹A
 ¯2   1
  1   0

¯13  ¯5
  8  ¯3
      B←2 2ρ1 0 0 1
      B⌹A
 ¯2   1
  1   0

¯13  ¯5
  8  ¯3
      C←2 3 4ρ⍳24
      C⌹A
 11  10   9   8
  7   6   5   4
  3   2   1   0

  1   2   3   4
  5   6   7   8
  9  10  11  12


 52  44  36  28
¯20 ¯12  ¯4   4
¯12 ¯20 ¯28 ¯36

¯31 ¯26 ¯21 ¯16
¯11  ¯6  ¯1   4
  9  14  19  24
```

Acknowledgements: The following people contributed to this SATN: R. Bernecky, K.E. Iverson, E.E. McDonnell, R.C. Metzger, J.H. Schueler.

5 AUG 83

TITLE: ENHANCEMENTS PROVIDED IN UPDATE #1 TO THE MAY 1983 RELEASE OF SHARP APL

AUTHOR: Eric B. Iverson

Overview

In August 1983 a set of enhancements called "Update #1" was made available to sites running the MAY 1983 release of SHARP APL.

The changes were in eight general areas:

1. The APL character set has been extended to include percent, ampersand, and twelve new elements designated as national use characters. The old elements dollar and cent (⎕*AV*[⎕*IO*+4 5]) are considered to be national use characters, giving a total of 14 national use characters.

2. The 14 national use characters, percent, ampersand, and the lower case alphabetics are now valid in quotes, in comments, and in ⍞ input.

3. SAPV now supports an overstrike escape character to allow entry of APL programming symbols not supported on an APL 3270.

4. SAPV now supports PFK7 and PFK8 for paging in S mode as well as PFK1 and PFK2.

5. The SAPV, MPXH, and NTOH terminal handlers have new translate tables to allow the display and entry of these new graphics and other, previously unsupported, graphics.

6. The extension to the APL character set required a new arrangement of the CODE S table. CODE S is used in the TSIO, AP123, and FCAP auxiliary processors.

7. Several new AP124 and SAPV ARBIN facilities have been added. Character attributes can now be set and reported by AP124. AP124 and ARBIN can now return screen image information in a device independent manner (APL characters, rather than device codes).

8. HCPRINT now supports a new set of request functions as well as the old ones. The new ones capture the screen images in a device independent manner. This allows HCPRINT to drive printers regardless of the printer buffer size, and also allows it to use SCS data streams which provide simpler and more efficient use of the printers.

National Use Characters

The concept of 14 national use characters comes primarily from IBM's definition of the 3270 character sets in various languages. The implementation of the concept is similar to and has benefitted from the IBM implementations VSAPL and APL2.

The APL character set now contains 14 elements designated as national use characters. These elements may have different graphics at different sites. They are intended to allow sites to support graphics required by end user applications (e.g., accented alphabetics required in a particular language, and national currency symbols). They are not APL "programming" symbols and will never be used as APL primitives. In general they should be used only as text, and giving them special significance in applications should be avoided if possible. It is very important to realize that the national use characters may well display differently at different sites. In general, the national use characters will be the same across sites in the same country (hence the name). The scheme applied here is essentially the same one that IBM uses for 3270 character sets in different countries, and the intent is that in each country the graphics chosen for the 14 national use elements in the APL character set will be the same as the IBM national use characters for the 3270 in that country.

National use characters can be very confusing. Not only can their graphic displays change from site to site, but their displays can be the **same** as (or similar to) another element of ⎕*AV*. For example, a site with the English (US) national use characters has ⎕*AV*[⎕*IO*+193] as a stile as well as ⎕*AV*[⎕*IO*+34] which is the APL primitive symbol. Remember, ⎕*AV*[⎕*IO*+4] will display as $ in North America and as an A overstruck with a small circle in Sweden!

The following table shows the national use character positions in ⎕*AV* and their English (US) graphics.

National Use Use Character	⎕*AV* element (0-origin)	English (US) Graphic
*NU*1	5	cent sign
*NU*2	193	stile
*NU*3	194	exclamation mark
*NU*4	4	dollar sign
*NU*5	196	PL1 not
*NU*6	197	broken stile
*NU*7	198	accent
*NU*8	199	pound sign
*NU*9	200	at sign
*NU*10	201	double quote
*NU*11	202	high not
*NU*12	203	left brace
*NU*13	204	right brace
*NU*14	205	backslash

SAPV Overstrike Support

SHARP APL currently requires eight symbols for programming that are not supported by IBM 3270 terminals with the APL feature. The symbols that cannot be directly entered form the keyboard are ⋄ diamond, ⊣ lev, ⊢ dex, { left brace, } right brace, ≡ match, ⍤ paw, and Ö hoof. Note that some terminals do have left and right braces on the keyboard that are enterable with APL OFF, but these are national use symbols.

An overstrike sequence can be used to enter these symbols. An overstrike sequence is three characters: the overstrike escape character and the two symbols, in either order, which form the overstrike. The overstrike pairs, and their resulting symbol are as follows:

<>	⋄	diamond
v^	⋄	diamond
-)	⊣	lev
(-	⊢	dex
[°	{	left brace
°]	}	right brace
=¯	≡	match
°¨	⍤	paw
O¨	Ö	hoof

The overstrike escape character can be set by the system command *OEC*.

The default when a session starts is that there is no overstrike escape character. The following examples should make clear the use of the system command:

```
      )OEC
IS OFF

      )OEC ¯
WAS OFF

      )OEC
IS ¯

      )OEC OFF
WAS ¯
```

A valid overstrike sequence is resolved to the single character. The resolved character has a single graphic that is used for display of that symbol. The overstrike escape sequence is used **only** for keyboard input. An invalid overstrike sequence is left untouched. The exact graphic for displaying the symbols entered with an overstrike sequence may vary from site to site depending on the translation tables in use at a site. The defaults are that ⋄⊣⊢{} are displayed with graphics that look correct and that ≡, ⍤, and Ö will be displayed respectively as + over -, degree symbol (small, high circle), and a solid circle. Trying the following examples should help make the use of overstrike sequences clear:

```
      )OEC ¯
WAS OFF
```

equivalent to

```
      23¯v^24          23 ⋄ 24
```

```
23¯∧∨24                    23 ◇ 24

23¯<>24                    23 ◇ 24

23¯=_24                    23 ≡ 24

'¯-)¯(-¯[∘¯]∘'             '⊣⊢{}'

'¯∨∧¯=_¯⍤∘¯⍤⍥0'            '◇≡⍤⍥⍤'

¯23+4                      ¯23+4

'AB¯CD'                    'AB¯CD'
```

Overstrike sequences are processed in session manager input, AP124 readscreen input, and *⎕ARBIN* input with translation (command number 6). They are **not** processed in AP126 or in *⎕ARBIN* input without translation.

The overstrike support in SAPV, although quite different from the "printable backspace" support in IBM's APL2, did benefit from the study of that facility.

⎕ARBIN/⎕ARBOUT Support for Setting and Reporting the Overstrike Escape Character

The overstrike escape character can be set and reported under program control using the *⎕ARBIN/⎕ARBOUT* interface. The general *⎕ARBIN/⎕ARBOUT* interface is documented in SATN-37. The following examples should make the facility clear.

```
⎕ARBOUT ⎕AV[⎕IO+24↑15],'¯'   ⍝ Set OEC to ¯

⎕AV[⎕IO+¯1↑⎕ARBIN ⎕AV[⎕IO+24↑15],'⍥']   ⍝ Set OEC to ⍥ and return previous setting

⎕ARBOUT ⎕AV[⎕IO+25↑15]   ⍝ Set OEC to 1↑⎕AV, which is OFF

⎕AV[⎕IO+1↑33↓⎕ARBIN 24↑0]   ⍝ Current OEC setting from ARBIN inquiry
```

In summary, a new command (15) has been added for *OEC* control, and the *OEC* current value is now returned in the *⎕ARBIN* inquiry command (command 0).

SAPV Character Set

SAPV is the terminal handler for IBM 3270 terminals. The mappings between keyboard entry, screen display, and elements of *⎕AV* can be customized by a site if necessary. The default mappings are as follows:

1. All keys on all terminals (both with APL ON and APL OFF) can be entered between quotes, in comments, and in ⍞ input.

2. The national use characters map to the national use elements in *⎕AV*. The national use characters are entered as expected on a non-APL 3270 and are entered on an APL 3270 when APL is OFF.

3. The percent and ampersand characters map to ⎕*AV*[206+⍳2] repectively.

4. Overstrike sequences can be used to enter the APL programming symbols not supported by the IBM 3270 APL terminals.

5. Output is made as readable as possible by mapping the three APL alphabets (upper case, underscored, and lower case) onto the alphabets available on the terminal. The following table shows the mappings.

Terminal Type	APL Upper Case IN/OUT	APL Underscored IN/OUT	APL Lower Case IN/OUT
APL	*A*/*A*	$\underline{A}$/$\underline{A}$	a/a
NON-APL	*A*/*A*	can't be input/*A*	a/a
UPPER CASE ONLY	*A*/*A*	can't be input/*A*	can't be input/*A*

MPXH Character Set

MPXH is the terminal handler for asynchronous (non-3270) terminals supported through the I.P. Sharp network control program, IBM EP (emulator program), or equivalent.

APL terminals supported by MPXH can now display and enter all of the APL character set. The changes are as follows:

1. Lower case alphabetics are entered and displayed as the upper case alphabetics overstruck with the overbar.

2. Percent and ampersand (⎕*AV*[⎕*IO*+206 207]) are entered and displayed respectively as ÷ overstruck with / and ⍺ overstruck with \.

3. English (US) national use characters are entered and displayed as follows:

National Use Character	⎕*AV* Element (0-origin)	Graphic	Overstrike
*NU*1	5	cent sign	c |
*NU*2	193	stile	| ⊥
*NU*3	194	exclamation mark	∘ '
*NU*4	4	dollar sign	*S* |
*NU*5	196	PL1 not	~ /
*NU*6	197	broken stile	! ,
*NU*7	198	accent	¯ \
*NU*8	199	pound sign	*N* =
*NU*9	200	at sign	*Q* ∘
*NU*10	201	double quote	¨ '
*NU*11	202	high not	~ -
*NU*12	203	left brace	[○
*NU*13	204	right brace	○]
*NU*14	205	backslash	| \

These overstrikes were chosen so as to give unique display and entry of the national use characters and to allow those with the same graphic as an APL programming symbol to be distinguished.

Remember, these characters may be quite different overstrikes at a site not running with the English (US) national use characters.

Non-APL terminals supported by MPXH (historically referred to as TY33 terminals) have the following changes.

1. Previously the backslash character (\) was used as a backspace on input and as the canonical illegal character on output. It now is simply the backslash character on both input and output.

2. A backspace is now entered as CTRL-H.

3. The canonical illegal character is now ⍰.

4. Previously accent was an entry error. Accent now maps to ⍰ on entry and display.

5. The characters ⋄⊣⊢⌿⍀⍲≠⍰ and ⍇ can now be entered and displayed as their standard overstrikes.

6. National use characters, percent, ampersand, lower case alphabetics, and several other characters are not supported on non-APL terminals through MPXH.

At some point in the future an alternative non-APL support will be selectable that is identical to the NTOH support for non-APL terminals, as discussed in the section on NTOH.

NTOH Character Set

NTOH is the terminal handler for asynchronous (non-3270) terminals connected through the IBM product NTO (Network Terminal Option) and for SNA connected IBM 3767 terminals.

APL terminals supported by NTOH can now display and enter all of the APL character set. The support for APL terminals is identical to the support provided by MPXH (see the earlier section on the MPXH Character Set).

The NTOH support for non-APL terminals is similar to the SAPV support, **not** the MPXH (TY33) support. In particular, national use graphics map to the national use elements of ⍞⎕AV, $\underline{A}$ maps to A, $\underline{\underline{A}}$ maps to $\underline{A}$, a maps to a, and there are no "special" mappings as there are in MPXH (e.g., accent to ⍰).

CODE S

CODE S is a unique rearrangement of ⎕AV. Each element of ⎕AV maps to a unique element of CODE S and vice versa. CODE S can be considered an APL extended version of the IBM EBCDIC code. Several auxiliary processors translate APL data from their ⎕AV positions to their CODE S positions. This then allows the data to be processed as EBCDIC data by non-APL applications. In particular, TSIO when used with CODE=S or CODE=A translates APL character data to and from CODE S as appropriate.

Because the translation table has changed, some old data written to CODE S will not be read properly. This will only be true if characters outside of the normal set of EBCDIC graphics are involved.

SAPV AP124 Enhancements

Several enhancements are provided for AP124. AP124 is documented in SATN-37 **IBM 3270 USER GUIDE** (Rev. 3). Only changes and new facilities are documented here, and this material should be used in close conjunction with SATN-37.

Two functions have been extended and four new functions have been added. These functions can be found in workspace 5 *IBM*3270.

Format

The *FORMAT* function now supports new values in the blank processing specification (SATN-37, Rev. 3, page 12).

Previously, trailing blanks **written** to a field could be displayed as blanks or as nulls (allowing insertion). However, **reading** a field always filled with trailing blanks, regardless of whether the field had trailing blanks or nulls. This loss of information caused problems in some applications. Now it is possible to specify that reads from a field will fill with nulls or blanks according to what was actually in the field. This allows an application to distinguish between blanks and nulls on input.

Blank Processing is one of:

0 - nulls read as blanks, trailing blanks write as blanks.

1 - nulls read as blanks, trailing blanks write as nulls.

2 - nulls read as nulls, trailing blanks write as blanks.

3 - nulls read as nulls, trailing blanks write as nulls.

FIELDATTR

The *FIELDATTR* function now supports the new values required by the extension to blank processing (SATN-37, Rev. 3, page 14).

The right argument to *FIELDATTR* is:

0 - nulls read as blanks,
trailing blanks write as blanks, no autoskip.

1 - nulls read as blanks,
trailing blanks write as blanks, autoskip.

2 - nulls read as blanks,
trailing blanks write as nulls, no autoskip.

3 - nulls read as blanks,
trailing blanks write as nulls, autoskip.

4 - nulls read as nulls,
trailing blanks write as blanks, no autoskip.

5 - nulls read as nulls,
trailing blanks write as blanks, autoskip.

6 - nulls read as nulls,
trailing blanks write as nulls, no autoskip.

7 - nulls read as nulls,
trailing blanks write as nulls, autoskip.

WRITEATTR

A new function *WRITEATTR* writes character attributes. This function allows the specification of the attribute (color and highlight) of each character in a field. The arguments are the same as for *WRITE*, but instead of specifying the character to display, the attributes of the character are specified.

The character attributes are encoded in the 8 bits in a character as:

SSSCCCHH

where S is for program symbol set (unsupported), C is for color, and H is for highlight.

Color Values are

0 - default
1 - blue
2 - red
3 - pink
4 - green
5 - turquoise
6 - yellow
7 - white

Highlight values are

0 - default
1 - blinking
2 - reverse video
3 - underscore

The attribute character is ⎕*AV*[⎕*IO*+(color×4)+highlight]. For example,

