Microcomputer software design—A checkpoint

by GARY A. KILDALL
Naval Postgraduate School
Monterey, California

INTRODUCTION

The general availability of low cost microcomputers has revolutionized digital design and digital applications. Using LSI chip technology, microcomputers are no more than scaled-down central processing units with minicomputer capability, and are treated as component computers at the heart of a digital design. Thus, microcomputers find wide application in both dedicated and general purpose roles, ranging from simple controllers through smart terminals and test instruments to small business data processing systems.

In each application, hardware and software modules are intermixed to minimize unit cost. As a result, the overall quality of a microcomputer-based product is directly determined by the quality of its hardware and software components. Similar to its hardware counterparts, the product's programmed subsystems must be well specified and engineered for long term reliability. In fact, well-engineered software has never been as important: packaged systems are often produced in the hundreds or thousands, where each program is permanently stored in unalterable ROM (Read-Only-Memory). Unreliable programs have far-reaching effects, while ill-specified software hinders product adaptability.

A particular high level language has emerged as an aid to the microcomputer software engineer which forecasts some industry standardization. This paper briefly reviews current design aids, with particular emphasis on applicability of high level languages in the microcomputer environment. A particular project case study is presented which exemplifies current design methodology, followed by projected trends in microcomputer software aids.

BEYOND THE DATA SHEET

In essence, a microcomputer is simply another integrated circuit chip set, with somewhat more than average capability. In fact, many design engineers consider a microcomputer CPU as simply a ROM-driven LSI chip which, with proper arrangement of 1's and 0's in the external ROM, can be tailored to act like a custom chip. The design engineer breadboards a circuit including the microcomputer, fills the ROM's with binary codes which drive the chip, and proceeds to debug with logic probe and scope. Although costly in development and maintenance time, this approach is quite popular since no external support is required beyond the chip's data sheet.

At the opposite end of the applications spectrum, the microcomputer is considered just another processor which, independent of physical characteristics, provides a key to product update and new marketing areas. Often from a minicomputer background, customers are unwilling to return to primitive programming tools and meager design support.

As a result of demands from a broad customer base, many of today's semiconductor houses find themselves in the software business. A recent survey cross-references ten microcomputer manufacturers by the software design aids which they support.[1] Of these manufacturers:

all ten support a cross-assembler,
four offer resident assemblers,
three provide a resident editor,
eight support relocatable or absolute loaders,
five provide primitive debugging facilities,
six offer cross-simulators, and
two support a high level language.

The cross products all require a larger host computer for actual execution. That is, cross-assemblers are usually written in ANSI standard FORTRAN to allow some measure of machine independence. The customer either purchases the program directly from the manufacturer, or contracts with a timesharing service which supports the manufacturer's software.

Resident software systems, on the other hand, execute using microcomputer developmental hardware. Most manufacturers offer a built-up microcomputer prototyping system as a hardware developmental aid, including CPU, memory, I/O access, and front panel control. In this configuration, the microcomputer has minicomputer characteristics, and thus can support its own software systems, including assemblers, paper tape editors, loaders, and debuggers. Although some of these resident software tools are quite comprehensive, current manufacturer's offerings are hindered by limited I/O facilities. As a result, resident software tools are less convenient than cross systems, but are generally less expensive to support.

Although similar in capability to a minicomputer, developmental systems generally incorporate features peculiar to microcomputer systems development. National's IMP-16P prototyping machine, for example, contains spe-

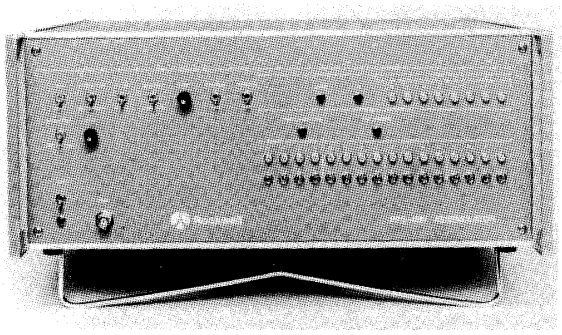

Figure 1—Rockwell's PPS-4 microcomputer development system

cial circuitry for loading reprogrammable ROM's, while Rockwell's "assemulator," shown in Figure 1, contains a built-in assembler and CPU emulator for programming and debugging their PPS-4 microcomputer. Thus, the manufacturer's developmental systems are generally inappropriate as end-user products.

