The PC as a Productivity Tool in the Microprocessor Laboratory

I. Scott MacKenzie, *Member, IEEE*

Abstract—**The PC microcomputer can serve as a host system for developing microprocessor-based products. The educational potential for the PC is investigated citing the low cost and widespread availability of systems and PC-hosted products to facilitate hardware and software design. New laboratory methodologies result which prevent students from getting bogged down in learning the basic functions of an expensive and unique name-brand development system. By focusing lab activities on the PC and a powerful set of add-on tools, students readily undertake the synthesis of complete projects, working from concept to product.**

Introduction

COMPUTERS are everywhere. This is evident to most people, but particularly to engineers and engineering educators. In the curriculum for electrical and electronics engineering, not only are many courses "about" computers, but others are often enhanced through the use of computers as learning vehicles. This paper investigates the means and methods through which the PC family of microcomputers (e.g., the PC/XT or PC/AT) can facilitate the achievement of advanced instructional objectives in the microprocessor laboratory.

The sphere of activities in the microprocessor lab is extremely rich, ranging from hardware design, construction, and testing to software coding, translation, debugging, and integration. With such diversity, the potential for chaos looms. It is shown in the present study that the PC, acting as an "activity center," can unify many of the tasks undertaken by students in learning about microprocessors. The result is increased productivity and the possibility of undertaking more "complete" projects, working from concept to product.

The simplest example perhaps is the creation of source program files using text editing software, followed by the translation to machine language using an assembler program. The PC, or any other computer for that matter, is a powerful tool for such tasks. This is only the starting point though. As we shall see, the widespread acceptance of the PC standard, along with inexpensive utility software and add-on hardware, has made the PC a pervasive force to reckon with in many laboratory settings.

Evolution of the Microprocessor Laboratory

In this section, four distinct modes of laboratory instruction are cited which have evolved through technological developments in the computer and microelectronic fields. The mode adopted often emerges from departmental planning and budgetary constraints. When a substantial investment in equipment in made, however, the mode of instruction is set for several years, regardless of newer developments that may arrive. Hindsight is often a nagging reminder to us of what we should have done or might have done.

The following chronology suggests the availability and viability of each mode. Although each mode is found in institutions today, there is a trend toward the central use of the PC in development environments for microprocessor-based applications.

Single-Board Computers (1971–1978)

It's hard to imagine a world of electronic tools and toys without the microprocessor, yet this single-chip wonder is just approaching its twentieth birthday. In 1971, Intel introduced the 4-b 4004, widely regarded as the first microprocessor, and then followed with the 8-b 8008 and 8080. Shortly thereafter, Motorola, RCA, and then MOS Technology and Zilog, introduced similar devices: the 6800, 1801, 6502, and Z80, respectively. Alone, these IC's were rather helpless (and they remain so!), but as part of single-board computers they became the central component in useful products for learning about and designing with microprocessors. These SBC's, of which the D2 by Motorola, KIM-1 by MOS Technology, and SDK-85 by Intel are the most memorable, quickly found their way into the labs of electronics departments at colleges and universities.

As a learning vehicle priced around $400, they were perfect. The typical setup employed a monitor program in ROM and about 1 kilobyte of RAM, a hexadecimal keypad for input, a seven-segment LED display for output, and a rather arcane form of mass storage utilizing audio cassettes and an analogue interface. (Some readers may remember the "Kansas City Standard"!)

Programming involved the laborious task of "hand assembly." Assembly language programs were written with pencil and paper followed by the conversion to machine language by looking up the instruction opcodes in the IC manufacturer's tables. Programs were entered into the SBC for execution using a hex keypad and a primitive set of commands.

Although today we might balk at such inefficiency, a good argument still exists that introductory courses in microprocessors are best served using SBC's and hand assembly: 1) students learn the architecture and instruction set of the microprocessor by noting the encoding scheme for registers, operations, and addressing modes within the instruction opcodes; 2) hand coding brings the students as close to the hardware as they are ever likely to come; and 3) students learn, because of the time invested, to exercise care and adopt a methodical approach to programming [1]. These benefits may reduce the "black box" mentality that often results by beginning at too high a level. It

Manuscript received October 16, 1989.
The author is with the Seneca College of Applied Arts and Technology, North York, Ont. M2J 2X5, Canada.
IEEE Log Number 9041393.

may be overkill, however, to base an entire course on microprocessors using an SBC and hand programming. At the very least, an assembler program (to convert mnemonics to opcodes) and an effective form of mass storage are needed.

Time-Shared Minicomputers (1978–1983)

As microprocessor IC's migrated into commercial and industrial products in the late 1970's, there was a need for more productive design environments. These were usually based on powerful time-shared minicomputers with a cross assembler and other utility programs that facilitated the programming and debugging stages of design. Minicomputers were expensive though and access to such resources tended to arrive through the back door (or, should we say, through ceilings and current loops!) from computer science or administration departments.