```
1 WRITEATTR ⎕AV[⎕IO+9 24]
```

would make the first character of the field blinking red, the second character yellow, and would leave the rest of the field unchanged.

Like *WRITE*, *WRITEATTR* simply updates the information about the field and does not cause data to be written to the screen.

The CTL value for *WRITEATTR* is 35 and DAT contains the character attribute data.

IMMWRATTR

A new function *IMMWRATTR* is the same as the new function *WRITEATTR* except that it causes the screen to be written immediately. This is the same as the relationship between *WRITE* and *IMMWR*.

The CTL value for *IMMWRATTR* is 37 and DAT contains the character attribute data.

GETSCREEN

The new function *GETSCREEN* returns a character matrix the same size as the screen (rows by columns) containing the data for that screen. Previously this was done with a complex set of APL functions. The result of *GETSCREEN* is device independent and can be easily used for displaying an AP124 screen image on other devices. *GETSCREEN* is used by the new *HCPRINT* request functions.

The CTL value for *GETSCREEN* is 40 and DAT is not used.

GETSCREENX

The new function *GETSCREENX* returns a character matrix the same size as the screen (rows by columns) of the character attribute information for each display character in the screen image. The character attribute information is encoded in the same way as for *WRITEATTR*.

The CTL value for *GETSCREENX* is 41 and DAT is not used.

CTL/DAT Summary for the New Functions. (SATN-37 Rev. 3, page 17.)

CTL	DAT	Description
35,fld	character attributes	Buffered write of character attributes
37,fld	character attributes	Immediate write of character attributes
40	*	get screen data
41	*	get screen attributes

SAPV ARBIN Enhancements

Several enhancements are provided for SAPV □*ARBIN* and □*ARBOUT* support. The general facilities are documented in SATN-37, **IBM 3270 USER GUIDE** (Rev. 3). Only the new facilities are documented here, and this material should be used in close conjunction with SATN-37.

Command	⎕*AV*	Meaning
FSMPFMAP	12	Turns on and off the mapping of PFK's 13-24 to 1-12. This feature is of use with terminals whose righthand keypad PF keys are numbered 13-24. The element following the 24-element header is 1 to turn on mapping or 0 to turn off mapping. If ⎕*ARBIN* is used, the result is the previous setting.
FSMGETS	13	Returns a screen image (translated to APL) and attributes, in vector form, of a SAPV session manager standard screen. The page number is passed in four data characters. The page number is 256⊥⎕*AV*[1 4]+*DS*. In the result, the first three elements following the 24-byte header are used to reshape the remaining elements into a rank three array where the screen image is in the first plane, and the encoded attribute information (same as for AP124 *WRITEATTR*) is in the second plane. For example, to retrieve information for screen number 4: *R*←⎕*ARBIN* ⎕*AV*[(24↑13),(4⍴256)⊤4] *R*←⎕*AV*[(3⍴24↓*R*)⍴27↓*R*]
FSMGETF	14	As for *FSMGETS*, but applies to ⎕*ARBIN*/⎕*ARBOUT* full screens.
FSMOEC	15	Sets the overstrike escape charcter (OEC). The element following the 24 element header is the ⎕*AV* index of the character to be used, or zero to turn off OEC. If ⎕*ARBIN* is used, the result is the previous setting.

Workspace 5 *IBM3270* contains functions *R←MAPPFK* for *FSMPFMAP*, *R←GETSSCREEN* for *FSMGETS*, and *R←GETFSCREEN* for *FSMGETF*.

The command *FSMINQ* (⎕*AV*[0]) now includes, in position 8, a value indicating whether PFK's 13-24 are to be mapped to PFK's 1-12, and in position 9, the value of the overstrike escape character. Positions 8 and 9 were previously reserved.

1 October 1983

TITLE: IBM 3270 USER GUIDE (IDSH)

AUTHOR: Mike Symes

ABSTRACT: SHARP APL under MVS supports the IBM 3270 display stations through an auxiliary processor called IDSH (Information Display Station Handler). This document describes how the IDSH Session Manager supports the IBM3270 display station as an APL terminal and how full screen management is provided by AP124, AP126, and the arbitrary input/output interface. AP6, the session manager command processor, is also described.

TABLE OF CONTENTS

INTRODUCTION

SHARP APL under MVS supports the IBM 3270 display stations through an auxiliary processor called IDSH (Information Display Station Handler). IDSH uses the S-task facility to interface with SHARP APL and uses VTAM to support the IBM 3270's. When IDSH connects a 3270, it determines what features the device has and whether or not it has the APL character set.

The IBM 3270 device operates in two modes: standard screen and full screen. In standard screen mode, the device operates as an APL terminal with system defined characteristics. While in full screen mode, the screen is under the complete control of an APL application program.

This document provides a full description of how the device operates in both modes. Included is a description of workspace 5 *IBM*3270 which contains functions that provide full screen management support using AP124.

For an in-depth description of 3270s, see the IBM manual GA27-2749, IBM 3270 **Information Display System Component Description**.

NOTE: Origin 0 is used throughout this document.

A QUICK GUIDE TO THE 3270 SESSION MANAGER

The SHARP APL 3270 session manager enables a user to run an APL session from a 3270 terminal in a way that somewhat resembles a "line-by-line" scrolling terminal. In addition, a "log" of the session-to-date is kept so that browsing back through the session is possible.

The following is a quick guide to how to use the session manager, intended to help a user become minimally aquainted with the facilities. A detailed account of the facilities is given in the next section.

Getting Connected to the Session Manager

The procedure to gain an initial connection to the session manager will vary from one installation to another. Once this is achieved, the session manager will attempt to determine the exact characteristics of the terminal. If it is unable to do so, a "menu" of possible terminal types will be presented. You will have to select (by number) the type that most closely describes your terminal.

The Session Manager Screen

The initial session manager screen will then be presented. The top row will appear as follows:

U INP *LINE* 0 *COL WRAP*

indicating that the session manager is expecting input from you (*U*, for **unlocked**), APL is expecting input (*INP*), we are currently on line 0 of the session log, and lines of the session will be displayed **wrapped** (i.e. displayed on more than one row of the screen if necessary). The remainder of the screen will display the first part of the session log (containing a banner-like logo), and the cursor will be placed on the first unused row of the screen, indicating where you are expected to type (the **APL input area**).

Signing-on To APL

Enter your APL user number and password in the usual way:

```
)1234567:lock
```

and press ENTER. Anything typed here will not be displayed, to protect your signon password. The normal APL signon messages will be displayed on the screen immediately below where you typed.

Conversing With APL

You can now proceed through a normal APL "conversation" in the same way: enter input where the cursor indicates (after the last prompt from APL), and output will be displayed following your input. When the screen fills up without displaying all available output, the left end of the top screen row will appear as follows:

U H

The *H* indicates that the session manager is **holding** output.

To display this held output, press ENTER. The held output will be displayed starting a few rows down from the top of the screen. The first such line becomes the "referenced line", its number appears in the first screen row after the *LINE* heading, and the line itself is usually highlighted in some way on the screen (e.g. painted a distinguishing colour).

When APL is not ready for you to type, the left part of the top screen row will appear as follows:

L RUN

The *L* indicates that the screen is **locked**. To send an "attention" signal to APL in this state, press PA1.

If APL is ready for input, and you wish to send an input interrupt to APL, press PA1 also.

Vertical Scrolling, Paging and Searching

To "browse" through the session log, you can use a number of session manager **commands**. For example:

To scroll backward 5 lines type:

```
-5
```

after the *LINE* heading on the top screen row, and then press ENTER. The **referenced line** will decrease by 5.

To scroll forward 14 lines, type:

```
+14
```

To force the referenced line to be line 23, type:

```
23
```

in the same field.

To "page" backward exactly 1 screen display type:

```
PAGE-1
```

in the field following the *U INP* status display (the **command area**), and then press ENTER.

To page forward, type *PAGE*+1 in the same field.

To search backwards from the referenced line for the string *)LOAD* type *BACK* ')*LOAD*' in the command area. The referenced line will become the first preceding line that contains the string (if such a line exists).

To search forward, type *FORW* ')*LOAD*' in the same area.

If, as a result of scrolling or paging, the area for typing APL input is no longer on the screen, press ENTER (with no other changes to the screen) to revert to an "end of log" display that does contain that area.

by typing over it (or using the insertion and deletion facilities of the terminal).

If you use a combination of the above devices, the input line will be replaced with the catenation of the lines.

Automatic insertion: assign a PF key (say, PF key 6) with an *INS* command:

```
PFK 6,')INS C,'')LOAD 666 BOX'''
```

Then, with the cursor in the input area, press PF key 6. The string *)LOAD 666 BOX* will be inserted where the cursor indicates. If you also wish the line to be submitted to APL immediately, assign the PF key as follows:

```
PFK 6,')INS C,'')LOAD 666 BOX''/ENTRY F'
```

(This assigns a sequence of 2 commands: the second specifies that entry to APL is to be **forced**).

Normally, the submission to APL of the input line you have prepared will take place when you position the cursor somewhere on the line and press ENTER (without, at the same time, "moving" (e.g. scrolling) the screen). You can avoid submitting the line to APL, therefore, by placing the cursor on the first screen row (e.g. press ALT BACKTAB), and pressing ENTER. The input line will not be submitted, and the cursor will be placed at the end of the line.

Some Other Commands

HELP - to display a list of commands.

HELP LINE - to remind you of the syntax of the *LINE* command.

DEL C - to delete lines from the log from the cursor to the end of the log.

There are also commands to control the screen overlap of lines (*ROLL*), whether to pause when the screen becomes full (*AUTOPG*), and to control the colour/highlight attributes of the screen fields (*FIELD*).

Disconnection From The Session Manager

To disconnect yourself from the session manager, sign off the APL task as normal (*)OFF*); then press PA1 or type *)OFF* in the APL input area.

If you sign off the APL task but do nothing more, you will be disconnected automatically after a few minutes.

Introduction

The APL interpreter can be considered from an "outside" perspective to be a transaction processor: in response to a line of input from the user, APL produces one or more lines of output. Taking this sequence of lines of input and output as the manifestation of a "session", the **SHARP APL Session Manager** is a facility, operating outside APL, which manages such a session in the following sense:

- the next line of APL input is maintained in such a way that it can be submitted to APL when the user is ready to do so;
- a more or less permanent log of the session's input and output lines is kept (as a sequence of lines) which can be processed to some extent by the user; for example, browsing through, or retrieving, information in the log.

The session manager can be considered to be a facility operating at the "gateway" to (but outside) APL.

The 3270 **terminal handler** is a facility that enables a 3270 terminal to be used to run an APL session. The user of the session gains access to APL through the facilities of the session manager; for example,

- parts of the session log can be displayed on the terminal screen;
- the the next input line can be displayed on the screen for possible amendment, and submission to APL.

Summary of Facilities

The SHARP APL session manager provides the following facilities:

- **next input line:** the next line of input for eventual submission to APL is maintained in such a way that it can be changed repeatedly prior to its submission.
- **the session log:** when a line is submitted to, or received from, APL, it is appended to the session log. It is possible to submit lines to, and receive lines from, the session manager itself (i.e. such lines are never seen by APL). These are also appended to the session log.
- **Session Manager Commands:** a command language is available, by which the user may explictly obtain some of the services of the session manager.
- **Program Function Keys:** certain command sequences can be associated with numbered "program function keys", so that they may be conveniently invoked.
- ***)SESM* Command:** this pseudo-APL system command can be used as one of several ways of entering session manager commands.
- **Session Manager Command Auxiliary Processor (AP6):** an auxiliary processor is available for submitting session manager commands under the control of an APL program.

The 3270 terminal handler provides the following facilities:

- **Screen Display:** selected (contiguous) parts of the session log, and the next input line can be displayed on the screen.
- **Input line amendments:** the next input line can be altered by amending the line as displayed on the screen; by amending a line or lines of the session log as displayed on the screen; or simply by placing the cursor on such a displayed line.
- **Sending input to APL:** the user can control when the next line of input is sent to APL, by positioning the cursor on the screen.
- **Display of session status:** some aspects of the the status of a session are dispayed on the screen (e.g. whether APL is in "run" or "input" mode).
- **Session manager row:** an area of the screen is available as one of several ways of submitting session manager commands.
- **3270 terminal commands:** some special session manager commands are available which pertain to the terminal itself (e.g. a command to control the colour/highlighting of areas of the screen).
- **Data and non-data keys:** actions are associated with the various keys: PA1, PA2, CLEAR, etc.
- **□*ARBOUT*/□*ARBIN* interpretation:** an interpretation of APL *ARBOUT* strings is provided which is meaningful especially (and only) for 3270 terminals. This consists primarily of the ability to write and read 3270 datastreams.
- **Fullscreen auxiliary processors (AP124/AP126):** two auxiliary processors are available for writing fullscreen applications.

It is useful to distinguish the session manager from the terminal handler, but in what follows, the combination of the two will simply be called the **session manager**.

There follows a more detailed account of the facilities and terms summarized above.

Mode Of APL

The **mode of APL** at any time is either:

- **input** mode: APL has sent the user a prompt and is expecting input from the user, or
- **run** mode: APL has not yet produced the next user prompt.

The Next Input Line

The **next line of APL input** may be changed any number of times prior to its submission to APL, and may be changed whether APL is in "run" mode or "input" mode. (See below for details on ways of changing the line).

The line is submitted to APL when:

- APL is in input mode;

- the preparation of the line is considered to be complete (see below for details)

After submission to APL, the next input line becomes an empty line. When APL changes from run mode to input mode, the last line of output (the **prompt**) is catenated to the front of the next input line.

The Session Log

When a line is submitted to, or received from APL, it is appended to the session log. This log is permanently available until the user disconnects (or is disconnected) from the session. A line of output from APL is considered complete when a carriage return or linefeed character is received. These characters are not included in the line, and a received linefeed character causes an appropriate number of blanks to be placed at the beginning of the next line. Any backspace characters are resolved by moving back in the "line so far" so that subsequent characters will **replace** those already received. The first 2 characters of *⎕AV* are suppressed.

As described later, it is possible to submit lines to, and receive lines from, the session manager itself (i.e. such lines are never seen by APL). These are also appended to the session log.

The lines in the log are stored in such a way that it is possible to determine whether the line was a line to or from APL, or to or from the session manager. In the case of lines submitted to APL, the length of any false prompt (a prompt which was produced with the use of *⎕ARBOUT* '') is also kept.

The lines in the log are numbered sequentially from 0. Since there is a physical limit on the size of the session log, when a line is appended to the log, room will be made for it if necessary by deleting lines from the beginning of the log. The lines are not renumbered when this is done, however.

A number of quantities are associated with the session log which are useful in dealing with it:

The Referenced Line:

At any time, one of the lines of the session log (or the next input line) is designated as the **referenced line**. It can be thought of as "the line we are currently looking at". It can be changed in a number of ways, including explicit session manager commands, and provides the basic means of "browsing" through the session log.

The Referenced Column:

Similarly to the referenced line, one of the columns is designated as the currently **referenced column**. It can be changed by session manager commands, and provides a means for horizontal browsing. The leftmost column is numbered as column 0.

A Display Page:

For display purposes, lines from the session log (and the next input line) can be arranged on a rectangular page. Normally, a long line will be wrapped onto as many page rows as are needed to display the whole line.

The **page width** and the **page depth** are the number of columns and rows, respectively, of the display page.

The Line Display Mode:

For display purposes there are two **line display modes**:

- **wrapped**: in which a line of the session log will be displayed in full, occupying as many page rows as necessary;

- **clipped**: in which only as much of a line as will fit on one page row will be displayed. In this case, the first character of the line displayed will be that specified by the referenced column. In this mode, it is possible to perform horizontal browsing.

The Referenced Line Offset:

The **referenced line offset** is the number of page rows which will normally precede the referenced line when displayed. When >0, this provides a way of displaying some of the preceding context of the referenced line.

Session Manager Commands

Some of the facilities of the session manager can be explicitly requested by submitting session manager commands. There are a number of mechanisms for submitting such commands:

- *)SESM* pseudo-APL system command;
- the session manager row on the 3270 screen;
- defining and invoking program function keys;
- the session manager command auxiliary processor (AP6).

(See below for details on these various devices.)

The general form of a command is:

commandword argumentlist

It is possible to submit a sequence of commands by making the first character a non-alphabetic character and using this as the command separator. For example:

/LINE I0/ENTRY F is a sequence of 2 commands

The execution of a command results in:

- A non-negative integer completion code. 0 indicates "successful" completion of the command; the non-zero "failure" codes are listed in the section titled "Miscellaneous Messages". These non-zero codes are only of significance when the command was issued by means of the session manager command auxiliary processor (AP6).

- Zero or more character vector results. These results are manifested in ways which depend on how the command was issued. For instance, if the command was issued from the session manager row on the 3270 screen, the result(s) may appear in the command area of that row, or be appended to the session log.

The commands in a command sequence will be executed one after the other up to and including the first one which results in a non-zero completion code.

Program Function Keys

Up to 24 **program function keys**, numbered 1 to 24, can be defined. Each such key can be defined to be a session manager command or command sequence, and provides a convenient way to invoke commonly used commands. The keys can be defined by means of the session manager command *PFK*, and they can be invoked by means of the command *XPFK*, or by pressing a 3270 physical program function key.

)SESM Pseudo-system Command .

The system command *)SESM* is provided as one of the means of submitting session manager commands. It can be used in any circumstance where an APL system command could be used (i.e. immediate execution), but it is intercepted by the session manager and is never seen by APL. Its argument can be any session manager command or command sequence, and its result(s) are appended to the session log. For example:

