

An APL at CRMS User's Manual for APL\360 Programmers

Wiley Greiner (1 August 1975)

[Typeset version by Paul McJones and Academia.edu Co-scientist, 2026]

Foreword and forewarning

Parts of this manual were written before the fact.

Logging on and off the CRMS system

A terminal is *sleeping* when no account name is logged on, and *awake* while an account name is logged on. A sleeping terminal will ignore without echoing all input, except the escape key. Typing the escape key initiates the CRMS logging on procedure by waking the terminal. The awakened terminal gives the prompt,

```
CRMS TIMESHARING  
NAME
```

and waits a little while for input. If input is not entered quickly enough, the terminal will print the message,

```
YOUR TIME IS UP
```

and go back to sleep. The terminal waits long enough for input to be entered at a relaxed typing speed. The initial input is expected to be an account name, followed by a carriage return.¹ Account names are issued through the CRMS office. They are often the name of a class, of a sponsor, or of the user himself. If the input is a recognized account name, the terminal will ask for a password.

```
CRMS TIMESHARING  
NAME CS-100  
PASSWORD:
```

Whenever a password is the expected input, the terminal does not echo any character entered (until the next carriage return, which terminates input). This is one of many security precautions taken by CRMS. If the password is OK, the terminal prints 'OK', meaning: the terminal will stay logged on (and awake) until logged off. But if the password is not OK, the terminal requests an account name, again. If too many wrong passwords are typed in a row, the proper authorities are notified, as a further security precaution.

¹Be sure that the symbols echoed on the terminal paper are the same as the symbols pasted to the keys typed. This guards against APL(ASCII) signals being sent to an ASCII(APL) terminal. More will be said about the two types of signals when double print set programming is discussed.

```
CRMS TIMESHARING
NAME AMATEUR
PASSWORD:
NAME PROFESSIONAL
PASSWORD: OK
9/14/75 13:09:23
```

COMMAND:

Terminals are logged off by responding to the 'COMMAND:' prompt with 'OFF' (carriage return).

COMMAND: OFF

The terminal acknowledges with the date and time, then falls asleep.

The same logging on and off procedures apply to dialed terminals, too. A logged off dialed terminal goes to sleep, but stays connected indefinitely (unlike APL\360 which hangs up).

Active and saved workspaces

The APL subsystem is entered by responding to the 'COMMAND:' prompt with 'APL' (carriage return).

```
COMMAND: APL
ACTIVE WS NAME: ACTIVE
)WSID
CLEAR WS
A←110
)SAVE IT2
B←⊖A°.≥A
)OFF
OFF ACTIVE
```

COMMAND:

Entering the APL subsystem does not involve an APL\360 like sign on. In fact, the APL subsystem may be exited and reentered repeatedly once an account name is logged onto the terminal.

Each account name may be logged onto two or more terminals simultaneously. So, an instructor may either provide each student with his own account name, or make one account name, say 'BUSINESS•147', available to his entire class, whichever seems more appropriate.

When the same account name enters the APL subsystem from two or more terminals, each terminal has access to a common library of APL workspaces. This library contains two types of APL workspaces: saved and active workspaces. Saved workspaces may be copied, dropped, loaded and saved from any terminal. Active

²These examples will omit the time-stamp messages that acknowledge such events as entering and exiting the APL subsystem, and copying, dropping, loading and saving APL workspaces. The omissions are intended to make the examples clearer.

workspaces may not. Each terminal that enters the APL subsystem has its own active workspace: no two of these terminals may share the same active workspace. So, each student using the previously mentioned *BUSINESS•147* account name should designate a unique active workspace name. This designation is requested immediately upon entering the APL subsystem. See the above example. The active workspace name was designated in response to the prompt,

ACTIVE WS NAME:

It is an error for the designated active workspace to be active (for another terminal). The designated active workspace may have been saved. If so, the type of the designated workspace changes from saved to active. Alternatively, the designated active workspace may be new. If so, it is implicitly created, cleared, and entered into the account name's library (as an active workspace).

Variables and functions in the active workspace are modified by executing immediate statements and entering function definition mode, respectively. Both may be modified by the *)COPY*, *)CLEAR* and *)LOAD* commands, but not by the *)SAVE*³ and *)DROP* commands.

The *)OFF* command changes the type of the designated active workspace from active to saved. The newly saved workspace may (like all saved workspaces) be copied, dropped, loaded or re-saved, and contains all of the functions and variables that were in the active workspace at the time of the *)OFF* command. Here is a continuation of the above example:

```
COMMAND: APL
ACTIVE WS NAME: JOHN_DOE--BUSINESS•1474
      )WSID
CLEAR WS
      )LOAD IT
      )VARS
A
      )LOAD ACTIVE
      )VARS
A B
      )DROP ACTIVE
      )WSID
ACTIVE
      )OFF
OFF JOHN_DOE--BUSINESS•147

COMMAND:
```

The '*OFF JOHN_DOE--BUSINESS•147*' message indicates where a saved version of the active workspace may be found. When the active workspace name is *CONTINUE*, the *)OFF* command acts like the APL\360 *)CONTINUE* command. The *)CONTINUE* command would be redundant at CRMS, and so is not supported.

