The design and implementation of an experimental APL system on the small, sensor-based System/7 is described. Emphasis is placed on the solution to the problem of fitting a full APL system into a small computer.

The system has been extended through an I/O auxiliary processor to make it possible to use APL in the management and control of the System/7 sensor-based I/O operations.

An APL interpreter and system for a small computer

by M. Alfonseca, M. L. Tavera, and R. Casajuana

System/7[1] is a small-size, special-purpose IBM computer for sensor-based applications. It has a limited memory of two to 64 kilowords, 16 bits per word, not being byte-oriented. Its arithmetic unit includes an accumulator, one instruction address register, and seven work registers as well as several indicators. The central processing unit (CPU) is hardwired (not microprogrammed), with a fast cycle of 400 nanoseconds, and accepts up to about 40 machine language instructions where bit and byte handling, fixed point multiplication and division, and all the floating point operations are absent. Only fixed point 16-bit integer numbers are supported.

The standard peripheral equipment is likewise limited: small capacity disks (1.2 megawords),[2] a tape attachment, slow card reader and punch, matrix printer, and a teletype console. However, telecommunication equipment and sensor-based input-output (I/O) equipment are available in large variety, including direct CPU attachment to other computer systems, and analog and digital input and output. Four levels of operation, with four complete arithmetic unit register sets, are supported in order to serve the different I/O interruptions.

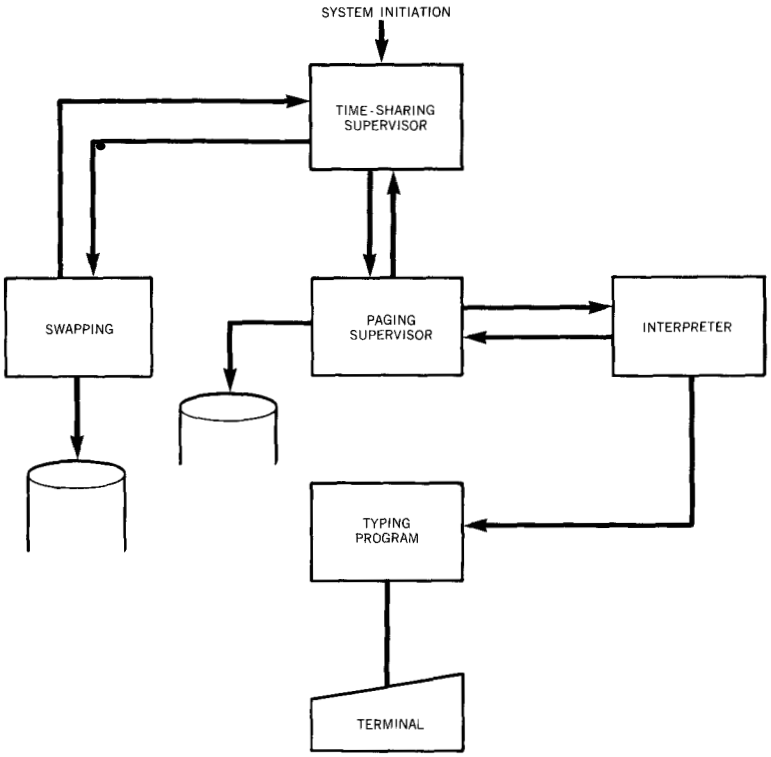
The computer was originally meant to be used primarily as an intelligent, sensor-controller terminal, connected to a host computer (e.g., the System/370 series) where the preprocessed data would be sent and further processed. Therefore, it was announced with almost no stand-alone software, including only a

disk support system and a primitive assembler with the absence of any operating system, but a sophisticated assembler and a FORTRAN compiler as well as an automatic program generator (a PL/I-based interactive support program to help users design their application programs) could be used at the host computer to prepare System/7 programs. As time proceeded, stand-alone software began to increase, and the aforementioned assembler and FORTRAN compiler were made available within the configuration, as well as a sensor-controller programming system (LABS/7[3]).

Before much of this had happened, we had come to the conclusion that a stand-alone System/7 provided with an interactive, sensor-based, high-level control language was justifiable. Yet, process control applications for which System/7 may be applicable could be eased if this high-level language had a matrix-handling facility, because process control problems commonly give rise to matrix equations. We chose APL for our implementation for the following reasons:

- APL[4] is a highly sophisticated high-level language, having a large number of symbolic built-in functions (primitive functions) and operators that render it possible to write complicated programs within a simple syntax in a concise form.
- Primitive APL functions and operators take arrays as well as scalars as their working objects, so that loopless programs may be written, in this way counteracting the loss in translation time inherent to interpretive systems as compared to compiling systems. Besides, many of the common array-handling operations are primitives to the language, such as matrix products, matrix inverses, and so forth, so that the language is especially useful for problems having to do with matrix computation, as is the case with process control and other sensor-based related problems.
- APL systems usually give the user access to an active work space in the main memory where he may store and execute his defined functions and variables. He may further save those work spaces in a disk library and recall them at his will, in this way giving to each user a flexible and easy procedure for building and using his own application packages.
- The APL shared variable facility (see below) makes it very easy to include language extensions in the form of auxiliary processors built to cope with special tasks. This ability would be especially important in our case, for the sensor-based I/O capabilities of the System/7 had to be, and were not, dealt with in the standard APL language definition as was necessary if this language were to be the answer to our needs.