```
     )SESM PFK 1
PFK 1,'PAGE-1'
```

General Session Manager Commands

The general session manager commands are:

LINE: To alter the referenced line (and referenced line offset), or to query the number of the referenced line.

LINE [(+|-) **integer**
| **lineno**[,**integer**]]

LINE
- the result is the number of the currently referenced line.

LINE **lineno**
- the referenced line becomes the line represented by **lineno**, with the default offset.

LINE **lineno**,**integer**
- the referenced line becomes the line represented by **lineno**, with offset **integer**.

LINE +**integer**
- the referenced line is increased by **integer**.

LINE -**integer**
- the referenced line is decreased by **integer**.

lineno may be one of the following:

integer - representing line number **integer** of the session log.
*I*1 - representing the input line.
R - representing the currently referenced line.
C - representing the line containing the cursor on the 3270 screen display.

Z - representing the line immediately following the last line of the log.

If the line number specified is prior to the first existing line, the referenced line will become this first existing line; if the line specified is beyond the last existing line, the referenced line becomes one beyond the last existing line.

COL: To change the line display mode and, if the mode is "clipped", to change the referenced column or query its number.

COL [*W* | colno | (+|-) integer]

COL W
- the display mode becomes "wrapped", and the referenced column becomes 0.

COL colno
- the display mode becomes "clipped" and the referenced column becomes **colno**.

colno may be one of the following:

integer - representing column number **integer**
R - representing the currently referenced column
C - representing the column containing the cursor on the 3270 screen

COL + **integer**
- the display mode becomes "clipped" and the referenced column is increased by **integer**.

COL - **integer**
- the display mode becomes "clipped" and the referenced column is decreased by **integer** (to a minimum of 0).

PAGE: To increase or decrease the number of the referenced line so that the displayed page will skip forward or back an exact number of display pages.

PAGE (+|-) **integer**

PAGE + **integer**
- the referenced line is changed so that the displayed page will skip forward exactly **integer** pages, unless this would place the referenced line beyond the end of the session log, in which case the referenced line becomes the next input line.

PAGE - **integer**
- the referenced line is changed so that the displayed page will skip backwards exactly **integer** pages, unless this would place the referenced line before the first existing line, in which case the referenced line becomes this first existing line.

FORW: To search forward for a string in the session log.

FORW **string**

where **string** is a quoted string

The referenced line is increased to be the first line beyond the current one which contains the string **string**, if such a line exists. If there is such a line, the referenced line offset is set to the default. If there is no such line, the command fails, with completion code 100, and character result '* *STRING NOT FOUND*'. The referenced line and offset are not changed.

BACK: To search backward for a string in the session log.

BACK **string**

where **string** is a quoted string

The referenced line is decreased to be the first line before the current one which contains the string **string**, if such a line exists. If there is such a line, the referenced line offset is set to the default. If there is no such line, the command fails, with completion code 100, and character result '* *STRING NOT FOUND*'. The referenced line and offset are not changed.

DEL: To delete any trailing section of the session log.

DEL **lineno**
- the lines of the session log from **lineno** to the end of the log are deleted.

lineno can be:

integer - representing session log line number **integer**;
R - representing the currently referenced line;
C - representing the line of the session log containing the cursor of the 3270 screen.
Z - representing the line immediately following the last line of the log.

INS: To change the next input line by inserting a specified string, either in the next input line itself, or a copy of a session log line.

INS **charpos,string**

where **string** is a quoted string, and

where **charpos** represents a line, and a character within that line, and can be one of the following:

integer1 integer2
- representing the **integer2**'th character of session log line number **integer1**;

*I***1 integer2**
- representing the **integer2**'th character of the next input line;

C - representing the line, and the character within the line, indicated by the cursor on the 3270 screen. The line may be either a session log line or the next input line. If the cursor is not on the APL part of the screen, it is taken to represent the end of the next input line.(If the cursor is not on the APL screen, *C* represents the end of the next input line).

The next input line is amended as follows:

- if **charpos** represents the next input line, the string **string** will be inserted immediately before the character indicated;
- if **charpos** represents a line of the session log, the next input line will be replaced by the result of inserting the string **string** immediately before the character indicated, in that line of the session log. The session log line itself is not altered, however.

HELP: To obtain some prompting about commands and how to use them.

HELP [**commandword**]

HELP
- the result is a list of valid session manager command words.

HELP **commandword**
- the result is an indication of the syntax of the command **commandword**.

ENTRY: To explicitly control whether or not the next input line is submitted to APL.

ENTRY F | B

ENTRY F - the next input line will be submitted to APL if APL is in input mode;

ENTRY B - the next input line will be prevented from being submitted to APL even if APL is in input mode.

The *ENTRY* command can be used in conjunction with the *INS* command to specify whether or not the line constructed by the *INS* command is to be submitted to APL immediately. Thus:

/INS C,')LOAD 666 *BOX'/ENTRY F*

or

/INS C,')LOAD 666 *BOX'/ENTRY B*

or in conjunction with the *LINE* command, for instance to combine a scrolling of the display with the immediate entering of the next input line. Thus:

/LINE I0,0/ENTRY F

ROLL: To set or query the default value of the referenced line offset.

ROLL [**integer**]

ROLL
- the result is the current default value of the referenced line offset.

ROLL **integer**

- the default value of the referenced line offset becomes **integer**.

PFK: To set or query the strings associated with the program function keys.

PFK [**integer**[,**string**]]

where **string** is a quoted string

PFK
- the results are the strings currently associated with all program function keys (excluding those that have empty strings associated with them). Each result is in the form:

  PFK **integer**,**string**

PFK **integer**
- if the string currently associated with PFKEY number **integer** is empty, there is no result; otherwise the result is that string, in the same form as above.

PFK **integer**,**string**
- the string associated with PFK number **integer** becomes **string**. The string can be empty.

XPFK: To issue the command or command sequence associated with a program function key.

XPFK **integer**
- the string associated with PFKEY **integer** is processed as a command or command sequence.

GET: To retrieve a block of lines from the session log.

GET **lineno1** **lineno2**
- the results are lines **lineno1** to **lineno2** inclusive of the session log.

Where **lineno** can be as for the *DEL* command.

If either **lineno1** or **lineno2** are outside the limits of the session log, they will be forced to the first or last lines of the log.

(Note: If **lineno2** is Z, the results are the lines from **lineno1** to the end of the session log.)

The principal use of the *GET* command is to retrieve data from the session log by issuing the command via AP6. However, if issued from the 3270 screen, it has the effect of copying a block of lines to the current end of the log.

Some commands which are related exclusively to the operation of the 3270 screen are detailed later.

The 3270 Screen

The 3270 screen is divided into two areas:

- The **session manager row**: to communicate exclusively with the session manager;

- The APL screen: to communicate principally with APL.

The Session Manager Row:

The first row of the screen is used as follows:

- Screen indicator:

  'U' : means that the 3270 screen is "unlocked". In this state the "next turn" with the screen is the user's: the session manager will not update the screen until the user has entered a screen transaction (i.e. pressed a data or non-data entry key);

  'L' : means that the 3270 screen is "locked". In this state the "next turn" belongs to the session manager: anything entered by the user will be ignored, except the pressing of a non-data key.

- Session Log Indicator:

  'H' : ("Holding") indicates that there are lines in the session log (or all or part of the next prompt from APL) that have not been seen by the user (this normally occurs when the screen is full).

- APL Mode Indicator:

  INP : means that APL is in input mode;
  RUN : means APL is in run mode;
  KEYB : means that APL is in run mode because the APL system command *)KEYB LOCK* has been issued.

- Command area: a field for the user to enter a session manager command or command sequence, and for a result of a command to be displayed.

- "Line" Area: a field for the user to enter an argument to the *LINE* command, and for the current number of the referenced line to be displayed. The area is preceded by a protected field containing the text *LINE*.

- "Col" Area: a field for the user to enter an argument to the *COL* command, and for the current column mode (or if this is "clipped", the current number of the referenced column) to be displayed. The area is preceded by a protected field containing the text *COL*.

The screen, session log and APL mode indicators are protected fields; the command, *LINE* and *COL* areas are unprotected.

Any time that the screen is unlocked, session manager commands can be entered, either in the command area, or in the case of the *LINE* and *COL* commands, by entering just the command argument in the line or column area. Any single result, if it will fit in the command area, will be displayed there. Otherwise the command (with the characters *)SESM* prepended), and its result(s), will be appended to the session log, and hence displayed on the APL screen.

The APL screen:

The remaining rows of the screen are devoted to displaying parts of the session log and the next input line. The contents of the display are normally determined as follows:

- The lines of the session log displayed will be a block of consecutively numbered lines;

- The referenced line will be displayed starting *N*+1 rows down the APL screen (where *N* is the referenced line offset);

- The parts of each line to be displayed will be determined by the column mode ("wrapped" or "clipped"), and if the latter, by the value of the referenced column;

- If the end of the session log is reached before filling the screen, the next input line, or as much of it as will fit, will be displayed immediately following the last session log line.

Each line of the session log occupies a separate, unprotected, 3270 screen field (possibly taking up more than one row if the display is in wrapped mode); the next input line occupies a separate unprotected field, extending over several rows but not necessarily to the end of the screen. The area beyond the next input line is protected.

Preparation and Submission of Next Input Line

The next input line can be changed in several ways:

- by the use of the *INS* session manager command (note that this can even be done when the screen is locked, by using AP6);

- if the screen is unlocked, by using one or more of the following devices in a single screen transaction:

  1) altering the displayed next input line itself;

  2) altering a displayed line of the session log, leaving the cursor somewhere other than on this line;

  3) positioning the cursor somewhere on a displayed line of the session log, with or without altering the line.

The next input line will be replaced by the result of catenating the individual lines (amended or not), in the above order. In the case of 2), if more than one such line is amended, the order of catenation will give precedence to the lowest numbered line.

Note that any such preparation of the next input line can be carried out regardless of whether APL is in input or run mode. Hence, preparation in anticipation of APL's next prompt is possible. For example, unlock the screen by pressing PA2 (or its equivalent), enter the next line of input on the screen, and press ENTER to lock the screen again.

The preparation of the next input line is considered to be complete when:

- the screen is in a "resting state" (see below); and
- the screen did not "move" as a result of the last transaction (i.e. the referenced line and offset did not change); and

- a command was not entered from the session manager row; and
- a program function key was not pressed; and
- the cursor was placed somewhere on the next input line.

(In brief, everything is as stationary as possible, and the cursor is on the next input line.)

If APL is in run mode when the preparation is complete, the screen will simply be locked.

If APL is in input mode, the next input line will be submitted to APL if either:

- the command *ENTRY F* was issued; or
- the command *ENTRY B* was not issued, and preparation of the next input line is considered to be complete according to the above list of conditions.

Non-Data Entry Keys

The non-data entry keys, PA1, PA2, and CLEAR have the following meanings:

PA1 - communicates (mostly) with APL:

- when signed-on to APL:
  - if APL is in run mode: an attention is sent to APL;
  - if APL is in input mode: if the screen is not in a resting state, no action is taken; if it is, the character sequence *O*, backspace, *U*, backspace, *T* is catenated to the end of the next input line, which is immediately submitted.

- when not signed-on to APL:
  - the terminal is disconnected from the session manager (alternatively, submitting *)OFF* as the next input line when not signed-on also causes a disconnection).

PA2 - communicates with the session manager
  - if the screen is unlocked: the screen is completely refreshed, and any lines at the end of the session log that have not yet been displayed, are deleted from the session log.

  - if the screen is locked: the screen is completely refreshed, and unlocked. Also, any "long" session manager command that is currently being processed may be interrupted and aborted. Such commands are *GET*, *FORW* and *BACK*; i.e. those for which the amount of processing required is dependent on the amount of session log data involved.

CLEAR - the session manager command '*LINE I*1,0' is executed and the screen is refreshed. Thus, a clear APL screen, with the next input line at the top is displayed.

Note: On some terminals, when the screen is locked the only keys that can be pressed are ATTN, and SYSREQ. On such terminals, the following equivalences apply when the screen is locked:

- ATTN is equivalent to PA1
- the keystroke sequence SYSREQ, SYSREQ, ATTN is equivalent to PA2.

Updating of the APL Screen

The APL screen will be updated when lines are appended to the session log, until the screen becomes full. If there are lines not yet displayed at this point, the referenced line will be advanced to the first session log line not yet seen. Thus the next screen display will show the further lines of the session log, normally with an overlap of the few screen rows (due to the referenced line offset). This is the normal "forward moving" display of the session log, but it can be altered at any time by explicitly setting the value of the referenced line with the session manager commands *LINE*, *PAGE*, *FORW*, or *BACK*; i.e. scrolling, paging or searching can be performed at any time.

In general, therefore, the display on the APL screen may be in either:

- a resting state: the screen is displaying the last zero or more lines of the session log, and all of the next input line; or

- a paged-back state: any non-resting state.

The general rules for transition from one screen display to another are as follows:

- an explicit setting of the referenced line (and possibly the referenced line offset) will determine the contents of the next display; otherwise,

- if a command is entered on the session manager row, or a program function key is pressed, then the referenced line and offset will not change (hence, the display will not "move"), unless of course the commands entered were ones that explicitly change these quantities; otherwise,

- the referenced line and offset will be set such that the last display that was in a resting state will be re-displayed.

Thus, if nothing was entered on the session manager row, pressing ENTER will result in a display in a resting state.

The rules for where the cursor will be positioned when a screen display is generated are as follows:

- if the screen is locked: the cursor will be placed at the end of the APL mode indicator (a protected position);

- if the screen is unlocked:

- if, as part of the preceding screen transaction, the user entered a command argument into the *LINE* or *COL* areas of the session manager row, the cursor will be placed in that area; otherwise,

- an attempt will be made to position the cursor in the next input line: if the next input line is the result of an *INS* command, at the end of the insertion; otherwise at the end of the next input line; if this position is not on the screen:

- the cursor will be positioned on the session manager row, at the start of whichever of the command, *LINE* or *COL* areas was last used as part of a screen transaction.

In other words, the cursor will tend to stay in the *LINE* or *COL* areas as long as one of them is used. Failing that, it will try to find a sensible place in the next input line; failing that, it will resort to whichever part of the session manager row was most recently popular.

The *ROLL* Setting

The *ROLL* setting, a non-negative integer less than the number of rows on the APL screen, specifies the default value of the referenced line offset. Hence, it controls the amount of screen overlap in normal "forward moving" display of the session log. It can be set by means of the *ROLL* session manager command.

The *AUTOPAGE* Setting

When the screen becomes full during the normal forward moving display of the session log, the user can control whether or not the screen is to be immediately updated with the next display. The *AUTOPAGE* setting controls this decision:

- *OFF*: when the screen becomes full, an *H* will be displayed in the session log indicator, and the screen will unlock, giving the user control over when to display the next page. (Note: this will not happen if the last session log line on the APL screen is input to APL);

- *ON*: when the screen becomes full, it will not unlock, but will immediately display the next page.

The *AUTOPAGE* setting can be controlled by means of the *AUTOPG* session manager command.

The Overstrike Escape Character

It is possible to enter certain characters, for which there is no key on the terminal, as a sequence of 3 characters: an "overstrike escape character" (OEC), and 2 other characters defined at a particular installation to be the "overstrike" combination which replaces the unavailable character. For instance, if the OEC is ¯, the sequence ¯<> might be defined to be the APL "diamond" character.

The OEC can be defined by means of the *OEC* session manager command.

Input Line Overlaying

When the display is in a paged back state, the next input line cannot be displayed. In this state, it is possible to specify that the bottom n rows of the APL screen be devoted to display of the next input line. This affords the user the convenience of being able to amend the line whilst in this state. This device is available through the *INP* session manager command.

Colour/Highlight Attributes

The colour and highlight attributes of the various fields on the screen can be controlled by means of the *FIELD* session manager command. There are a number of distinguished **field types**:

MODE - the screen, session log, and APL mode indicators
TITLE - the (protected) *LINE* and *COL* titles

PARMS - the *LINE* and *COL* areas, and the command area.
REFLINE - the referenced line
UNSEEN - when scrolling forward, the first line of the session log which was not displayed on the previous display; when scrolling backward, the last line that was not displayed on the previous display
SESM - a line of the session log that was input to or output from the session manager
INOUT - a line of the session log which was input to APL
OUT - a line of the session log which was output from APL
PROMPT - a "false" APL prompt (i.e. that part of the prompt which was produced by using □*ARBOUT* '')
INPUT - the next input line

In the case of a 3270 terminal with extended features, the colour and highlight attributes of these field types can be specified (including a specification of "neutral"); for a terminal without extended features, each field type may be specified to be either "high" or "neutral". If either of the attributes for field type *REFLINE* are specified to be "neutral", that attribute will default to that of *SESM*, *INOUT* or *OUT* as appropriate. The same is true for field type *UNSEEN*. The specifications for *REFLINE* take precedence over those for *UNSEEN* if a line falls in both categories.

3270 Session Manager Commands

The following session manager commands are related specifically to the 3270 screen:

AUTOPG: To control or query the autopage setting.

AUTOPG [*ON*|*OFF*]

AUTOPG
- the result is the current value of the autopage setting ("on" or "off").

AUTOPG ON - the autopage settings becomes "on".

AUTOPG OFF - the autopage setting becomes "off".

FIELD: To specify or query the colour/highlight attributes of the distinguished screen field types.

FIELD [fldtype[,[colour [highlight]]]]

where fldtype is one of the distringuished field types;
colour is *NEUTRAL* or one of the colours:

BLUE, *RED*, *PINK*, *GREEN*, *TURQUOISE*, *YELLOW*, *WHITE*

or, in the case of a terminal without extended features:

HIGH;

highlight is *NEUTRAL* or one of:

BLINK, *REVERSE*, *UNDERLINE*

FIELD
- the results are the colour/highlight settings for all field types, each one in the form:

  FIELD fldtype,colour highlight

FIELD fldtype
- the result is the colour/highlight setting for the field of type **fldtype**, in the above form.

FIELD fldtype,
- the colour/highlight setting for the field of type **fldtype** becomes *NEUTRAL/NEUTRAL*.

FIELD fldtype,colour
- the colour setting for the field of type **fldtype** becomes **colour**.

FIELD fldtype,colour highlight
- the colour/highlight setting for the field of type **fldtype** becomes **colour/highlight**.

INP: To take advantage of the "input overlaying" device.

INP integer
- if the screen is in a paged back state, the bottom **integer** rows of the screen will be devoted to display of the next input line. Amendments to this line can then be made in the usual way. The effect of this command persists through changes in the referenced line as long as the screen remains in a paged back state, but is cancelled as soon as the screen returns to a resting state.

OEC: To establish (or revoke) the overstrike escape character.

OEC [*OFF*|character]

where **character** is a single quoted character,

OEC
- the result is the current OEC, or *OFF* if there is none.

OEC OFF
- the current OEC is revoked, so that no OEC is defined.

OEC character - the OEC becomes **character**.

The Session Manager Command Auxiliary Processor (AP6)

The session manager auxiliary processor allows the APL task to which the session is attached to issue session manager commands from an APL program. The most immediate use of this AP is to set the various session parameters (referenced line offset, *AUTOPG* setting), the program function keys, and the field colour/highlight attributes with an APL function, but there are other possible applications such as automated browsing through the session log.

The processor id of the AP is 6, and communication with the APL task is via 1 or 2 shared variables. The name of the first

variable must begin with the characters *CTL*, and the second with *DAT*. If 2 variables are used, their names must be identical after the first 3 characters. (The only reason to use 2 variables is if an application has problems with space in the workspace.)

Communication with the AP via a *CTL* variable only is achieved by sending a chain of command sequences as follows:

CTL ← (0 ⊃ *CMDSEQ*1),(0 ⊃ *CMDSEQ*2), ... ,(0 ⊃ *CMDSEQ*(*N*))

where each *CMDSEQ* is a character vector consisting of a session manager command sequence. The sequences in the chain will be executed by the session manager one after the other, up to the first element in the chain which is not of the form 0 ⊃ *CMDSEQ*. The command sequence in a chain element will be executed by the session manager up to the first command in the sequence which gives a non-zero completion code.

On completion of processing, *CTL* will be set to:

CRC ⊃ (<*ERC*1 ⊃ *ERES*1),(<*ERC*2 ⊃ *ERES*2), ... ,<*ERC*(*M*) ⊃ *ERES*(*M*)

where:

- *M* is such that the (*M*+1)th chain element was the first invalid element
- *CRC* is 0 if all chain elements were valid; otherwise, a positive integer code indicating the problem with the first invalid chain element. The meanings of these codes are given in the section titled "Miscellaneous Messages";
- *ERC*(*I*) is the completion code of the last command executed in the *I*'th command sequence.
- *ERES*(*I*) consists of the results of the commands in the *I*'th sequence:

  (<*CRES*1),(<*CRES*2), ... ,<*CRES*(*S*)

  where *CRES*(*J*) consists of the results of the *J*'th command in the *I*'th command sequence:

  *RES*1 ⊃ *RES*2 ⊃ ... ⊃ *RES*(*C*)

  Each such *RES*1 is a character vector in which the first 2 characters are control information as follows:

  The 1st character is an encoding of the type of the result, ⎕*AV*[⎕*IO*+2⊥*LOGF*,*SSMF*,*INPF*], where:

  LOGF - 0 means the result is a line from the session log; 1 means it is not;

  and if *LOGF* ↔ 0:

  SSMF - 0 means the line was to/from APL; 1 means the line was to/from the session manager;

  INPF - 1 means the line was input to APL or the session manager; 0 means it was output.

If the result is a session log line which was input to APL in which the prompt resulted from a use of ⎕*ARBOUT* '', the 2nd character is ⎕*AV*[⎕*IO*+*L*], where *L* is the length of this prompt.

Communications with the AP via *CTL* and *DAT* variables is achieved as follows:

DAT ← *CMDSEQ* ⋄ *CTL* ← 0

where *CMDSEQ* is a (character vector) command sequence as above.

On completion of processing, *DAT* will be set to *ERES* as above, and *CTL* will be set to *ERC* as above.

Note: Workspace 5 *IBM3270* contains a function *IDSHPROF*, which is an example of the use of AP6 to establish PF key definitions, etc.

Miscellaneous Messages

The non-zero completion codes for commands submitted to AP6 are:

```
  1  - invalid command word
  2  - invalid parameter to a command
  3  - command recursion too deep
  4  - command aborted by user
  5  - result too big