³The *)SAVE* command (as well as the *)WSID* command) may modify the WSID of the active workspace. The WSID is not the same as the active workspace name, which is (as you may recall) specified in response to the '*ACTIVE WS NAME:*' prompt.

⁴Workspace names may be as long as desired (instead of being limited to 11 characters, as is the case with APL\360), and may contain almost all the printable characters (instead of being limited to letters and numbers).

The `)LIB` command lists the names of all workspaces in the account name's library. Active workspaces are surrounded by parentheses. Parentheses are not allowed in workspace names.

```
)LIB  
IT  
(JOHN_DOE--BUSINESS•147)
```

If the `)LIB` command has an argument, it is interpreted as another account name.

```
)LIB CS-100
```

Here, the workspace names listed are those that CS-100 has given permission to read (load and copy), or to read/write (load, copy and save but not drop). Read permission of a workspace is given to a specific account name by first loading that workspace, then typing the `)RPERMIT` command.⁵

```
)RPERMIT BUSINESS•147
```

Read/write permission is given to a specific account name by typing `)RWPERMIT` instead of `)RPERMIT`.

```
)RWPERMIT BUSINESS•147
```

If the arguments to `)RPERMIT` or `)RWPERMIT` are omitted, then all users are given read or read/write permission, respectively.

```
)LOAD GOSSIP  
)RWPERM
```

Permission may not be rescinded, except (indirectly) by dropping the workspace. Read and read/write permission apply to saved workspaces only. It is always an error for any account name to attempt to `)COPY`, `)DROP`, `)LOAD` or `)SAVE` any active workspace. Workspaces cannot be locked. APL\360 users, in particular, should remember that the colon is considered a valid character in a workspace name. Mistakes like

```
)LIB  
SECRET:LOCK
```

can be embarrassing.

The public workspaces may be listed by typing,

```
)LIB PUBLIC  
STATISTICS_PACKAGE  
PLOT  
INTRO-T0-APL-AT-CRMS
```

This last workspace may be loaded with the command,

```
)LOAD PUBLIC INTRO-T0-APL-AT-CRMS
```

Every account name has permission to read all public workspaces.

⁵The first four letters after the right parenthesis uniquely describe all APL\360 and CRMS commands. So, both `)RPER` and `)RPERMISSION_T0` are interpreted as the `)RPERMIT` command.

Variable and function names

Variable and function names may be as long as desired. They may contain the quad ($\square$), the underbar ($_$), and both underlined and non-underlined numbers, letters and the delta (Δ).

Names may not begin with any number (but may begin with any underlined number). Some system variables and primitives begin with the quad, but user defined names may not. All attempts to do so are thwarted at compilation time. User defined names may contain the quad elsewhere, however. For this reason, the quad is not delimited the way it is in APL\360. In APL\360, for example,

```
XYZ $\square$ ←3
```

would output three, then pass three to the monadic function *XYZ*. In CRMS, on the other hand, the variable *XYZ $\square$* would become three. The earlier interpretation would require a blank between *XYZ* and $\square$ ←3.

The internal workspace environment

Suspended execution

The *)ENV* command, which is not supported by APL\360, makes it possible to execute an immediate statement in the environment of any function on the SI stack. The *)SI* and *)SIV* commands have been changed slightly from APL\360 to accomodate the new *)ENV* command: the + symbol is printed by *)SI* and *)SIV*, rather than the * symbol. The + symbol indicates the present environment.

```
      )SIV
THIS[1] +   LOCAL
THAT[7]     LOCAL VARIABLE S
THIS[4]     LOCAL

      LOCAL
1
      )ENV THAT
      LOCAL
7

      )ENV THIS 2
      )SI
THIS[1]
THAT[7]
THIS[4] +

      )POP
POPPED FUNCTION(S): THIS THAT

      )SI
THIS[4] +
```

A *suspended* function appears at the top of the SI stack, and nowhere else on the SI stack.⁶ Any function on the SI stack that is not suspended is said to be *pendant*. So, if a function at the top of the SI stack appears more than once, it is pendant rather than suspended.

Pendant functions, and the header line of suspended functions may be displayed but not revised in function definition mode. All illegal revision attempts are thwarted by the text editor. The bodies of suspended functions may be revised, however. But, if this is done the line number of the suspended function will not be printed by the)SI and)SIV commands.

The)POP command can remove suspended and pendant functions from the SI stack, and can make a pendant function suspended.

The)POP and the)ENV commands make it more likely that an experiment can be debugged on the fly, and less likely that an experiment must be started all over again.

```

      ▽ INPUT; INSTR
[1] INSTR←'TYPE A, B OR C'
[2] □←INSTR
[3] X←'ABC'⌞1↑□
[4] INSTR←'FOLLOW INSTRUCTIONS, PLEASE ', INSTR
[5] DISPATCH X
[6] 'HURRAY!:' ▽

```

```

      ▽ DISPATCH Y
[1] →(LINE_A, LINE_B, LINE_C) [Y]
[2] LINE_A: . . .
      .
      .
      .
      ▽

```

```

      MAIN_FN
TYPE A, B OR C
WHAT?

