WRITING BEHAVIORAL EXPERIMENTS IN CRMS APL:

PROGRAMMERS' MANUAL
(PRELIMINARY VERSION)

Paul Gee
Wiley Greiner
Sheldon Linker
David Redell

July 15, 1974

Systems Group
Technical Document
Center for Research in Management Science
University of California
Berkeley

This work was done as part of the Systems Development effort under National Science Foundation Grant GS-32138.

1. Introduction

APL\360 and most subsequent implementations of the APL language have been oriented around a view of man-machine interaction in which each person at his terminal is served by a single sequentially executing program, over which he exercises complete control. While this view certainly reflects accurately a majority of applications of interactive computing, it fails rather badly to cope with several aspects of on-line control of multi-subject behavioral experiments. In particular, it assumes:

(a) That there is no direct interaction between the programs serving different people.

(b) That there is a one-to-one correspondence between programs, people, and terminals.

(c) That all program activity is initiated by a person at a terminal, who may interrupt, modify, and otherwise interfere with the program at will.

At least one new variant of APL\360 ("APL SV") attempts to eliminate assumption (a) above, but it has left assumptions (b) and (c) in force. The system described below is a much wider departure from APL\360, which attempts to provide a more suitable environment for experiment control by eliminating all three of the unsuitable assumptions mentioned above.

2. Processes

Following standard operating-system terminology, we refer to "a program in execution" as a process. It is intended that an experiment be written as a collection of several cooperating processes, started at appropriate (perhaps different) places in the same APL program. Logically, these processes are best thought of as separate, disjoint copies of the program. (The fact that certain parts of the program are shared in order to conserve memory is invisible; in particular, there is no logical sharing of any variables, local or global, between separate processes.)

It would be possible, of course, to avoid the idea of processes by forcing a single experiment program to explicitly deal with multiple subjects at terminals, experimenter intervention, data logging, and so on. Experience with other systems has shown, however, that such intrinsically parallel tasks can be handled more conveniently by several "copies" of a program, running in parallel (i.e., several processes).

In an experimental situation, the people interacting with terminals (or other interface equipment) are not "users" in the normal sense and must be given neither privileges nor responsibilities normally associated with "users". In particular, they must be neither responsible for the initiation of program execution (process creation) nor capable of termination of program execution (process destruction). Moreover, some processes within an experiment may not be associated with any terminal at all.

For the above reasons, the system allows processes to be created and destroyed by other processes. Except for the initial process (which

is created by a command from the controlling terminal--see USERS' GUIDE) all processes in an experiment are created in this way.

3. The Two Levels of Library Functions

The library functions available to users for constructing multi-process experiments are divided into two groups:

(a) "High-level" functions which allow easy construction of standard experiment configurations.

(b) "Low-level" functions which manipulate the basic building blocks of experiments in more flexible but rather more complicated ways.

The low-level functions are used in a stylized fashion by the high-level functions to construct the most frequently used types of configurations.

Section 4 describes the high-level functions in sufficient detail for the construction of standard configurations. Section 5 describes the basic building blocks and the low-level functions for manipulating them. This information allows the programmer to completely ignore the high-level functions and construct any unusual configuration desired. Between these two extremes lies the approach of constructing a standard configuration using the high-level functions, and then slightly rearranging things using a few calls on low-level functions. In general, this is much less work than starting "from scratch", even though it requires some knowledge of how the high-level functions use the basic building blocks. This information is contained in Section 6.

Sections 1 through 4 and the appendices are sufficient for writing most experiments; hence Sections 5 and 6 may be skipped on a first reading.

4. High-Level Functions for Standard Configurations

The standard experiment configuration is shown in Figure 1. Here, the initial process is the experimenter's process which creates and supervises as many subject processes as desired. The bi-directional communication channels, shown as double-ended arrows, allow the processes to communicate with each other and with their terminals. A terminal connected to a subject process is called a subject terminal. The terminal connected to the experimenter's process is called the experimenter's terminal, and is not, in general, the same as the controlling terminal from which the experiment was started. The data log is a file in which the experimenter's process may log any desired data as the experiment progresses.

