An APL system for the IBM Personal Computer

by M. L. Tavera
M. Alfonseca
J. Rojas

This paper discusses the design and building of an APL interpreter for the IBM Personal Computer. Discussed is the writing of the interpreter itself, which required the use of an intermediate language designed by the authors. This machine-independent language also made possible the development of APL interpreters for two other systems—System/370 and Series/1. The particularizing of the interpreter required a compiler, which in the case of the Personal Computer produced Intel 8088 and 8087 assembly language code. The matching of the APL interpreter to the operating system (DOS) required an APL supervisor, which is also discussed in this paper. The provision of the APL character set presented problems, the solutions of which are also presented. Other topics discussed are the display, the keyboard, and the session manager.

The programming language APL is a powerful general-purpose interactive language that handles scalars and n-dimensional arrays of numeric and literal data in a very flexible way via a large set of primitive functions. These are used to form APL sentences, which can be grouped in defined functions that make it possible to write very complex applications in APL. The APL language is also a mathematical notation that can be used as such without involving a computer. To execute APL programs, it is necessary to write both an interpreter that runs in a particular machine and an APL supervisor consisting of a set of routines that interface the interpreter with the operating system of the machine.

The APL language itself is hardware independent. Successive IBM implementations of the language have adhered to the APL language standard, so that applications that use only the APL notation are portable between different APL systems. However, that does not mean that the APL systems neglect machine hardware. To the contrary, the standard makes provision for a tool to make machine hardware accessible to APL without modifying the language. This tool, called the Shared Variable Processor, is an interface between the APL interpreter and the outside world. If a piece of hardware is to be related to the APL system, an auxiliary processor has to be written. The communication between the APL system and the auxiliary processors is established via the Shared Variable Processor, using shared variables that can be set and referenced by both the APL system and the auxiliary processor. Usually an IBM APL system includes several auxiliary processors to perform the most useful I/O operations, but most of them, including the one discussed in this paper, give the users the information and the tools needed to build their own auxiliary processors.

Having an APL interpreter in a machine increases the system power. The building of an APL interpreter, however, is a long and difficult task because it has to be written in a low-level language, usually assembly language. After an APL interpreter has been written, obtaining a second interpreter presents little difficulty unless the machine is special in some way.

Although the algorithms do not have to be redesigned, writing a second interpreter is time-consuming because everything has to be rewritten in a different assembly language. The only part that has to be specially created in every case is the APL supervisor.

We designed the ad hoc language that we called intermediate language (IL).

In 1976, these considerations convinced us that we needed a method to speed up the construction of an APL interpreter after the first one had been written. This observation was based on our experience in developing APL/7, an APL interpreter for the IBM System/7.[1]

To solve the problem, we developed[2] the following three-step process:

1. We wrote an APL interpreter (IAPL) in an intermediate machine-independent language (IL).
2. We write a compiler to translate IL programs into the assembly language of a given machine, e.g., the IBM System/370.
3. We compile the IL interpreter only once per machine. The generated code is an APL interpreter written in the chosen language, e.g., that of System/370.

To obtain the first APL interpreter using this technique, we used all three steps. To obtain other APL interpreters for different machines, we now rewrite only the code generator of the compiler in step 2 and repeat step 3.

In any case, the final product we obtain is an APL interpreter written in the assembly language of the chosen machine. To make it directly executable in that machine, we have to assemble the APL interpreter and link it with the corresponding APL supervisor.

We want to stress the fact that we have developed the procedure just mentioned to speed up the construction of APL interpreters. The user of the APL interpreter thus generated is not aware of that procedure.

To choose the most convenient intermediate language, we studied the possibilities that were available in the mid-1970s, such as systems programming languages, macrolanguages, and high-level languages. Because none of these languages fully suited our needs, we designed the ad hoc language that we called intermediate language (IL).

Using this methodology, we developed three APL interpreters: (1) for the IBM System/370, which we used as a test case; (2) for the IBM Series/1; and (3) for the IBM Personal Computer. The IBM Personal Computer APL is an IBM product that was announced in the United States and Europe in June 1983.

The IBM Personal Computer

The IBM Personal Computer (PC) is a powerful small computer that offers a wide variety of options to give the user the ability to tailor his system to meet his present needs while providing growth potential for the future.[3] The PC has two major elements, a system unit and a keyboard.

