COBOL/2: The next generation in applications programming

by R. Sales

IBM COBOL/2 is a new compiler and debugger system for the Personal System/2® product range developed by Micro Focus Group PLC of the United Kingdom. In this paper, Robert Sales, a software development manager for Micro Focus who was instrumental in creating the COBOL/2 system, describes how COBOL/2 breaks new ground in providing support for many disparate COBOL language dialects and standards, as well as in providing support for OS/2™ on the Personal Computer architecture.

COBOL was conceived as a business-oriented language and was designed to meet the needs of data processing applications. It is particularly strong in the area of character handling and data presentation, allowing clear and concise specification of the fields making up file records. A wide range of advanced facilities are built into the language, including indexed file handling, sorting and merging data files, and report generation. These facilities are fully implemented in COBOL/2 and undoubtedly represent the principal value of the system. Together with specialized syntax for interactive screen and keyboard handling, business applications can be developed that make full use of the immediacy of microcomputers with all the power offered by COBOL on mainframes. Alternatively, the system can be used for developing, testing, and maintaining programs which are to be run on the mainframe.

The use of COBOL as a data processing language on mainframe computers is well known. This paper discusses some of the features of COBOL/2 which make it well suited to other application areas—in particular, how it has been adapted for use on microcomputers.

COBOL/2, the ANS85 standard, and the support of different dialects

The first COBOL standard was published in 1968 by the American National Standards Institute (ANSI). It was followed six years later by the 1974 standard, which introduced new features, tightened up the definition of others, and removed some facilities, such as the NOTE statement, which had become obsolete with the introduction of comment lines. By contrast, the next standard took eleven years to emerge and was the subject of intense debate.

The long negotiation for the ANS85 standard demonstrated the importance attached to forward compatibility in the computer industry, particularly in regard to COBOL, which has been the vehicle for massive investments in applications software. Despite the obvious benefits of the new standard, in particular the new structured programming constructs, its adoption was continually held back by objections to any incompatibilities it introduced. Publication was possible only after many features originally destined for deletion (such as the notorious ALTER verb) had been reinstated as "obsolete features to be deleted from the next revision of standard COBOL."

Whatever arguments may have been generated by the arrival of the ANS85 standard, it remains true that

very few commercial applications even conform to the ANS74 standard. The most widely used COBOL compiler, IBM's OS/VS COBOL, not only offers a wide range of its own extensions, but continues to support much of ANS68 COBOL.

To meet the needs of existing programs while allowing new programs to be developed to the specifications of the new standard, COBOL/2 incorporates the syntax of many different COBOL dialects. In order to facilitate migration of programs to the new standard, it is permitted to mix features from more than one dialect within a single program.

Two series of compiler directives are available. One of these series is positive; it enables the use of constructs from a specified dialect. The main effect of these directives is to cause the reserved words for that dialect to be set up in the compiler's internal reserved-word table. For instance, the directive OSVS will add all reserved words specific to OS/VS COBOL, thus allowing the use of such statements as EXHIBIT and TRANSFORM. Conversely, NOOSVS will remove these reserved words from the table and allow EXHIBIT and TRANSFORM to be used by the program as data names. Where there is an incompatibility, these directives will also determine the behavior of the compiled code. Thus, the directive ANS85 not only adds all the new reserved words but changes the meaning of such statements as IF data-name IS ALPHABETIC from the ANS74 meaning (uppercase only allowed) to the ANS85 meaning (uppercase or lowercase permitted). Dialects recognized include OS/VS, VS COBOL II, ANS85, ANS74, and, for compatibility with IBM PC COBOL Version 1.00, PC-1.

Even if NOANS85 is specified, constructs which do not require any new reserved words will still be compiled correctly, so that it is possible to make use of these features while maintaining forward compatibility.

The other series of directives is restrictive in the sense that it enables the user to choose one dialect and create a listing on which any departures from the syntax permitted in that dialect are flagged. For instance, the directive FLAG (ANS74) will cause any use of in-line PERFORMs or other features not available in the ANS74 standard to be flagged. Similarly, the directive FLAG (SAA) will identify any departures from standard COBOL as defined in IBM's Systems Application Architecture.

The combination of these series of directives gives great flexibility in controlling migration between environments. As an example, if it were required to convert a program originally developed with IBM's OS/VS compiler to run under VS COBOL II, it could be compiled using the directives OSVS FLAG (VSC2). This would ensure that all constructs specific to OS/VS were understood by the compiler, while at the same time warning of any such usages that are not part of the VS COBOL II language.