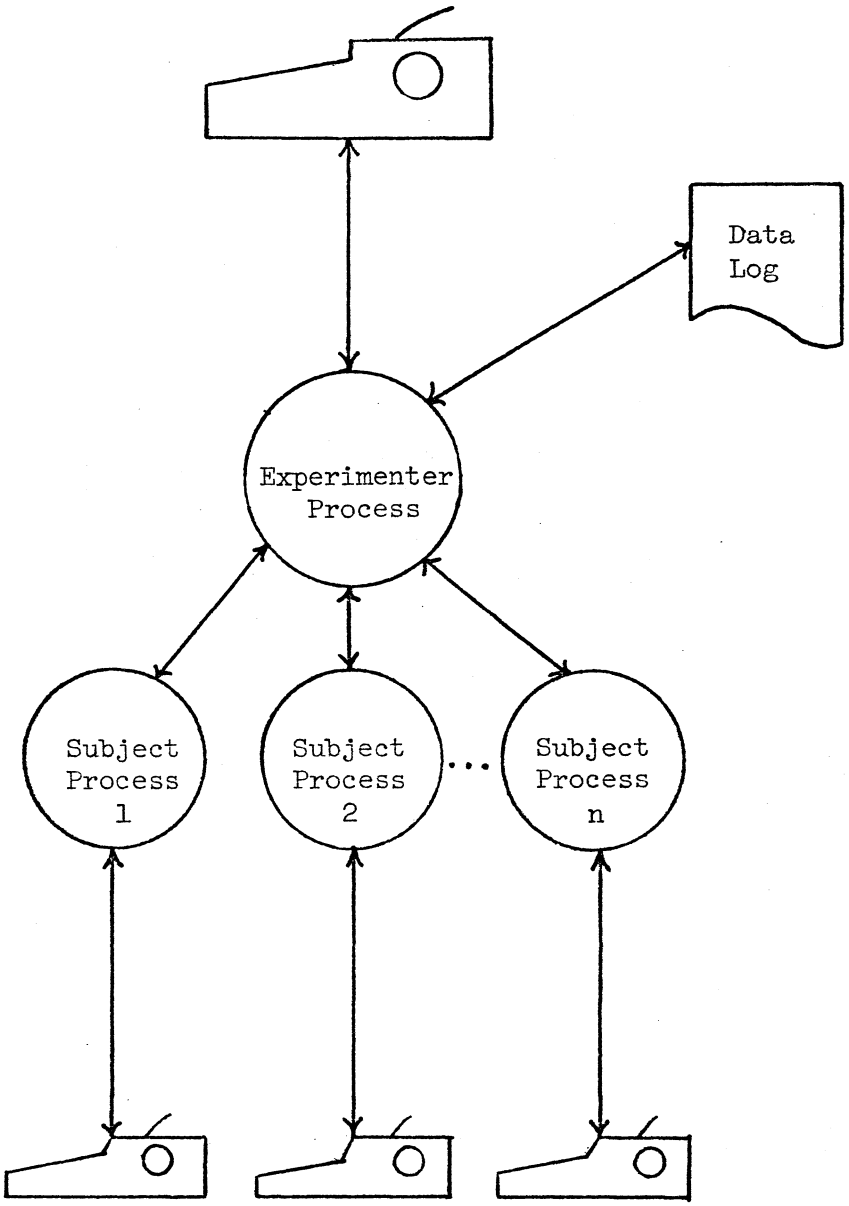
The experimenter's process creates subjects by calling the "create subjects" function[†]:

∇ ⎕*CR_SUBJECTS*[o*A*;o*T*] *FD*

This creates one or more subject processes attached to the terminal(s) specified by *T* . The number of subjects created is ρ*T* . If *T* is omitted, the experimenter is asked to enter the value of *T* from his terminal. The subject processes start execution in the function whose "function-descriptor" is *FD* . (The function-descriptor of *F* can be obtained using monadic $--i.e., $*F* .) This function must be monadic, and receives as its argument the value of *A* . (Thus, the new process starts running as if it had executed the statement *F A* .) If *A* is omitted ⍳0 is passed.

There is no rule against calling ⎕*CR_SUBJECTS* more than once, hence the subject processes may be created individually, in several batches, or all at once. At all times, the matrix ⎕*SUBJECTS* contains information on all existing subject processes. In particular, the correspondence

[†]Note CRMS APL argument notation: o means "optional", ρ means "by reference".



Standard Configuration

Figure 1

between subject processes and subject terminals is kept track of by making the value of ⎕*SUBJECTS* [*I*;2] be the terminal number of the terminal attached to the Ith subject process. (There is other information in ⎕*SUBJECTS* , but this can be ignored for standard configurations. It is discussed in Section 6.)

The experimenter's process creates the data log using the "create log" function:

∇ ⎕*CR_LOG*[⍬*NAME*]

NAME is the filename of the logging file. If *NAME* is omitted, its value is requested from the experimenter's terminal. If the named file already exists, it's old contents will be overwritten.