100  - string not found: FORW, BACK
101  - next input line too long: INS
102  - PFK definition too long: PFK
```

The codes resulting from incorrect chain elements submitted to AP6 are:

```
 11  - CTL rank error
 12  - CTL length error
 13  - CTL domain error
 21  - DAT rank error
 22  - DAT length error
 23  - DAT domain error
 41  - no DAT variable value
 42  - output buffer overflow
```

The following messages from the session manager may be appended to the log:

APL NOT RESPONDING
This usually means either that APL is not running, or that resources required for the session manager to connect to APL are not available

NO SVP
A fundamental required resource, the SHARP APL shared variable processor, is not available.

Note: the spelling of these messages can be altered by a site.

In addition, it is possible for the session manager to issue PA messages to all its users.

There are three full screen management interfaces.

AP124 is the simplest and most efficient and should be adequate for most applications. It uses a CTL/DAT type shared variable interface.

AP126 is an interface to the IBM product GDDM (Graphical Data Display Manager) and provides all of the facilities of AP124 plus extensive colour graphics capabilities. It uses a CTL/DAT type shared variable interface.

The ARBIN Manager is the lowest level interface and gives the user **complete** control of the device. It uses the □*ARBIN* and □*ARBOUT* primitives to send and receive arbitrary data streams. In addition the ARBIN manager supports program control of some parts of the Session Manager.

Using AP124, an application may define and manipulate rectangular areas called SCREEN FIELDS, or FIELDS. In addition to area, a field has attributes, such as highlighting, input protection, or color. Area and attributes can be set during the initial field definition and reset afterward using full screen commands. These specifications are made in the form of a format matrix (discussed later).

Once the appropriate variables have been shared with AP124, an application can assign operation codes directly, or through the use of cover functions contained in one of the full screen management workspaces in public library 5. Full screen commands can be grouped into three areas, namely

1) Operations to define fields
2) Operations to write and read
3) Miscellaneous operations

AP124 allows an application to pass full screen operations and data using a pair of shared variables (prefix CTL and DAT) or in a single CTL variable if the data is in enclosed form. Below is a list of operation codes and their descriptions.

CTL	DAT	Description
0	*	no operation
1	fmt matrix	format screen into fields
1,fld	fmt matrix	reformat screen into fields
2,fld	data	write to screen immediately
3	*	read from screen and wait
4,fld	data	buffered write to screen
5,fld	*	get data
6,fld	type	change field type
7,fld	intensity	change field intensity
9	*	read format matrix
11	*	sound alarm
12	position	set cursor position
16	attribute	change field attributes
20	*	erase the screen
24,fld	color	change field color
26,fld	highlight	change field highlighting
28	*	return raw data stream for this screen
30	*	get data for modified fields
31	*	get data for modified fields but in enclosed form
35,fld	character attributes	buffered write of character attributes
37,fld	character attributes	immediate write of character attributes
40	*	get screen data
41	*	get screen attributes

Several pairs of variables may be shared simultaneously with AP124 allowing several independent logical screens. To allow this, full screen variable naming conventions have been adopted. The first three letters of the control variable must be CTL, and similarly, DAT for the data variable. Each variable may be up to eleven characters long and are paired by suffix to identify a

logical screen. Since AP124 has no outstanding variable offers, the degree of coupling will be:

1 - after the application's initial offer
2 - when AP124 accepts the offer

For operations requiring both a control variable and a data variable, the data variable is assigned first. A non zero return code in the control variable means that an error occurred during the operation. Below is a list of return codes and their descriptions.

```
  0 NORMAL RETURN -  OPERATION SUCCESSFUL
 11 CONTROL VARIABLE RANK ERROR
 12 CONTROL VARIABLE LENGTH ERROR
 13 CONTROL VARIABLE DOMAIN ERROR
 14 INVALID COMMAND
 15 REQUEST TO POSITION CURSOR IN UNDEFINED FIELD
 21 DATA VARIABLE RANK ERROR
 22 DATA VARIABLE LENGTH ERROR
 23 DATA VARIABLE LENGTH ERROR
 24 DATA VARIABLE NOT SHARED
 30 INVALID FIELD NUMBER
 32 DEFINED FIELD EXTENDS BEYOND THE SCREEN
 33 REFERENCE OUTSIDE FIELD DEFINITION
 35 LIGHT PEN FIELD STARTS IN COLUMN 1
 36 LIGHT PEN FIELD (HEIGHT 1) NOT CONTAINED
    IN 1 PHYSICAL SCREEN LINE
 37 INVALID FIELD TYPE
 38 INVALID FIELD INTENSITY
 99 UNKNOWN 3270 DEVICE ERROR
201 INVALID COLOR
202 INVALID HIGHLIGHT
204 INVALID PROGRAM SYMBOL SET
205 INTERNAL BUFFER OVERFLOW
206 INVALID SKIP COLUMN IN FORMAT MATRIX
207 INVALID NULLS COLUMN IN FORMAT MATRIX
```

An application may request many full screen operations with a single shared variable use by passing several CTL and DAT values together as an enclosed vector. In this case, the CTL control variable is used for both input and output, and the DAT data variable need not be shared. When the CTL argument is enclosed, AP124 automatically returns an enclosed result, consisting of an enclosed return code element, followed by enclosed data elements resulting from full screen operations and normally found in DAT. When disclosed, the return code element is simple and has two numbers, specifying completion code (as discussed earlier), and the number of operation codes processed. Processing terminates when an error occurs, so in the event of a non zero completion code, the second number indicates how many operation codes were sucessfully processed.

1) OPERATIONS TO DEFINE FIELDS

FORMAT the SCREEN

```
CTL=1
DAT=format matrix
```

The format matrix is numeric with a row describing each screen field. Columns specify starting row, starting column, depth, width, type, intensity, color, program symbol set, highlighting, skip, and trailing blank processing. A format matrix with fewer than eleven columns assumes default values from the following default format matrix.

1,1,device rows,device cols,2,1,0,0,0,0,1

a) TYPE is:
- 0 - character input/output allowed
- 1 - numeric character input/output allowed
- 2 - character output only (default)
- 3 - character output/light pen interruptable
- 4 - character output/light pen selectable

A field type of 2, 3, or 4 causes that field to be "protected", so data cannot be entered by the user.

b) LIGHT PEN FIELDS

Terminals with light pen support can have fields defined as being light pen sensitive as well as "normal" fields. A screen is "mixed" if it contains both light pen and non-light pen fields. The two types of light pen fields are "selectable" and "interruptable", and a light pen interruptable field can return a "light pen" or "enter" completion code.

All light pen fields must have a designator character written in the first column of the data. Also, the attribute and designator characters must appear on the same screen row (so light pen fields may not begin in column one). A maximum of twelve light pen fields, or fourteen mixed fields may precede the final light pen field on any row.

SELECTABLE FIELDS

In the case of selectable fields, one of two designator characters may be used: question mark (?) or greater-than (>). The ? indicates that the selectable fields has not been modified, and the greater than sign indicates that a field has been modified. Therefore an application may pre-select a field by initially writing a column of >'s. A selectable field switches between unmodified and modified each time the light pen is applied. If the designator character is illegal or omitted, AP124 changes the first column to "?" (unmodified). A field whose designator character is ">" upon input completion, is considered to have been modified.

INTERRUPTABLE FIELDS

An interruptable field has a designator character of blank or ampersand (&) (types 1 and 2 respectively). When a light

pen is applied to a type 1 field, input is completed, a light pen completion code is returned, and field numbers for modified selectable fields (modified non-light pen fields) are ignored. Type 2 interruptable (3276, 3278, 3279 only) fields return a completion code identical to that of "enter", and all modified field numbers, so it behaves as if the user had pressed the "enter" key. An illegal or omitted designator character is changed to a blank character.

c) INTENSITY is: 0 - do not display data
1 - normal intensity (default)
2 - high intensity

d) COLOR is: 0 - green (default)
1 - blue
2 - red
3 - pink
4 - green
5 - turquoise
6 - yellow
7 - white

e) PROGRAM SYMBOL SET: must be zero (default)

f) HIGHLIGHTING is one of: 0 - no highlighting (default)
1 - blink
2 - reverse video
4 - underscore

g) SKIP is: 0 - skip to next unprotected field when the cursor is advanced past the end of the current unprotected field (default)

1 - allow the cursor to move into the next screen area regardless of whether it is protected or not protected

h) BLANK PROCESSING is one of:
0 - nulls read as blanks, trailing blanks write as blanks
1 - nulls read as blanks, trailing blanks write as nulls
2 - nulls read as nulls, trailing blanks write as blanks
3 - nulls read as nulls, trailing blanks write as nulls

Before and after each row of each field, is an attribute character defining the following screen area. These attribute characters occupy a physical space, so allow for this when designing a panel (don't include this space as part of the width specification). When the format matrix SKIP column is zero (default), undefined areas are skipped when the cursor moves there during input.

REFORMAT the SCREEN

```
CTL=1,field numbers
DAT=format matrix
```

Used to reformat selected fields, the format matrix is identical to that defined in the FORMAT section except that each row now has a corresponding field number. Existing data is not erased during reformatting.

READ the FORMAT MATRIX

```
CTL=9
DAT=(none)
```

The result is the format matrix used to define the current screen.

SET FIELD TYPE

```
CTL=6,field numbers
DAT=type numbers
```

Field types for selected fields are reset.

SET FIELD INTENSITY

```
CTL=7,field numbers
DAT=intensity numbers
```

Intensities for selected fields are reset.

SET FIELD COLOR

```
CTL=24,field numbers
DAT=color numbers
```

Colors for selected fields are reset.

SET FIELD HIGHLIGHTING

```
CTL=26,field numbers
DAT=highlighting numbers
```

Selected field numbers' highlighting is reset.

SET FIELD ATTRIBUTES

```
CTL=16,field numbers
DAT=attribute numbers
```

Attribute numbers range from zero to seven and control the SKIP and BLANK PROCESSING bits described in the FORMAT section. Attribute values are:

0 - nulls read as blanks,
trailing blanks write as blanks, no autoskip

1 - nulls read as blanks,
trailing blanks write as blanks, autoskip

2 - nulls read as blanks,
trailing blanks write as nulls, no autoskip

3 - nulls read as blanks,
trailing blanks write as nulls, autoskip

4 - nulls read as nulls,
trailing blanks write as blanks, no autoskip

5 - nulls read as nulls,
trailing blanks write as blanks, autoskip

6 - nulls read as nulls,
trailing blanks write as nulls, no autoskip

7 - nulls read as nulls,
trailing blanks write as nulls, autoskip

2) OPERATIONS TO WRITE AND READ

BUFFERED WRITE

```
CTL=4,field numbers
DAT=data
```

Data written using this operation goes into the screen's buffer and is displayed during the next physical write. The data is a character matrix and each row is written to the row whose number is in the corresponding position in the list of field numbers. DAT may be a character vector if only one field is being written.

BUFFERED WRITE ATTRIBUTES

```
CTL=35,field numbers
DAT=attributes
```

Attributes of characters in a field are specified. The relationship between 'field numbers' and the DAT argument is the same as for 'BUFFERED WRITE', but instead of specifying the characters to display, DAT specifies the attributes of the characters.

The character attributes are encoded in the 8 bits in a character as:

SSSCCCHH

where S is for program symbol set (unsupported), C is for colour, and H is for highlight.

```
Colour values are: 0-  default
                   1 - blue
                   2 - red
                   3 - pink
                   4 - green
                   5 - turquoise
                   6 - yellow
                   7 - white

Highlight values are: 0 - default
                      1 - blinking
                      2 - reverse video
                      3 - underscore
```

The attribute character is $\square AV[\square IO+(COLOR\times 4)+HIGHLIGHT]$.

IMMEDIATE WRITE

```
CTL=2,field numbers
DAT=data
```

DAT is a character matrix as defined for the buffered write operation, except that it is displayed immediately.

IMMEDIATE WRITE ATTRIBUTES

```
CTL=37,field numbers
dat=attributes
```

As for 'BUFFERED WRITE ATTRIBUTES', except that it causes the screen to be written immediately.

READ and WAIT

CTL=3
DAT=(none)

This operation causes a physical write to the screen of any buffered data, then waits for the user to complete input. The resulting DAT contains a completion code indicating how input was caused, and, depending on the completion code, a modified and/or list of modified field numbers as defined below.

USER ACTION	*COMPLETION CODE*	*COMPLETION MODIFIER*	*CURSOR POSITION FIELD*	*CURSOR POSITION ROW*	*CURSOR POSITION COLUMN*	*FIELDS*
ENTER KEY	0	0	*FLDNUM*	*ROW*	*COL*	*FLDNUMS*
PF KEYS	1	1-24	*FLDNUM*	*ROW*	*COL*	*FLDNUMS*
LIGHT PEN	2	0	*FLDNUM*	*ROW*	*COL*	*FLDNUMS*
PA KEYS	4	1-3	-	-	-	-
CLEAR KEY	5	-	-	-	-	-
NO INPUT	6	-	-	-	-	-

From the result, an application can ascertain if and how input was caused and what fields were modified.

GET FIELD DAT

CTL=5,field numbers
DAT=(none)

The result contains a character matrix where a row contains data for the field number in the corresponding position of the field number list.

GET MODIFIED DAT

CTL=30
DAT=(none)

Data is returned in DAT for fields which were modified during the last READ AND WAIT operation.

GET MODIFIED DAT EXTENDED

```
CTL=31
DAT=(none)
```

As for GET MODIFIED DAT except that the data is returned as a vector of enclosed elements. Each element's data has been stripped of trailing blanks.

GET SCREEN

```
CTL=40
DAT=(none)
```

The result is a character matrix the same size as the screen (rows by columns) containing the data for this screen.

GET SCREEN ATTRIBUTES