Cross simulators are also used on larger host computers to programmatically simulate actions of the microcomputer. The primary problem, however, is that extensive program testing and simulation of real-time external events, such as signals input from a device controller, is tedious and expensive. Thus, cross simulators are principally used to step-through subroutines and program modules independent of the electronic environment. A simulator is extremely useful, however, when exact execution time must be determined for time-critical program segments.

Two major manufacturers of microcomputer chip sets are currently supporting a particular subset of PL/I as a base language for their products. Intel's language, called PL/M, has been available since mid-1973 through a cross-compiler, while National's product, called PL/M+, will be available in mid-1975 as an integral part of their resident developmental system. Intel's PL/M provides a base language for their 8-bit processors, and National's PL/M+ is designed for the IMP-16 and PACE microcomputers. The two languages are basically compatible, thus allowing transportation of customer software between these two manufacturers.

SYSTEMS LANGUAGES

As interest grows in PL/M-like languages for microcomputer systems development, one immediately questions the suitability of high-level languages in such an environment. First, does a language such as PL/M support necessary low-level control functions which occur in microcomputer systems, or does the designer "lose control" of his machine? Second, how memory-efficient can a translator for such a language be? The cost of high-quantity electronics products is largely determined by component count, and high-level language translators are notorious for their inefficient code sequences, resulting in excessive memory requirements in the final product. Thus, the discussion focuses on experiences with Intel's product as a benchmark for this class of languages.

First, a few general comments on PL/M itself. The language is modest in structure and scope: basic operators are tied closely to the capabilities of 8- and 16-bit processors, augmented by structures for writing assignments, simple expressions, conditional statements, looping control, and subroutine mechanisms. The result is a language which simplifies the expression of microcomputer systems, while allowing access to all machine functions, without becoming completely dependent upon a particular CPU organization. The language has facilities which are reflected within the capability of the microcomputer, and, similarly, each machine function is reflected in some high-level statement. Architecture-oriented languages of this sort, often referred to as systems languages, are traditionally used to implement the lowest level system functions to avoid the rigidities of assembly language coding. In the larger computer environment, systems languages are often used to implement operating systems, language processors, utilities, and some applications software. Thus, they are themselves self-supporting, generally requiring little existing system support. As illustrated in the examples which follow, this close relationship between the language and the machine architecture holds also for PL/M.

The Appendix contains a sample PL/M program which indicates the basic facilities of the language. This particular language has global characteristics of the "PL-family," but derives its basic structure from the microcomputer problem environment, as described above.[2]

As a final comment, one notices that after decades of ad hoc programming, there is finally an emerging body of theory and practice concerning software engineering[3,4,5,6] which is gaining industrial acceptance. Languages such as PL/M, which provide clear representation of control flows, are important tools in support of structured programming techniques. When combined with professional project management and programming practices, the result is usually well-specified, reliable, and efficient software systems.[7,8,9]

A CASE STUDY

Given the current level of support, how does one approach a microcomputer project which involves a total system design? Non-trivial projects are generally evolutionary in nature, where each phase of development and testing is a controlled experiment. In the case of software generation, the designer starts with cross systems for initial program development and testing, gradually moving to resident developmental systems, and then to a breadboarded prototype. Since system malfunctions can occur at any level, from low voltage power supplies through marginal IC's to programming blunders, this evolutionary approach isolates the range of errors at each

stage. A particular microcomputer project is outlined below which demonstrates this approach.

A dedicated computer system was recently constructed at the Naval Postgraduate School to be used by Navy divers while working underwater for extended periods. The device monitors the dive time and depth, and produces a continuous read-out of the "safe ascent depth." The safe ascent depth is the depth to which the diver can ascend from his current depth without contracting the "bends." As the diver descends, his blood takes on nitrogen, and as he ascends, the nitrogen is given off. Depending upon the length of time he has worked at various depths on a particular dive, he can rise only to the safe ascent depth before nitrogen gases form in the blood. Thus, the computer keeps the diver informed of this depth. The diving computer has four principal functions to perform:

- compute partial pressures of nitrogen for several controlling tissues,
- monitor external parameters such as elapsed time and current dive depth,
- drive simple displays with the current and safe ascent depths, and
- control the sequencing of external monitoring, computing, and display.