Once the experiment is set up and running, the various processes must be able to communicate data over the appropriate channels. The interface to the terminal channels is simply the familiar input/output convention of APL\360 with certain local modifications. (For example, quad-input is weakened considerably to prevent a subject at a terminal from interfacing with the subject process controlling that terminal, as discussed in Section 1.) Input/output is discussed in more detail in Sections 5.5 and 5.6.

The channels between the experimenter and the subjects are used by calling four special functions. Two of these are for the subject processes:

∇ ⎕*GIVE M*

∇ *M* ← ⎕*GET*

Calling ⎕*GIVE* passes message *M* to the experimenter's process. Calling ⎕*GET* returns the next message sent to this subject process by the experimenter's process. The other two functions are for the experimenter's

process:

```
∇ M □GIVETO I

∇ M ← □GETFROM ρI
```

Calling *□GIVETO* sends message *M* to the *I*th subject process. Calling *□GETFROM* returns the next message sent by the *I*th subject process. If *I* is a vector, the message may come from any of the specified subjects. When *□GETFROM* returns, *I* is set to the number of the subject process which is sent *M* .

The channel to the data log is accessed by the "log" function:

```
∇ □LOG X
```

Calling *□LOG* writes the value of *X* on the data log.

There are two useful variations on the standard configuration which are also supported by the high-level functions. The first is the simple one-terminal experiment, as shown in Figure 2. The programmer here writes the program for the subject process, but need not write a program for the experimenter's process. Instead, a standard experimenter program named *□SINGLE* may be started, which will start the subject program in its first function and turn over the experimenter's terminal to it. If several such one-terminal experiments are desired, as in Figure 3, starting the experimenter program *□MULTIPLE* will request that the terminal numbers be entered at the experimenter's terminal, and then will start a subject process for each terminal specified. Of course, the subject programs must refrain from using the *□GIVE* and *□GET* functions, since there is no real experimenter process for them to talk to.

The second variation is an elaboration on the standard configuration which allows the substitution of "robot" subjects for any number of the

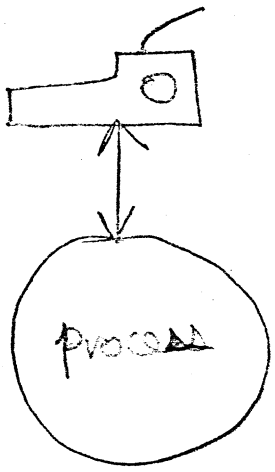


Figure 2 : Single-terminal experiment

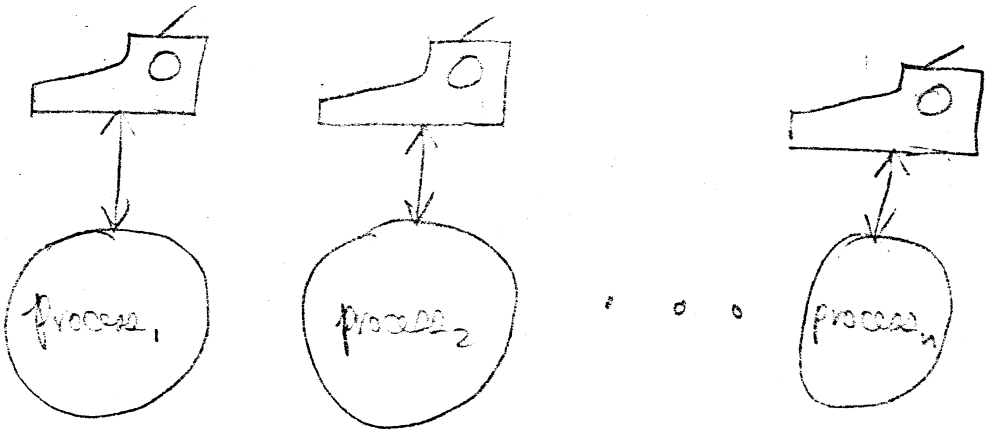


Figure 3 : parallel single terminal experiments

terminals interfacing with real human subjects. This is done in two stages. First, the robots are created by the experimenter's process using the "create robots" function:

∇ □*CR_ROBOTS*[o*A*;o*N*] *FD*

This creates *N* robot processes. If *N* is omitted, the value is requested at the experimenter's terminal. The use of *FD* and *A* is exactly as in □*CR_SUBJECTS* . Subsequently, when the subjects are created, any terminal number which is negative (i.e., -J) will be interpreted as a request to connect the corresponding subject process to the Jth robot instead of a real subject terminal. Subsequently, any output from the subject process will arrive as input for the robot, and vice-versa. Thus, a configuration such as shown in Figure 4 can be constructed with only minimal modification of the experimenter's process, and no change whatsoever to the subject processes.