The system unit contains an Intel 8088 microprocessor, read-only memory (ROM), read/write memory (RAM), power supply, audio speaker, and system expansion slots for the attachment of options. The PC also contains the necessary hardware to support the Intel 8087 math coprocessor. Input to the system unit is by way of an 83-key keyboard that also includes a numeric keypad and ten function keys.

The Intel 8087 math coprocessor[4] is an adjunct to the 8088 processor and performs arithmetic and comparison operations on a variety of numeric data types. The math coprocessor also executes numerous built-in transcendental functions (e.g., tangent and logarithm functions). The 8087 expands the register and instruction set of the host 8088 and adds several new data types as well. The programmer does not perceive the 8087 as a separate processor. Instead, the computational capabilities of the 8088 appear to be greatly expanded. The 8087 supports the following data types: (1) three different kinds of integers (16, 32, and 64 bits long); (2) a packed decimal (80 bits long); and (3) three different kinds of floating point data (32, 64, and 80 bits long). Its main instructions include several kinds of operations, such as data transfer, arithmetic, comparison (compare, examine, test), transcendental (tangent, arctangent, exponentiation, logarithm), and processor control.

In addition, a variety of options are offered, including one or two diskette drives with adapter, which together can be housed inside the system unit, a fixed disk (in the IBM Personal Computer XT or in the expansion unit), an IBM monochrome display, an IBM color display, an IBM 80-character-per-second graphics printer, two display adapters, storage increments to 640K bytes, an asynchronous communications adapter, a printer adapter, a game control adapter, and a prototype card.

There is a monochrome display and printer adapter that supports the monochrome display and the graphics printer. The monochrome display is supported in text mode only. It has a ROM chip with the character set definition programmed in it. Every character has a resolution of 14 by 9 picture elements (pixels).

Both BIOS and DOS interrupts can be dynamically called from the user program.

The graphics adapter supports the IBM color graphics display in both text and graphics mode. In text mode, the definition of the character set is stored in a ROM chip. Every character is given a resolution of 8 by 8 pixels. In graphics mode, the definition of the characters with an ASCII code smaller than 128 is stored in the ROM basic input/output system (BIOS) described below. The rest of the characters are not defined. The BIOS provides a pointer to allow the user to point to the zone in read/write memory where he has previously defined those characters.

The software is also very powerful. The computer comes with a ROM-programmed basic input/output system (BIOS) that handles the system I/O at a very low level by means of interrupts. Since the BIOS is programmed in ROM, it cannot be changed. However, the interrupt vector is in read/write storage. Therefore, if a particular BIOS interrupt does not suit the programmer's needs, its address can be changed to point to a user-defined routine. The BIOS also bootstraps the system.

The most used software operating system is the disk operating system (DOS),[5] which provides a set of commands that handle the system interactively. It also has a debugger, a line editor, and a set of interrupts that support I/O at a higher level than BIOS. Both BIOS and DOS interrupts can be dynamically called from the user program.

The IBM Personal Computer APL system

Working with a fully available machine is very different from working with a machine that is still under development. Fortunately, a development system was not needed, because the machine was provided from the beginning with an editor, assembler, linker, debugger, and operating system. The system could also work in a stand-alone mode. The following is a summary of the procedure we followed in studying the IBM PC hardware and software configuration needed.[6]

- We compiled the IAPL interpreter into 8088 assembly language code. Because this operation took place in the IBM System/370 where the source code was stored, we wrote a compiler for translating IL code to 8088 assembly language. Also, because the IAPL makes heavy use of floating point operations, we needed some kind of floating point management, a fact that became a point of issue.
- After we had compiled the IAPL, the object code had to be downloaded from the System/370 to the PC. Thus, a communication adapter and the software to support it were needed. This also made it possible to write auxiliary processors to communicate between PC APL and VM/370 later on.
- The APL supervisor, together with the auxiliary processors—all of them fully machine dependent—had to be written. Therefore, some macros or interrupts were needed to provide an interface between the APL interpreter and the operating system, as well as to manage the peripherals.
- Finally, everything had to be assembled and linked, which made necessary an assembler and a linker.
- To fit the full APL interpreter, the APL supervisor, a sizable workspace, and the operating system into the IBM PC, we needed a minimum of 128K bytes of read/write memory.
- Because files of several types are a very important part of an APL system, a fast auxiliary storage device was needed to load the APL-executable module and to manipulate user-created files, APL workspaces, transfer files, etc.