```

```

INDEX ERROR
DISPATCH[1] →(LINE_A, LINE_B, LINE_C) [Y]

```

```

)SI
DISPATCH[1] +
INPUT[5]
MAIN_FN[65]

```

```

▽DISPATCH[0.5] →0 □IF AGAIN←4=Y ▽

```

```

)SI      Ⓢ NOTE THE FIRST LINE NUMBER
DISPATCH[-] +
INPUT[5]
MAIN_FN[65]

```

```

▽DISPATCH[0] AGAIN←DISPATCH Y ▽
DEFN ERROR: HEADER IS READ-ONLY

```

⁶This definition must be modified slightly when substacks are introduced.

```

[0] ▽
      ▽INPUT[5] →3 □IF DISPATCH X ▽
DEFN ERROR: FUNCTION IS READ-ONLY
[5] ▽
      )ENV INPUT
      6ρINSTR
FOLLOW
      →3
BRANCH OR RETURN IN BAD CONTEXT
      →3
      ^

      )POP
POPPED FUNCTION(S): DISPATCH
      )SI
INPUT[5] +

      ▽DISPATCH[0□] AGAIN←DISPATCH Y ▽
[0] DISPATCH Y

      ▽INPUT[5□] →3 □IF DISPATCH X ▽
[5] DISPATCH X

      →5
FOLLOW INSTRUCTIONS, PLEASE TYPE A, B OR C
A
HURRAY!!

```

Multiple executions

APL360 allows a number of executions to be suspended at the same time. CRMS only allows one suspended execution at a time. The executions stay suspended until a niladic right arrow is entered as an immediate statement. A *primary immediate statement* is the first immediate statement to be invoked after a right arrow is typed. All other immediate statements are said to be *debugging immediate statements*. So even if the primary immediate statement completes successfully, the next immediate statement will be a debugging immediate statement (unless of course a right arrow is typed first).

The execution invoked by a debugging immediate statement is never reflected on the SI stack, whether the execution terminates normally or not.

```

      )SI
PRIMARY[7] +

      DBG      ⌘ THIS IS A DEBUGGING IMMEDIATE STATEMENT

VALUE ERROR
DBG[1]  ATTEMPT←ATTEMPT+1

      ATTEMPT←0 ⌘ THIS IS ANOTHER DEBUGGING IMMEDIATE STATEMENT

```

⌘ IT MAY BE TEMPTING TO TYPE →1 HERE

```
)SI      ⌘ BUT THAT WOULD BRANCH TO PRIMARY[1], NOT DBG[1]
PRIMARY[7] +
```

The SI stack reflects a single primary execution. But an SI stack actually consists of one or more substacks, one substack per process.

```
)SI
MESSAGE_SWITCHING[3] +
MAIN[7]
```

```
PROCESS 1
LOGGING[7] +
SUB_TASK[5]
```

```
PROCESS 2
BATCH_JOBS[91] +
```

```
PROCESS 3
BATCH_JOBS[17] +
```

The only suspended functions here are *MESSAGE_SWITCHING* and *LOGGING*. The other functions in this SI stack are pendant. The suspended functions appear exactly once in the SI stack: at the top of exactly one substack.

The argument of the *)DB* command is a process number. The specified process is the context for all subsequent debugging immediate statements, and the *)ENV*, *)POP*, *)WIDTH*, *)DIGITS* and *)VARS* commands.

```
)DB 1
)ENV SUB_TASKS

)POP
POPPED FUNCTION(S): LOGGING

→5
```

Multiple processes

A primary immediate statement *implicitly* creates one process. This is process number zero, and is usually called the experimenter's process. It is initially the only process existing in the active workspace. Other processes may be *explicitly* created by a previously existing process. The creating process calls one of a number of primitives to explicitly create new processes. It is an error to call one of these primitives in a debugging execution. Explicitly created processes have a startup function, which is a niladic or monadic user defined function. The setup function parameter (if required) may be given by the creating process. Otherwise, a default parameter is passed. The startup function is the top function in the substack of the newly created process.

All processes may call any user defined function. Two processes may invoke the same user defined function simultaneously. In fact, there is no way for one process to gain exclusive access to a user defined function.

Variables, on the other hand, may not be shared between processes in the *.APL.SV.* sense. A copy of a variable may, however, be transferred from one process to another, instead. Modifying the copy does not change the original, of course. A variable is transferred between processes in a first in first out buffer (FIFOB).