Figure 1 System structure



The implementation of a full APL system in a small computer met a number of difficulties. Although several attempts have been made along this line, such as the APL-1130[5] and the APL-3[6] systems, all of them had tried to implement only a subset of the language,[7] whereas in our case, our purpose was precisely the opposite, that is to say: to enhance the language. Besides, it was just not possible to take the APLSV source code and translate it into System/7 instructions, because several APLSV features were not desirable, such as multiple active work spaces, while the absence of most arithmetic operations in System/7 machine language rendered the translation impossible. It was also our purpose to design our own algorithms in order to try to optimize several features for storage occupation, thus leading, for instance, to our having developed a double table-driven syntax analyzer unlike any of the APL implementations known to us. This analyzer uses the well-known transition matrix approach,[8] but splits up the table into two parts to save space.

The implementation of an APL system on a small computer, such as System/7, presented the following challenges:

- Representation and management of data.

- The difficulty of fitting a full APL system into a small-size computer.
- Implementation of a time-sharing system.
- Management of peripherals.

The Computer Science Department of the IBM Scientific Center of Madrid has built an experimental system,[9] called APL/7, implementing the full APL language on a 16K-word System/7, with I/O typewriters (IBM Model 735) as terminals, connected through the digital I/O groups of the System/7. The objectives of the work were:

- To demonstrate the feasibility of solving the aforementioned problems.
- To provide our System/7 with an interactive high-level language.
- To extend APL to sensor management in order to analyze its suitability for process control and laboratory automation applications.

The system has been completely designed and implemented. Figure 1 shows the general structure of the system. The subsequent sections of this paper describe the development of the APL/7 system in meeting the objectives.

Data representation and management

The previously mentioned arithmetic limitations presented us our first problem in the internal representation and handling of the different data types (characters and numeric) existing in APL (giving rise internally to Boolean, integer, floating point numbers, and characters), since the System/7 features only permit the management of integers.

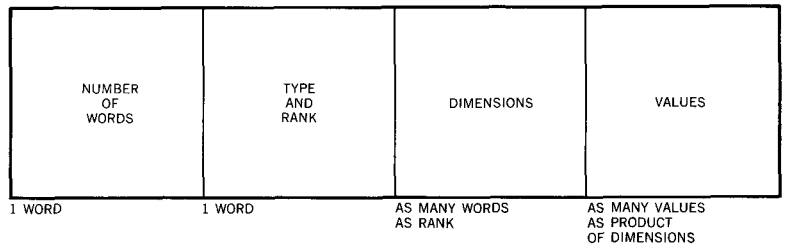
For floating point numbers, instead of the classical solution of simulating binary floating point arithmetic, we considered a pseudo-decimal approach. A floating point number is represented in four consecutive System/7 words, the first containing the exponent (base 10) and the other three the mantissa in base 10^4 (i.e., each of them is a fixed point integer smaller in absolute value than 10^4); the sign is carried by the three mantissa words.

If we call w_1, w_2, w_3, and w_4 the contents of these words in the order indicated, the represented floating point number value would be taken as:

$$10^{w_1} \times ((w_2 \times 10^8) + (w_3 \times 10^4) + w_4)$$

This representation allows up to 12 decimal digit numbers with an exponent up to ±9999.

Figure 2 General internal data format



A right-justified normalization was used so that the representation is unique and with minimal exponents. This representation gives rise to the following advantages:

- It is possible for the system programmer to process the exponent part directly.
- The register overflow condition cannot happen when adding or subtracting two numbers in this representation, since twice 10^4 is less then 32767, which is the maximum positive number a System/7 register may contain.
- Larger exponents are possible, with respect to most programming languages, and correspondingly reduced the possibility of real overflow (numbers larger than the maximum allowable.)
- Any integer number with less than 13 digits is represented exactly, with no truncation error, reducing to a minimum the need for comparison tolerances.
- The process of translation into external characters (decimal) is simplified.

The general internal format of the data is shown in Figure 2. Depending on the type, the values occupy the following amounts:

Type 3 (characters)	1 byte (½ word) per data item
Type 2 (floating point)	4 words per data item
Type 1 (integer)	1 word per data item
Type 0 (Boolean)	1 bit per data item

In the first and fourth cases, if the number of values is such that they do not fill a whole word length (odd number of characters, number of Boolean values not multiple of 16), the last word is left-justified and the rest of the word is not used. In this way, it is possible to process the APL object (scalar, vector, matrix, high-order arrays) from left to right, and to know, from the type, whether the processor must look for data every bit, byte, word, or group of four words, starting after the dimensions, the number of which is also given in the same word as the type.

The work of writing the code for the primitive functions started by designing the basic arithmetic and logic subroutines using this internal representation. Thus, later on it was possible to undertake the task (anyway tedious) of the design of algorithms, some of them standard, some new, for the huge amount of more sophisticated operators existing in the APL language.

Size of the system

The small size of the main storage, reduced in our case to 16K words (16 bits per word) and in the largest possible System/7 to 64K words, made it necessary to find a feasible solution for the problem of squeezing a complete APL system into that size. "Standard" APL systems run on a minimum of 128K bytes, approximately equivalent to the largest possible System/7 main storage (APLSV, for instance, needs a minimum of 256K bytes).