Highly productive microprocessor learning environments have been described using the HP2000, PDP-11/70, PDP-11/750, and PDP-11/34 time-shared minicomputers with attached terminals [2]–[5]. Using a cross assembler as the basic tool, these systems permit "symbolic" programming through mnemonics and labels thus avoiding direct coding of addresses and data in binary or hexadecimal formats. Some systems also have a "debugger" and "simulator" which model the target microprocessor's architecture and "execute" a program while reporting the content of CPU registers and memory locations for the programmer's scrutiny.

Single-board computers or prototypes of microprocessor-based products are still used in these environments but share the link to the host minicomputer with the terminal. Assembled programs are downloaded from the host to the SBC for "real" execution and further debugging. With an immediate increase in productivity, more comprehensive projects are possible.

Centralized storage in the form of a hard disk is standard equipment and further unburdens students from the labors present in a lab based solely on single-board computers. Another advantage of this approach is that the minicomputer is often owned, maintained, and managed by another department at the institution [2].

Name-Brand Development Systems (1983–1987)

The ultimate tool for developing microprocessor-based products is the "development system." These all-in-one systems were introduced in the late 1970's by microprocessor chip manufacturers such as Intel (Series III) and Motorola (EXORciser), and by test instrument manufacturers such as Hewlett-Packard (HP64000) and Tektronics (TEK8001). As well as a powerful assortment of software utilities, these systems added a new tool: the in-circuit emulator (ICE). Allowing software to execute in the system-under-test under control of the development system, the ICE dramatically increases productivity and has become an essential tool for professional product development.

Along with the powerful features of development systems comes prohibitive price tags. The purchase of one commercial microprocessor development system may consume a full year's budget for an entire electrical engineering department [2]. Prices may top \$50 000 for a fully equipped workstation [7]. Other than the base price of the system, the in-circuit emulator is the most expensive component. The latest 80386 in-circuit emulators tip the scales at about \$16 000 [8].

Although prices have dropped in recent years, equipping an entire student lab with, say, a dozen name-brand development systems, is still too extravagant for most departments. Reports of development system-based labs indicate environments conducive to product development but with a limited number of stations per lab [9], [10]. At the author's college, three Intel iPDS100 development systems were acquired in 1985 and used for several years before being shelved in lieu of ten PC-based development systems. Limited resources often require that lab facilities remain open many hours each week [11] or even 24 h per day [4].

Although problems such as a long learning curve and expensive maintenance contracts have been cited [1], a major advantage of development systems is the degree of consistency offered across all tools, whether in the command-line format for programs or in the layout of the documentation. Environments which are PC-based or based on a central time-shared minicomputer, are likely to incorporate tools from different manufacturers with different user interfaces. This may frustrate students as they adapt to different environments.

PC-Hosted Development Systems (1987–present)

The date of 1987 is roughly set as the arrival of PC-hosted development systems. Due the presence of "compatibles" and "clones," prices dropped to the point where microcomputers truly became "personal." The widespread migration of microcomputers into the home and onto the desk at work caused the market to grow and become stable. At the same time, a wide variety of PC-hosted add-on products entered the market, such as EPROM programmers, cross assemblers, simulators, in-circuit emulators, etc.

The tools have arrived that make the PC the logical choice for many courses in the engineering curriculum. The rest of this paper will elaborate on the "why" and "how" of the PC-based microprocessor laboratory.

Advantages of PC-Hosted Development Tools

In this section arguments are presented for basing a microprocessor laboratory (for teaching any target microprocessor) on the PC. Advantages are categorized by cost, standardization, and methodology.

Cost

Each year the January issue of *IEEE Spectrum* features a technology update, a comprehensive survey and barometer of the electronics industry. It has been evident for the past several years that the PC has become the de facto standard for engineering: "Users are buying millions of PC/AT-bus clones each year . . . the cheapest 8088- and 8086-based PC clones stayed at \$500, and PC/AT clones at \$1200 [13]."

One need only look in a newspaper or electronics magazine, such as *Byte*, to see the cost-effectiveness of equipping a student lab with PC's. There are basically three choices: an IBM product, a compatible, or a clone. The difference between a compatible and a clone is that a compatible is designed from scratch by a major computer company, such as Olivetti or Hewlett-Packard, and reproduces the architecture of the PC (with some enhancements, presumably), whereas a clone is a chip-by-chip copy of a PC (or a gate-by-gate equivalent consolidating large sections of the design into LSI gate arrays). Although initially unreliable, clones have matured and are generally the preferred choice today. They are legal since IBM's original basic input-output system (BIOS) EPROM is omitted in favour of an alternative BIOS provided by a specialty company (e.g., Phoenix

Technologies Ltd., Norwood, MA). The operating system, MS-DOS, is also copyright and is acquired as a separate purchase.

The cost savings do not stop with the purchase of the system. There are a large number of vendors for hardware and software add-ons, with competition pushing prices down. Some useful software is even available free from on-line bulletin boards [12].

Standardization

Part of the success of the PC family is due to upward compatibility through the various models allowing customers to run old software on new systems. This is important for engineering departments that must invest in software and equipment for long-term use.