- The APL language has a special character set not supported by the IBM PC. Thus, we needed a graphics display where the special APL character set could be generated.
- In the case of the printer, we also had the problem of the APL character set representation, which required the use of a graphics printer.

The compiler

The compiler for translating IL code was written in APL. Compiler speed was not an important factor because we executed it only once. We had already written most of the parts of the compiler (the lexical and the syntax analyzers). Only the code generator remained to be written again. The object code was to be that of the 8088. This objective, although simply stated, introduced two issues that affected the design of the whole interpreter: floating point management and the 8088 address space. Floating point management affected mainly the compiler design, and we discuss that in this section. Address space, which affected both the compiler and the supervisor, is discussed later.

Floating point management. The intermediate language (IL), in which the APL interpreter (IAPL) is written, handles the following four data types: boolean (one bit per element), character (one byte per element), integer (16 or 32 bits per element), and floating point (eight bytes per element). The APL system performs data-type conversion automatically, whenever possible, to minimize storage space.

Floating point operations form a very important part of an APL interpreter. These operations include addition, subtraction, multiplication, division, modulus, absolute value, change of sign, integer part, conversion to and from integer, conversion to and from boolean, and six different types of comparison relations. In addition, several of the APL language primitive functions nearly always apply to floating point data.

From the very beginning, our plan was twofold. First, we would translate the IL floating point primitives into 8087 instructions directly, using the compiler. Second, we would design the APL primitives just mentioned from scratch using the full power of the 8087 primitives. We would not compile the corresponding IAPL modules because they use primitives of much lower level.

In general, the PC APL floating point management could be designed in either of two ways. One possibility, that of using the 8087 math coprocessor, offered the advantage that the final performance of the APL system would be greatly improved. However, at the time we were doing our planning no 8087 instruction assembler was available to us. The other possibility involved emulating the IL floating point instructions by means of 8088 instructions. This involved three disadvantages. The design of the IL floating point instruction emulation would require between six and eight weeks. The code would require about 6K bytes of storage. There would be a loss in performance, mainly in those applications such as mathematical calculations that use many APL floating point primitives.

It was easy to emulate the assembly of these instructions by means of the 8088 macroassembler macro language.

We chose the first solution because it was the faster and the more efficient of the two alternative methods. It was possible to design the floating point part of the compiler in such a way that—without loss of efficiency—the set of 8087 instructions it used was very small. Therefore, it was easy to emulate the assembly of these instructions by means of the 8088 macroassembler macro language. Of the seven different data types supported by the 8087 we needed only the 16-bit integer and the 64-bit and 80-bit floating point data types. Also, the assembly and debugging of the specially designed transcendental APL functions could be postponed because the rest of the system did not depend on them.

During the design stage we overcame two problems. Neither the assembler nor the debugger supported 8087 instructions. We solved the first problem by writing a set of macros for the 8088 macro assembler.[7] This solution made the assembly of the modules very slow. It was also a source of errors, because we had to debug the 8087 macros at the same time we debugged the code. The difficulty with the debug-

ger was that it did not support 8087 instructions. Each time we wanted to read the 8087 registers, with no computer help we had to insert a series of 8087 instructions to dump its stack on memory, read from it, and interpret the result by hand.

The APL supervisor

The APL supervisor is the machine-dependent interface between the APL interpreter and the operating system DOS that manages connection, initialization, and disconnection. It also handles the floating point interrupts (underflow, overflow, etc.), system error recovery, and execution interrupt. Other functions of the APL supervisor are I/O (from the display, keyboard, printer, etc.) and file management. These functions have been implemented using BIOS and DOS interrupts. When designing those routines we were faced with several issues that we now discuss.

APL workspace size. The size of the PC APL workspace was greatly affected by the way the 8088 microprocessor handles its address space.[8] The 8088 is a 16-bit addressing microprocessor; it can directly access only a maximum of 64K bytes of memory, called a *segment*. To overcome this condition, the processor was provided with segment registers that allow it to access up to one million bytes.

Every memory location can be considered to have a physical address and a logical address. A physical address is the 20-bit value that uniquely identifies every byte location and can range from hexadecimal 00000 to hexadecimal FFFFF. Every time memory is accessed, the physical address is used. To map physical addresses onto 16-bit registers, the microprocessor deals with logical addresses. A logical address consists of a segment value and an offset, where the former locates the first byte of the segment, and the latter shows the distance in bytes from the origin of the segment to the target location. The segment value is stored in a segment register, and the offset is stored in an index register.