Usually, it will not be necessary for the robot processes to communicate directly with the experimenter process. If this does become necessary, however, it can be done using the □*GIVE* and □*GET* functions in the robot programs, and the □*GIVETO* and □*GETFROM* functions in the experimenter program. The □*GIVETO* and □*GETFROM* functions interpret a negative argument (-J) as indicating the Jth robot process.

At any time, a process may wait for some period of time using the "wait" function:

∇ □*WAIT M*

Calling this function delays the process for *M* milliseconds. One may also wish to request input from a subject terminal with the understanding that if no input is received within a specified time limit, the request should time-out. To this end, each of the input functions has an optional

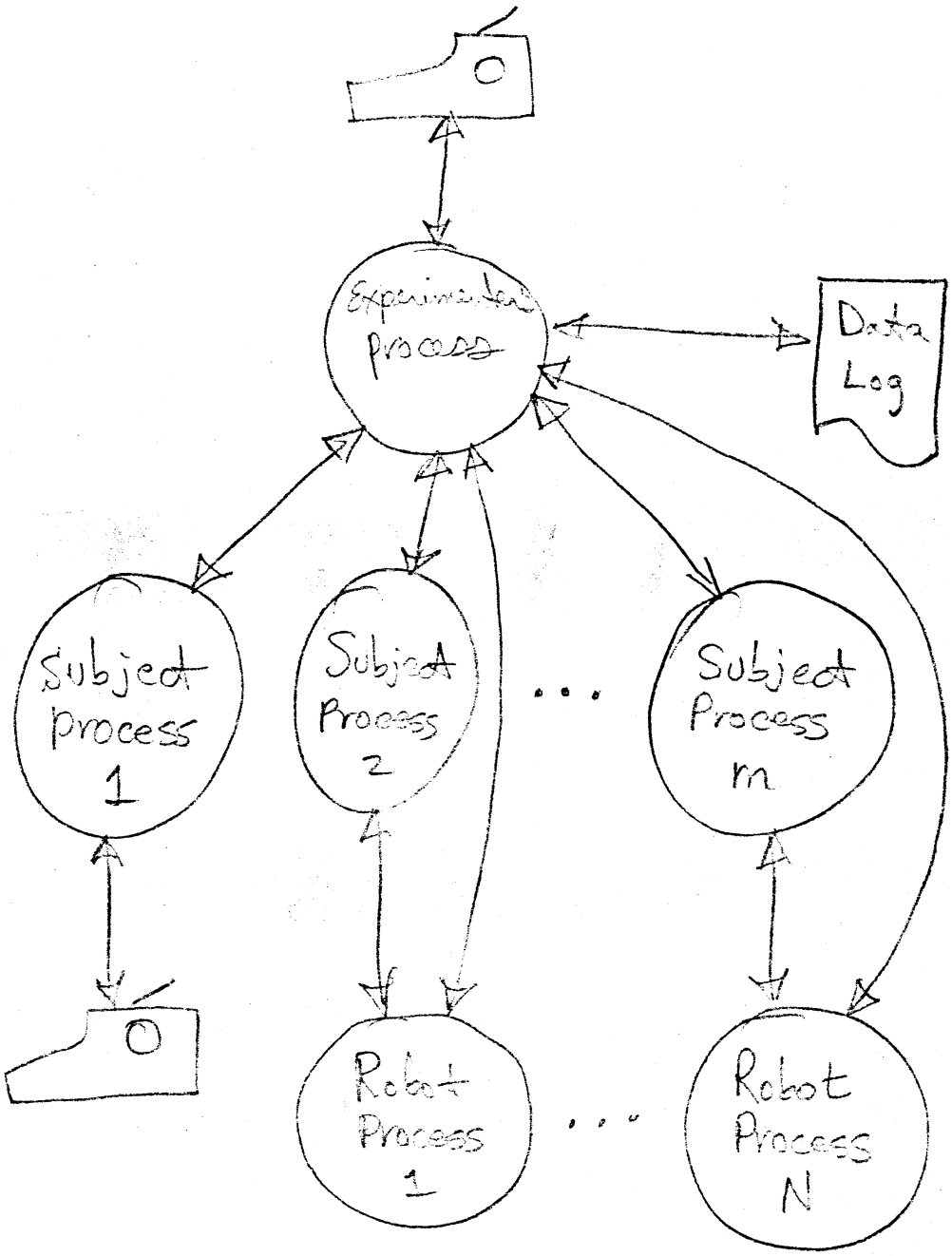


Figure 4: Experiment with Robots

argument which is a time limit in milliseconds:

```
∇ R← ⎕INNUM[⍥TL]
∇ R← ⍞[⍥TL]
∇ R← ⎕INMIX[⍥TL]
```

If the time limit expires, ⍳0 is returned. (See Appendix for an explanation of input functions.)