The ubiquitous PC permits students to work on lab projects outside of scheduled lab time. This is a major point since for security or insurance reasons engineering departments often limit access to labs, yet the nature of projects necessitates extra work. There are several possibilities to increase access to PC's: other labs at the institute may be available, students may have their own PC's, or students may have access to PCs at part-time jobs.

Compatibility exists not only in the system, but in the add-on products. Hundreds of companies offer PC-hosted development products, such as cross assemblers (for any target microprocessor), cross compilers (C, Pascal, Fortran, etc.), simulators and debuggers, linkers, locators, and librarians, conversion utilities, terminal emulators, file transfer programs . . . the list goes on.

Noteworthy hardware products have embraced the PC standard with bus-compatible plug-in cards. The author's institution recently retired a veritable monster of an EPROM programmer packaged in a heavy suitcase and priced at around $2000, replacing it with several PC-based EPROM programmers at $150 each. Each comes with a plug-in card and software on diskette.

PC-based in-circuit emulators have recently arrived at the sub-$2000 mark; however, average prices are still around $4000, a touch dear when equipping, say, ten stations! A recent survey [14] cites 24 manufacturers of PC-based in-circuit emulators with prices starting at $550.

In summary, just about any development product of interest is available in a PC-hosted version at a price much less than stand-alone or name-brand versions.

Laboratory Methodology

The widespread availability of PC's offers new possibilities for laboratory methodology. Experience shows that students take charge and learn MS-DOS on their own (or from their peers). This results in a tremendous shift in the nature of supervised lab activities—a shift away from learning basic system functions to the synthesis of student-initiated projects.

Since many editors are available, students can be expected to provide their own editor and to develop proficiency with it outside of scheduled laboratory time. Dedicated lab activities are not needed to teach students the operating system and editor of a unique development system. Students weaned on word processors, however, must learn to use ASCII or "nondocument" mode when editing to keep files free of formatting control codes. (They quickly learn the difference!)

When a name-brand development system or time-shared minicomputer is used, no such assumptions can be made; students tend to learn the operating system, editor, and various tools only to the extent that they are exposed to them during scheduled lab periods. When PC-based development systems are used, students learn from one another, both in the lab and outside of it.

Tools

This section examines tools for developing microprocessor-based products, categorizing them as software, hardware, or hybrid.

Software Tools

Software tools are MS-DOS commands, utility programs, or application programs. Typical MS-DOS tools are commands such as DIR, TYPE, DEBUG, COPY, and FORMAT. Attaining proficiency with MS-DOS is not a problem as students unfamiliar with MS-DOS quickly get up to speed. Capable students create their own working environment using batch files, subdirectories, macro key definitions, terminate-and-stay-resident programs, ramdrives, etc. Others catch on quickly.

Utilities are programs exclusive of MS-DOS. The main utility program is the assembler. If we assume the microprocessor of interest is a 6809, Z80, 8085, 8051, or some other IC which is not of the 8088/8086 family native to the PC, then a "cross" assembler is needed, i.e., a utility that assembles a program for a CPU other than the one in the host system. PC-hosted cross assemblers are available at around $300 for most microprocessors.

Object programs created by a cross assembler cannot execute on the PC. Another utility program, a "simulator," is needed. Program execution is simulated with the results immediately evident on the PC display. The contents of CPU registers, status bits, and memory locations are displayed, as is a disassembled listing of the program. Commands are provided to set breakpoints or execute the program in single-step mode. An example of a simulator display is shown in Fig. 1 for an 8051 product called 8051SIM (HiTech Equipment Corp., San Diego, CA).

Other utility programs include cross compilers to translate high-level language programs to the machine language of the target microprocessor, linker/locators used to combine relocatable object modules created by the cross assembler or cross compiler, and, of course, text editors or word processors. Some editors are specifically "program editors" providing automatic indenting, or an integrated environment whereby program assembly or compilation that terminates in an error automatically returns to the line in question.

Another tool is the conversion utility that translates object files to hex files or vice versa. Other programs, such as those for EPROM programming, often provide small conversion programs to bring files into the necessary format.

Fig. 2 illustrates a sequence of commands to assemble a program, type the assembled result on the console, convert the object file to a hex file, and dump the contents to the console. The object bytes in the listing file are clearly evident in the hex file. A hex file is needed when transferring an object file from the host system to the target system, EPROM programmer, in-circuit emulator, or another system. Essential information, such as load address, byte count, and a checksum, is present without any control codes (except carriage return).

A terminal emulation program is also needed to connect the PC to an SBC. Dedicated programs are available for this purpose, or communication programs such as Xmodem or Kermit may be used. Thus, the PC serves not only as a development