To obtain the physical address from the logical address, the segment value is left-shifted 4 bits and then added to the offset. For example, if the segment value is hexadecimal 1234 and the offset is hexadecimal 5678, the physical address is the result of adding 12340 to 05678, which gives hexadecimal 179B8. All of this is performed automatically by the hardware microprocessor. When the segment and the index registers have been set, the physical address is fixed.

The problem arises when the value in the index register is incremented or decremented. In the incremented case, if an overflow occurs, the corresponding segment register is not incremented accordingly. Therefore, the addressing wraps around within the 64K-byte segment pointed to by the segment register.

To use all available memory in the machine, we devised an elastic workspace.

That is, when the index register value is hexadecimal FFFF (pointing to the last byte in the segment), and it is incremented by one, its next value is 0000. However, because the segment register value has not been modified accordingly, the target location of the physical address is the first byte of the same segment, instead of being the first byte of the next sequential segment. The same happens when the index register is decremented and an underflow occurs.

When we wrote our machine-independent APL interpreter, the only condition we imposed was that the machine memory addresses had to be sequential, e.g., that address hexadecimal 0FFFF precede hexadecimal 10000 and follow hexadecimal 0FFFE. That simple condition was not fulfilled by the IBM PC, if we were to consider workspaces greater than 64K bytes. Thus, any time we changed the value of an index register, we had to ask whether an overflow or underflow had occurred, so as to modify the segment register accordingly, if necessary. Because this would increase the size of the interpreter and decrease its performance, we decided to limit the workspace size to 64K bytes.

To use all available memory in the machine, we used an idea that we had devised for and implemented in the APL interpreter for the IBM Series/1. We called that idea the "elastic workspace."

The elastic workspace. The elastic workspace approach consists in the following. The workspace (WS) is split into two parts, the main WS (MWS), where all

computations take place, and an extension called the elastic WS (EWS), where objects not needed in the MWS may be stored. Whenever a computation needs space greater than that available in the MWS, objects not currently needed are sent to the EWS and are erased from the MWS. This increases the size of the free space in the MWS. If an object in the EWS is needed in the MWS, it is moved back again. The space in both workspaces is dynamically managed in a manner that is transparent to the user.

The elastic workspace does not decrease the performance.

Whenever the memory size allows it, we assign 64K bytes to the MWS and the rest of the available memory (with no limit of size) to the EWS. If the memory available is less than 64K bytes, it is all assigned to the MWS.

Every APL object has a pointer to its location in the MWS and another to its location in the EWS. Both pointers are 16 bits long, the MWS one because the MWS maximum size is 64K bytes, and the EWS one because the APL objects are stored in the EWS on 16-byte boundaries. Thus, the least significant hexadecimal digit in the EWS physical address is always 0, and only 16 bits are needed to store the four most significant hexadecimal digits.

The elastic workspace does not decrease the performance because it is placed in main memory, and the 8088 microprocessor is provided with very efficient string manipulation primitives.

Summing up, we have designed the workspaces to be (1) independent of machine configuration because PC APL can work with a minimum of 128K bytes of main memory, and (2) as large as possible, through the use of elastic workspace.

The APL character set. The APL language has its own set of 135 characters that can be divided into the following four main classes:

- Alphabetic, consisting of the Roman alphabet in uppercase and uppercase underlined form, delta, and delta underlined.
- Numeric, including the digits 0 through 9.
- Blank.
- Special APL characters, seventy in all.

Over one third of these characters are not included in the extended ASCII character set supported by the IBM PC. Because there are currently 256 characters—the maximum allowed—there is no room to add new ones. Therefore, some of the existing characters had to be replaced to accommodate the APL characters. Thus, the issue presented two different aspects: (1) How many characters should be replaced, and which characters should they be; and (2) How do we implement a replaced character set?

The ideal solution would be to do without replacing any of the characters. In this way, APL could support either the monochrome or the color display and the printer straightforwardly. This solution was not feasible because a large number of the APL characters do not resemble any of the existing characters. We decided, however, to reduce as much as possible the number of characters to be replaced by making compromises and using as APL characters those that are similar to them. For example, the logarithm is represented in APL as a star inside a circle. We used instead ASCII character 15, which is represented by a circle inside a star. Another important reduction occurred when we decided to use the lowercase alphabetic characters (a through z) instead of the standard APL uppercase underlined characters. Finally, we were left with thirty characters for which we had no compromise solution. We used the following conditions for these characters:

- The first 128 characters were not used, because when the graphics display is working in graphics mode, the descriptions of these characters are read from BIOS, which itself cannot be changed.
- The character sets of the five most widely used European languages, i.e., English, French, German, Italian, and Spanish, were not modified.
- The graphics line-drawing characters were also left intact.