5. Low-Level Functions and Basic Building Blocks

The standard configurations described in section 4 are implemented utilizing a set of primitive objects which can be connected together in a variety of ways. The primitive objects are:

(a) Processes

(b) Mailboxes

(c) Terminals

(d) Files

(e) Alarm clocks

Each of these objects and the functions for manipulating them are discussed below.

5.1 Processes

The notion of a process has already been discussed in Section 2. Processes are created by calling the "create process" function:

```
∇ ID ← A □CR_PROC[○MAX;○STATE] FD
```

The new process starts executing in the function whose function-descriptor is *FD* , which must be monadic and receives as its argument the value of *A* . *MAX* is the maximum number of mailboxes the new process may specify in any single call on □*RECEIVE* (see below about □*RECEIVE*). If *MAX* is omitted, the default is 5. *STATE* contains the initial values of certain global state variables in the new process whose significance will be explained later.

```
□WIDTH ←      STATE[1]     (default = 132)
□DIGITS ←     STATE[2]     (default = 6)
□PROC ←       STATE[3]     (no default)
□IO_BOXES ←   STATE[4 5]   (no default)
□IPC_BOXES ←  STATE[6 7]   (no default)
□ALARM ←      STATE[8]     (no default)
□RAND ←       STATE[9]     (default = 93117)
```

The value of *ID* returned by □*CR_PROC* is the "process-ID" of the new process.

Processes are destroyed using the "destroy process" function:

```
∇ □DE_PROC ID
```

The process identified by *ID* is destroyed.

5.2. Mailboxes

For a collection of processes to implement an experiment, at least three kinds of communications must be possible:

(a) Communication among processes.

(b) Communication between (at least some of) the processes and the terminals which interface with the subjects.

(c) Communication between some process(es) and at least one file for logging of experimental data as it is collected.

All three of these functions are accomplished through objects called mailboxes. A mailbox is simply a first-in-first-out queue of APL data items (i.e., scalars, vectors, matrices--anything which can be stored in an APL variable). The following description treats case (a) in which items are being communicated among processes. Later sections describe the use of mailboxes in cases (b) and (c), involving terminals and files.

Mailboxes are created using the "create box" function:

```
∇ ID ← □CR_BOX[⍜N] MAX
```

If *N* is absent, this function creates a new mailbox and returns the integer mailbox identifier *ID* to be used later in referring to the new mailbox. As many as *MAX* processes may request messages from the mailbox at any given time. If *N* is given, then *N* mailboxes are created and *ID* is a vector of *N* mailbox identifiers. Mailboxes are destroyed using the "destroy box" function:

```
∇ □DE_BOX ID
```

This function destroys the mailbox identified by *ID* .

5.3. Sending Messages

A message may be sent to a mailbox using the "send" function:

```
∇ BOX □SEND M
```

If a receiver is already waiting for the message, it is transferred immediately. Otherwise, the message waits in the mailbox until it is requested. In either case, the sender continues execution immediately. If *BOX* is a multi-element vector, a copy of message *M* is sent to <u>all</u> mailboxes specified.

Using the "deliver" function:

```
∇ BOX □DELIVER M
```

is identical, except that the delivering process is delayed until reception of the message(s) by the receiver(s) has occurred.

5.4. Receiving Messages

A process can obtain a message from a mailbox using the "receive" function:

```
∇ M ← □RECEIVE ⍴BOX
```

When *□RECEIVE* is called, *BOX* identifies the mailbox from which the message *M* is to be received. If *BOX* is a vector of different mailbox identifiers, the message may come from any of the specified boxes. When

□*RECEIVE* returns, *BOX* is set to the identifier of the mailbox from which the message was received. (Note that the length of *BOX* may not exceed the value *MAX* specified when the process was created.)

5.5. Terminal Input/Output

A terminal can be attached to a pair of new mailboxes using the "boxes for terminal" function:

∇ *BOXPAIR* ← □*BOXES_TERM TERM*

which returns a 2-element vector (*BOXPAIR*) specifying the two new boxes: an input mailbox and an output mailbox, in that order. *TERM* is the terminal number.

The exact format of terminal input/output is discussed in the next section. Generally speaking, on output, characters are printed literally, and numeric values are always converted to their character-string equivalents. The same is normally true on input, but the conversion of numeric substrings in the input line may be controlled using the "set convert" function:

∇ *INBOX* □*SETCONV CFLAG*

This function turns conversion off (on) if *CFLAG* = 0 (1) .