The final prototype was developed in two man-months, with approximately three weeks devoted to software development, and the remainder in hardware design and debugging.

With the overall analysis of the dive problem complete, a BASIC program was written which computed test values. The computations involved 32-bit signed integer values with fixed precision. Since the 8 bit processors support only simple operations on 8-bit quantities, subroutines were written in PL/M to provide necessary functions. Each subroutine was compiled using the PL/M cross-compiler on the school's IBM S/360, and the machine code was read-in by another program, called INTERP/8, which simulates 8008 CPU actions. Using the break point and display commands of the simulator, the numeric subroutine package was checked-out, using only the S/360, with no physical microcomputer hardware.

The numeric subroutines were augmented by additional PL/M coding which evaluated standard formulae (essentially the same as those of the BASIC program) for determining the partial pressures of nitrogen for a particular depth. Again, these subroutines were checked-out under simulation by inserting test values in simulated memory, running a single computation, and displaying the values resulting from the simulation. A control and sequencing program was then written which simulated a complete dive by looping through a predetermined dive profile of times and depths. Using the simulation, several complete dive profiles were run, and the intermediate and final results were compared with the BASIC program. Extensive testing was infeasible, however, since a simulated fifteen minute dive to a depth of 130 feet required over thirty minutes of S/360 CPU time.

Transition to real microcomputer hardware thus became necessary to complete the testing. From this point on, the program was compiled using the cross PL/M compiler on the S/360, but executed in real time using a developmental system. A paper tape was produced from the S/360 compilation containing the 8008 machine code which was then loaded through the Teletype reader into the memory of the developmental system, and executed.

In order to properly check-out the central algorithms, another set of subroutines was written in PL/M which provided basic communication between the program and Teletype, allowing the program to read commands, write test results, and read and print 32-bit fixed point numbers. These subroutines formed a software test bed which would eventually be discarded. Each test involved a dive profile with various times and depths preset from the Teletype console. The program would run the dive profile and print the safe ascent depth at crucial points in the test. The computations executed in five times real time (a 30 minute dive was completed in six minutes of 8008 time), and thus it was possible to verify results by comparing with both the BASIC program and standard Navy diving tables. After check-out, the central algorithms were separated from the test environment, and set aside for the final prototype.

At this point, it was determined that there were several disadvantages in using the 8008 for the final prototype, including factors such as power consumption and compactness. Thus, the design was altered to incorporate the newer 8080 microcomputer. Because of its increased speed, the 8080 could be "shut-down" for longer periods between each computation, resulting in significant power savings (partial pressures were updated every two seconds, and could be computed in 50 milliseconds). The PL/M language is upward compatible along this processor line, and thus the program was recompiled using the 8080 version of PL/M.

The prototype was constructed and debugged, and, upon completion, I/O drivers were coded in PL/M, placed into erasable ROM in the prototype, and independently tested. The I/O drivers were then combined with the core computation and control algorithms. The total program was compiled on the S/360, placed into ROM in the prototype and checked-out. As shown in Figure 2, the completed prototype is contained on a single 7×9 wirewrap board with space for 2K bytes of erasable ROM (the program currently uses 1.2K), and 1024 bytes of random access memory.