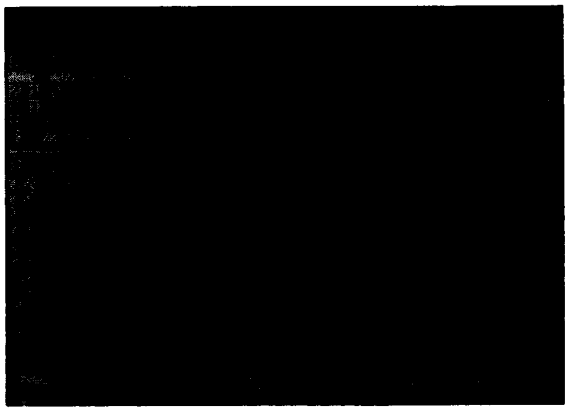


Fig. 1. A typical display for the 8051SIM simulator program. Clockwise from the top, there are separate display areas for CPU registers, internal RAM, external RAM, program memory (hex format), and program memory (disassembled format). Menu commands are at the bottom of the display. Currently active is the program command from the RAM Alter menu.

```
A:\>asm51 test.src

A:\>type test.lst

LOC  OBJ       LINE    SOURCE

0100             1               org     0100h
0100 7860        2               mov     r0,#60h
0102 7F20        3               mov     r7,#32
0104 7655        4     loop:     mov     @r0,#55h
0106 08          5               inc     r0
0107 DFFB        6               djnz    r7,loop
0109 22          7               ret
                 8               end

A:\>oh test.obj

A:\>type test.hex

:0A01000078607F20765508DFFB22AF
:00000001FF

A:\>
```

Fig. 2. The command sequence shows a program (test.src) being assembled and converted to a hex-format file. The contents of the listing file and the hex file are dumped to the console for inspection.

host, but also as a dumb terminal dedicated to the single-board computer. Many terminal emulation programs remain resident when not in use, with a single keystroke switching between terminal mode and host mode.

Hybrid Tools

Hybrid tools—those that are both hardware and software—are of two types: EPROM programmers or in-circuit emulators. These are purchased as one or more programs on diskette and a plug-in card or an external unit that connects to the PC's serial or parallel port.

Fig. 3 illustrates a PC-based development system with an EPROM programmer's zero-insertion-force socket mounted on the system chassis. A ribbon cable connects the ZIF socket to the plug-in card inside the system.

The menu provided by the EPROM programmer software (barely evident in Fig. 3) is shown in Fig. 4. Commands can be seen to select EPROM type, blankcheck an EPROM, load disk file into buffer, read EPROM contents into buffer, copy buffer contents to EPROM, verify programmed EPROM against buffer, or select a range of addresses to be programmed. The concept of a "buffer" is central to the operation of the EPROM programmer. Indeed, the use of the host system's RAM for this purpose is one of the reasons PC-hosted EPROM programmers are so inexpensive in comparison to stand-alone units (which must include a significant amount of buffer RAM as well as a CPU, serial port, etc.).

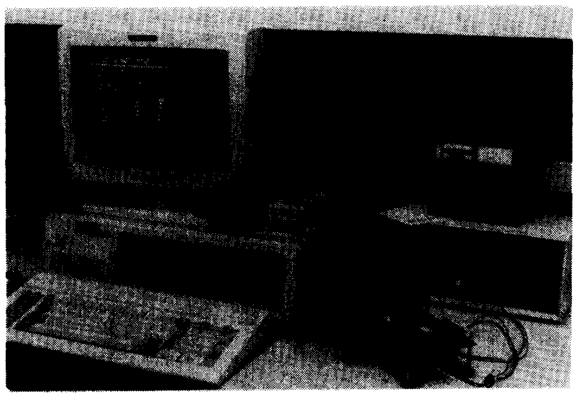


Fig. 3. A typical PC-based development system. Note the EPROM programmer on the system chassis with a ribbon cable connecting to an interface card inside the system. A single-board computer connects to the PC serial port via an RS-232C cable.

```
      Sunshine  Eprom writer card        V-5.7
        model : EW-901B, EW-904B  (C) 1986
*****************************************************
Select from the following :
  <E>: Eprom type / Vpp ---- (2764 /21V)
  <T>: Textool size -------- (1)
  <Q>: Quit.
  <L>: Load disk object file in buffer.
  <S>: Save buffer on disk.
  <D>: Display, modify, checksum or print buffer.
  <B>: Blank check.
  <R>: Read -----------  in  buffer address 0000.
  <V>: Verify --------- with buffer address 0000.
  <C>: Copy ----------- from buffer address 0000.
  <1>: Read(A) --------  in  buffer  any address.
  <2>: Verify(A) ------ with buffer  any address.
  <3>: Copy(A) -------- from buffer  any address.
  <4>: Copy(B) --------  Blank  check  and  Copy.
  <5>: Verify(B) ------ Verify  &  Display error

Result:
```

Fig. 4. Typical commands for the software accompanying an EPROM programmer.

The ultimate tool (with the ultimate price tag) is the in-circuit emulator. Fig. 5 shows an Intel iPDS100 development system with an EPROM programmer and an in-circuit emulator consisting of a bus-interface unit, a ribbon cable, and a CPU emulation pod. The pod contains a CPU wafer with "bond-out" wires allowing the development system to control the operation of application software in the system-under-test in real time, halting execution at will and displaying the contents of CPU registers or system memory.

Hardware Tools

Hardware tools consist of the development system, a serial cable, a printer, and test equipment. We may also consider the SBC and power supply as necessary hardware equipment (see Fig. 3). The SBC has a monitor program in EPROM providing a primitive set of commands to examine and change memory locations, download hex files from a host system, and begin execution of an application program.