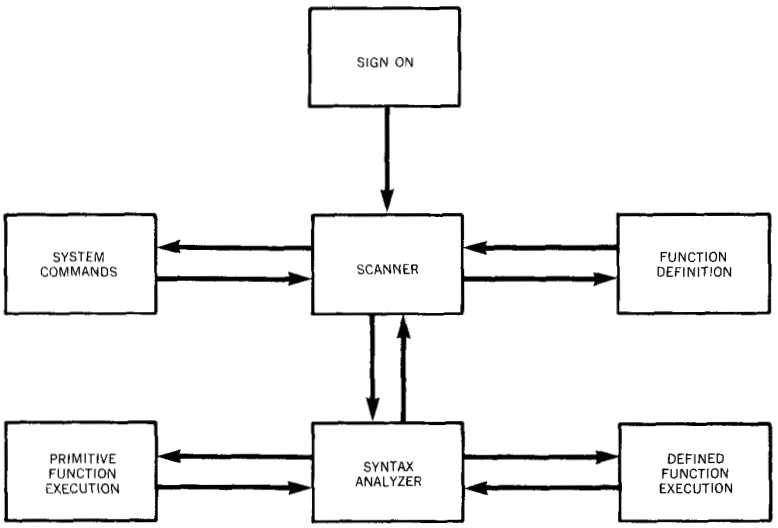
Up to now, there has been some work in the direction of implementing APL in small machines (e.g., the IBM 1130 computer,[5] the System/3[6]), but all of them solved the size problem by reducing the APL power, using only a subset of it. Our intention was to have a full APL system, with no reduction in the applicability compared to the standard ones. Our approach was a virtual memory allocation processor controlling a modular interpreter. Response time would be traded off against work-space size.

Since the System/7 is not a microprogrammable machine and has no virtual storage hardware processor, the virtual memory allocation had to be software simulated. The problem was solved through the following steps, which describe the paging algorithm:

1. The interpreter code was written in modules, a module being defined as a relocatable program occupying up to 128 words. Since these modules will be transferred to the main storage and given control in different situations depending on the APL programs to be executed, the modules had to be strictly relocatable, and hence, all the System/7 instructions that use as arguments absolute addresses had to be avoided when the code was written.

   Given the relocatibility requirement, the size of 128 words is a compromise between the number of I/O operations, which obviously increases as the size of the modules decreases, and the simplicity of the programming task, since the System/7 instructions permit direct references (relative addressing) to other instructions in the program provided the difference between their addresses is lower than 128.

Figure 3 Structure of the interpreter



Despite the restrictions, 128 was found to be a quite adequate size to fit the programs for the very basic (and hence very often called) APL operations. If a larger size had been used, the modules would have been able to contain more than one of these basic operations, and time would have been wasted in transferring the code to main storage for these extra operations when only one of them was requested.

2. Transference of control from one module to another must always be done via the supervisor. The transference of control can be called in coroutine mode (no return) or in subroutine mode (with return to the point immediately after the call when the subroutine execution is completed). A macroinstruction (pseudo supervisor call) was built for this purpose. Its argument must be the number of the requested module, positive for a coroutine call, negative for a subroutine call.

3. The virtual interpreter supervisor has access to a table containing as many words as there are modules contained in the real memory allocated to the interpreter. Each word in the table is related to a real module address and contains either zero if that space is empty or the module number, seizing it if it is not, with an indication (the sign) about the state of the module in question: positive if it is no longer needed in real storage for the moment, negative if it is still needed.

4. Whenever a pseudo supervisor call is issued, the supervisor searches the module table for the appearance of the requested module number. In the case where it is found, it is given

control immediately. In the opposite case, a new search is done to seek either an empty space or one occupied by a program no longer needed. Then the requested module is read from the interpreter disk file onto the assigned space, the module table is updated, and the program is given control.

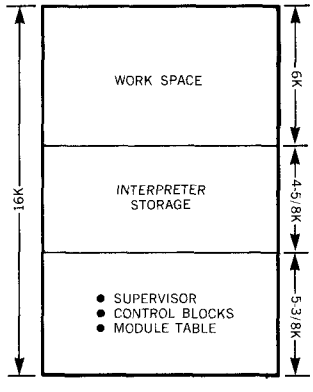
5. Parameters are passed from one module to another by means of the registers of the System/7. Their values are saved by the supervisor and restored in such a way that when control is given to a coroutine or subroutine, the registers have the same values as before the call, and the same is true for the return of a subroutine. No matter how many intermediate nested calls have appeared during the subroutine execution, when control is resumed, the registers contain the same values as before the call. Figure 3 presents the structure of the interpreter.

Implementation and performance

A full APL interpreter has been built, including the primitive functions and operators added to the APLSV system. It also includes a set of APL system functions and variables,[4] and a work-space management system consisting of a work-space library containing a directory where information is stored concerning the degree of library occupation, users, and public library numbers, and the work spaces related to each. The rest of the file contains the actual work spaces. The APL command language has been implemented to manage the work-space library and the active work space.

Our APL/7 prototype runs on a System/7 provided with only 16K words of memory. This memory is split up between the different parts of the system as Figure 4 indicates. The supervisor is resident in memory, controlling time-sharing and virtual interpreter allocation.

Figure 4 Division of memory



Control blocks are provided for each APL terminal to be used for disk file and sensor-based operations. They consist of a number of words containing information about the I/O processes. The modular interpreter table contains the number of those modules present in memory at any time (see above). The total size of this section of the system (in our prototype, 5.373K) depends on the number of terminals connected (up to four) and I/O operations allowed (number of control blocks) as well as on real interpreter size (size of the module table).