The characters taken out were ASCII 144, 145, 146, 152, 157, 159, 172, 174, 175, 226 through 231, 234 through 237, 240, 241, 244, 245, 247, and 249 through 254. Most of these characters correspond to Greek characters and special mathematical symbols. Next we had to implement the replaced characters both in the display and in the printer.

In the display. We could implement the APL character set on the monochrome display, providing a hardware solution, or on the graphics display, providing a software solution.

At first sight, the easier solution seemed to be to use the graphics display in graphics mode. In this way the characters with ASCII code equal to or greater than 128 could be graphically represented by software. One difficulty with this was that the resolution of every character is 8 by 8 picture elements (pixels), which yields very poor resolution for the APL character set. Many of the APL characters are similar, and it is important to distinguish one from another easily. Also, the graphics display working in graphics mode does not show a cursor, an indispensable feature in an interactive system such as APL.

By using the monochrome display, we would have a much better resolution for the APL characters, i.e., 14 by 9 pixels each. This solution would involve hardware because the monochrome display and printer adapter that controls the monochrome display does not work in graphics mode. The Image Processing Department in the Madrid Scientific Center studied the logic diagrams of the monochrome display and printer adapter and found that the ROM where the character set is defined is not welded to the board, but rather inserted into a socket, and thus easy to remove. In place of the original ROM they used a compatible Erasable Programmable Read-Only Memory (EPROM) with the new character set definition programmed in it. Thus the APL characters could be displayed on the monochrome display. The EPROM was programmed in the Madrid Scientific Center. Parenthetically, we published this solution, and people found it so useful that we were asked to program EPROMS with other character sets. To mention but a few, we have programmed Hebrew, Eastern European languages, Scandinavian languages, and Portuguese.

Our original idea was to support both displays, each one with its own solution, so that people could work in APL with either display or with both. It was decided, however, that the APL PC should support only the APL character set in the graphics display. The monochrome display is supported with the original character set.

APL PC can work sequentially (without leaving APL) with both monitors in the same session. In the graphics monitor, it can display 80 and 40 characters per line. The APL system can switch modes either interactively, by pressing function keys, or dynamically, using the AP205 auxiliary processor.

In the printer. The APL character set could be included for the printer via either hardware or software. The hardware solution consisted in replacing the chip with the printer character set by another chip that included the APL character set. The software solution consisted in directly printing the characters that were originally in the ASCII set (text mode) and the remaining characters in graphics mode. We adopted the software solution as an easier one to implement. Experience and time have proved this to be the preferable solution.

Printing in the graphics mode has three drawbacks that affect the printing of APL programs. For one thing, it is very slow. Also, every time the printer switches from text mode to graphics mode, the printing head moves back to the beginning of the line. Not only the APL characters but also those characters with an ASCII code less than 32 have to be printed in graphics mode, because the printer takes the latter as control codes.

When we speak of a keyboard, we are referring to the software that translates the keystrokes to ASCII codes.

To speed up the printing of a line, we do the following:

- If the line contains only characters that can be printed either in text mode or in graphics mode, it is printed in one pass.
- If the line contains characters of both types, it is printed in two passes. The characters to be printed in graphics mode are printed from left to right, leaving blank gaps that will be filled in when the remaining characters are printed in the second pass from right to left.

The printer is handled from APL through the AP80 auxiliary processor in three different ways: (1) to print the contents of the currently active screen; (2) as a log of the session; and (3) to print selected APL objects dynamically.

The APL keyboard. Nearly every country has a keyboard of its own. When we speak of a keyboard, we do not mean simply the physical 83-key device. We are referring to the software that translates the keystrokes, either single or combined with shift keys, to ASCII codes. The differences between the keyboards lie in the fact that the character obtained when a particular key or key combination is depressed in every keyboard does not have to be the same. The IBM PC BIOS supports the United States keyboard, and DOS brings with it one keyboard program for each of the following European countries: France, Germany, Italy, Spain, and the United Kingdom. Each of these programs is called a national keyboard. The APL systems also have a special keyboard.