Implementation on an 8086- or 80286-based system of a high-level language must resolve the problem of data segmentation.

COBOL/2 and large applications

Every implementation on an 8086- or 80286-based system of a high-level language permitting large blocks of contiguous data must resolve the problem of data segmentation. Whereas COBOL allows the declaration of data items and groups of data items of any length, the internal architecture of the CPU only allows data to be addressed within a maximum of 64K bytes from a base address defined by a "segment register."

On an 8086 system, or equivalently an 80286 system working in the so-called "real mode," the value contained in the segment register corresponds to an actual machine address. In fact, the corresponding machine address is determined by multiplying the contents of the segment register by 16 (or if, as is usual, the value is expressed in hexadecimal form, by adding a 0 on the end). A segment register can thus be made to point to any machine address which is a multiple of 16.

It is therefore relatively simple to point at whatever data are required for a particular COBOL statement. For instance, if a table is declared as

```
03 X1 PIC X(10) OCCURS 3000 INDEXED BY I1
```

a statement of the form

```
MOVE "ABCDEFGHIJ" TO X1 (I1)
```

could be executed on the 8086 by calculating a value for the segment register based on the value of the index I1 such that the required instance of X1 would start within 16 bytes of the base address.

The fundamental difference in "protected mode" working on the 80286 microprocessor is that these segment registers (renamed "selectors" in this case) no longer contain real machine addresses, but rather indices into a table maintained by the operating system where the actual addresses are stored. Whenever a value is placed into a selector (by instructions of the form *mov ds, ax*, or *pop es*), an interrupt is generated by the hardware, which transmits a call to the operating system to ensure that the new value of the segment register is valid. It will only be considered valid if it has been allocated previously by the operating system, either when the program was loaded or by subsequent calls made by the program to the operating system requesting further memory. Invalid contents will cause the process to halt.

The COBOL/2 compiler does not distinguish between real mode and protected mode.

The COBOL/2 compiler does not distinguish between real mode and protected mode, and produces identical object programs for both cases. It follows that the kind of calculation described in reference to real-mode working is not permitted. Instead, a COBOL data division of more than 64K bytes is viewed as one or more 64K blocks of data followed by a final segment up to 64K bytes long. Individual data items may cross the boundary between one segment and another and, for each item referenced in a procedure division statement, the compiler determines whether this is or may be the case (the doubt arises when the data item is subscripted and the table of which it forms a part itself crosses a segment boundary). If boundary crossing is possible, the compiler will generate code to do one of two things. For small operands (including numeric operands which have a well-defined maximum length), if the operand is a source field for the operation, it is first copied to a dedicated area within the first data segment and the operation is performed on this copy. If it is a target operand, the result is evaluated in the first data segment and copied into place afterwards. For large operands, calls are made to specialized routines which do not assume that data are contained within a single segment but check whenever they increment data pointers to see whether the current segment is exhausted.

COBOL/2 and communication with OS/2™ functions and other languages

One of the major advances offered by OS/2 over earlier DOS versions is the introduction of the Application Programming Interface (API). Parameters to DOS functions were passed in the internal registers of the CPU and could therefore be accessed only through Assembler programs. Parameters to the OS/2 API functions are passed in a standard way on the machine stack, which means that high-level languages such as COBOL may now call the operating system directly. To give COBOL the flexibility to call any of these functions, certain syntax extensions have been found necessary. These are detailed below, and an example using all of these features follows. These same features also enable COBOL programs to call subprograms written in C and, provided a stack-based interface is used, Assembler.

Call by value. All parameters to OS/2 functions are passed on the stack. However, the stacked item may be either the address of the parameter or its value, and a given function may have parameters of both kinds.

A parameter whose address is stacked is said to be passed BY REFERENCE; data passed in this way may be both read and modified by the called function. If only the value is stacked, the parameter can serve only as input to the function, and it is said to be passed BY VALUE.

One of the extensions introduced by ANS85 is the provision of parameter passing BY CONTENT as well as the default BY REFERENCE. BY CONTENT is similar to BY VALUE as described above, in that parameters

passed in this way serve only as input and cannot be modified by the called program. However, it is a consequence of the ANSI definition that a called program does not need to know whether the parameters passed to it were BY REFERENCE or BY CONTENT. A BY CONTENT parameter is therefore implemented by allocating a work area within the data space of the calling program, into which the parameter is moved before the call. This copy of the parameter is then passed BY REFERENCE to the called program, which may change it while the original data remain unchanged.