The number of modules simultaneously present in memory is the number of times 128 is contained in the real interpreter memory size. The number is also the size of the module table in

words, this number influencing the relation of APL execution time to supervising time, and therefore the speed of the system.

A trade-off can be made between the space allocated in main storage for the interpreter (response time) and the work-space size, within reasonable limits. For a general system, minimum space for the interpreter is established as 4K words. Work-space size should be larger than 6K words, smaller than 48K words, and a multiple of 3K words, because this is the length of a disk track.

A configuration program has been built that conversationally asks the operator of the system his desired configuration in terms of the number of terminals, number of I/O control blocks, and work-space size. The program then calculates the real interpreter size, and if it is within the limits, the new configuration is written up in the system loader.

The complete APL interpreter and auxiliary processor comprise 289 (128 words) modules. Our prototype allows the simultaneous presence in real memory of 37 modules. Thus, the real/virtual relation is equal to 1:7.8, much worse than the normal maximum recommended relation in standard virtual storage systems.

With this configuration, performance tests were run, separating APL execution time from supervising time. An average relation of about 1:30 between both times was observed, the results depending on the APL primitive function to be tested.

A second test panoply was run with a different APL/7 configuration. In this case, work-space size was reduced to only 3K words, supervisor and control blocks remaining the same. The number of modules in real memory was thus incremented to 61, corresponding to a real/virtual relation of 1:4.7. In this case, the average relation between APL execution time and supervising time was equal to about 1:3, APL execution time remaining the same as in the preceding case, as expected. Supervising time was thus shortened by one order of magnitude, meaning that disk I/O operations were drastically reduced.

No other performance tests were run on the supervising time response, because the small size of our System/7 did not allow the necessary flexibility.In any case, it is possible to theoretically calculate the effect of further increases of real interpreter size on the system response.

The maximum interpreter size is the one allowing permanent presence of all interpreter modules in real memory and equals about 36K words. In this case, supervising time would be re-

Table 1 Performance measurements

Primitive function		*CPU Time (microseconds) Argument*			
		Boolean	*Integer*	*Floating Point*	*Literal*
Dyadic	,	750	500	500	850
	+	650	700	2050	–
	-	900	600	1350	–
	×	800	1300	9100	–
	÷	1100	11800	18000	–
	⌊	800	900	1550	–
	⌈	650	1250	–	–
	∨	650	–	–	–
	=	550	750	1150	950
Monadic	+	550	100	100	–
	-	350	150	250	–
	?	–	7450	–	–
	×	450	300	1000	–
	⌈	650	550	1950	–
	~	350	–	–	–
	⌈	650	300	900	–
Other operations	4*4	–	5400	–	–
	4*30	–	55100	–	–
	4*.5	–	–	72100	–

duced to a minimum, being a fraction of APL execution time. Minimum System/7 memory size to attain this condition (with a work-space size of 6K words) would thus be equal to 48K words. The maximum System/7 memory size of 64K words would then allow an APL/7 system to operate at maximum speed with an available work space of 22K words.

The maximum recommended virtual storage relation (1:2.5), corresponding to 116 modules simultaneously in memory, would be possible for a minimum System/7 memory size of 28K words (with a work space of 6K words). Supervising time would then be one to two times as much as APL execution time. The maximum System/7 memory size of 64K words would allow in this case a reasonable speed for an APL system with an available work space of 42K words.

Table 1 depicts the performance measurements in terms of APL execution time for different primitive functions. The measurements were taken by executing the following APL functions:

```
    ∇ Z←G
[1]   Z←2 0⍴0
[2]   I←0
[3] E:Z←Z,F
[4]   →(6≠I←I+1)/E
[5]   Z←(+/Z)÷6
    ∇
```

```
    ∇ Z←F
[1]   K←0
[2]   T←⎕AI[2 3]
[3] L:A←X,Y
[4]   →(200≠K←K+1)/L
[5]   Z←0.001×⎕AI[2 3]-T
```

Function F computes the APL execution time and total time needed to execute 200 times the loop formed by instructions in lines 3 and 4, which include the function whose performance we are measuring (in the example, concatenation). Different primitive functions may be tested by changing line 3 in function F. Function G takes six different measurements by means of F and gives the average of the six measurements as its result. Times are obtained in seconds.

The loop execution time is obtained by substituting line 3 of function F by L:A ← X and executing G again. The difference between the preceding measurement and this time is the net time to execute the given function. Times given in the table have been obtained by dividing the resulting APL execution time by 200, and in most cases (where applicable), correspond to the operation of the tested primitive function on one or two vectors of two elements.

Times obtained are comparable to those of APL running on a System/360 Model 50. If we take into account the response degradation due to supervisor time, the APL/7 system response would be similar to that of APL on a System/360 Model 40.

Time-sharing

Although the development of a single terminal system was an option considered to be useful in itself, it was decided to implement a time-sharing system. As terminals for our prototype, we chose I/O typewriters (IBM Model 735, with slightly modified correspondence wiring) driven through a modified standard terminal control program[10] and accepting the connection of up to four terminals.

Again, the classical solution was too cumbersome for our simple System/7. With a standard time-slicing method, it would have been necessary not only to handle specially dedicated clocks, queues, etc. but also to solve other problems related to the interruption and resumption of the execution of modules while they are needed, with the corresponding need of saving internal variables, etc.