The PC APL system has been designed to support the APL keyboard program, the United States keyboard, and the national keyboard programs at the same time. A national keyboard is supported only if it has been previously loaded by the user. When the APL system is loaded, the APL keyboard becomes active. To switch to a national keyboard and back, the Ctrl-Bksp combination of keys has to be pressed. To pass from this keyboard to the United States keyboard and back, the standard combination of keys, Alt-Ctrl-F1 and Alt-Ctrl-F2, respectively, must be used. Finally, to return to the APL keyboard, the Ctrl-Bksp combination is pressed. If a national keyboard has not been loaded, the Ctrl-Bksp combination toggles between the APL and the United States keyboards.

The PC APL system supports every keyboard program that fulfills the following conditions. It must trap the BIOS interrupt number 9, and in byte number 3 in the keyboard program there must be a switch to indicate whether the national or the United States keyboard is active.

The session manager. We have designed a session manager in which every line of the screen can be active. The input lines have a maximum length of 79 characters. The cursor is moved by the keys in the numerical keyboard. A line becomes active when the cursor is on it. To execute the active line, only the Enter key has to be pressed. When a line is executed, it is copied at the bottom of the screen (unless it is the last one), and it is passed to the APL interpreter. The screen contents scroll up.

Both screens work in the same fashion. Both support the Insert and Delete keys. In the graphics display we had to simulate a blinking cursor, because the graphics adapter does not support a cursor when working in graphics mode. When the cursor is on a character, it is seen in reverse, black pixels on bright background.

The Insert key toggles between the insert and replace states. When the APL system is in the insert state and the monochrome display is active, the cursor fills up the space assigned to the character it is on. When the graphics display is active, the cursor blinks twice as fast as it does in the replace state.

The IAPL interpreter and the PC APL interpreter follow closely the VS APL interpreter.

Extensions to the APL interpreter

We designed the IAPL interpreter and, therefore, the PC APL interpreter, to follow closely the VS APL interpreter, which is a kind of standard for the language. In May 1982, we were asked to add new features to the PC APL system, some of which had been included in VS APL,[10] and other features belonging to the APL2 language.[11] We added these features in both the IAPL and the PC APL interpreters. These features are as follows:

- Execute alternate is used for error trapping and can be very useful in system design and emulation.
- Dyadic grades sort character arrays according to a collating sequence that the user provides.
- Picture format is a powerful formatter of figures.
- Execution of machine-code subroutines can be used to make APL perform at a very low machine level.
- A command termed)RESET cleans the execution stack in one operation.
- Ambivalent dyadic functions, which are dyadic defined functions that can be called either monadically or dyadically, are useful to define options by default. When the function is executed, the class of the left argument can be used to distinguish whether the function has been called with one or two arguments.

APL workspace data interchange

To make possible the interchange of data between two APL systems, a system function, the transfer form, and two commands—)OUT and)IN—have been added to the APL PC. The transfer form changes an APL object from its workspace internal form to the transfer form common to all APL systems.

An auxiliary processor is an interface between the APL system and external hardware or software.

The)OUT command takes an APL workspace or a subset of it, translates it to the transfer form, and stores the result in a DOS file. The name of the file is given by the user. Later, the whole file or a subset of the objects contained in the file can be copied into the APL workspace using the)IN command.

These commands are very useful because they allow data interchange between different APL systems. They also are a substitute for the)COPY command, which is very difficult to implement.

Auxiliary processors

An auxiliary processor (AP) is an interface between the APL system and external hardware or software. In our APL system, both the shared variable processor and the APs are modules that are different from the interpreter. Thus, they can be separately loaded. Also, the user can have as many APs as desired from which he can choose the ones that he is going to use during an APL session. The APs used during a working session are invoked when the APL system is loaded. A maximum of six APs can be used at the same time. At load time, the APL system is loaded in the low end of main memory. The APL supervisor then loads the chosen APs in the high end. The remaining memory is assigned to the workspace.

The PC APL system is distributed with six auxiliary processors:

- AP80 handles the printer.
- AP100 provides an interface to generate BIOS or DOS interrupts or function calls.
- AP205 is a full-screen auxiliary processor for both the monochrome and graphics displays.
- AP210 is used to handle DOS files.
- AP232 provides an interface for communications between the IBM Personal Computer and a host (e.g., IBM System/370).
- AP440 produces music through the attached loudspeaker.