ADDITIONAL APPLICATIONS

The case study given above serves to illustrate current methods used to develop dedicated microcomputer software. In addition, the application involves both bit-level and simple numeric processing, which are both handled well in this particular high level language. To

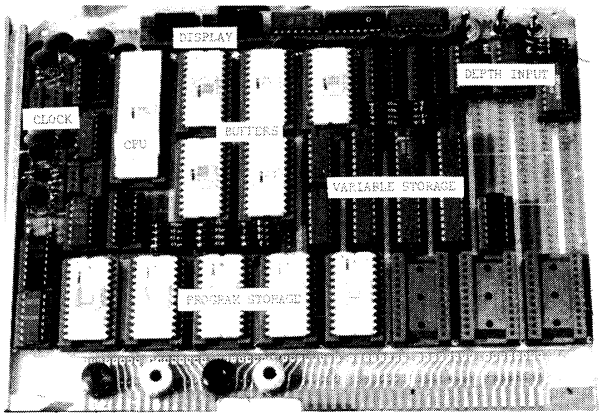

Figure 2—Navy SCUBA diving computer, using the Intel microcomputer

illustrate the range of applicability of PL/M, however, additional projects from more traditional computer areas are considered.

There is current industry-wide interest in incorporating today's low-cost peripherals with microcomputer devices to build inexpensive general purpose processors for resident microcomputer development and end-user applications. One such computer system, shown in Figure 3, includes a floppy disk operating system, which implements a named file structure with dynamic disk allocation on multiple disks, sequential or random access, and optimal disk arrangement strategies. When combined with the system's loaders, language processors, editors, and debuggers, the resulting facility rivals that of most time-sharing services for microcomputer program development. All software modules are written in PL/M including basic file management subroutines (3K), transient console command handler (2K), and various utility programs. An indefinite number of programs and subsystems can be supported since they reside on disk, and are loaded into memory on demand. Clearly, this particular application of a microcomputer heavily overlaps traditional general-purpose minicomputer areas.

A number of language processors have been implemented in PL/M, including a translator for the BASIC language as an aid in developing microcomputer programs which make heavy use of floating point operations. The BASIC translator operates under the disk system described above, and produces code which is executed interpretively by a special run-time subroutine package. More importantly, any translated program can optionally be loaded into ROM with the run-time subroutines, and placed into a circuit with a microcomputer which executes the program repetitively at the push of a button.

The translator for BASIC was itself written in PL/M (5K), and demonstrates its use as an implementation language. That is, PL/M has only simple operations, and thus is relatively easy to implement for any microcomputer. Given that PL/M exists, further special-purpose programs, such as the BASIC translator can be coded easily. As a result, all system software can be transported between different architectures if the base language can be transported. It is reassuring to know, for example, that the disk system software, BASIC translator, and BASIC programs will execute on Intel's 8008 and 8080 machines, as well as National's IMP-16 and PACE microcomputers with little modification.

SUITABILITY OF PL/M

These examples indicate the suitability of one high-level language in microcomputer systems design. Based upon this implementation, the most straightforward applications were those which the basic machine could already perform, including bit-level I/O control and character manipulation found in word-processing, operating systems, and language processors. In these cases, the algorithms were easy to express, and simple to debug and maintain. The operating system application, however, contains heavier use of table subscripting and run-time address computations. Although these functions were easy to express in PL/M, the underlying computations are more complicated for Intel's 8-bit machines. General floating point applications were by far the most complicated to code and debug in PL/M and, in general, resulted in a sequence of unintelligible mainline calls on these numeric subroutines.

The question of memory-efficiency is also a part of the suitability discussion. Again, the bit-level and character processing functions result in short code sequences which are quite competitive with good assembly language programming. The 16-bit address computations found in operating system work cause excessive program length unless the programmer uses techniques, such as localizing computations to common subroutines, which minimize this overhead. The general floating point application took an inordinate amount of program storage, due principally to the lack of basic machine facilities to perform these functions. One should consider implementing basic arithmetic functions of this sort in PL/M-compatible

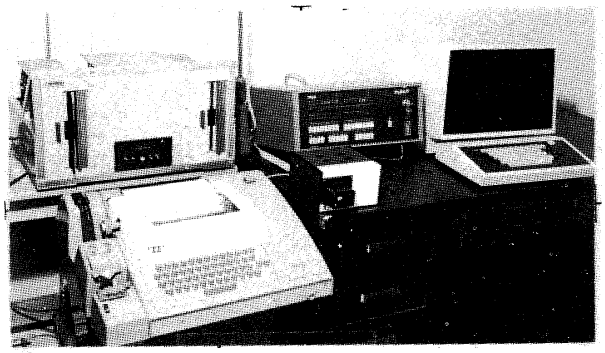

Figure 3—A disk-based microcomputer development system

assembly language where the side-effects of the machine can be more easily exploited. In any case, measured overhead for PL/M is in the range 10 percent to 35 percent when compared with assembly language coding, based upon experienced programmers and the current PL/M compiler.[9]

One can conclude, however, that the most suitable problems for expression in PL/M are precisely those problems which are most appropriate for the 8-bit processors. That is, the low-level functions are all present in PL/M, and the high-level functions are not. Further, the low-level functions are exactly the ones which are most memory-efficient.