Keeping in mind the simplicity of the design and the efficiency of the system to reduce supervising time in order to gain calculation time, we took advantage of the fact that between two consecutive modules the control is always transferred to the supervisor, in the following way. By fixing the maximum number of consecutive calls to the supervisor allowed to each user, a time-sharing controller was constructed, which is called by the virtual interpreter supervisor whenever a user either: (a) exhausts the allowed number of calls or (b) goes into the wait state because of an I/O request to the terminal.

The time-sharing controller then goes into a cycle until it finds the first user requesting the interpreter. If this one and the currently active user are the same, nothing is done, and control is given to him. In the opposite case, the active work space is swapped out, and the new user work space is swapped in, whereupon he becomes the active user. Reenterability of the interpreter was, of course, required.

Disk file management and sensor-based input/output

The I/O operations with a System/7 can be classified into two groups: disk file operations and analog-digital I/O operations. Two approaches were considered:

1. To extend the APL language with special I/O (quad) symbols.
2. To include the shared variable concept (or a subset of it) and to build an auxiliary processor to handle the I/O operations.

Approach 2 was chosen since it does not require modification of the APL language, and an auxiliary processor, called TSIO7, was designed. This auxiliary processor is contained in a disk and is loaded into main memory through the same paging algorithm used to manage the interpreter. The same module slots in real memory are shared by both the interpreter and the auxiliary processor.

shared variables

Two different processors can communicate, and thereby made to cooperate, if they share one or more variables. Such *shared variables* constitute an interface between the processors through which information may be passed to be used by each for its own purposes. In particular, variables may be shared between an APL work space and some other processor that is part of the overall APL system to achieve a variety of effects including the control and utilization of peripheral devices (e.g., magnetic disk storage units).

At any instant, a shared variable has only one value: the last assigned to it by any one of its owners. Characteristically, how-

ever, a processor using a shared variable will find its value different from what it might have set earlier. For example, let X be a variable shared by processors A and B:

```
Processor A      Processor B
   X←2
                      2×X
                 4
                      X←3
   2×X
6
```

Sharing can be retracted by the monadic function $\square SVR$ applied to the variable name. After this function is executed, the variable stops being shared. Retraction of sharing is automatic upon completion of the function in which it was defined if the connection to the computer is interrupted, if the user signs off or loads a new work space, if the variable is erased, or if it is a local variable.

disk file operations

The auxiliary processor, TSIO7, allows the APL/7 user access to disk files, in this case, IBM 5022 files. Six different operations are allowed: sequential read/write, indexed read/write, delete, and rename. The system operator has access to every file. Other users may read from every file but can write, delete, and rename only their own files, which must be defined as members of a directorized data set. The directorized data sets must be defined outside of the APL/7 system, and the System/7 operator may do it by means of standard utilities. The user may create new members to his directorized file.

auxiliary processor sign on

Any user may sign on to the auxiliary processor by sharing any number of variables with it. A variable is offered to the auxiliary processor by means of the following sentence:

```
99 □SVO 'name of the variable'.
```

The result of this expression is a 1 if the variable has been correctly offered, and a 0 in the opposite case. A variable is retracted by using the same procedure as in APLSV.

auxiliary processor commands

When a variable has just been shared, the auxiliary processor expects the variable to be assigned a command. The commands are character strings that explain to the auxiliary processor the disk file operation desired as well as the necessary parameters to define where and how this operation must be done. A command may have any one of the following forms:

```
SR   vol,dsn,mn{,code}
IR   vol,dsn,mn,DAT←identifier{,code}
IW   vol,dsn,mn,DAT←identifier{,code}
```

SW vol,dsn,mn[,NEW,[FX,lrecl / TX] / ,OLD]{,code}, S←*n*

DL vol,dsn,mn

RN vol,dsn,mn,nn

The commands consist of the following parts:

1. The operation code:

   SR: sequential file reading (records are read in the order they are encountered).
   SW: sequential file writing (records are sequentially written on the disk).
   IR: indexed file reading (records are read in any order the user specifies).
   IW: indexed file reading and/or writing (records are either read or written in the order the user specifies).
   DL: file deletion.
   RN: file renaming.

Immediately after the operation code, at least a blank is expected.

2. The complete file name, consisting of the following subparts, separated by commas:

   vol: disk name.
   dsn: file name.
   mn: member name if a member must be accessed. If this is not the case, and more parameters must be given in the command, the absence of the member name must be indicated by writing two consecutive commas.
   nn: in case of renaming, the new name is given at the end of the command.

The four subparts stated above are positional. Their order cannot be changed.