Since BY CONTENT is unsuitable for use in passing values to OS/2 functions, it was necessary to introduce BY VALUE as a new category of parameter-passing method in COBOL/2. This method may not be used to pass parameters to other COBOL programs since, as mentioned above, there is no way for a COBOL program to define the kind of parameter it is expecting as input. A possible extension would be the provision of syntax of the form

```
PROCEDURE DIVISION [USING [ [ BY REFERENCE ] {data-name} ... ] ...]
                           [ BY VALUE     ]
```

Pointer variables. Pointer variables are a feature of the IBM VS COBOL II compiler, from which the syntax used in COBOL/2 is adapted. They are used to hold pointers to data areas either within or outside the data space of the program in which they are declared.

The statement

SET pointer-variable TO ADDRESS OF identifier

is used to initialize a pointer variable to the address of an identifier, while the statement

SET ADDRESS OF linkage-record TO pointer-variable

will map the linkage record onto the address held in the pointer-variable. Following a SET statement of this kind, the linkage record, and all data items contained within it, may be accessed by the COBOL program in the normal way.

Pointer variables in COBOL/2 are held as four-byte fields in standard 8086 pointer format, i.e., two-byte offset followed by a two-byte segment. A special value NULL is used to indicate a pointer which is not pointing to any valid address. NULL pointers are represented by binary zeros.

Return-code. This is a feature of both VS COBOL II and OS/VS COBOL. RETURN-CODE is one of a class of special data items known as "special registers" that are not declared explicitly in the COBOL data division but may be referenced within the procedure division. The value held in RETURN-CODE at the moment when a called program executes an EXIT PROGRAM is passed back and placed in the RETURN-CODE belonging to the calling program. This value normally represents the success or failure of the call, with 0 meaning success and other values indicating a particular error code.

A different data type, COMP-5, has been introduced to use the storage convention of the 8086.

In COBOL/2 the return code is passed via the *ax* register of the 8086, so that the code generated for EXIT PROGRAM includes a move from the return code into the *ax* register. Similarly, for a COBOL CALL statement, following the call of the subprogram, the value of *ax* is moved into the calling program's return code.

Since the OS/2 functions always return a value in *ax* indicating success or failure, calls to these functions will automatically set up the COBOL RETURN-CODE on return.

COMP-5 data. In most environments, binary data longer than a single byte are stored with the most significant byte at the lowest address. It is a characteristic of the 8086 series chips that this convention is reversed so that the machine instruction which stores a two-byte value to memory will place the least significant byte at the lower address. As a consequence, most languages implemented on these chips hold binary values with the least significant byte first.

In the case of COBOL, COMPUTATIONAL data are stored in binary form in most implementations, but data

items of this type frequently form part of records written to files, and considerations of portability dictate that they be stored in the conventional manner, with the most significant byte first. A different data type, COMP-5, has therefore been introduced which uses the storage convention of the 8086.

The example in Figure 1 shows how a COBOL program can call the operating system to allocate a shared memory segment and initialize the data so obtained via a linkage section record. Following the call, independent programs running concurrently, whether COBOL or not, would be able to access this shared segment.

ANIMATOR is the symbolic debugger supplied with COBOL/2.

COBOL/2 and animation

ANIMATOR is the name given to the symbolic debugger supplied with COBOL/2. Although ANIMATOR offers a very wide range of functions, perhaps its principal strength, and certainly the chief quality aimed for during its original development, is its "obviousness."

In the animation of a program, the first thing the user sees is a page of the source program presented on the screen, together with two menu lines listing the functions available. This screen of text will include the first executable statement of the program, which is distinguished by highlighting. From then on the user can simply type "S" (for Step) to execute the current instruction and move the highlighting to the next executable statement. The screen is refreshed as necessary so that at each state it includes this next statement.

When the current instruction is a PERFORM, a Step would normally cause ANIMATOR to display the text of the performed routine with highlighting on its first statement. However, commands are available both to "step over" an entire PERFORM statement and, once a performed subroutine has been entered, to complete it and return to the statement following the PERFORM. The other main execution command is "Z" (for Zoom), which suspends animation, allowing the user program to execute continuously until either a breakpoint is reached or the user breaks in by entering Ctrl Break on the keyboard.