Since the PC serial ports (COM1 or COM2) are configured as DTE's (data terminal equipment) and the SBC serial port is

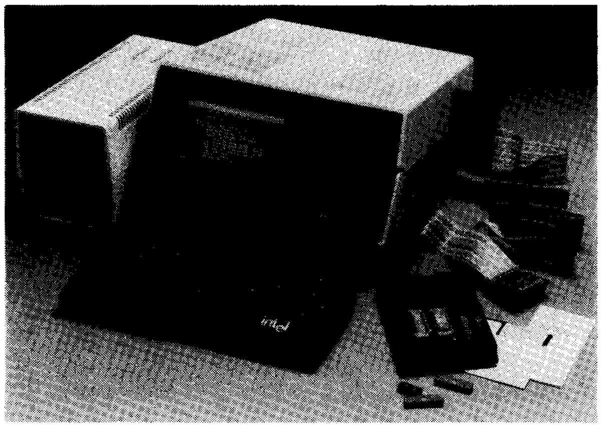


Fig. 5. An Intel iPDS100 development system with a 8051 in-circuit emulator installed. The emulator pod replaces the CPU chip in the system-under-test (Courtesy Intel Corporation).

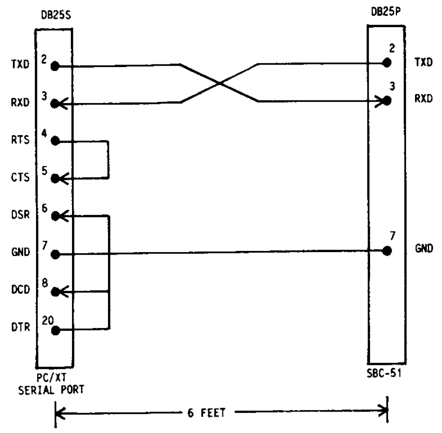


Fig. 6. Typical wiring for a cable to connect a PC serial port to a single-board computer. Wrap-back connections are needed at the PC end of the cable to fool the system into thinking a modem is attached.

most likely a DTE as well, the serial cable must cross the transmit (TxD) and receive (RxD) wires. (Such a cable is commonly called a "null modem.") As well, since the PC serial ports are configured with the full complement of modem control signals, wrap-back connections are needed to "fool" the system into thinking a modem is connected, online, and ready for data (see Fig. 6).

The Development Cycle

In progressing from concept to product, hardware and software are designed separately, with a final integration stage at the end (see Fig. 7).

One may observe that there is nothing particularly "cyclic' about the development cycle shown. Indeed, the ideal and impossible scenario of no "breakdowns" is shown. Of course problems are omnipresent. Debugging occurs at every step in the development cycle with corrections introduced by reengaging in an activity earlier in the cycle. Depending on the severity of the error, the correction may be trivial or, in the extreme, may return the designer to the concept stage. Thus there is an implied connection in Fig. 7 from the output of any stage in the development cycle to any earlier step. Each stage in Fig. 7 is now described.

Software Development

Specifying software is the task of explicitly stating what the software will do. This may be approached as a user interface problem, i.e., how will the user control the system, and what effects will be observed for each action taken? Students can be asked to produce screen layouts for the user interface before writing any code. If switches, dials, or audio or visual indicators are employed on the prototype hardware, the explicit purpose and operation of each should be stated.

Designing the software is another task that students are likely to jump into without a lot of planning. There are two common techniques for designing software prior to coding: flowcharts and pseudo-code. Flowcharts need no introduction here as they have proven a useful prelude to programming since the earliest days of Fortran. Pseudo-code is a Pascal-like language that allows a relatively free use of language in describing operations within the framework of standard looping and choice statements such as WHILE, IF/THEN/ELSE, etc. A good argument can be made for the use of pseudo-code since it enforces good programming techniques (such as the use of modules, structures, and hierarchies), and allows program design to be undertaken using a text editor rather than pencil, paper, and template [15].

The editing, translation, and preliminary testing stages of software design take the most time, at least in student projects. Errors detected by the assembler are quickly corrected by editing the source file. Run-time errors will not show up until the program is executed by the simulator or in the target system. These errors may be elusive since they require careful observation and often are uncovered only by single stepping the program or executing up to an address specified as a breakpoint. Uncovering the bug may require that the student monitor changes in memory locations, CPU registers, status bits, or input-output ports.

Hardware Development

Specifying the hardware involves assigning quantitative data to system functions.For example, a robotic arm project should be specified in terms of number of articulations, reach, speed, accuracy, torque, power requirements, etc. Students should provide a specification sheet analogous to that accompanying the purchase of an audio amplifier or VCR.

The conventional method of hardware design, employing a pencil and logic template, is often enhanced through computer-aided design (CAD) software. The learning curve for CAD programs is substantial though, and, unless specific course activities are provided, these packages are too complex for casual use.

There are pathetically few shortcuts for the labors of prototyping. Whether breadboarding a simple interface to a bus or port connector on an SBC, or wire wrapping an entire controller board, the techniques of prototyping are only developed with a great deal of practice. Such manual skills are likely to be emphasized more at the vocational level than at the university level; nevertheless, students sooner or later must put it all together and make it work. No silver bullet here!