3. The file parameters, separated by commas. The following ones are accepted:

   Disposition (NEW, OLD). If this parameter is not given, OLD is assumed, meaning that the accessed file already exists. NEW must be stated to create a new member (SW) and is only accepted with this operation code. (See Example 1 in Figure 5.)
   Record organization (FX, TX). If it is not given, the old organization is assumed. In case the member is being created, this

Figure 5 Example 1

```
      CREATING A MEMBER
      99 ⎕SVO'X'
1
      X←'SW APL7,APL70166,MYMB,NEW,FX,128,S←10'
      X
0           (RETURN CODE OF THE OPERATION)
      A MEMBER, MYMB, HAS BEEN CREATED IN THE
      DIRECTORY APL70166 (BY THE USER NUMBER 166)
      IN THE DISK APL7, WITH 10 FIXED LENGTH RECORDS
      OF 128 WORDS. THE FILE IS OPENED TO WRITE
      REGISTERS SEQUENTIALLY WITH APL CODE.

      X←2 2⍴⍳4
      X
0
      A FIRST RECORD HAS BEEN WRITTEN, CONTAINING
      AN APL 2×2 MATRIX.
      X←'ABCDMNPQ'
      X
0
      A SECOND RECORD HAS BEEN WRITTEN.

      X←''
      THE FILE IS CLOSED, THE VARIABLE RETURNS TO
      THE COMMAND STATE.
```

parameter is compulsory. TX corresponds to a compressed EBCDIC text organization, and FX to a fixed length record organization.

Record length: lrecl. It must be a multiple of 128, and less than 32768. It defines the number of words in a fixed length record.

Code: The form in which the values in each record are codified. The following codes are accepted:

A: The APL object is read or written such as it is, including dimension, type, and rank headings.
B: An $N \times 16$ binary matrix is transferred, N being the number of words in the record.
C: APL character vector without heading.
D: APL floating point vector without heading.
E: EBCDIC character vector.
F: APL integer vector without heading.
H: $N \times 4$ matrix, where N is the number of words in the record, which are transferred in hexadecimal form.

Figure 6 Example 2

```
  READING A DATA SET
     X←'SR APL7,APL70166,MYMB,A'
     X
0
     THE ABOVE CREATED MEMBER IS OPENED FOR APL
     CODE READING.
     X
1 2
3 4
     X
ABCDMNPQ
     X

     (AN EMPTY VECTOR)
     THE FILE IS CLOSED.
```

Space: This parameter must always be given in the case of an SW operation. It is stated as S ← *n*, where *n* is the number of sectors to be assigned to the member (the size of the file).
Data variable: DAT ← name of the variable. In the case of IR/IW operations (see Example 3 in Figure 7), two variables must be shared. One will be the control variable as explained above which now is first assigned the opening command and afterwards the indexing subcommands. Each indexing subcommand consists of a vector of two integers: the first one is 0 or 1 depending on the kind of operation (read or write) to be performed on the contents of the record indicated by the second element of the subcommand.

If it is a read operation, the contents of the indicated record are read into the data variable. If it is a write operation, the contents of the data variable will be written into the record. The data variable should have been shared before the command is assigned to the control variable.

All the preceding parameters are not positional, but keyword controlled, and they may be given in any desired order. Examples of various operations are shown in Figures 5, 6, and 7.

sensor-based I/O

The problem of controlling analog-digital sensors presents the following properties:

- The theoretical solution of the problem in this context commonly gives rise to matrical equations; a sophisticated mathematical support is thus needed.

Figure 7 Example 3

```
  INDEXED ACCESS TO A DATA SET.
      99⎕SVO'Y'
1
      X←'IW APL7,APL70166,MYMB,A,DAT←Y'
      X
0     (THE FILE IS OPEN FOR INDEXED ACCESS)
      X←0 1       (SUBCOMMAND MEANING 0: READ AND
                  1: RECORD NUMBER 1)
      X
0                 (THE OPERATION IS SUCCESSFUL)
      Y           (RECORD NUMBER 1 HAS BEEN READ
                  INTO THE DATA VARIABLE Y)
ABCDMNPQ

      X←0 0       (READ RECORD 0)
      X
0
      Y           (NOW RECORD NUMBER 0 HAS BEEN
                  READ INTO Y)
1 2
3 4
      Y←1 2 3 4
      X←1 0       (THE VALUE OF Y IS WRITTEN IN
                  RECORD 0)
      X
0
      X←0 0       (THE VALUE OF RECORD 0 IS READ
                  INTO Y)
      X
0
      Y
1 2 3 4
      Y←0
      X←1 6       (THE CURRENT VALUE OF Y IS
                  WRITTEN IN RECORD 6)
      X
0
```

- Interactive systems are convenient, allowing the user to respond immediately to any eventuality. Therefore, a more direct control on the process under consideration is feasible.
- Response time is a must only for concrete applications.

The sensor-based operations can be divided into the following classes:

- Analog input reading (AI).
- Analog output writing (AO).

- Digital input reading (DI).
- Digital output writing (DO).

For sensor-based applications[11] on System/7, the I/O module is the basic building block. Each module is self-contained, that is, it houses all the electrical/mechanical components necessary to serve the I/O operations (analog input, analog output, digital input, digital output) and to connect the System/7 internal interface. An I/O module may be physically placed in any module position of enclosure.

An I/O point is a two-wire connection. The points of the same type (analog/digital I/O), which are placed in the same module, are reunited in groups that can be simultaneously accessed.

analog input

The following is an example of a command for an analog input read:

```
AI MOD=5,POINT=14,INTV=200,NVAL=10
```

The command indicates that we want to read at analog input point number 14, located in the System/7 module 5, 10 consecutive values separated by an interval of 200 microseconds. In the case where the interval is not specified, a standard value is applied. The result of the operation is an APL vector whose dimension is equal to NVAL and whose values are the desired readings in millivolts.

analog output

A command example for an analog output follows:

```
AO MOD=5,POINT=1,INTV=200
```

This example indicates the wish to output consecutively through the analog output point number 1 in the System/7 module 5, the elements of an APL vector, with an interval of 200 microseconds.

digital input

A command example for digital input reading without process interrupt:

```
DI MOD=5,GRP=3,BITS=5 7 6 8,NVAL=10,INTV=200
```

This example commands that the bits, numbers 5, 7, 6, and 8 of digital input group number 3 in System/7 module number 5, must be read consecutively 10 times, with an interval of 200 microseconds. The result will be given as an APL binary matrix where the rows represent the successive readings and the four columns correspond to the values of the specified bits in the order indicated.