The system variables `□WIDTH`, `□DIGITS` and `□RAND` are the width, digits and random seed of a process. Each process has its own system variables. They may be accessed and assigned anytime. All three system variables of an explicitly created process are initially set to their respective values in the creating process. The width and digits in the experimenter's process are initially 78 and 6, respectively. The random seed is initially a clock-dependent value.

```

      ▽ X←RANDOM_NUMBER_GENERATOR_TESTER QUANTITY;YOU
[1] YOU←□CR_ROBOTS[QUANTITY;1] $COMPUTE_IT
[2] ASK: □←'BIGGEST, SMALLEST, OR AVERAGE?: '
[3] →ASK □IF 4=X←'BSA'⋮1↑□
[4] 'THE RANDOM SEED IS NOW '; □GETFROM YOU
[5] X←(□GETFROM YOU)[X]

```

```

      ▽ COMPUTE_IT Q
[1] Q←?Qp100      ⍝ ? AFFECTS □RAND
[2] □GIVE □RAND ⍝ TRANSFER THE RANDOM SEED
[3] Q←(⌈/Q),(⌊/Q),+/Q÷pQ
[4] □GIVE Q ▽

```

```

      RANDOM_NUMBER_GENERATOR_TESTER 39
BIGGEST, SMALLEST, OR AVERAGE?: HUH?
BIGGEST, SMALLEST, OR AVERAGE?: SMALLEST (SORRY)
THE RANDOM SEED IS NOW 61792
2

```

`□CR_ROBOTS` is one of the many primitives that explicitly create other processes. The rightmost bracketed argument, 1, means that exactly one process is created. The ID number of this process is returned to the variable, *YOU*. The other bracketed argument is the parameter to be passed to the startup function of the new process. The startup function, in the example, is specified as the monadic function, *COMPUTE_IT*. The dollar sign means that the function *COMPUTE_IT* is not to be called, but is to be passed to `□CR_ROBOTS` instead.

The newly created other process executes *COMPUTE_IT*, while the experimenter's process continues past line [1] of the *RANDOM_NUMBER_GENERATOR_TESTER* function. Notice that the other process performs time-consuming number crunching, while the experimenter's process produces output and waits on input.⁷ Depending on the human subject's input, the input and output may be performed an arbitrary number of times. The `□GETFROM` primitives synchronize the two processes. `□GETFROM` does not return until the other process calls the `□GIVE` primitive. This is because `□GETFROM` waits on the FIFOB, introducing a delay. `□GIVE`, on the other hand, never introduces a delay since the value given (a copy of `□RAND` or *Q* in this case) stays in the FIFOB until a recipient comes along. The FIFOB can hold both `□RAND` and *Q* at the same time if necessary. `□GIVE` and `□GET` are used by a subject or robot process to give values to,

⁷It is generally more efficient to perform a compute-bound computation and an I/O-bound computation simultaneously rather than sequentially.

and get values from, the experimenter's process. □*GIVETO* and □*GETFROM* are used by the experimenter's process to give values to, and get values from, a subject or robot process. The concepts of subject and robot processes are further developed in the standard configurations subsection.

An experiment

Double print set APL programming and the CRMS character set

The CRMS computer can interact with humans through either the APL or the standard ASCII print set. Of course, the two print sets are designed for different man-machine applications. All APL programs are written with the APL print set; while many large APL programs communicate to their human subjects in ASCII. Actually, it is not possible to enter function definition mode with the ASCII print set since that set lacks the ∇ symbol. Although lacking the numerous Greek symbols found on the APL print set, the ASCII print set contains more natural looking English sentences: here, lower case letters are understandably more useful than fancy squiggles like iota and rho.

Proper use of the APL(ASCII) print set requires:

- that the APL(ASCII) print wheel be physically mounted on the terminal,
- that the terminal keyboard be the APL(ASCII) keyboard layout, and
- that the computer interpret transmissions between it and the terminal as APL(ASCII) signals.

Physically mounting a new print wheel involves removing the ribbon then the old wheel before installing the new wheel then the ribbon. The details of this procedure may be found in the DTC manual.⁸ A different procedure is required to change the print set of the graphic display units.⁹ Figure [SO AND SO] demonstrates both the APL and the ASCII print.

Permanently attached to every terminal is either the APL or the ASCII keyboard layout. Both keyboard layouts are shown in figure [..]. The semicolon, asterisk, left and right parentheses and brackets, and the dollar sign keys are among the keys located in different places on the two keyboard layouts. Fortunately, keys A through Z are located in the expected places on both keyboard layouts. Unfortunately though, capital A through Z are *not* shifted on the APL keyboard layout, but *are* shifted on the ASCII keyboard (unless the 'ALL CAPS' mode is selected). The APL keyboard layout - to - APL print wheel mapping algorithm changes drastically when the knob is turned between 'LOCAL' and 'LINE.' The -, ~, >, and ≠ keys (when the knob is turned to 'LINE'), for example, become the ×,),] and v keys (when the knob is turned to 'LOCAL'). The ASCII keyboard layout - to - ASCII print wheel mapping algorithm also changes when the knob is turned, but much less drastically.

⁸The *DTC-300 Users Manual* is available from Data Terminals and Communications, 1190 Dell Avenue, Campbell, California 95008.

⁹Graphic Display Unit Manual

The computer cannot sense whether it is connected to an APL or an ASCII print wheel, or to an APL or ASCII keyboard layout. Instead, the computer must be told whether to send, and expect to receive, APL or ASCII signals. This is done by logging-on, taking care that the symbols printed on the terminal look like the symbols pasted on the keys. (Sometimes it may be necessary to enter or exit from the 'ALL CAPS' mode, first.) If the user inadvertently indicates the wrong type of signal, the computer will send, and expect to receive, ASCII signals from an APL terminal, or APL signals from an ASCII terminal. Once the mistake is made, the rectifying log-off message must be sent, in the type of signal expected by the computer.