FUTURE TRENDS

Microcomputer development practices seem to change on a monthly basis as manufacturer support increases, and hardware component costs decrease. Although any projections are questionable in light of this advancing technology, several trends are evident. First, the use of inconvenient and expensive cross development tools will be short-lived. Although the cost for cross assembly and cross compilation is comparable, either approach can rapidly consume project funds. Inexpensive disk-based resident developmental machines are becoming commercially available which, although still somewhat primitive, can be purchased for the price of the timesharing services necessary for even a moderate project. National's PL/M+, for example, will be available in mid-1975 as an integral part of their floppy disk-based development system, while numerous independent companies are providing add-on equipment for Intel, Rockwell, and other manufacturers. Due to the developmental nature of these systems, resident language processors will soon be augmented by comprehensive debuggers which provide high level reference through symbolic names and statement context.

Current interest in PL/M as a base language indicates that high level language standards are possible to some degree in the 8-bit processor category. Although there are obvious customer benefits in training, documentation, benchmarking, program portability, and machine independence, standardization also benefits the manufacturer. The present similarity between Intel's PL/M and National's PL/M+ allows the companies to "second source" one another at the language compatibility level. Thus able to share customer bases, their products can compete on a meaningful level: questions of suitability are settled by benchmarked performance and cost, not simply on the cycle time of the CPU. The role of the microcomputer has expanded since the initial introduction of PL/M, however, and thus the language must evolve to suit these applications. Nearly all major manufacturers have investigated the implementation of a PL/M-like language for their processors, and one can only guess whether these factors will lead to a unified base language, or simply a maze of confused dialects.

REFERENCES

1. Falk, H., "Microcomputer Software Makes its Debut," *IEEE Spectrum*, Vol. 10, No. 11, October, 1974.
2. *A Guide to PL/M Programming*, Intel Corporation, 3065 Bowers Ave., Santa Clara, Ca., 95051.
3. Buxton, J., *Software Engineering Techniques*, Nato Science Committee, OTAN/NATO, 1110 Bruxelles, Belguim, April, 1970.
4. Dahl, et al., *Structured Programming*, Academic Press, 1972.
5. Kernighan, B., et al., *The Elements of Programming Style*, McGraw Hill, 1974.
6. Yourdon, *Advanced Programming Techniques Volume 1: Program Structure and Design*, Yourdon, Inc., New York, N.Y., 1974.
7. Davidow, W., "Processors and Profits: How Microprocessors Boost Them," *Electronics*, July 11, 1974.
8. Metzger, *Managing a Programming Project*, Prentice Hall, 1973.
9. Kildall, G., "Systems Languages: Management's Key to Controlled Software Evolution," *Proceedings of the 1974 Western Electronics Show and Convention*, September, 1974.

APPENDIX

The listing given in Figure 4 is an example of an 8080 PL/M program which executes on an Intel developmental system. The purpose of the program is to test a procedure which keeps track of the elapsed time since system start-up. After each minute of elapsed time, the program prints:

hh HRS mm MINS

at the teletype, where hh and mm are decimal values for the hours and minutes of elapsed time.

The following run-time environment is assumed. A Teletype is connected to the 8080 CPU through a UART (Universal Asynchronous Receiver-Transmitter). In addition, an external interrupt is generated every 1/60th of a second, and is used for the basic program timing.

The program consists of a number of procedures followed by calls on these procedures. The mainline procedures are listed below along with their function in the program:

PRINTCHAR	print the single ASCII character in CHAR
CRLF	send a carriage-return and line-feed
PRINTBCD	print two decimal digits
PRINT	print a sequence of characters

One "interrupt procedure," called TIMEKEEPER, is defined with the attribute INTERRUPT 2. This interrupt attribute results in control transfer to TIMEKEEPER whenever interrupts are enabled and the external interrupt occurs.