```
CTL=41
DAT=(none)
```

The result is a character matrix the same size as the screen (rows by columns) containing the character attributes, encoded in the same as for the 'WRITE ATTRIBUTE' operation, for this screen.

3) MISCELLANEOUS OPERATIONS

ERASE THE SCREEN

```
CTL=20
DAT=(none)
```

The full screen display is cleared.

SET CURSOR POSITION

```
CTL=12
DAT=position
```

Position is field number, row, and column. After the next write operation, the cursor is placed at the row and column of the selected field number. When the field number is zero, the row and column are taken from the screen's upper left hand corner. The cursor position is automatically updated when a field is modified by the user.

SOUND THE ALARM

```
CTL=11
DAT=(none)
```

The audible alarm is sounded after the next physical write.

PUBLIC FULL SCREEN WORKSPACES

At present, there are three public workspaces located in public library 5 to aid a full screen application in the use of AP124. Each one uses the same basic operation codes, but differ in other ways. The three workspaces are:

```
5 IBM3270
5 AP124
5 AP124E
```

Following is a description of each workspace.

Workspace 5 *IBM*3270

This workspace contains functions with the same names as those supplied with IBM's AP124x, so that an applications programmer, used to working with IBM's functions will feel comfortable using this workspace. In addition, this workspace contains functions to control color, and highlighting, as well as extended capabilities peculiar to SHARP APL's AP124.

FUNCTION NAME AND SYNTAX	CTL	DESCRIPTION
FORMAT fmtmatrix	1	fmtmatrix is the format matrix containing four, six, or eleven columns
fld *REFORMAT* fmtmatrix	1	Format matrix fmtmatrix, is applied to field numbers fld.
R←READFORMAT	9	The result is the current format matrix
fld *FIELDTYPE* typ	6	Field numbers in fld are reset to the values in typ.
fld *INTENSITY* int	7	Field numbers in fld are reset to the values in int
fld *PSS* ps	-	Field numbers in fld will use the program symbol sets in ps (currently unsupported).
fld *COLOR* col	24	Color values of field numbers in fld are reset to col.
fld *HIGHLIGHT* hi	26	Highlight values for fields in fld are reset to hi.
fld *FIELDATTR* att	16	Attribute values in att are applied to field numbers in fld
fld *WRITE* data	4	Data in a row of character matrix data is written to the buffer of the field whose number is in the corresponding position of fld.

fld *WRITEATTR* att	35	Encoded character attributes in a row of att are written to the buffer of the field whose number is in the corresponding position of fld.
fld *IMMWR* data	2	As for *WRITE* except that the data is written to the screen immediately.
fld *IMMWRATTR* att	37	As for *WRITEATTR* except that the attributes are written to the screen immediately.
R←READSCREEN	3	Buffered data is written to the screen and the full screen manager waits for the user to complete input.
R←GETFIELDS fld	5	Data as a character matrix is returned for field numbers fld.
R←READDATA	30	Data in character matrix form is returned for fields which were modified during the last *READSCREEN* operation.
R←READDATAX	31	As for *READDATA* except that the data is returned as a vector of enclosed elements.
R←GETSCREEN	40	A character matrix representing the current characters on the screen is returned.
R←GETSCREENX	41	A character matrix representing the current character attributes on the screen is returned.
ERASESCREEN	20	The full screen display is erased.
fld *SETCURSOR* rowcol	12	The cursor is placed at the row and column of rowcol in the field number fld.
ALERT	11	The alarm is sounded.
STATE	ARB	*STATE* reports the session manager status including the number of □*ARBIN* screens, first and last session manager line numbers, and screen dimensions. This function uses the □*ARBIN* interface.

Workspace 5 *AP124*

This workspace contains all of the shared variable functions of workspace 5 *IBM3270* and none of the □*ARBIN* functions. The functions are in stripped down form and assume that variables *CTLS* and *DATS* are shared and fully coupled, otherwise an error message results. The function *SHAREAP124* shares these variables and returns a 1 when coupling is complete.

Workspace 5 *AP124E*

Functions in this workspace are identical to those of workspace 5 *AP124* except that they use the COMMAND CHAINING feature. Full screen operations are stored in a workspace resident buffer until being forced out by the application. The letter E (for Enclosed array) prefixes all function names so that they may be distinguished from those of the other workspace. Although all commands are buffered and are in enclosed array form, the buffer contents may optionally be sent after each operation. The functions *EFRCON* (turn on automatic buffer force) and *EFRCOFF* (turn off automatic buffer force) set the appropriate state which may be checked using the function *ESTATE*. The workspace buffer is cleared after its commands are sent, but it may be cleared by the application at any time using the function *ECLRBUF*. See *DESCRIBE* in this workspace for a more detailed description of its operation.

AP126 FULL SCREEN MANAGER

AP126 provides a CTL/DAT type shared variable interface to the IBM product GDDM (Graphical Data Display Manager). AP126 provides all of the facilities of AP124 plus extensive colour graphics capabilities. If an application doesn't require colour graphics, then it is better to use the simpler and more efficient AP124.

The key concepts in AP126 (e.g.,field and format matrix) are essentially the same as in AP124, and familiarity with AP124 will help in understanding AP126.

AP126 calls functions provided by GDDM. These functions are documented in the IBM manuals **Graphical Data Display Manager User's Guide** and **Presentation Graphics Feature User's Guide.**

AP126 can use any GDDM or PGF call except SPINIT, FSINIT, FSRNIT, FSTERM, FSEXIT, and CHART.

To request a GDDM call, you must specify the CTL variable with the AP126 request code that corresponds to the GDDM call mnemonic. The following table contains the AP126 request codes for each of the GDDM and PGF functions.

GDDM Control Requests

ASREAD	101	FSSHOW	105	FSALRM	109
FSFRCE	102	FSQERR	107	FSQDEV	110
FSREST	103	FSTRCE	108	ASTYPE	111
FSSAVE	104				

GDDM Symbol Set Requests

GSDSS	201	PSRSV	206	PSQSS	211
GSLSS	202	GSRSS	207	SSQF	212
PSDSS	203	PSRSS	208	SSREAD	213
PSLSS	204	GSQNSS	209	SSWRT	214
PSLSSC	205	GSQSS	210		

GDDM Page Requests

FSPCLR	301	FSPQRY	304	FSQUPG	307
FSPCRT	302	FSPSEL	305		
FSPDEL	303	FSQCPG	306		

GDDM Alphanumeric Requests

ASDFLD	401	ASFMOD	412	ASCGET	422
ASDFMT	402	ASFOUT	413	ASCHLT	423
ASDTRN	403	ASFPSS	414	ASCPUT	424
ASFCLR	404	ASFTRN	415	ASCSS	425
ASRFMT	405	ASFTYP	416	ASMODE	426
ASDFLT	406	ASRATT	417	ASQCOL	427
ASFCOL	407	ASQFLD	418	ASQHLT	428
ASFEND	408	ASQMAX	419	ASQSS	429
ASFHLT	409	ASQMOD	420	ASFCUR	430
ASFIN	410	ASCCOL	421	ASQCUR	431
ASFINT	411				

GDDM Graphics Requests

GSCLP	501	GSMIX	518	GSQCEL	535
GSFLD	502	GSMS	519	GSQCLP	536
GSPS	503	GSPAT	520	GSQCM	537
GSVIEW	504	GSARC	521	GSQCOL	538
GSWIN	505	GSAREA	522	GSQCP	539
GSCLR	506	GSCHAP	523	GSQCS	540
GSSCLS	507	GSCHAR	524	GSQCUR	541
GSSDEL	508	GSENDA	525	GSQLT	542

GDDM Graphics Requests (cont.)

GSSEG	509	GSLINE	526	GSQLW	543
GSCA	510	GSMARK	527	GSQMAX	544
GSCB	511	GSMOVE	528	GSQMIX	545
GSCD	512	GSMRKS	529	GSQMS	546
GSCM	513	GSPLNE	530	GSQPAT	547
GSCOL	514	GSVEC	531	GSQPS	548
GSCS	515	GSQCA	532	GSQVIE	549
GSLT	516	GSQCB	533	GSQWIN	550
GSLW	517	GSQCD	534		

GDDM Print Requests

FSCLS	601	FSLOG	603	GSCOPY	605
FSCOPY	602	FSOPEN	604		

PGF Control Functions

CHSTRT	703	CHRNIT	704	CHTERM	705

PGF Chart Format Definition

CHAREA	721	CHYTTL	738	CHGATT	754
CHBATT	722	CHXRNG	739	CHXDTM	755
CHCGRD	723	CHYRNG	740	CHYDTM	756
CHHMAR	724	CHXINT	741	CHDATT	757
CHVMAR	725	CHYINT	742	CHNOTE	758
CHKEY	726	CHXTIC	743	CHNOFF	759
CHKATT	727	CHYTIC	744	CHNATT	760
CHKEYP	728	CHXSCL	745	CHCOL	761
CHHATT	729	CHYSCL	746	CHLT	762
CHHEAD	730	CHLATT	747	CHMARK	763
CHXSEL	731	CHXLAB	748	CHPAT	764
CHYSEL	732	CHYLAB	749	CHGAP	765
CHXSET	733	CHXMTH	750	CHGGAP	766
CHYSET	734	CHYMTH	751	CHVCHR	767
CHAATT	735	CHXDAY	752	CHPIER	768
CHTATT	736	CHYDAY	753	CHVATT	769
CHXTTL	737				

PGF Chart Drawing

CHDRAX	790	CHHIST	794	CHPIE	796
CHPLOT	791	CHNUM	794	CHVENN	797
CHSURF	792	CHBAR	795	CHSET	798

The AP126 request code and numeric parameters are set in CTL and any character data is set in DAT. A return code and numeric results are returned in CTL and any character results are returned in DAT.

More than one GDDM call can be specified in a single specification of DAT and CTL. The data for the requests are simply catenated together and set into DAT and CTL as appropriate. The results come back catenated in the same form.

The result in CTL is at least four elements for each request. The four elements are:

1. Return code: 0 - success
   1 - AP126 error
   4,8 or 12 - GDDM error
2. Reason code, or 0 if the return code was 0.
3. The number of numeric results returned in CTL for this request.
4. The length of character data returned in DAT for this request.

AP126 Service Requests

In addition to GDDM calls, AP126 will also process "service requests". Service requests are indicated by negative numbers.

Service Request	Code
Query GDDM calls	-1
Set Error Threshold	-2
Set EBCDIC Translation	-4
Set AP Options	-6
Query AP Options	-7

Query GDDM Calls

CTL Input	-1
CTL Output	rc vector,AP 126 request codes
DAT Output	GDDM request code mnemonics

Returns the available GDDM calls and the corresponding AP 126 request codes. The request codes are catenated onto the 4-element result and the 8 character GDDM call names are returned in *DAT*.

Set Error Threshold

CTL Input	-2,threshold value
DAT Output	rc vector

This request sets a threshold value for GDDM error severities. When a GDDM error is encountered that has a severity greater than or equal to the threshold value, AP126 terminates processing the current string of requests. This threshold applies only to GDDM requests.

A valid threshold value is any non-negative integer. The default threshold value is 8, which causes processing to continue when a warning is encountered.

Set EBCDIC Translation

CTL Input	¯4,trans value
DAT Output	rc vector

Normally AP126 will translate all characters in a request from APL internal representation to EBCDIC and from EBCDIC to APL internal representation in a result. However, the character data parameter of SSREAD, SSWRT, PSDSS, GSDSS, and the table value parameters and character results for ASDTRN will not be translated by AP126. That is, the symbol set values, and translate table values are not translated.

This request enables you to set the translate value off, resulting in no translation by AP126, or to set the value on again. Trans value can be one of the following:

0 = set translation off
1 = set translation on (the default)

Set AP Options

CTL Input	¯6,count,AP options vector
DAT Output	rc vector

This request sets the three control requests described above in one request. A 1- to 3-element vector of options must be specified. A negative one (¯1) entry in any position in the vector retains the current value being used (it may be the default). Elements of the options vector are as follows:

1. threshold value
2. key
3. trans value

Query AP Options

Numeric Input	¯7,count
Numeric Output	rc vector,AP options vector

This request enables you to query the AP options, as described above. A numeric vector is returned containing **count** elements in the following order:

1. threshold
2. key
3. trans value

AP126 CTL Return and Reason Codes

Return	Reason	Description
0	0	No error.
1	7	Data variable length error.
1	11	Control variable rank error (must be a vector).
1	12	Syntax error in request.
1	13	Control variable domain error (must be numeric).
1	14	Invalid request code.
1	21	Data variable rank error (must be a vector).
1	23	Data variable domain error (must be characters).
1	24	Data variable not shared.
1	34	User not authorized to use this request.
1	41	Data variable has not been specified.
1	53	CTL or DAT variable too large.
1	60	GDDM not available.
1	61	Invalid parameter for AP control request.
1	62	Invalid count, code, or length on GDDM request.
1	63	Hardcopy translate not available.
1	64	Reserved page identification.
1	65	Invalid hardcopy destination.
1	66	Hardcopy destination unavailable.
1	67	Hardcopy destination already open.
4, 8 or 12	nnn	Return code is the GDDM severity code and nnn is the GDDM error code.

This section describes how an application can make complete use of the IBM 3270 device's capabilities through the primitives *⎕ARBIN* and *⎕ARBOUT*. The standard screen uses only a subset of the device's capabilities. A user can construct a raw data stream as outlined in the IBM manual GA27-2749 (IBM 3270 Component Description) and transmit it using either of the primitives. In this way an application program has control over screen format, highlighting, color, intensity and graphics (if the device is so equipped). The IBM data stream, exactly as described in the IBM manuals, can be used, but in SHARP APL the data stream has been simplified by introducing SHARP Order (SO) codes. In addition to attribute control, SOs allow an application program to:

- Define a logical screen
- Define logical screen fields on any logical screen
- Write to any logical screen or logical screen field
- Read from any logical screen or field

Constructing a Data Stream

This section outlines the formation of a valid IBM 3270 data stream with emphasis on the SHARP Order extensions.

1) The Header

A data stream consists of a character vector containing IBM data stream commands and data (for write and read commands only). To the front of this data stream is added 24 bytes of control information which tell the device-controlling software what area of terminal control should be addressed. This header currently includes a command and a logical screen number (the remaining bytes are reserved). When formed, the header and data stream are transmitted as a character or integer vector using *⎕ARBIN* or *⎕ARBOUT*. A typical transmission might look like:

```
R←⎕ARBIN HDR,DS
```

where *HDR* is the 24 byte character vector and *DS* is the data stream. The header determines how the data stream is to be applied. The first byte of the header, the command, must be one of:

Command	□*AV*	Meaning
FSMINQ	0	Device inquiry.

Position	Meaning
0	IDSH version
1	reserved
2	graphic escape character
3	number of rows
4	number of columns
5	unused
6	number of logical screen pages
7	extended capabilities (0 or 1)
8	PFK 13-24 mapped to 1-12
9-11	reserved
12-15	first session log line number (256 base)
16-19	last session log line number (256 base)
20-27	VTAM device ID (Z-code)

The result may be extended in the future.

These values are decoded by the function *STATE* in workspace 5 *IBM*3270.

FSMWRT	1	The data stream contains information to be written to the screen.
FSMRD	2	The data stream contains information to be written to the screen. After the write, wait for the user to cause input (ENTER key, PA1, PA2, CLEAR or any PFK) and return the input to the application program. This information includes termination type (ENTER, PFK, etc.), cursor position (row and column) and data. Data is returned in IBM EBCDIC device codes.
FSMSWCH	5	Switch logical screens. *FSMSWCH* is followed by □*AV*[**logical screen number**]. The logical screen image is copied from the current screen to that logical screen number.
FSMRDT	6	Read and Translate (same as *FSMRD* except the data is translated). Data that is read from the screen is translated into Z-codes using the default device translate table. The advantage of *FSMRDT* over *FSMRD* is that the application does not have to do the translation from device codes to Z-codes after a read.
FSMWRS	7	Write to Screen file. The full screen image currently displayed is written to a screen file and may be recalled with a read command.
FSMCLR	9	Clear the Session Manager Standard screen by setting the reference line to be the next input line, and the reference line offset to be zero.
FSMSSCR	11	Retrieves from the screen file the raw data stream from which a single logical screen image was composed. The logical screen number normally following the header element is not used and must be set to zero. Instead, the screen number is passed as four 256 base elements in the data stream following the header.

FSMPFMAP	12	Turns on and off the mapping of PFK's 13-24 to 1-12. This feature is of use with terminals whose righthand keypad PF keys are numbered 13-24. The element following the 24-element header is 1 to turn on mapping or 0 to turn off mapping. If □*ARBIN* is used, the result is the previous setting.
FSMFETF	14	As for *FSMGETS*, but applies to □*ARBIN*/□*ARBOUT* full screens.
FSMOEC	15	Sets the overstrike escape character (OEC). The element following the 24-element header is the □*AV* index of the character to be used, or zero to run off OEC. If □*ARBIN* is used, the result is the previous setting.

Following the header's command is the logical screen number. Its range is from □*AV*[0-255] for screen numbers 0-255. Element number 7 of the *FSMINQ* result is the number of logical screens available.

2) The Data Stream

Following the 24 byte header is the data stream, which is valid for write and read commands only. Refer to IBM manuals GA27-2749 (3270 Component Description) and GL27-6999 (3270 Programming) for a more detailed data stream description. The commands in the data stream perform screen writes and reads as well as assignment of cursor position and attribute characters. A writing data stream must begin with a write command followed by a write control character. After that, the data stream contains any combination of orders and data. Although a screen read can be done by transmitting a read command (*FSMRD* or *FSMRDT*) with an empty data stream, data can be transmitted simultaneously. In this way information can be displayed and a response solicited.

The data stream is determined to be all IBM codes or all SHARP Order codes by checking the value of the first byte. Below is a list of valid SHARP Order codes:

a) First Byte (the command)

Command	□*AV*	Meaning
SOWRT	13	Writes the following orders and data to the screen. This command adds to any existing screen image.
SOEWC	11	The screen is cleared and reset to its primary size before processing the rest of the data stream.
SOEWA	12	As for *SOEWC* except the alternate (larger) screen size is available for writing.

b) Second Byte

The data stream's second byte is a write control character (*WCC*). The *WCC* character controls, among other things, keyboard resetting and alarm sounding (for devices so equipped). A keyboard reset clears the input inhibited condition. The most common *WCC*'s are listed below but refer to the IBM manual for a

complete breakdown of the *WCC* byte (each one resets the modified data tag bit).

Command	⎕*AV*	Meaning
WCR	3	Reset the keyboard after the write and do not sound the alarm.
WCN	1	Do not reset the keyboard and do not sound the alarm.
WCRA	7	Reset the keyboard and sound the alarm.
WCNA	5	Do not reset the keyboard but do sound the alarm.

c) The Remainder

The remainder of the data stream consists of SHARP Order (SO) codes and data. SHARP Orders are used to position, define and format data, erase parts of the screen or insert the cursor. Below is a list of SHARP Orders with a brief description of any bytes that must follow, then a descriptive paragraph of each order.

Order	⎕*AV*	Bytes	Meaning
SOALLE	0		The following data is all in IBM data stream format (EBCDIC).
SOE	1	⎕*AV*[256 256⊤*COUNT*]	The following encoded *COUNT* bytes are in IBM data stream format (EBCDIC).
SOALLZ	2		All of the following data is in Z-codes
SOZ	3	⎕*AV*[256 256⊤*COUNT*]	The following encoded *COUNT* bytes are Z-codes.
SOSBA	4	⎕*AV*[*ROW*,*COL*]	Set the buffer address to row *ROW* and column *COL* on the screen.
SOEUA	5	⎕*AV*[*ROW*,*COL*]	Erase all unprotected areas from the current buffer address to location *ROW* and *COL*.
SOSF	6	*ATTRIBUTE*	Start a screen field at the current buffer location and assign it attribute *ATTRIBUTE*
SOSA	7	*TYPE*,*VALUE*	Set attributes for characters to be displayed.
SOSFE	8	⎕*AV*[*COUNT*],(*COUNT*×2)⍴*TYPE*,*VALUE*	Start field extended. *COUNT* is the number of type/value pairs.

SORA	9	□*AV*[*ROW*,*COL*],*CHAR*	Repeatedly write character *CHAR* from the current buffer address to address *ROW*,*COL*
SOMF	10	□*AV*[*COUNT*],(*COUNT*×2)⍴*TYPE/VALUE*	Modify field. To the attribute character starting at the current buffer address, set the attributes described in the type/value pairs that follow.
SOAID	14	*AID*,*ROW*,*COL*	Attention identification. Inbound data stream information including *AID* and cursor *ROW*, *COL*umn at input termination.
SOLF	15	*ROW*,*COL*,*DEPTH*,*WIDTH*,0,0,0,*FLAGS*,*PAD*,*PREFIX ATTRIBUTE*,*SUFFIX ATTRIBUTE*,*SOZ*,*COUNT*	Logical field description. Includes start positions, dimensions, pad character, leading and trailing attributes. *ROW*, *COL*, *DEPTH*, *WIDTH* and *COUNT*, are 256 base, 2-byte counts.
SOIC	16	□*AV*[*ROW*,*COL*]	Insert cursor at the current buffer address.

SHARP Order Codes - Detailed Description

1) All EBCDIC (SOALLE)

This order says that all following data and orders are in EBCDIC. If some of the orders or data following are not in EBCDIC, use SOE (see below) instead.

2) EBCDIC (SOE)

A 2 byte count follows. That many characters are then considered to be in EBCDIC. SOALLE and SOE are used when transmitting an order which has no SHARP Order counterpart or non Z-code data. Counts for this order are 256 base and 2 bytes long, so encode them using □*AV*[256 256⊤⍴*EBCDIC-DATA*].

3) All Z-codes (SOALLZ)

As for SOALLE, the entire data stream from this point is considered to be Z-code. Use this order when writing text to the screen which requires no conversion.

4) Z-codes (SOZ)

As for SOE, there is a 2 byte, 256 base count immediately afterward indicating how many Z-code characters follow.

5) **Set Buffer Address** (SOSBA)

This order looks for 2 bytes indicating the row and column to which the buffer address should be set. The row and column are direct indices into ⎕*AV*, so for example, *SOSBA*,⎕*AV*[10 20] sets the buffer address at row 10, column 20. Use this order in cases such as starting a field (see SOSF and SOSFE) which begin at the current buffer address.

6) **Erase Unprotected to Address** (SOEUA)

This order is followed by a row and column address. All unprotected data is erased from the current buffer address to the address ⎕*AV*[*ROW*,*COLUMN*].

7) **Start Field** (SOSF)

A field is started at the current buffer address. The field attribute character which must follow this order remains in effect until the next field attribute character is encountered. When the screen contains a single attribute character, the entire screen assumes the characteristics of that attribute. Attributes must be one of the following:

⎕*AV*	Meaning
76	alphanumeric input/output, non-display.
64	alphanumeric input/output, normal display.
200	alphanumeric input/output, high intensity.
92	numeric input/output, non-display.
80	numeric input/output, normal display.
216	numeric input/output, high intensity.
108	alphanumeric output, non-display.
96	alphanumeric output, normal display.
232	alphanumeric output, high intensity.
124	numeric output, non-display (note: any of these 3 cause cursor skip).
240	numeric output, normal display.
248	numeric output, high intensity.

Note: A field attribute character occupies physical space on the screen and appears as a blank character.

8) **Set Attribute** (SOSA)

This order is used to control extended features (on units with that option). Included in the extended features are color, extended highlighting and program symbols. Attributes set by the SOSA order remain in effect until another SOSA order is encountered, and override field attributes set in SOSFE orders. Subsequent data has those characteristics.

9) **Start Field Extended** (SOSFE)

SOSFE looks for a count byte (⎕*AV*[*COUNT*]) which indicates how many type/value pairs follow (again, refer to the IBM manual GA27-2749 for a complete breakdown of types and values). SOSFE differs from SOSF in that it is used to assign extended attributes as well as standard attributes starting at the current buffer address. SOSFE and SOSF control field attributes, while SOSA controls the attributes of single characters in the data stream.

10) **Repeat to Address** (SORA)

Writes the character following the specified buffer address from the current buffer address to the specified one.

11) **Modify Field** (SOMF)

Used to selectively change attribute characters previously set by SOSF or SOSFE at the current buffer address. The address must be that of an existing field attribute character and the data following it is identical to that of SOSFE.

12) **Erase Write Clear** (SOEWC)

The screen is cleared and reset to its primary size before processing the rest of the data stream.

13) **Erase Write Alternate** (SOEWA)

As for SOEWC except the alternate (larger) screen size is available for writing.

14) **Write** (SOWRT)

Writes the following orders and data to the screen. This command adds to any existing screen image.

15) **Attention Identification** (SOAID)

The next 3 bytes are an *AID* character and a 2 byte buffer address. From the AID character, an application knows how fullscreen input was caused. Termination codes include the enter key, clear key, a program function key or a PA key. A numeric value for the 2 byte address is achieved by *ROWCOL*←⎕*AV*⍳*ADDR*. Following the address is the data from any modified fields. SOAID is valid for an incoming data stream only (resulting from a SORD or SORDT).

AID	⎕AV
ENTER	125
PF1	241
PF2	242
PF3	243
PF4	244
PF5	245
PF6	246
PF7	247
PF8	248
PF9	249
PF10	122
PF11	123
PF12	124
PF13	193
PF14	194
PF15	195
PF16	196
PF17	197
PF18	198
PF19	199
PF20	200
PF21	201
PF22	74
PF23	75
PF24	76
PA1	108
PA2	110
PA3	107
CLEAR	109

16) **Logical Field** (SOLF)

Use the SOLF order to describe a logical screen field. A logical field has a defined area and attributes. SOLF is followed by 2-byte, 256 base ⎕*AV* indices for starting row, starting column, depth, width, 3 reserved bytes, flags (explained later), a pad character, two properly formed SOSF, SOSFE, or SOMF orders, SOALLZ or a properly formed SOZ order, then any text to be written in that field. Even when data is not present, the SOZ order must be present with a length count of ⎕*AV*[0 0]. Row, column, depth, and width are 256 base, 2-byte counts.

A suffix field is required since the prefix field attribute remains in effect until another field attribute is encountered. Flag bits are used to inhibit display of all or part of the SOLF data. This is desirable since the application may need to replace only an attribute (such as color) or just the data without affecting the rest of the field. The first 3 bits of the flag byte control writing of the prefix attribute, suffix attribute and data respectively. When a bit is on, that write is inhibited, so, to write data only, use ⎕*AV*[2⊥1 1 0 0 0 0 0 0] as the flag byte (note: the trailing 5 bits are reserved). Any non-overstruck character may be used as a pad character, although the most common are null (⎕*AV*[254]) and blank (⎕*AV*[152]).

For example, to write *HI THERE* and attributes in a field whose start position is row 1 column 5 and whose depth is 10 and length 20, the SOLF order is:

```
SOLF,⎕AV[0 1,0 5,0 10,0 20, 0 0 0 0 152],
SOSF,ATTA,SOSF,ATTZ,SOALLZ,'HI THERE'
```

In the above example, *ATTA* and *ATTZ* are the prefix and suffix attribute characters as described in the IBM 3270 component description manual. Several SOLF order streams may be transmitted at once allowing several simultaneous screen field writes.

17) **Insert Cursor** (SOIC)

The cursor is placed at the current buffer address. For example, to place the cursor at row 10, column 20, use
SOSBA,⎕*AV*[10 20],*SOIC*.

Transmitting a Complete Header and Data Stream

By constructing a data stream for the device and a controlling header, ⎕*ARBIN* and ⎕*ARBOUT* allow an application complete control over an IBM 3270 screen (see the sections on header and data stream formation above).

Transmission is one of the following:

```
R←⎕ARBIN (CS,22ρ⎕AV[0]),DS
```

where *CS* is a command and a screen number and *DS* is an IBM data stream, or

```
⎕ARBOUT (CS,22ρ⎕AV[0]),DS
```

⎕*ARBIN* differs from ⎕*ARBOUT* in that it gives a result, which is necessary in the case of a screen read. Although either ⎕*ARBIN* or ⎕*ARBOUT* can be used in most cases, ⎕*ARBOUT* is preferable in a fully debugged system as its response time is less. All data streams (incoming or outgoing) using ⎕*ARBIN* have a 24 byte header (provided by the application in the outgoing and by the system incoming). The result of ⎕*ARBIN* contains ⎕*AV*[0] at position 4 if the operation was successful and an error code otherwise.

Workspace 5 *UTIL*3270 contains functions to format and transmit data streams and to decipher error codes.

Error	Meaning
0	All OK
1	Send access method failure
2	Receive access method failure
3	Output exceeds buffer size
4	Command illegal
5	Header too short
6	Reserved fields not 0
7	Read data exceeded buffer size
9	Invalid logical screen number in header
11	Invalid length
12	Invalid count in SHARP data stream
13	Translation of SHARP data stream too large for buffer
14	Undefined SHARP Order code in data stream
15	SHARP Order, reserved fields not 0
16	SHARP Order, logical field order invalid
17	Invalid logical screen number as argument
18	Attempt to retrieve stored logical screen failed.

30 NOV 83

TITLE: Workspace 1 *HCPRINT*

AUTHOR: John D. Burger

Functions in this workspace are used to submit print requests to an IBM 3287 type printer, or to inquire about previously submitted requests. The source of data for these requests may be a 3270 screen or an APL file, and requests may be submitted interactively or non-interactively using different functions. When the non-interactive mode is used, a request number is returned as a result. If the interactive mode is used, a message consisting of a unique request number and timestamp information is returned. A request may be in one of several states. Immediately upon submission its state is QUEUED, during processing its state is ACTIVE, and after normal completion, its state is COMPLETED. Requests which cannot be processed terminate abnormally with a state of FAILED, and a request that has been restarted by the system has a state of REQUEUED which then becomes ACTIVE as processing continues. It is possible to withdraw a request before it completes, in which case its state is WITHDRAWN.

FUNCTIONS TO SUBMIT REQUESTS

The two function names are *HCSUBMIT* and *HCREQ*. *HCSUBMIT*, the non-interactive version, takes processing specifications as its right argument, and depending on the request type, additional information as its left argument. *HCREQ* prompts for all information. Responses to *HCREQ* need not be enclosed in parentheses.

In this document, user responses are indicated by ▒▒▒▒. The symbol ▒cr▒ means a user-supplied carriage return.

Examples:

```
      328 496 HCSUBMIT 'STD,PRINT(DEST=PRT1),TO(JOHN BURGER)'
6115


      HCREQ
1. PROCESSING SPECS                - STD,PRINT(DEST=PRT1)
8. STD,PRINT(DEST=PRT1) - OK? YES
9. STD SCREEN LINE RANGE(S)  □:
                                     300 310
HCREQ NO. 6119 FILED 21.55.31  MON 19 DEC 1983


      HCREQ
1. PROCESSING SPECS                - cr
2. TO                              - LHG
3. STD, FULL, FULL2, FILE OR HSP - FILE
5. FILE NAME(S)                    - DATAOUT
   ERASE OR NOERASE                - ERASE
6. PRINTER ID                      - PRT1
7. COPIES                          - cr
8. PRINT(DEST=PRT1),ERASE,FILE(DATAOUT),TO(LHG) - OK? YES
HCREQ NO. 6120 FILED 21.56.33  MON 19 DEC 1983
```

If the source is a 3270 screen, the print may be from the standard screen or a full screen. The standard screen print may be from either the SAPV or IDSH session managers and the full screen print may be from a screen created using AP124 or □*ARBIN*. The user specifies the print type as part of the processing specifications.

Type - Additional Information To Be Specified

STD
A numeric scalar, vector, or matrix indicating a line number range (IDSH), or a scalar or vector of page numbers (SAPV). *HCREQ* prompts for this additional information whereas *HCSUBMIT* expects it in the left argument. When the information is not supplied, the current page is printed. In the case of IDSH, this means one page calculated by LINE-ROLL. Also, for IDSH, the line ranges to be printed are supplied in pairs. If the second number in any pair is 0, it is taken to mean one page of screen depth, from the reference line. A scalar or vector with an odd number of elements will have 0 appended to it, then will be processed according to the above rules.

Other Options: *TO*, *COPY*, *EJECT*, *PRINT*, *PRIORITY*.

FULL
The names of shared control variables (optionally just the *CTL* suffixes) from which the full screen image is to be extracted. *HCREQ* prompts for this information, *HCSUBMIT* expects it as the left argument. Default is *S*.

Other Options: *TO*, *COPY*, *EJECT*, *PRINT*, *PRIORITY*.

FULL2
A numeric scalar or vector of □*ARBIN* full screen numbers. *HCREQ* prompts for this information, *HCSUBMIT* expects it as the left argument.

Other Options: *TO*, *COPY*, *EJECT*, *PRINT*, *PRIORITY*.

FILE(file names)
File names or file tie numbers. *HCREQ* prompts for this information, *HCSUBMIT* expects file tie numbers as the left argument, or file names as a parameter in the form *FILE*(1854339 *FPRINT*1,314158 *FPRINT*2). To submit a file print, your account number must be able to read the files access matrices, and be able to set them if account number 1002 or account number 0 do not have sufficient access to process the request.

Other Options: *TO*, *COPY*, *ERASE*, *EJECT*, *PRINT*, *PRIORITY*, *HSP*.

OTHER OPTION NAMES - Meaning

TO(name)
Delivery instructions only. This option does not affect the processing of a print. Default is *TO*().

COPY(number)
Number of copies, in the range 1-99. Default is *COPY*(1).

PRINT(*DEST*=destid)
The name of the printer to which output is to be directed (see the description of *HCPRINTERS* below). A site may define a ring

of printers whose name may be used here, or a site may define a printer to be the nearest to a terminal, in which case *NEAREST* may be used as a printer name here.

ERASE/NOERASE
After printing the data from the files that have been specified, they can be erased. Use of this option requires that the account number be able to set the file access matrix if user numbers 1002 or 0 do not have sufficient access. Default is *NOERASE*.

EJECT/NOEJECT
After printing each page or file component, the printer performs a forms feed. Default is *EJECT*.

PRIORITY(priority)
The request is processed with this priority. Range 0-15. Default is *PRIORITY*(0).

HSP
While processing a file request, observe and process □*OUT* file control messages.

During any session, your option settings are kept by the *HCPRINT* processor, so that with the exception of the print type, you need not respecify these settings. You may choose to reset these options during any print submission, in which case the new settings are kept. These settings are kept as long as your current session lasts and are discarded at sign-off time. The minimum information required during your initial print request is the print type (must always be specified), and the destination id (printer name).

FUNCTIONS TO INQUIRE ABOUT A REQUEST

REQ←HCQUEUED
Returns the numbers of your requests which have not yet been processed.

REQ←HCREQUEUED
Returns the numbers of your requests which had started to be processed but were restarted by the system.

REQ←HCACTIVE
Returns the numbers of your requests which are now being processed. This usually means that they are currently being printed, but it is possible for a requests printing to be temporarily or permanently inhibited.

REQ←HCCOMPLETED
Returns numbers of your requests which completed normally, and without error.

REQ←HCFAILED
Returns the numbers of your requests that were processed, but terminated abnormally (see *HCDISPLAY* and *HCINQ* below).

REQ←HCLAPSED
Returns the numbers of your requests which have been processed and completed successfully or otherwise. The result includes the same information returned from *HCCOMPLETED*, *HCFAILED*, and *HCWITHDRAWN* (described below).

REQ←HCWITHDRAWN
Returns the numbers of your requests which were withdrawn from the system by you or by a system administrator, using the function *HCWITHDRAW* (see below).

REQ←HCREQUESTS
Returns the numbers of all of your requests that are still on file, regardless of their status.

INQ←HCINQ REQ
Returns a matrix of information for each of your specified request numbers. This information consists of the request number, status, error code, 0, and the requesting user number. Possible status settings are:

0 - requeued
1 - queued
2 - active
3 - completed
4 - withdrawn
5 - failed

The error code column is normally 0, unless a requests status is 5 (failed). Error codes have the following meaning:

1 *LOGICAL UNIT NOT FOUND*
The printer whose name you supplied is valid according to *HCPRINT*, but no such printer exists on the system.

2 *LOGICAL UNIT NOT AVAILABLE*
The printer exists, but is not currently available for some reason.

3 *UNIT NOT A HARD COPY DEVICE*
The printer name that you supplied was not a 3287 type printer.

7 *UNABLE TO TIE FILE*
You submitted a request involving a file that could not be accessed either because it didnt exist at print time, or *HCPRINT* did not have access to it.

9 *UNABLE TO ERASE FILE*
You used the *ERASE* option, but *HCPRINT* was unable to erase the file that it was processing. The access matrix may have changed, or the file may have been erased by someone else.

10 *TRANSMISSION DATA CHECK*
Any error, not handled specifically by *HCPRINT*, is mapped to this error code.

499 *PROCESSOR ERROR*
While processing your request, the *HCPRINT* system had an APL error. Report this to the system administrator.

OTHER FUNCTIONS

YN←HCWITHDRAW REQ
HCPRINT attempts to withdraw the requests whose numbers are specified, and returns a Boolean 1 if withdrawn, or a 0 if the withdraw was not performed.

REQ←REQ HCRESUBMIT SPECS
Resubmit an existing request, if possible, using new processing specifications supplied in the right argument (an empty argument uses current specifications). The left argument is the number of the inactive request to be resubmitted.

PRINTERS←HCPRINTERS
Returns information about printers to which you may direct output. The result consists of printer name, VTAM id, 0 or user numbers to which use of this printer is restricted, and a description of the printer. Either the printer name or the VTAM id may be used with the *PRINT* option described above.

HCDISPLAY REQ
Produces a report including request number, submitting account number and name, timestamp, status, and processing specifications.

SAMPLE REQUESTS

- Print lines 328 to 496 at the printer named *PRT*1:

```
      328 496 HCSUBMIT 'STD,PRINT(DEST=PRT1),TO(JOHN BURGER)'
6115
```

- Print one screen depth starting at line 450. Unspecified parameters are taken from the previous request.

```
      450 0 HCSUBMIT 'STD'
6116
```

In this example, the 0 is optional.

• The following are displays of the previous two requests.

```
      HCDISPLAY 6115
HCREQ NO.          - 6115
REQUESTOR          - 1491625 JDBURGER
DATE SUBMITTED     - 21.52.25  MON 19 DEC 1983
STATUS             - ACTIVE
PROCESSING SPECS   - STD,COPY(1),TO(JOHN BURGER),PRINT(DEST=PRT1),
                     NOEJECT


      HCDISPLAY 6116
HCREQ NO.          - 6116
REQUESTOR          - 1491625 JDBURGER
DATE SUBMITTED     - 21.52.59  MON 19 DEC 1983
STATUS             - QUEUED
PROCESSING SPECS   - STD,TO(JOHN BURGER),PRINT(DEST=PRT1),COPY(1),
                     NOEJECT
```

• Print one page starting at line 12 and print lines 765 to 840. Print each range as a new page and make two copies.

```
      12 0 765 840 HCSUBMIT 'STD,PRINT(DEST=PRT1),
                   TO(LESLIE GOLDSMITH),COPY(2),EJECT'
6118
```

HCREQ is interactive. All processing specs can be entered after the first prompt *PROCESSING SPECS* alternatively, responding with CR results in a prompt for each parameter:

```
      HCREQ
1. PROCESSING SPECS                - STD,PRINT(DEST=PRT1)
8. STD,PRINT(DEST=PRT1) - OK? YES
9. STD SCREEN LINE RANGE(S)  □:
                                     300 310
HCREQ NO. 6119 FILED 21.55.31  MON 19 DEC 1983


      HCREQ
1. PROCESSING SPECS                - cr
2. TO                              - LHG
3. STD, FULL, FULL2, FILE OR HSP   - FILE
5. FILE NAME(S)                    - DATAOUT
   ERASE OR NOERASE                - ERASE
6. PRINTER ID                      - PRT1
7. COPIES                          - cr
8. PRINT(DEST=PRT1),ERASE,FILE(DATAOUT),TO(LHG) - OK? YES
HCREQ NO. 6120 FILED 21.56.33  MON 19 DEC 1983


      HCDISPLAY 6120
HCREQ NO.          - 6120
REQUESTOR          - 1234567 LHGOLDSMITH
DATE SUBMITTED     - 21.56.33  MON 19 DEC 1983
STATUS             - QUEUED
PROCESSING SPECS   - FILE(1234567 DATAOUT),TO(LHG),
                     PRINT(DEST=PRT1),COPY(1),ERASE,NOEJECT
```

∘ Print the AP124 screen named *CTLS*:

```
'CTLS' HCSUBMIT 'FULL,PRINT(DEST=PRT1),TO(LHC)'
6121


    HCREQ
1. PROCESSING SPECS                  - cr
2. TO                                - cr
3. STD, FULL, FULL2, FILE OR HSP     - FULL
   EJECT OR NOEJECT                  - cr
6. PRINT ID                          - PRT1
7. COPIES                            - cr
8. PRINT(DEST=PRT1),FULL - OK? YES
9. CTL/DAT VARIABLE NAMES            - CTLS
HCREQ NO. 6122 FILED 21.58.06  MON 19 DEC 1983
```