Whatever execution commands are used, any screen displays generated by the user program are automatically diverted to a separate logical screen which can be viewed at any time. Any statement requiring keyboard input will cause the ANIMATOR screen to be temporarily replaced by the user screen while the input is in progress.

Breakpoints can be unconditional or conditional in the sense that they will only be active when a COBOL conditional expression typed in by the user becomes true. They may also have an associated iteration count so that they only take effect when control passes through the statement a given number of times. The execution path can be reviewed by means of a backtracking facility.

Other important commands include "R" (Reset), which allows program flow to be changed so that alternative program branches can be tested, and "Q" (Query), which is used to examine and change the contents of data items and tables of data items; these are displayed either in text form (with conversion for nonDISPLAY fields) or in hexadecimal form. The Query command is also used to examine the status of files and condition-names.

Some of the more advanced features of ANIMATOR are made possible by the fact that it is integrated with the compiler, which it can use to parse COBOL syntax. Thus, the condition associated with a conditional breakpoint can be a combined conditional expression of any complexity. It is also possible to set up such a condition which is not attached to a particular point of the program but is tested after each statement, with execution halting at whatever point the condition becomes true.

Another powerful facility which relies on the resources of the compiler is the "D" (for Do) command, which allows the user to type in a COBOL statement for immediate execution. These Do commands may alternatively be associated with breakpoints, with the effect that they are executed whenever control passes through that breakpoint.

Figure 1 A COBOL/2 program calling an OS/2 API function (in this case DosAllocShrSeg)

```
working-storage section.
01   num-bytes              pic 9(4) comp-5 value 4096.
01   selector-name.
     03                     pic x(18) value "\SHAREMEM\COBOLSEG".
     03                     pic x     value low-value.
01   returned-pointer       usage pointer value null.
01   redefines returned-pointer.
     03  zero-offset        pic 9(4) comp-5.
     03  returned-selector  pic 9(4) comp-5.

linkage section.
01   link-record            pic x(32000).

procedure division.

* The specification for DosAllocShrSeg is
*
*    EXTRN DOSALLOCSHRSEG:FAR
*
*    PUSH  WORD Size        ; Number of bytes requested
*    PUSH@ ASCIIZ Name      ; Name string
*    PUSH@ WORD Selector    ; Selector allocated (returned)
*    CALL  DOSALLOCSHRSEG
*
* Any parameter without an "@" sign, such as the size field in
* this example, is expected to be passed by value.
* All other parameters (those with the "@" sign) must be passed
* by reference (which is the default).
* An ASCIIZ string is a series of ASCII characters terminated
* by a binary zero.
* The function name is preceeded by two underscores, which tell
* the compiler that the function name is an external symbol to
* be resolved by the linker.  The compiler will strip these
* underscores from the name.
* As in most other languages, the first parameter specified in
* a COBOL CALL is pushed last, so parameters are written in
* reverse order as compared with the specification above.
* Note the use of reference modification to access only that
* portion of the linkage record allocated by the call.

call "__DOSALLOCSHRSEG" using
     by reference returned-selector
                  selector-name
     by value     num-bytes.
set address of link-record to returned-pointer.
move low-values to link-record (1:num-bytes).
```

COBOL/2 and interactive applications

If the most obvious advantage of a small desktop computer over a mainframe is the immediacy of interaction with the user, it is important that a language designed for the small machine should enable the user to take full advantage of the facilities available.

One of the microcomputer-oriented features of COBOL/2 is the screen-section code, which allows the complete specification of screens in much the same way that the report-section code allows the definition of reports. The screen-section syntax of COBOL/2 is based on that of IBM PC COBOL Version 1.00, but includes certain extensions such as the use of OCCURS to allow for display or entry of all the elements in a table by means of a single ACCEPT or DISPLAY statement.

The following example of a screen section shows some of the options offered. To make use of these screens, the procedure division simply uses statements of the form DISPLAY PASSWORD-FORM, which will clear the screen and display the message

"Please enter password < >,"

and ACCEPT PASSWORD-FORM, which will allow the user to enter a password. The phrase REQUIRED prevents processing from continuing until a password has been entered, while the phrase ENSURE means that the characters entered are not echoed to the screen.