A command example for digital input reading with process interrupt:

```
DI MOD=5,GRP=0,BITS=0,REF≠0
```

It means that as soon as bit 0 in the digital input group number 0 in System/7 module 5 has a value of 1 (REF ≠ 0), an interruption must be produced. The REF parameter is a reference to which the digital input is compared. As soon as the given condition is satisfied, an interruption is triggered.

To solve this case, two new system variables, $\square EV$ and $\square XV$ have been added to the system. $\square XV$ is to be assigned an executable character matrix prior to the starting of the interruption process. When the interruption is served, $\square EV$ will be assigned a code, which identifies the point where the interruption has been detected.

Let Z be a shared variable that has been assigned a digital input command with process interrupt:

```
Z←'DI location, bits, reference, interrupt control'
```

The next step is to ask if the command has been accepted. This question requires an access to Z, an access that enables the interruption.

When the interruption arrives, the current execution is halted, and the interruption is served in another level, that is to say, $\square EV$ is assigned a code that identifies the point where the interruption has been produced. The system returns to its previous level of interruption and execution is resumed. Every time a user accesses the virtual storage supervisor, $\square EV$ is scanned. If an interruption has been produced, the APL function being executed is suspended and $\square XV$ is executed. If $\square XV$ is an empty array, the terminal is opened. The effect in this case is equivalent to pressing the attention key. If several interruptions are to be successively input with the same command, the variable Z has to be accessed after every interruption is served to enable the next interruption.

digital output

The following is a command example for digital output writing:

```
DO MOD=5,GRP=3,BITS=5 7 6 8,INTV=200
```

It means that we want to write the elements of an APL binary matrix (it must be a four-column matrix, each column for every element of BITS), and every 200 microseconds send the successive rows of the matrix through the bits 5, 7, 6, and 8 respectively) of digital output group number 3, located in System/7 module number 5.

application example

As an example, we present the following case to be solved: Every 10 milliseconds, digital input group number 2, in System/7 module 5, must be read until the reading has taken place 100 times. Then the arithmetic average of the read values must

be computed, and the output sent through analog output number 0 in System/7 module number 3. Each input reading will be considered as a positive integer with 16 binary digits. The process must begin when bit 0 in the digital input group number 0 in module 5 is set.

The following APL function would solve this example:

```
    ∇ DIRW; A
[1] ⍝ LET X,Y,Z BE VARIABLES SHARED WITH THE AUXILIARY PROCESSOR
[2] ⎕XV ← 3 11⍴ 'A←0.05×2⊥X Y←(+/A)÷⍴A →L       '
[3] X ← 'DI MOD=5,GRP=2,NVAL=100,INTV=10000'
[4] → (X≠0)/E
[5] Y← 'AO MOD=3,PT=0'
[6] → (Y≠0)/E
[7] Z ← 'DI MOD=5,GRP=0,BITS=0,REF≠0'
[8] L: → (Z≠0)/E
[9] →L+1
[10] E: 'ERROR'
    ∇
```

Line 3 opens the digital input group 2 in module 5, and line 5 opens the analog output point 0 in module 3. Line 7 means that as soon as bit 0 in the digital input group 0 in module 5 has a value of 1 (REF ≠ 0), an interruption must be produced. Line 8 enables the interruption.

Line 9 is a closed loop. The system is waiting for the interruption to come. When that happens, ⎕*EV* is set, and at the next call to the supervisor, ⎕*XV* (whose value had been set in line 2) is executed. Now, the 16 bits of the digital input group 2 are read 100 times, with an interval of 10 milliseconds; they are coded into decimal base, multiplied by a scalar factor 0.05, and assigned to a local variable *A*. In the second line of ⎕*XV*, the arithmetic average of *A* is assigned to *Y*, and the output goes through the analog output.

The last line in ⎕*XV* is a return to *L*, where a new access to *Z* is made to enable the next interruption, and the whole process will be repeated again.

Summary

The implementation of an APL system on a small computer (System/7) presented the following challenges:

1. The limited set of machine language instructions, where bit handling, floating point, multiplication, and division operations do not appear, made it necessary to simulate most of them by means of software. In particular, a special decimal floating point representation was devised.

2. The small size of the main storage was a problem that we did not want to solve by reducing the power of the language. It was thus necessary to simulate a virtual memory allocation processor controlling a modular interpreter.
3. The implementation of a time-sharing system capable of supporting several terminals was solved in a nonstandard fashion, taking advantage of the fact that the interpreter continuously calls the virtual memory supervisor. This fact gives rise to a time-sharing system with variable time-slicing.
4. Disk file management and sensor-based I/O operations were problems that we decided to solve by making use of the shared variable concept.

In solving the problems thus presented, the experimental APL/7 system[12] was built and is fully operational, including the sensor-based I/O operations. The extension of the system to sensor management makes it possible to analyze the suitability of APL for process control applications.