A terminal may be detached from a pair of mailboxes using the "detach" function:

∇ □*DETACH BOXPAIR*

5.6. Terminal Input/Output Formatting

In an experimental situation, a high degree of control over the level of interpretation of input is desirable. Therefore, besides the usual Quad and Quotequad input forms, a third form of input interpretation is provided. Generally, it interprets the input as a vector with characters distributed

to elements in two ways. Each longest initial substring of characters that can be interpreted as a number forms one numerical element, and all other characters individually form character elements.

Examples:

3¯2 is interpreted as the two-element vector 3, ¯2

3-2 is interpreted as the three-element vector 3, -, 2

1.2 A-24.6E¯1EZ is interpreted as the seven-element vector

1.2, blank, blank, A, -2.46, E, Z

All numeric output conversions are done using "free format". Conversion specifications need not be given by the programmer; they are chosen for him according to the output values each time. If the programmer wants his output to be of a specific form, he may use the "specified-format" conversion discussed in Section 5.9.

Output is converted as follows:

(a) SCALAR - A scalar character is printed literally; a scalar number is printed after being converted into its character string equivalent in 'integer', 'floating-point', or 'exponential' format. The format is automatically chosen depending upon the number's magnitude and whether or not it is an integer.

(b) VECTOR - Each element of a vector is converted as if it were a scalar, and the entire vector is printed with the insertion of two blank characters between the adjacent numerical elements.

(c) MATRIX - All characters in the matrix are printed literally; for the numerical elements, a single minimum width format is first chosen such that all the numbers in the matrix can be represented with at least one preceding blank character. Then

all numbers of the matrix are converted using that format. Each row of the matrix is started on a new line. Thus, if each column of the matrix is of the same type (either character or number), the columns of the output will be aligned.

(d) RANK-N ARRAY - Rank-n arrays are displayed as sets of matrices using additional linefeeds to indicate the shape of the higher ranks.

(e) EMPTY ARRAY - The empty array is signified by a carriage-return followed by a line feed.

5.7. File Input/Output

A process may create and destroy files using the monadic functions:

```
∇ □CR_FILE NAME
∇ □DE_FILE NAME
```

where *NAME* is a character-vector containing the symbolic name of the file.

A file is attached to a new pair of mailboxes by using the "boxes for file" function:

```
∇ BOXPAIR ← □BOXES_FILE NAME
```

where a 2-element vector (*BOXPAIR*) is returned (as in *BOXES_TERM*). Sending messages to the output mailbox causes data to be written on the end of the file. Receiving messages from the input mailbox causes data to be read sequentially (starting from the beginning of the file when it is first attached). At any time, input may be restarted from the beginning of the file using the "rewind file" function:

```
∇ □RW_FILE BOXPAIR
```

As in the case of terminals, a file may be detached from a pair of mail-

boxes using the "detach" function:

```
∇ □DETACH BOXPAIR
```

The existence of a file named *NAME* may be determined using the "exists file" function:

```
∇R ← □EXISTS_FILE NAME
```

where 1 is returned if the file exists, and 0 otherwise.

5.8. Alarm Clocks

An alarm clock can be attached to a mailbox, which will cause a single message to appear in the mailbox when the alarm "rings". This is useful in two situations:

(a) A process may wish to simply "go to sleep" for some predetermined length of time (as in the *□WAIT* function of Section 4).

(b) A process may wish to receive a message while insuring that it will not wait forever if the message does not arrive (as in the time-limit option on the terminal input functions, also described in Section 4).

In both cases, the process should call *□RECEIVE*, specifying the alarm mailbox and, in case (b), the other mailbox(es) on which the time-limited message is expected.

Each process is automatically given a private mailbox preattached to an alarm clock. The mailbox-identifier is stored in the global variable *□ALARM* in the process. New alarm mailboxes may be created using the "box for alarm" function:

```
∇ B ← □BOX_ALARM
```

Note that any attempt to send or deliver messages to an alarm mailbox is illegal.

An alarm clock is set using the "set alarm clock" function:

∇ □*SET_ALARM*[o*ALARM*] *TIME*

This causes the alarm clock to "ring" in *TIME* milliseconds. If *ALARM* is omitted, □*ALARM* is used.