The phrase TO P-PASSWORD means that the field entered will be moved into the data item P-PASSWORD, which must be defined in the working-storage section. Screen-section items declared with the TO phrase only participate in ACCEPT statements and are ignored during execution of DISPLAY statements. Items which are only meant to participate in DISPLAY statements must use the FROM or VALUE phrase, while items declared with the USING phrase take part in both ACCEPT and DISPLAY operations.

Other phrases used in this example include LINE and COLUMN to define screen position, FULL to indicate that the field must be entered in full before progressing to the next field, and BLINK, meaning that the message will be displayed blinking. This last is one of many options allowing use of the various screen attributes available on the PC screen, such as underlining, reverse video, high intensity and, on a color screen, all the combinations of foreground and background color. Figure 2 illustrates the screen-section code.

COBOL/2 as a systems programming language

If COBOL is best known as a language for data processing applications, the development of COBOL/2 demonstrates that it can also be effective for systems programming. The compiler is itself written in

COBOL offers a multiplicity of storage formats for numeric data items.

COBOL, as are many run time support functions such as the indexed file handler, the screen handler, and even ANIMATOR, the symbolic debugging tool. This self-residency has played an important role in the development of the product, encouraging the emergence of new features and exposing problems as they arise. In particular, ANIMATOR is routinely used to "animate" itself and is even capable of animating itself animating itself to any level of nesting.

Among the strengths of COBOL in the context of systems programming is the simplicity of string manipulation. This has been much enhanced by the concept of reference modification introduced by the ANS85 standard, which allows any portion of a string to be accessed directly by specifying a start position and a length. For instance, the statement MOVE SPACE TO TEXT-LINE (START-POS: FILL-LENGTH) will move spaces into the portion of TEXT-LINE starting at position START-POS and for a length of FILL-LENGTH. Similarly, the introduction of loop constructs (in-line PERFORMs) and scope delimiters for conditional statements (particularly END-IF) has added much to the pleasure of working with COBOL.

However, COBOL is not like C. It was not designed with bytes and bits in mind; to support this kind of low-level working, certain enhancements have been made to the language.

COBOL offers a multiplicity of different storage formats for numeric data items. In the standard format

Figure 2 Example screen section code

```
screen section.

01   password-form.
     02   blank screen
     02   value "Please enter password  <".
     02   pic x(20) to p-password, required, secure.
     02   value ">".

01   check-form.
     03   blank screen.
     03   line 2, column 35, value "Check".
     03   line 4.
     03   occurs 4.
          04  pic x from "<", line, column 10.
          04  pic x(15) using s-to.
          04  pic x from ">".
     03  value "Recipient Name and Address", line + 2, column 6.
     03  value "<", line 14, column 10.
     03  pic zzz,zz9.99 using s-amount, required.
     03  value ">".
     03  value " Amount", line 15, column - 9.
     03  value "<" line 5, column 40.
     03  pic x(10) using s-for.
     03  value ">".
     03  value "   For", line 6, column - 9.
     03  value "<", line 5, column 60.
     03  pic x(10) using s-refno, full.
     03  value ">".
     03  value "Reference", line 6, column - 9.
     03  value "Date: ", line 14, column 46.
     03  value "<", column day-col.
     03  pic zz    using s-day, prompt is "d", auto, required.
     03  value "/", column month-col.
     03  pic zz    using s-month, prompt is "m", auto, required.
     03  Value "/", column year-col.
     03  pic zzzz  using s-year, prompt is "y", full, required.
     03  value ">  ".

01   error-messages, blink.
     02  clear-msg   line 24, blank line.
     02  msg1        line 24, value "Invalid day  ".
     02  msg2        line 24, value "Invalid month".
     02  msg3        line 24, value "Invalid year ".
```

they are held as printable characters (USAGE DISPLAY), with the sign represented either as a separate character at the beginning or end of the field or as a transformation of the first or last character of the field itself.

Most implementations also offer a binary format (USAGE COMPUTATIONAL or BINARY in ANS85) and packed decimal (COMPUTATIONAL-3 or PACKED-DECIMAL). Calculations may be performed on data items of any of these types and may involve different types in the same statement. However much the compiler attempts to optimize such code, it is clear that format conversions will degrade performance. Even if calculations are confined to binary fields, efficient implementation is hampered by the fact that the ANS standard requires such fields to behave in the same way as DISPLAY fields; e.g., a data item defined as PIC 9 (4) and occupying two bytes should only hold values up to 9999.