Preliminary testing of hardware is undertaken in the absence of any application software. Stepwise testing is important—there's no point measuring a clock signal on an oscilloscope

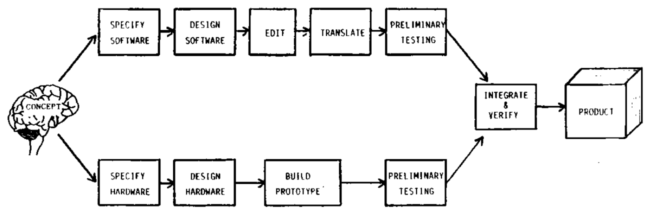


Fig. 7. The development cycle. From concept to product, there are separate paths for hardware and software development with a final step for integrating the two.

before the presence of power supply voltages has been verified. The following sequence is proposed: visual checks, continuity checks, dc measurements, ac measurements. After verifying the connections, voltages, and clock signals, debugging becomes pragmatic: is the prototype functioning as planned? If not, corrective action may take the student back to the construction, design, or specification of the hardware.

Integration and Verification

The most difficult stage in the development cycle occurs when hardware meets software. Some very subtle bugs that have eluded the simulation stage appear under real-time execution. The problem is confounded by the need for a full complement of resources: hardware such as the PC development system, target system, power supply, cables, and test equipment; and software such as the monitor program, operating system, terminal emulation program, etc.

Once a satisfactory degree of performance is obtained through execution in RAM (or via in-circuit emulation), the software is burned into EPROM and installed in the SBC as "firmware,"

Of course the truly final stages of manufacturing, marketing, and servicing are not undertaken with student projects, but they are sometimes noted and undertaken as a hypothetical step in a term project. For example, students could develop a plan for diagnosing problems in the field.

Commands and Environments

In this section the overall development environment is considered. We present the notion that at any time the student is working within an "environment" with commands doing the work. The central environment is, of course, MS-DOS. As suggested in Fig. 8, some commands return to the environment upon completion, while others evoke a new environment.

Invoking Commands

One of the drawbacks of MS-DOS is the inconsistency in user interfaces from one application to the next. The growing awareness of the Apple Macintosh's superior interface, whereby developers are expected to conform to interface specifications, is perhaps signaling an end to the winner-take-all approach to software development.

MS-DOS commands are either resident (e.g., DIR) or transient (e.g., FORMAT, DISKCOPY). Application programs are similar to transient commands in that they exist as an executable disk file and are invoked from the MS-DOS prompt. However, there are still many possibilities. Commands or applications may be invoked as part of a batch file, by a function key, or from a menu-driven user interface acting as a front-end for MS-DOS.

If command arguments are needed there are many possibilities again. Although typically entered on the invocation line following the command, some commands have default values for arguments, or prompt the user for arguments. Unfortunately, there is no standard mechanism, such as the "dialog box" used in the Macintosh interface to retrieve extra information needed for a command or application. Some applications, such as word processors, "take over" the system and bring the user into a new environment for subsequent activities.

Environments

As evident in Fig. 8, some software tools such as the simulator, ICE, or EPROM programmer evoke their own environment. Learning the nuances of each is a problem for beginners due to the great variety of techniques for directing the activities of the environment: cursor keys, function keys, first-letter commands, menu highlighting, default paths, etc. Often hard copy manuals are poor or nonexistent. This is consistent with the general trend away from manuals in lieu of "user friendly" and "help-driven" interfaces.

Methodology

As research in artificial intelligence and cognitive science has discovered, modeling human "problem solving" is a slippery business. It is of little benefit to force students to adhere to the step by step approach to development shown in Fig. 7. Humans appear to approach the elements of a situation in parallel, simultaneously weighing the relative importance of possible actions and proceeding by intuition. The methodology suggested here is to show the students the development cycle to clarify the use of tools and the techniques to realize each step, but to leave the overall process to the individual. Past efforts to serialize student work in a stepwise manner have not worked; students solve problems using a personal style and meet the instructor's checklist later. Although specific documentation requirements and deadlines are necessary, the day-to-day process is best left to the individual. When "breakdowns" occur, the tools become exploratory vehicles to discover and correct problems.

The basic operation of commands is "translate," "view," or "evoke a new environment." Students should view the results of translation to verify results. They can be told, in jest, not to believe the outcome of any translation (assembling, EPROM programming, etc.), and to verify everything by viewing results. Tools for viewing are commands such as DIR (were the expected output files created?), TYPE (what's in the output file?), edit, print, etc.