Double print set programs are written in APL to communicate to subjects in ASCII. *Single print set* programs are written in APL to communicate exclusively in APL. Single print set programs can often be coded, debugged, edited and executed without ever leaving the terminal or changing its print wheel. Single print set programming can be appropriate for number-crunching, non-interacting applications, or applications translated from APL\360, which supports just the APL print wheel. Compared to single print set programs, double print set programs are both (1) easier on the subject, and (2) more challenging to (read: harder on) the programmer.¹⁰

Programmers can create lower-case output with the `⌵LOWER` primitive:

```
V←'T', ⌵LOWER 'HIS IS A LOWER-CASE SENTENCE.'
```

assigns to *V* the 30 element character vector 'This is a lower-case sentence.'. The `⌵LOWER` primitive has no affect on any characters besides capital A through capital Z. Capital A through Z, however, are converted into lower case a through z, respectively. The argument passed to `⌵LOWER` may be a scalar or an array of any rank whatsoever, but may not contain any numbers. `⌵LOWER` always returns a result whose shape is equal to that of the argument.

Sending the variable *V* to a terminal properly outfitted with the ASCII print set causes the following line to be typed:

```
This is a lower-case sentence.
```

But sending *V* to a terminal properly outfitted with the APL print set causes

```
T$$$ $ $ $$$$$-$$$$ $$$$$$.
```

to be typed instead, which is rarely the desired output.

Characters which cannot be represented by the APL(ASCII) print set-but can be represented by the ASCII(APL) print set-are output as dollar signs. Characters which cannot be represented by either print set are called non-printing characters. Non-printing characters are not output as dollar signs by either print set.

Both print sets can represent all CRMS numeric output. CRMS however, represents negative numbers, and numbers with a negative E-exponent, different than APL360 does: the '-' symbol is used rather than the '⌵' symbol. So, the APL immediate statement,

```
⌵1E-30-1E-30
```

causes '-2E-30' to be printed.

¹⁰In spite of the difficulties, many large CRMS experiments are double print set programs. Although not recommended for novices, double print set programming becomes easier with time. Double print set programming is also made easier by a number of non-APL360 features provided by CRMS, including the `⌵LOWER` primitive, the escape character '⌵', literal input, and the translate primitive. Each of these features will be discussed later in this section.

The CRMS character set

There are 256 characters in the CRMS character set. Certain characters are non-printing—such as the tab and reverse line feed printer-control characters. Other characters can be represented with just one of the two print wheels. The remaining characters can be represented with both print wheels. But the two print wheels use different symbols to represent some characters.

The character zero, for example, is represented with the 0 symbol on the APL wheel, and the Ø symbol on the ASCII wheel. Some of the differences are even more trivial: the APL asterisk has five points, while the ASCII asterisk has six. Certain other characters have two representations which are more noticeably different. In most cases, however, the two symbols are at least vaguely related. Here are some characters with noticeably differing representations on the print wheels:

The APL '""' and the ASCII double quote

The APL 'α' and the ASCII at sign

The APL '~' and the ASCII raised not (a Spanish accent symbol)

The APL '◇' and the ASCII percent sign

The APL '↑' and the ASCII cap

The APL '≠' and the ASCII number sign

The APL '⋈' and the ASCII ampersand

Among the most commonly used overstrike characters are the underlined capital letters, and the underlined numeric digits. (The latter are not available in APL\360.)

The only character represented by both a non-overstrike ASCII symbol and an APL overstrike is the exclamation point!

Certain overstrikes supported by other APL installations are not needed, and hence are not supported by CRMS. The non-overstrike symbols $\vdash$ $\dashv$ $\{$ $\}$ $\$$ and X should be used instead of the $\{$ $\}$ $\mathfrak{f}$ $\mathfrak{d}$ $\$$ and $\mathfrak{x}$ overstrikes, respectively. The $\mathfrak{f}$ overstrike is supported by CRMS.

The escape character '⋈'

Every character in the CRMS character set has its own unique character code. Each character code, in turn, is a unique integer between zero and 255. Figure [XXX] lists the character codes, *in octal*, for all characters in the CRMS character set. Barely half of these can be represented by the ASCII print set, and well over half can be represented by the APL print set.

It is sometimes desirable to specify a character that cannot be represented by the print set currently in use. This specification is possible with the escape character '⋈' (shift X for X-scape). The APL constant ' $\mathfrak{N}$ ' specifies the character with character code of N , *in octal*.

```
'⋈41'      ↔ 'A'
'⋈41⋈42 ⋈43D' ↔ 'AB CD'
'⋈418'      ↔ 'A8' ␣ '8' IS NOT AN OCTAL DIGIT
'⋈377'      ␣ THIS IS THE LARGEST CHARACTER CODE
'⋈417'      ␣ CODE TOO BIG, SO TRANSLATOR GIVES ERROR
```