To ensure a true binary format that is not connected with decimal arithmetic, we have defined a new data type, COMP-X. The PICTURE string for such data may be written either with 9s or with Xs, where each X indicates one byte. Thus PIC XX USAGE COMP-X is exactly the same as PIC 9 (4) COMP-X, but gives a more accurate indication of the nature of the data. These data items may have any length up to eight bytes and are treated as unsigned numeric data. The only thing that differentiates them from normal numeric items is the fact that they are subject to binary and not decimal overflow. This means that subtracting 1 from an item declared as PIC X COMP-X and containing a value of 0 will give a result of 255; adding 1 to it when it contains 99 will give 100.

Manipulation of bits within these fields is made possible by a number of system calls which perform such functions as unpacking the eight bits of a byte into eight contiguous bytes, rotating by a given number of bits, etc. There is some overhead in the use of CALLs, and simple language extensions to provide the logical operations of OR, AND, and XOR would be of benefit to programs making heavy use of bit manipulation.

Several options are available for optimization of suitably written COBOL programs. Among these is the NOTRICKLE option, which allows for fast execution of PERFORM statements by compiling the perform into an 8086 CALL instruction. All paragraphs or sections which are used as the end of a perform range, such as A in the statement PERFORM A and C in the statement PERFORM B THROUGH C, have an 8086 RET instruction planted after their last statement.

This optimization is not possible when a program contains a statement such as PERFORM A and where, in other circumstances, control is expected to fall through, or "trickle," from A into the succeeding paragraph, as would be the case if the program also contained the instruction PERFORM A THROUGH B.

OS/2 allows several COBOL programs to be run concurrently.

Translation of PERFORMs into CALLs allows the PERFORM to be used recursively, and the compiler makes some use of this possibility, for instance when dealing with arithmetic and conditional expressions. The program shown in Figure 3 demonstrates the use of PERFORM recursion to calculate the factorial for a number entered by the user. As in most examples of recursion, it could be coded more simply using a loop, as shown in the figure.

COBOL/2, file sharing, and network communication

OS/2 is a multitasking operating system, allowing several COBOL programs to be run concurrently on the same machine. If these programs make use of the same files, it is necessary to have a means by which one program can prevent the others from accessing files, or particular records within files, while it is making modifications. This is also important when machines are connected together on a network. In this case one or more of the machines may be set up as a "server," meaning that all its files or files belonging to particular subdirectories are available to any other machine on the network. Once a command such as NET SHARE DIR1 XYZ has been issued on the server, the directory DIR1 becomes known on the network by the name XYZ; any other machine can then issue a command such as NET USE E: server XYZ and access the files on the server directory *xyz* using the name *e:file-name.*

Figure 3 Example of a PERFORM recursion

```
working-storage section.
01   n1  pic 9(4).
01   n2  pic 9(18).
procedure division.
     display "Enter number: " with no advancing accept n1
     move n1 to n2
     perform p1
     if n2 = 0
          display "Number too large"
     else
          display "Factorial = " n2
     end-if
     stop run.
p1.
     subtract 1 from n1
     multiply n1 by n2
     on size error
          move 0 to n2
     end-multiply
          if n1 > 1
          perform p1
     end-if.
```

In COBOL/2, record locks may be obtained when a record is read to prevent anyone else from accessing that record until the lock is released. The SELECT statement for a file may include the phrase LOCK MODE IS AUTOMATIC, meaning that locks will be applied automatically to each record read, or LOCK MODE IS MANUAL, meaning that a lock will only be applied if the READ statement uses the WITH LOCK phrase. Additionally, the file may be defined as having LOCK ON RECORDS, meaning that each successive access to the file will release any previous locks, or LOCK ON MULTIPLE RECORDS, in which case locks are only released by means of the UNLOCK statement, which releases all locks on a specified file, or the COMMIT statement, which releases all locks on all files. COBOL/2 also permits a whole file to be locked, either by means of a LOCK MODE IS EXCLUSIVE phrase in the SELECT statement, or by specifying EXCLUSIVE in the OPEN statement.

Besides these facilities for sharing files, COBOL/2 also includes a means of passing messages between different programs running on a network, using one of the least-known and least-used areas of the COBOL language, the COMMUNICATIONS module.

The lack of interest in the COMMUNICATIONS module is no doubt due partly to the complexity of the language definition, but perhaps also to its lack of usefulness. In previous COBOL compilers on microcomputers, this facility was certainly implemented more because it was a requirement for ANS certification than because it had been demanded by users.