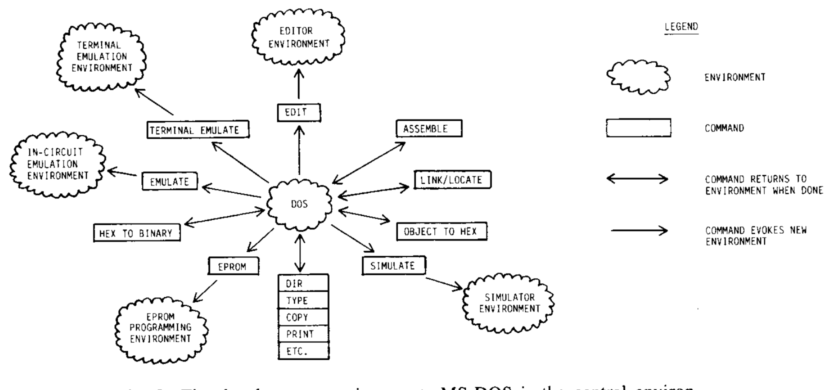


Fig. 8. The development environment. MS-DOS is the central environment and most commands return control to it upon completion. Some commands or applications evoke a new environment with their own set of controls and commands.

Advanced Tools and Methodologies

The approaches presented in this section must be evaluated with caution. In many instances the low level and primitive development environment is of prime educational importance in developing computer literacy. Just as high-level languages buffer programmers from hardware details, providing engineering students with sophisticated development environments often shields them from many of the problems they must eventually face in the working world when they are required, for example, to integrate a variety of software and hardware products into a working system. Although such tools may be the key to success in professional environments, they may end up akin to a security blanket when used by students.

Menu-Driven Environments

The simplest enhancement, perhaps, is in the use of a menu-driven environment that eliminates invoking commands from the MS-DOS prompt. These programs, such as QUICK DOS (Gazelle Systems, Provo, UT), provide the user with a full screen of commands, and a listing of the currently selected directory. Navigating about the system or executing commands is performed using the cursor keys or by entering the first letter of menu commands.

Although such environments can enhance productivity, they also introduce an extra layer of abstraction which hides a system's underlying behavior. Generally this is not a problem. Students using these programs have a good understanding of MS-DOS to begin with; others use MS-DOS and migrate to these environments when they are ready.

Development Using High-Level Languages

The introduction of high-level languages in microprocessor-based designs has always been controversial. Although development time may drop, learning time is longer and the compiled code is longer and slower than an equivalent assembly language program. Efficient programming languages, such as C, combined with optimizing compilers can, however, come close to producing code as efficient as assembly language programs. Regardless, the best of both worlds can be obtained by coding "critical sections" in assembly language and merging the result with sections written in the high-level language.

Educators have long debated the merits of using high-level languages in microprocessor courses. The literature reveals a moderate use of languages such as Basic [16], PL/M [17], and Pascal [18], with some educators even predicting the demise of assembly language [19].

As with the use of menu-driven environments, the main drawback with using high-level languages lies in the degree of abstraction. For engineering students, it is important to remain as close to the hardware as possible until a deep understanding of the architecture of microprocessors and microcomputers is developed. For this reason, assembly language is the best choice, at least initially.

Other PC-based Tools

A range of PC-based test equipment has emerged in recent years, including voltmeters, frequency counters, and digital pattern generators. As with PC-based EPROM programmers, designing these products around a PC negates the need for much of the internal circuitry which contributes to their high cost. Do not hold your breath though: the $1000, 500 MHz PC-based oscilloscope is not imminent.

A tool of great interest to electronics educators is the logic analyzer. Although stand-alone units have a history of stratospheric price tags (The Gould K450M costs $287 000!), the PC-based logic analyzer is now a practical reality. A recent survey [20] cites products by 18 manufacturers with prices ranging from $495 to $104 000, with average prices just under $2000. It is interesting that none of the companies surveyed make a logic analyzer hosted by the Apple Macintosh. The Mac simply has not made a mark in engineering applications.

The possibility of the PC acting as the host for the development tools as well as the test equipment, suggests the possibility of powerful development scenarios. Perhaps data gathered on system activities could be analyzed by automatic debugging, code generating and optimizing programs. Bugs could be found and new code generated. Such expert systems, like language translators, would only provide a rough first approximation; human processes would be required to complete the task.

Networking

A network contains one or more "file servers" and multiple "workstations." The workstations are PC's with plug-in local area network (LAN) cards, costing around $300, with a coax or twisted-pair links forming a "bus" or "ring" arrangement. Link speed is typically 10 Mbps.

One advantage is that mass storage is centralized in one powerful and expensive system, the file server, with the workstations remaining modest, e.g., a PC/XT with one or no floppy disk drive. The file server can be a PC/AT with 60-300 Mbytes of hard disk storage. The network approach may be cheaper than putting a hard drive in each system if such a configuration is being considered.

New possibilities exist with networked environments: interstation file transfers, cooperative work, e-mail, teacher monitoring of network activities, connection to BITNET for intercollege communications, etc. A dependency on the fileserver, however, means network activities may be halted or restricted in the event of a break down.

CAD Tools

As mentioned earlier, various computer-aided design packages are available to expedite the drafting and documentation of a design. Although many CAD tools are for the mechanical or civil engineering disciplines, some are specifically geared for electronic engineering. The two most common types are schematic drawing software such as SCHEMA (Omation, Inc., Richardson, TX) and printed circuit board (PCB) layout software such as smARTWORK (Wintek Corp., Lafayette, IN). Although these programs have a long learning curve, the results are impressive. Some schematic drawing programs produce files which can be read by PCB programs to automatically generate a layout.