A number of the commonly used printer-control characters are provided as reserved words, for programming convenience:

```
'␣147' ↔ □BELL  Ⓜ EXIT PLOT MODE AND RING BELL
'␣150' ↔ □BS    Ⓜ BACKSPACE
'␣151' ↔ □TAB   Ⓜ TABULATE
'␣152' ↔ □LF    Ⓜ LINE FEED
'␣153' ↔ □RLF   Ⓜ REVERSE LINE FEED

'␣154' ↔ □TOP   Ⓜ SKIP TO TOP OF NEXT PAGE
'␣155' ↔ □CR    Ⓜ CARRIAGE RETURN WITHOUT LINE FEED
'␣375' ↔ □CRLF  Ⓜ LIKE THE APL\360 CARRIAGE RETURN
'␣377' ↔ □EOF   Ⓜ SPECIAL END-OF-FILE INDICATOR
'␣173' ↔ □ESC   Ⓜ ASCII ESCAPE, DON'T MISTAKE FOR ␣
'␣376' ↔ □IDLE  Ⓜ DOES NOTHING, BUT DELAY A CYCLE
'␣374' ↔ □NULL  Ⓜ DOES NOTHING, PERIOD.
```

The null characters may be used as placeholding characters in large vectors. The null characters need not be compressed away before outputting the vector. In other words, outputting *VEC* is equivalent to outputting $(VEC \neq \square NULL) / VEC$.

Control characters are input by typing the control key and another key, simultaneously. If the other key is 'B' or '/' then we say that control-B or control-/ was typed. Control characters are used by CRMS for *editing* text, and so are not usually *entered* as text. There is, however, a procedure for entering control characters as text. This procedure is to type control-L (*literal* mode) immediately before typing the control character to be entered. Every control character may be entered this way, except control-M, the carriage return character.

So typing 'X', '←', quote, control-L, any control character except control-M, quote, and then carriage return, enters the specified control character as text between the two quotes, and then assigns the character constant to the variable *X*. The immediate statement would appear, on the terminal printout, as

```
X←'
```

if the specified control character were control-Z, or some other non-printing, non-printer-control character. It seems as if one quote is missing, but it isn't. Actually the second quote was printed on top of the first, and so the type appears darker and fuller than it might otherwise be.

Although the computer will never be fooled by these obscure visual anomalies, the same cannot be said for humans. For this reason alone, programmers who value documentation tend to shy away from *literal* mode, and use the escape character '␣' instead to specify control characters.

When *N* is a capital letter, the constant '␣*N*' specifies a character with a character code equal to that of *N* plus 100 octal or 64 decimal. This feature is provided at CRMS because many ASCII printer-control characters are defined as control characters. Control-*N* inputs a character with a character code equal to that of '␣*N*'. □*CR* and □*LF*, for example, may be specified as '␣*M*' and '␣*J*'. This dovetails nicely with the ASCII definition of carriage return and line feed: control-M and control-J, respectively.

The *T* primitive (translate) converts characters into their integer character codes,

```
T'ABCD' ↔ 33 34 35 36
```

and integers in the proper domain are converted into characters with the same code,

```
TT'XYZ'      ↔  'XYZ'  
TT'A' 1      ↔  'A' 1
```

The domain error is generated by any attempt to translate a number that is not a proper character code. The translate primitive is its own inverse, and both the domain and the range of the translate primitive is the CRMS character set *union* the set of integers between zero and 255.

Translate makes it possible to generate any character in the CRMS character set, at execution time. The escape character method of generating characters is evaluated at compile time.

Applications which use the translate primitive may assume that the digits '0' through '9' have contiguous character codes. One such application converts an integer into a character vector:

```
TC0+ 10 10 10 10 T3579  ↔  '3579'
```

where the constant $C0 \leftarrow T'0'$. Contiguous character codes may also be assumed by the inverse function, which converts a character vector into an integer:

```
MC0+10⊥T'9876'  ↔  9876
```

where the constant $MC0 \leftarrow -C0$.

The full duplex text editor

The internal construction of CRMS data

The six types of CRMS data are integers, floating point numbers, characters, arrays, pass by reference parameters, and the undefined value. Associated with each type of data is a unique differentiator. Although the APL machine can distinguish the six differentiators, their lengths are not all the same. The floating point number differentiator, for example, is a single bit, while the character differentiator is a 24-bit field. A more detailed, bit-level look at APL differentiators may be found in the APL Processor Reference Manual.

CRMS integers, floating point numbers, and characters contain a total of 32 bits, which is the memory's word length (no coincidence).

Integers and floating point numbers

CRMS integers and CRMS floating point numbers both contain 23 significant bits, which is slightly less than seven decimal digits. The expected precision of the APL mathematic primitives (* ⊗ ○ and !) is about five decimal digits. For this reason, certain computations may not be suitable for APL at CRMS.

Besides the 23 significant bits, CRMS integers and floating point numbers contain one sign bit. All APL numbers are represented in sign-magnitude form. This representation, unlike the IBM/360 representation of integers, is closed under negation:

any non-zero integer is negated by complementing its sign bit, and leaving its 23 significant bits unchanged. The sign bit of zero is always zero.

CRMS integers contain 23 significant bits, one sign bit, and an eight-bit *integer differentiator*, for a total of 32 bits. Integer differentiators are used to distinguish CRMS integers from all other CRMS data types.