An alarm clock may be reset at any time by calling the "reset alarm clock" function:

∇ □*REMAIN* ← □*RESET_ALARM*[o*ALARM*]

If *ALARM* is ommitted, □*ALARM* is used. This empties the alarm mailbox and if the alarm was previously set but has not yet "rung", the ring is cancelled. The time remaining until the cancelled ring is returned to *REMAIN* . (If no ring was pending, zero is returned.) Using these functions, a process can wait for some message *M* via mailbox *B* by executing:

```
□SET_ALARM LIMIT
M ← □RECEIVE BOX ← B,□ALARM
R ← □RESET_ALARM
WAIT ← LIMIT - R
```

In this example, if *M* never arrives, the process will wakeup after *LIMIT* milliseconds with *WAIT* set equal to *LIMIT* , and *BOX* equal to □*ALARM* . On the other hand, if *M* arrives before *LIMIT* milliseconds have elapsed, *WAIT* will contain the length of time the process waited for *M* , and *BOX* will be equal to *B* .

5.9. Specified-Format Conversion

Any data object in APL may be converted to its external form by the function:

∇ *RESULT* ← □*CONVERT* [o*FORMAT*] *OBJ*

where *OBJ* is the data object, *FORMAT* is a matrix of pairs of numbers,

and *RESULT* is the character matrix of *OBJ* in external form. Each pair of numbers in *FORMAT* has the meaning total field width and fraction field width, respectively. Each pair of numbers is used to control the conversion of a column of the array. The last pair controls all successive columns.

All aspects of specified-format conversion are the same as in free-format (see Section 7) if the optional argument, *FORMAT* , is not given. If *FORMAT* is given, then it is used to specify the numeric conversion formats. The total field width may be chosen by the user or, if set to zero, is chosen by the function such that at least one space will be left between adjacent numbers. The format to be used is determined by the fraction field width. If it is zero, integer format is used. It it is positive, then it indicates the desired number of digits after the decimal point. If it is negative, its absolute value minus one is the desired number of digits after the decimal point, and exponential format is used.

Specified-format conversion has to particular applications. The first is to allow better control of terminal output formatting. Terminal output of any form can be composed by first converting the output data using □*CONVERT* and then sending the resulting character matrix to the output terminal.

The second application involves writing data files which are to be used in other computer systems. When APL data objects are written on data files, the internal representation of that object is put on the file. However, this is inadequate if the file contains numbers and the file is used by another computer system because different computer systems may use different internal representation for its numbers. Thus, numbers must first be converted to an external from (character matrix equivalent) using □*CONVERT* before they are written onto the file.

6. Conventions Used by High-Level Functions

The conventions used in constructing the standard experiment configurations are relatively straightforward. Each of the bi-directional communication channels is a pair of mailboxes which are attached to a terminal or file in the appropriate cases. The mailbox identifiers of the various mailboxes are stored in certain standard global variables in the memory of the various processes. Every process has a scalar *⎕PROC* containing its own process number and a two-element vector named *⎕IO_BOXES*, specifying an input and an output mailbox which are used whenever the standard APL input/output operations are executed. Similarly, the subject processes and robot processes each have a two-element vector called *⎕IPC_BOXES*, specifying the mailboxes used by *⎕GET* and *⎕GIVE*, in that order. (In the experimenter's process, *⎕IPC_BOXES* is undefined.) The overall configuration of the experiment is recorded in the matrices *⎕SUBJECTS* and *⎕ROBOTS* in the experimenter's process. These are formatted as follows:

element	data from *I*th subject process
⎕SUBJECTS [*I*;1]	= process-identifier
⎕SUBJECTS [*I*;2]	= terminal number (or robot)
⎕SUBJECTS [*I*;3 4]	= *⎕IO_BOXES*
⎕SUBJECTS [*I*;5 6]	= *⎕IPC_BOXES*
⎕SUBJECTS [*I*;7]	= *⎕ALARM*

element	data from *J*th robot process
⎕ROBOTS [*J*;1]	= process-identifier
⎕ROBOTS [*J*;2 3]	= *⎕IO_BOXES*
⎕ROBOTS [*J*;4 5]	= *⎕IPC_BOXES*
⎕ROBOTS [*J*;6]	= *⎕ALARM*

Also defined after □*CR_LOG* has been called, is the two-element vector □*LOG_BOXES* , specifying the pair of mailboxes attached to the data-logging file.

Note that in an experiment using robots, if the *I*th subject process is connected to the *J*th robot process, that the terminal number of subject *I* (i.e., □*SUBJECTS*[*I*;2]) is $-J$, and that □*IO_BOXES* in subject *I* is just ◊ □*IO_BOXES* in robot *J* , thus setting up the connection between them.