Conclusion

The PC has been cited throughout this paper as a useful tool for centralizing student activities in the microprocessor laboratory. A good indication that the trend cited here is likely to continue is that the IC and test equipment manufacturers, such as Intel and Hewlett-Packard, now offer a selection of PC-hosted products. Cross assemblers and in-circuit emulators, once only available on expensive name-brand development systems, are now available in PC-hosted versions, even from major manufacturers. There is a clear trend toward the use of the PC as the host system in development environments incorporating a broad range of inexpensive add-on products. For educators, therein lies a unique opportunity to expand and enhance the instructional objectives in the microprocessor laboratory.

References

[1] C. E. Strangio, "Microprocessor instruction in the engineering laboratory," *IEEE Trans. Educ.*, vol. 31, pp. 172-176, Aug. 1988.

[2] J. H. Aylor, "Creating a microcomputer facility," *IEEE Trans. Educ.*, vol. E-24, pp. 47-50, Feb. 1981.

[3] D. J. Ahlgren, "A microprocessor laboratory work station," *IEEE Trans. Educ.*, vol. E-24, pp. 59-62, Feb. 1981.

[4] D. J. Ahlgren, "Synthesis of a small microcomputer—A project for undergraduate laboratories," *IEEE Trans. Educ.*, vol. E-28, pp. 65-68, May 1985.

[5] D. B. Gill and V. B. Hass, "Purdue's junior-level control laboratory with microprocessors," *IEEE Trans. Educ.*, vol. E-24, pp. 82-86, Feb. 1981.

[6] J. R. Glover and J. D. Bargainer, "Integrating hardware and software in a computer engineering laboratory," *IEEE Trans. Educ.*, vol. E-24, pp. 22-27, Feb. 1981.

[7] R. M. Lumley, "An industrial microcomputer education program," *IEEE Trans. Educ.*, vol. E-24, pp. 136-141, May 1981.

[8] D. Strassberg, "In-circuit emulators ease development of hardware for 80386-based applications," *EDN*, vol. 33, no. 5, pp. 69-73, Mar. 3, 1988.

[9] C. T. Wright, Jr., "An embedded computer systems laboratory," *Front. Educ. Conf. Proc.*, pp. 57-60, 1987.

[10] J. Bowen, "Software/hardware integration on microprocessors," *Microprocess. Microsyst.*, vol. 9, no. 6, pp. 8-14, Jan./Feb. 1985.

[11] R. J. Distler, "In-circuit emulators in the microprocessor laboratory," *IEEE Trans. Educ.*, vol. E-30, pp. 250-252, Nov. 1987.

[12] S. Ciarcia, "Why microcontrollers—part 1," *Byte*, pp. 239-245, Aug. 1988.

[13] J. Voelcker, "Technology '89—Personal computers," *IEEE Spectrum*, vol. 26, pp. 31-34, Jan. 1989.

[14] J. Novelino, "Focus on in-circuit emulators," *Electron. Design*, vol. 35, no. 27, pp. 139-144, Nov. 12, 1987.

[15] I. S. MacKenzie, "A structured approach to assembly language programming," *IEEE Trans. Educ.*, vol. 31, pp. 123-128, May 1988.

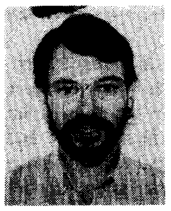
[16] F. DeCesare, S. M. Bunten, and P. M. DeRusso, "Microcomputers for data acquisition, control, and automation—A laboratory course for preengineering students," *IEEE Trans. Educ.*, vol. E-28, pp. 69-75, May 1985.

[17] J. F. Poiraudeau and S. Roux, "PL/M Code mixing on single-board systems," *Microprocess. Microsyst.*, vol. 8, no. 9, pp. 498-500, Nov. 1984.

[18] A. E. G. El-Dhaher, T. S. Hassan, and A. J. Ahmed, "Linking assembly code to PASCAL programs for microprocessors," *Microprocess. Microsyst.*, vol. 8, pp. 29-33, Jan./Feb. 1984.

[19] T. W. Schultz, "Teaching dedicated micros 14 years from now," *Front. Educ. Conf. Proc.*, pp. 582-585, 1987.

[20] D. Strassberg, "PC-based logic analysers," *EDN*, vol. 33, No. 21, pp. 134-148, Oct. 13, 1988.



I. Scott MacKenzie received the B.A. degree in music from Queen's University in 1975, the Diploma in electronic engineering technology from Durham College in 1978, and the M.A. degree in education from the University of Toronto in 1989.

He is currently a Ph.D. candidate in the Department of Education (computer applications) at the Ontario Institute for Studies in Education, University of Toronto. He joined the Department of Electronics, Seneca College of Applied Arts and Technology in 1983 where he is currently Professor of Computer Engineering Technology. His research interests include performance modeling for human-computer interaction, user interface design, development environments, and educational technology.

Mr. MacKenzie is a member of the IEEE Computer Society and Education Society.