CRMS floating point numbers contain 23 significant bits (called the mantissa), one sign bit, and an eight-bit exponent. Floating point numbers are always normalized, which means the leftmost significant bit is always on. This bit serves a further purpose: as the single-bit *floating point number differentiator*.

If S is the sign bit of a CRMS floating point number, and I is the binary integer representation of the 23 significant bits, and E is the binary integer representation of the eight integer bits, then the floating point number F , is

$$F \leftarrow (1 - S + S) \times I \times 2^{E - 128 - 22}$$

The constants mean that:

- The exponent is a power of 2 (and not the power of 16 used by IBM/360 implementations of APL).
- The exponent is interpreted as an excess 128 number, which means the true exponent is actually 128 less than the binary representation of E . Since E is eight bits, the true exponent is between -128 and +127.
- The radix point of the 23-bit mantissa is between the first and second bits, counting from the left; which is after the 22nd bit, counting from the right. (The IBM/360 implementations of APL have the radix point between the fourth and fifth bits, counting from the left). Since the leftmost significant bit is always on, the mantissa is never less than one, but always less than two. This is because the smallest mantissa is 1.000..., and the largest is 1.111... (base 2).

Binary values are always represented as 32 bit CRMS integers or floating point numbers. The binary primitives, such as `v` and `⋆`, however, always return binary CRMS integers. Binary CRMS integers are always returned by the comparison primitives, such as `<` and `≥`, too.

Adding, subtracting and multiplying two CRMS integers together returns a CRMS integer whenever possible. Otherwise a CRMS floating point number is returned. The multiplication

$$1 \ 2 \ 100000 \times 100000$$

returns two CRMS integers followed by one integral CRMS floating point number. Notice that integers and floating point numbers may be mixed within the same array.

Adding, subtracting and multiplying a CRMS integer with an integral CRMS floating point number always returns an integral floating point number. Division of anything by anything always returns a floating point quotient, which might possibly be integral. No result is returned, of course, by a primitive that generates an error. The divide by zero error is generated in lieu of a result whenever the denominator is zero and the numerator is not. If both are zero, the division returns the CRMS integer, one, which is in keeping with an (arbitrary) APL convention.

Addition, subtraction and multiplication of integers never generate either the floating point exponent overflow or the floating point exponent underflow errors.

The floating point exponent overflow error is generated whenever the magnitude of the true result of +, -, × or ÷ is too large to be represented as a CRMS floating point number.

The largest floating point number is about 3.04E38. Its reciprocal is almost exactly the smallest positive floating point number, which is about 2.95E⁻³⁹. This is because the reciprocal of 1.111... (base 2) times 2¹²⁷ is so close to 2⁻¹²⁸.

An underflow happens when the magnitude of the true result of +, -, × or ÷ (as well as <, ≤, ≥ and >, which internally computes a subtraction) is not zero, but is 'significantly' (meaning: to a correctly rounded 23 significant digits) less than the smallest (normalized) positive floating point number. The floating point underflow error is generated whenever both an underflow happens, and the underflow condition is enabled. When an underflow happens, but the underflow condition is disabled, zero is returned rather than the true result. The runtime mathematic primitives are designed to run with the underflow condition disabled. So, enabling the condition may cause unexpected floating point underflow errors to be generated by the mathematic primitives.

Numeric constants which are written in E-notation, or that contain a decimal point, are interpreted as floating point numbers whenever possible. The constant vector

1 1E0 1.

for example, contains a CRMS integer followed by two APL floating point numbers.

Zero cannot be represented as a floating point number, since floating point numbers are always normalized. So, the constant vector

0 0. 0.00 0E5

contains four CRMS integers. Consequently, the FORTRAN trick of adding 0.0 to an integer to produce an integral floating point number does not work at CRMS. Integers can always be *floated*, however, by multiplying (or dividing) them by 1.0. Integral floating point numbers that can be represented as integers can be *fixed* by using | or |.

The □COEF monadic scalar primitive extracts the coefficient (which is one-half the mantissa) of its argument. The range of coefficients is (̄1,̄0.5] union [0.5,1).

□COEF X+X ↔ □COEF X
□COEF 1 2 3 4 ↔ 0.5 0.5 0.75 0.5

The □BEXP monadic scalar primitive extracts the biased exponent (which is one plus the binary integer representation of the exponent) of its argument. The biased exponent is always an integer between 1 and 256.

□BEXP X+X ↔ 1+□BEXP X
(□COEF X)×2*(□BEXP X)-128 ↔ X ■ ASSUMING * IS ACCURATE.
□BEXP 1 2 3 4 ↔ 129 130 130 131

The □COEF_□BEXP dyadic scalar primitive creates a floating point number with a coefficient equal to that of the left argument, and a biased exponent equal to the right argument, which must be a CRMS integer.

(□COEF X) □COEF_□BEXP □BEXP X ↔ X

Characters and arrays

Characters contain an eight-bit character code (for the 2*8 characters in the CRMS character set), and a 24-bit *character differentiator*. This differentiator, together with the single-bit floating point differentiator and the eight-bit integer differentiator make it possible to mix any combination of integers, floating point numbers—both of which are called numeric—and characters in the same array. CRMS provides a primitive that indicates which elements of an array is numeric.