The first PL/M statement which is executed follows the TIMEKEEPER procedure. The four variables FRACS, SECS, MINS, and HRS are zeroed. The first variable, FRACS, is a byte variable which tallies the number of 1/60ths of a second which have elapsed during a one second interval. The remaining variables each hold a pair of BCD

```
00001  1    /* THE FOLLOWING 8080 PL/M PROGRAM COMPUTES AND DISPLAYS THE
00002  1       ELAPSED TIME SINCE SYSTEM START-UP.  THE ELAPSED TIME IS
00003  1       PRINTED AT THE TELETYPE CONSOLE EVERY MINUTE */
00004  1
00005  1    DECLARE
00006  1       /* LITERAL SUBSTITUTIONS IN THE PROGRAM */
00007  1       TRUE LITERALLY '1',
00008  1       FALSE LITERALLY '0',
00009  1       FOREVER LITERALLY 'WHILE TRUE',
00010  1
00011  1       /* TELETYPE CONSTANTS FOR UART */
00012  1       TTO LITERALLY '0',    /* DATA TO TTY IS OUTPUT(0) */
00013  1       TTS LITERALLY '1',    /* STATUS PORT IS INPUT(1) */
00014  1
00015  1       /* SPECIAL CHARACTERS (NON GRAPHIC) */
00016  1       BEL LITERALLY '7',    /* RING TELETYPE BELL */
00017  1       CR LITERALLY '15Q',   /* CARRIAGE RETURN (15 OCTAL) */
00018  1       LF LITERALLY '0AH';   /* LINE FEED (A HEXADECIMAL) */
00019  1
00020  1       /* TELETYPE OUTPUT SUBROUTINES */
00021  1
00022  1    PRINTCHAR: PROCEDURE(CHAR);
00023  2       DECLARE CHAR BYTE;
00024  2       /* PRINT THE 8-BIT ASCII CHARACTER IN 'CHAR' AT THE
00025  2       TELETYPE CONSOLE */
00026  2
00027  2           DO WHILE ROR(INPUT(TTS),2);
00028  2           /* WAIT FOR UART TRANSMIT READY */
00029  2           END;
00030  2
00031  2       OUTPUT(TTO) = NOT CHAR;
00032  2       END PRINTCHAR;
00033  1
00034  1    CRLF: PROCEDURE;
00035  2       /* SEND A CARRIAGE-RETURN FOLLOWED BY A LINE-FEED */
00036  2       CALL PRINTCHAR(CR); CALL PRINTCHAR(LF);
00037  2       END CRLF;
00038  1
00039  1    PRINTBCD: PROCEDURE(B);
00040  2       /* PRINT THE BCD-PAIR HELD IN THE 8-BIT VARIABLE 'B' */
00041  2       DECLARE B BYTE;
00042  2       CALL PRINTCHAR(SHR(B,4) + '0');
00043  2       CALL PRINTCHAR((B AND 0FH) + '0');
00044  2       END PRINTBCD;
00045  1
00046  1    PRINT: PROCEDURE(A);
00047  2       /* WRITE CHARACTERS TO THE TELETYPE STARTING AT ADDRESS 'A'
00048  2       IN MEMORY UNTIL THE FIRST '$' CHARACTER IS ENCOUNTERED */
00049  2       DECLARE A ADDRESS,
00050  2           (MESSAGE BASED A) BYTE;
00051  2
00052  2           DO WHILE MESSAGE <> '$';
00053  2           CALL PRINTCHAR(MESSAGE);
00054  3           A = A + 1;
00055  3           END;
00056  2
00057  2       END PRINT;
00058  1
00059  1       /* END OF TELETYPE OUTPUT SUBROUTINES */
00060  1
00061  1       /* FRACS HOLDS THE NUMBER OF 1/60THS OF A SECOND WHICH
```

```
00062  1        HAVE ELAPSED IN THE LAST PARTIAL  SECOND, WHILE
00063  1        SECS, MINS, AND HRS HOLD THE ELAPSED TIME COUNTS */
00064  1
00065  1        DECLARE (FRACS, SECS, MINS, HRS) BYTE;
00066  1
00067  1    TIMEKEEPER: PROCEDURE INTERRUPT 2;
00068  2        /* THE TIMEKEEPER PROCEDURE IS CALLED THROUGH AN EXTERNAL
00069  2        INTERRUPT (RST 2) EVERY 1/60TH OF A SECOND.  THE PROCEDURE
00070  2        UPDATES THE VALUES OF HRS, MINS, AND SECS SO THAT THE TOTAL