The ANS definition of COMMUNICATIONS, in both the 1974 and 1985 standards, speaks in terms of communication between a COBOL program on one side and a "communication device" on the other. COBOL/2 makes the relation symmetrical by allowing

Figure 4 Simple introductory COBOL program

```
Select infile assign infile-name line sequential.
fd infile.
01 in-rec pic x(120).
working-storage section.
01 eof-flag pic 99 comp.
procedure division.
     display "Enter file name: " with no advancing accept infile-name
     open input file
     move 0 to eof-flag
     perform until eof-flag not = 0
          read infile
               at end
               move 1 to eof-flag
               not at end
               display in-rec
          end-read
     end-perform
     stop run.
```

the "communication device" to be another COBOL program executing remotely on a networked machine, thus giving COBOL programs the ability to talk to one another over the network. Although the complete syntax for COBOL communications is complex, a simple subset is enough to establish contact between two such COBOL programs and pass messages between them.

Communication is initiated on both sides by means of the ENABLE verb and continues thereafter by means of SEND and RECEIVE verbs. Message destinations are defined within the COBOL program with symbolic names, and before the program can be run, the actual name must be associated with this symbolic name. Thus, the symbolic destination for a program sending messages is defined within the program, and this must be associated with the actual destination, which is another COBOL program. This association is established by running a utility program called MCSETUP, which creates an information file for each of the programs. The use of symbolic names means that network configurations can be altered without having to change the COBOL programs.

COBOL/2 as a vehicle for teaching

Since COBOL is widely regarded as a verbose language, it seems appropriate to start with the most simple program that can be created using COBOL/2:

```
Procedure division.
Display "hello world."
```

This program, which will simply display the message and stop, is not very useful in itself, but perhaps serves as a better introduction to computer programming than the normal approach to COBOL through a

solemn discussion of the identification, environment, data, and procedure divisions. In COBOL/2 none of these divisions is actually required (though the program will not do anything without the procedure division) and, if they are put aside initially, COBOL is revealed as being just as simple to use as BASIC or Pascal, though it offers unlimited growth to the novice programmer.

Creation of more realistic programs still requires a minimum of red tape, as can be seen from Figure 4, which reads a text file and copies its concepts to the screen. Note the use of lowercase letters, which some feel improves the readability of the program. Since the compiler in fact takes no notice of case, the user is free to adopt whatever convention is preferred.

This little program owes much of its simplicity to new features introduced by the ANS85 COBOL standard. In particular, the in-line perform statement permits the loop to be written without resorting to GO TOs and paragraph names. The alternative "not at end" branch of the READ statement serves a similar purpose since, in its absence, it would be necessary for the "at end" branch to jump off somewhere in order to avoid the display.

As a result of these and other improvements in the language, it is now possible to write well-structured COBOL programs to satisfy the most ardent advocate of "structured programming." It is perhaps still true that COBOL gives more scope than most languages for bad practices, but these practices can be avoided. The ALTER verb is part of the history of COBOL and, despite the temptation to withdraw it, remains a part of COBOL for historical reasons. But the need to support old programs does not mean that anyone should consider using such constructs in new programs.

The essential character of COBOL is a sequential, narrative style which is very close to the style of current machine languages (a COBOL "move" is similar in purpose to an 8086 "mov" even though it operates on much more complex entities). This narrative style, which makes COBOL such an intuitive language for a first approach to computing, is exploited and emphasized by ANIMATOR, the symbolic debugging tool discussed earlier. ANIMATOR is an excellent way to provide insight into the way in which a computer program functions.

General reference

American National Standard X3.23, Programming Language COBOL ANSI, ANSI COBOL Committee, ANSI X3.24, 1985.

Robert Sales *Micro Focus Inc., 2465 E. Bayshore Rd., Palo Alto, California 94303.* Mr. Sales has been with Micro Focus since 1979 and has been involved in developing and evolving most components of the Micro Focus COBOL system, including the original design and implementation of the ANIMATOR debugging tool in 1981. During 1985 and 1986 he was leader of the software group that developed the IBM COBOL/2 compiler. He graduated from the University of Cambridge, England, in 1971 with a degree in mathematics and completed a postgraduate course in number theory at Cambridge during the following year. Before joining Micro Focus, Mr. Sales worked as a mathematics teacher in London, as an English teacher in Rome, and with MAEL Computer in Carsoli, Italy, as a technical writer and software developer.

Reprint Order No. G321-5316.