The $\perp$ primitive is a monadic scalar primitive (called the indicator primitive) which returns a one if its argument is either an integer or a floating point number, and returns a zero if its argument is a character.

```
+/\110      ↔ 10
+/\110p'X'   ↔ 0
\X='16'+5    ↔ 0 0 1 0 1
(\V)/V       ↻ RETURNS THE NUMERIC ELEMENTS OF V
```

Array pointers are formed by a 16-bit *array differentiator* field and a 16-bit address field. The address locates an array body. An array body contains as many (32-bit) words as the array has elements, plus as many words as the rank of the array (one word per dimension), plus two words of overhead. The overhead words are used primarily by the APL machine's dynamic storage allocator. The smallest array body, three words, is that of the empty vector. The next smallest, four words, are those of single-element vectors and empty matrices. Array bodies reside in array storage areas. Each CRMS process has its own array storage area, and these areas are never shared by more than one process. If a CRMS process attempts to create an array body which would require more space than its array storage area could accomodate, then the error message,

ARRAY STORAGE NNN WORDS TOO SMALL

is printed, and the process stops. The programmer may restart his stopped process after increasing the size of the array storage area (up to an installation-defined limit), with the $\rangle$ SZ command. An alternative solution is to $\rangle$ ERASE one or more variable(s), and then restart.

Scalars never reside in array storage areas. So interestingly enough, assigning (the scalar) zero to a variable always frees as much space as is freed by erasing the variable.

The undefined value

The undefined value contains one field: the 32-bit *undefined value differentiator*. The undefined value has a symbol at CRMS: $\cup$. The undefined value symbol may appear to the right of an assignment arrow, in certain cases. The $\cup$ symbol can be used to re-initialize variables to their clear WS value.

```
A←B←\cup    ↔  )ERASE A B
```

If A or B is subsequently read without being set first, then CRMS will generate the value error. Certain illegal uses of $\cup$, such as

```

VEC[I]←u
2×u
u[1]

```

will cause the peculiar value error to be generated. This error is generated by an illegal attempt to read the undefined value from the stack. The undefined value can be loaded onto the stack with the `u` primitive. The value error, on the other hand, is generated by an illegal attempt to load the undefined value onto the stack.

If an APL function attempts to return the undefined value (which is not an error on APL\360), CRMS generates the value error.

The `)VARS` command does not display variables assigned the undefined value. The `)VARS *` command does.

```

)CLEAR
ACTIVE CLEARED

```

```

A←B←3

```

```

C←u

```

```

)VARS

```

```

A B

```

```

)VARS *

```

```

A B C*

```

The existence primitive, monadic `ε`, detects whether a variable is defined. The immediate statement,

```

εVAR

```

prints 1 if `VAR` is a defined variable, and prints 0 if `VAR←u`.

```

εVAR←VAR1

```

prints 1 if `VAR1` was a defined variable or a function which returns a result, and prints 0 if `VAR1` was undefined. Certain illegal uses of `ε`, such as

```

εVAR[I]
εFUNCTION CALL
ε2
εA+B

```

are thwarted at translation time. The last expression should not be mistaken for for the legitimate expression,

```

(εA)+B

```

Pass by reference parameter

The headers of user defined APL functions specify which parameters are to be passed by value, and which are to be passed by reference. The header,

$\nabla A \leftarrow B \text{ } AFN \text{ } \rho C$

for example, specifies that B is to be passed by value, and that C is to be passed by reference ($\rho \Rightarrow$ reference). All variables passed to all user defined APL\360 functions are passed by value. A variable passed by value, such as X in the expression,

$2+X \text{ } AFN \text{ } Y$

is not modified by the called function (AFN in this case). The variable Y , however, which was passed by reference, becomes equivalent to C , until AFN returns. In other words, any modification to either C or Y affects the other, equivalently. So, executing $C \leftarrow U$ in AFN , erases Y . And if Y were global, executing $Y \leftarrow U$ in AFN would erase C . Furthermore, if Y were undefined immediately before calling AFN , then C would be undefined immediately afterwards. Variables passed by reference may be undefined. Variables passed by value, however, must always be defined. Otherwise the value error is generated (as it would be on APL\360).

The need to distinguish between variables passed by reference, and variables passed by value is filled by the pass by reference data type. An attempt to access this data type (indirectly) accesses a global or local variable, instead. The variable is addressed by a 17-bit field in the pass by reference data type. One bit indicates whether the other 16 bits form the address of a global variable or a variable local to the calling function. The 17-bit field never locates an APL expression or constant, since these may not be passed by reference. All attempts to pass them by reference, such as,

$X \text{ } AFN \text{ } ,Y \quad \text{AND}$
 $X \text{ } AFN \text{ } 2$

are thwarted at translation time. Assignments, like

$X \text{ } AFN \text{ } Y \leftarrow A \oplus B$

however, are permitted. The other field of the pass by reference data type is the 15-bit *pass by reference differentiator*.