00071  2        ELAPSED TIME SINCE SYSTEM START-UP IS MAINTAINED IN
00072  2        BCD-PAIR FORM */
00073  2
00074  2        IF (FRACS := FRACS + 1) >= 60H THEN /* ONE FULL SECOND */
00075  2            DO;
00076  2            FRACS = 0;
00077  3
00078  3            IF (SECS := DEC(SECS + 1)) = 60H THEN /* ONE MINUTE */
00079  3                DO;
00080  3                SECS = 00H;
00081  4
00082  4                IF (MINS := DEC(MINS + 1)) = 60H THEN /* HOUR */
00083  4                    DO;
00084  4                    MINS = 0;
00085  5                    IF (HRS := DEC(HRS + 1)) = 24H THEN
00086  5                        /* ONE DAY ELAPSED */ HRS = 0;
00087  5                    END;
00088  4                END;
00089  3            END;
00090  2        END TIMEKEEPER;
00091  1
00092  1    /* SET COUNTERS TO ZERO */
00093  1    FRACS, SECS, MINS, HRS = 0;
00094  1
00095  1    /* START COUNTING TIME  */
00096  1    ENABLE;
00097  1
00098  1    /* WRITE INITIAL MESSAGE */
00099  1    CALL CRLF; CALL CRLF;
00100  1    CALL PRINT(.'** ELAPSED TIME COUNTER **$');
00101  1    CALL CRLF;
00102  1
00103  1    /* WRITE ELAPSED TIME EVERY MINUTE */
00104  1       DO FOREVER; /* OR UNTIL RESET, WHICHEVER COMES FIRST */
00105  1       IF SECS = 00H THEN
00106  2            DO; /* PRINT ELAPSED HOURS AND MINUTES */
00107  2            CALL CRLF;
00108  3            CALL PRINTCHAR(BEL); /* RING TTY BELL */
00109  3            CALL PRINTBCD(HRS); CALL PRINT(.'HOURS $');
00110  3            CALL PRINTBCD(MINS); CALL PRINT(.'MINS$');
00111  3            CALL PRINTCHAR(BEL);
00112  3            CALL CRLF;
00113  3
00114  3            /* NOTE THAT 'SECS' MUST HAVE CHANGED WHEN THE MESSAGE
00115  3            WAS SENT (ASSUMING 10 CPS TRANSMISSION RATE) */
00116  3            END;
00117  2        END;
00118  1    EOF
NO PROGRAM ERRORS
```

Figure 4—A sample PL/M program for the 8080 microcomputer

numbers. The ENABLE statement turns on the 8080 interrupt system.

At this point, the program execution must be considered in two parts: the mainline code which continues past the ENABLE statement, and the interrupt code which is executed each time an interrupt is generated. If the interrupt system had not been enabled, the mainline code within the DO FOREVER block would execute indefinitely, and, since the value of SECS remains at zero, the message

00 HRS 00 MINS

would print continuously.

Given that the interrupt system has been enabled, the interrupt which occurs 60 times each second causes the mainline code to stop at each interrupt. The TIMEKEEPER procedure immediately receives control, with the interrupt system automatically disabled and the machine state saved. Upon completion of the interrupt processing, control returns back to the interrupted mainline code to the point of interruption with the machine state restored, and interrupts enabled. As a result, the values of SECS, MINS, and HRS are continuously incremented as the mainline program executes. Thus, the program output will appear as follows:

00 HRS 01 MINS
00 HRS 02 MINS
00 HRS 03 MINS

and so-forth, with one minute intervals between each line of output.