Using the information in this section, the programmer can create modified versions of the standard configurations of Section 4. For example, to set up the situation in Figure 5, one can create a standard two-subject experiment and then establish the direct subject-to-subject channel by having the experimenter's process create two new mailboxes (identified by say *J* and *K*) and pass the pairs *J*,*K* and *K*,*J* to the two subjects, who can then communicate directly using □*SEND* and □*RECEIVE* .

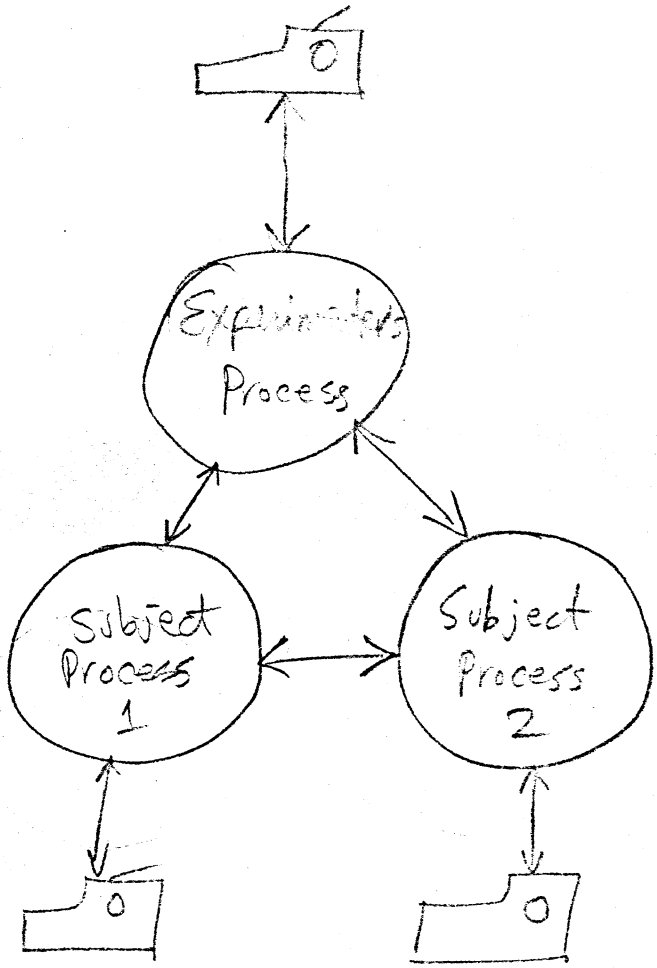


Figure 5: Modified standard configuration

APPENDIX 1

Operators not in CRMS APL

OPERATOR	NAME
⌹	Matrix division
⍢	Protected Junction
⍬	Rotate, 2nd dimension
⍞	Quad output

APPENDIX 2

Overstrike Transliterations

Currently, overstrike characters are not implemented. Operators with overstrike-character name are renamed as follows:

SYMBOL	REPLACEMENT SYMBOL	NAME
⍲	NAND	Nand
⍱	NOR	Nor
⍋	UPGRADE	Up grade
⍒	DOWNGRADE	Down grade
⍟	LN	Logarithm
!	EXCLAMATION	Factorial and Combination
⍉	TRANSPOSE	Transpose
⌽	ROTATE	Rotate and Reversal
⍝	∩	Comment
⍞	⊢	Quote-quad
'	⍞QUOTE	Quote input operator

APPENDIX 3

Additional primitive functions in CRMS APL

1) ∇ *R* ← ⊥ *A*

⊥ returns the type of *A* (or elements of *A*).

1 if the element is numeric, 0 if character

2) ∇ *R* ← ⊤ *A*

⊤ converts between character and its internal ASCII code equivalent.

3) ∇ *R* ← ∈ *A*

∈ returns 1 if *A* is defined, 0 otherwise

4) ∇ □*RANDSET A*

Set "random seed" to *A*

5) ∇ *A* ← □*INNUM* [o*TL*]

□*INNUM* returns a vector of numbers from the terminal. *TL* is time limit of wait. ⍳0 is returned if time limit is exceeded.

6) ∇ *A* ← □*INMIX* [o*TL*]

□*INNMIX* returns a vector of numbers and characters from the terminal. *TL* is the time limit.

7) ∇ *A* ← ⊢[o*TL*]

(same as □ returns a vector of characters from the terminal. *TL* is the time limit.

8) ∇ *A* ← □*TIME*

□*TIME* returns the time.

The values are:

A[1] = year	*A*[5] = minute
A[2] = month	*A*[6] = second
A[3] = day	*A*[7] = millisecond
A[4] = hour	*A*[8] = microsecond