Each auxiliary processor is accompanied by an application showing the use of its shared variables and commands.

Concluding remarks

We make one final observation on our experience in programming the PC APL system. The theoretically greatest part of the work was estimated at the outset to be the design and implementation of the APL interpreter. Actual experience proved this to be the least part, requiring only five months, because we used the machine-independent APL interpreter that we had devised. Thus, this procedure has fully proved its usefulness.

Acknowledgments

We have received help from many people at many IBM locations, and we would like to acknowledge each one individually. In doing so, however, we risk neglecting someone whose contributions we especially value. Therefore, let us simply say that we are fortunate to be able to honor you all in the system that embodies your thoughts and work.

Cited references

1. M. Alfonseca, M. L. Tavera, and R. Casajuana, "An APL interpreter and system for a small computer," *IBM Systems Journal* **16**, No. 1, 18–40 (1977).
2. M. Alfonseca and M. L. Tavera, "A machine-independent APL interpreter," *IBM Journal of Research and Development* **22**, No. 4, 413–421 (1978).
3. *IBM Personal Computer Technical Reference*, 6936895, IBM Corporation; available through IBM branch offices.
4. *The INTEL 8086 Family User's Manual, Numerics Supplement*, 121586-001, Intel Corporation (July 1980); available through the Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051.
5. *IBM Personal Computer Disk Operating System*, 6936836, IBM Corporation; available through IBM branch offices.
6. *IBM Personal Computer APL Interpreter*, 6024077, IBM Corporation; available through IBM branch offices.

7. *IBM Personal Computer Macro Assembler*, 6024002, IBM Corporation; available through IBM branch offices.
8. *The INTEL 8086 Family User's Manual*, 9800722-03, Intel Corporation (October 1979); available through the Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051.
9. M. L. Tavera and M. Alfonseca, "Elastic work space for an APL system," *IBM Technical Disclosure Bulletin* **TDB04-81,** No. SZ8800004, 5147–5148 (April 1981).
10. *APL Language*, GC26-3847, IBM Corporation; available through IBM branch offices.
11. *APL2 Programming, Language Reference*, SH20-9227, IBM Corporation; available through IBM branch offices.

Reprint Order No. G321-5238.

Maria L. Tavera *IBM Madrid Scientific Center, P. Castellana, 4.28046 Madrid, Spain.* Since joining IBM in 1974, Dr. Tavera has worked at the Madrid Scientific Center. Prior to that, she worked at the International Telephone and Telegraph Laboratory of Spain in the fields of computer design and computer-controlled digital exchanges. At the Scientific Center, she has worked on projects related to compilation, language design, and data structures. She received an industrial engineering degree from the Madrid Polytechnical University in 1968, an M.S. degree in computer science from the University of London in 1973, and a Ph.D. degree in industrial engineering in 1980, also from the Madrid Polytechnical University. Dr. Tavera was awarded the Pilar Carega Prize from the Civil Engineering Association in Spain. She also received an IBM Outstanding Technical Achievement Award for the work described in this paper.

Manuel Alfonseca *IBM Madrid Scientific Center, P. Castellana, 4.28046 Madrid, Spain.* Dr. Alfonseca joined IBM at the Madrid Scientific Center in 1972. There he has worked on a number of computer-related projects, including computer language translators, continuous simulation, and computer graphics. Prior to joining IBM, he worked at the Computing Center of the Universidad Computense of Madrid from 1970 to 1972. Dr. Alfonseca received an electronics engineering degree and a Ph.D. degree from Madrid Polytechnical University in 1970 and 1971, and the computer science Licenciature in 1972. Since 1977, he has lectured on formal languages at the postgraduate level as a member of the Computer Science Faculty of the Madrid Polytechnical University. He was awarded the National Graduation Award in 1971. He also received an IBM Outstanding Technical Achievement Award for the work described in this paper.

Juan Rojas *IBM Madrid Scientific Center, P. Castellana, 4.28046 Madrid, Spain.* Before joining IBM in 1977, Mr. Rojas worked at the Compañia Española de Petróleos, Sociedad Anónima, in the Department of Computer Science. In 1981, he became a member of the Madrid Scientific Center, where he has worked in the fields of compilation, databases, and microcomputer operating systems. He obtained a Licenciature in physics from the University of Seville in 1969 and a degree in industrial engineering, also from the University of Seville, in 1973. Mr. Rojas was awarded an IBM Outstanding Technical Achievement Award for the work described in this paper.