CITED REFERENCES AND FOOTNOTES

1. *IBM System/7, Functional Characteristics*, No. GA34-0003-3, IBM Corporation, General Systems Division, Atlanta, Georgia (1972).
2. An RPQ to connect IBM 3340 disks to the System/7 has recently been announced.
3. D. L. Raimondi, H. M. Gladney, G. Hochweller, R. W. Martin, and L. L. Spencer, "LABS/7—a distributed real-time operating system," *IBM Systems Journal* **15**, No. 1, 81–101 (1976).
4. *APL Language*, No. GC26-3847-0, IBM Corporation, Data Processing Division, White Plains, New York (1975).
5. *APL/1130 Primer*, No. C20-1697-1, IBM Corporation, Data Processing Division, White Plains, New York (1969).
6. H. Peñafiel, "Diseño de un sistema de APL para la computadora IBM S/3", No. CCAL-74-8, IBM Mexico Scientific Center, Mexico City, Mexico (1974).
7. The IBM 5100 APL machine is a different approach, contemporary to our work.
8. D. Gries, *Compiler Construction for Digital Computers*, John Wiley and Sons, New York (1971).
9. Because the system is an experimental development, it is not available for distribution outside of IBM.
10. *System/7 Input-Output Support for the IBM Selectric I/O Writer*, No. SB21-0660-0, IBM Corporation, General Systems Division, Atlanta, Georgia (1972).
11. *IBM System/7 System Summary*, No. GA34-0002, pp. 2.5–2.7, IBM Corporation, General System Division, Atlanta, Georgia.
12. M. Alfonseca and M. L. Tavera, "The APL/7 System," No. SCR-04.74, IBM Madrid Scientific Center, Madrid, Spain (1974).

Appendix

In the terminal sample session given below, the following APL features, as present in APL/7, are shown: function definition,

statement syntax, definition of shared variables, and disk file management. They are explained by means of APL comments (⍝).

```
)166

    APL\7
CLEAR WS

     ⍝ THE FOLLOWING FUNCTION DECODES THE CONTENTS OF
     ⍝ A MEMBER OF AN ASSEMBLER SOURCE LANGUAGE LIBRARY
     ⍝ CONTAINED IN A DISK FILE, AND LISTS THE MEMBER
     ⍝ AT THE TERMINAL.
     ∇DECODE X;A;K
[1]  ⍝ A COUNTER IS SET TO NUMBER THE STATEMENTS
[2]  K←20
[3]  ⍝ THE SHARED VARIABLE CTL IS ASSIGNED A COMMAND TO
[4]  ⍝ OPEN A TEXT ORGANIZED MEMBER WHOSE NAME AND LOCATION
[5]  ⍝ IS GIVEN IN ARGUMENT X, TO BE SEQUENTIALLY READ
[6]  ⍝ WITH EBCDIC CODE TRANSLATION.
[7]  CTL←'SR ',X,',E,TX'
[8]  ⍝ SUCCESS OF THE REQUESTED OPERATION IS CHECKED.
[9]  CHECK CTL
[10] ⍝ IF WE READ AN EMPTY RECORD, WE ARE DONE.
[11] L:→0 IF 0=⍴A←CTL
[12] ⍝ THE READ RECORD IS NUMBERED AND WRITTEN OUT.
[13] (¯4↑'0000',⍕K),' ',A
[14] ⍝ THE COUNTER IS INCREMENTED.
[15] K←K+20
[16] ⍝ LOOP
[17] →L
[18] V
     ⍝ THE FUNCTION TO CHECK SUCCESSFUL TERMINATION OF
     ⍝ DISK FILE OPERATIONS:

      ∇CHECK X
[1]   ⍝ SUCCESS IS INDICATED BY A ZERO VALUED RETURN CODE
[2]   →0 IF X=0
[3]   ⍝ MSG IS A LITERAL APL MATRIX CONTAINING THE POSSIBLE
[4]   ⍝ ERROR MESSAGES
[5]   MSG[X;]
[6]   ⍝ AFTER WRITING THE ERROR MESSAGE, WE QUIT EXECUTION
[7]   →
[8]   ∇

      ⍝ THE FUNCTION TO SIMULATE THE IF STATEMENT
      ∇Z←X IF Y
[1]   Z←Y/X∇

      ⍝ AFTER DEFINING THE FUNCTIONS, WE BEGIN IMMEDIATE EXECUTION.
      ⍝ THE VARIABLE CTL IS SHARED WITH TSIO7
```

```
      99 □SVO'CTL'
1
      ⍝ THE FUNCTION DECODE IS INVOKED TO LIST THE CONTENTS
      ⍝ OF MEMBER M1 IN FILE SLIB IN VOLUME APL7.
      DECODE'APL7,SLIB,M1'
0020  USING MEMOR,MR  M O D U L O 1
0040 IAM LI XEC,0
0060  N INF
0080  SZ
0100  B XC
0120  L QSB
0140  N INF
0160  SNZ
0180  B LL
0200  LI 1,2
0220  PSVC 6
  . . . .
      ⍝ WHEN THE LISTING IS FINISHED, WE SAVE THE WORK SPACE
      )SAVE LIST
      ⍝ HAVING FINISHED THE SESSION, WE SIGN OFF THE TERMINAL
      )OFF
TERMINAL DISCONNECTED
```

Reprint Order No. G321-5043.