PLZ: A Family of System Programming Languages for Microprocessors

Charlie Bass
Zilog, Incorporated

There is an art to the practice of any science, a point argued cogently by Donald E. Knuth,[1] author of the series of volumes deliberately titled *The Art of Computer Programming*. If "science" is knowledge which has been logically arranged and systematically codified, then "art" refers to the use of personal skill, guided by a sense of aesthetics, in applying these organized principles, whether they describe engineering, physics, mathematics, or computer programming. For, at a given stage in the translation of an art into the organized body of the corresponding science, that which is still art contains intuitive and aesthetic factors which defy precise formalization. Computer programming—with its scope extending from "arty" folklore to science-based automatic code generation and verification—is a prime example of these subtle differences. Now widely called computer *science*, the *art* of programming continues to challenge and often baffle the most scientific of managers and "computer scientists."

Programming languages are the tools with which computer scientists practice their particular blending of art and science. As every artisan and craftsman knows, having the right tools affects both the work and the results. The abundance of programming languages is an indication, in part, of the desire to provide the right tool for a myriad of programming situations.

The PLZ family of languages is intended to meet the needs of a relatively new experience—microcomputer system programming. At the core of each language is a common kernel which is expanded to make each separate language. The several languages of the family can each be applied to the tasks for which it is best suited and then later linked together into a single program.

In this article, after a perspective view of the setting for new language development, the key objectives which governed the development of PLZ are examined. Then the principal features of the new system-programming-language family are described. Two individual languages have been designed and are in use. Others are under consideration.

Perspective

The advent of microprocessor technology has extended the challenge of controlling software development into a wider domain and has introduced an abrupt increase in the concern for the economics of software development.[2] Whereas the cost of software often equals that of hardware in megacomputer systems, it can totally dominate the economics of a microcomputer-based system. Indeed, the fundamental issue which drives the current interest in "structured programming" is improvement in the program development process.[3] In this context, improvement is usually measured in terms of reliability, time to completion, and flexibility. One of the central issues in the study of structured programming is the effect which the programming language has on the program development process. While the results of this study are inconclusive, the evidence seems to indicate that Algol-like languages, which organize programs into procedures and blocks, tend to reinforce desirable programming practices. Linear languages, such as Fortran, Basic, and assembly language, require greater discipline to achieve coherent structure. This suggests that within a fixed time and money budget, Algol-like languages are generally better.

Language levels. A crude taxonomy of programming languages can be based on their degree of machine independence and on the types of problems for which they are best suited. Languages whose statements model machine operations are called low-level languages. Assembly language is the quintessential low-level language. Languages whose statements are algorithmic rather than machine oriented are called high-level languages, e.g., Fortran. Among high-level languages, those whose primitive operations manipulate arithmetic expressions, character strings, and I/O streams, and which are well suited for problem solving in science and business, are referred to as application languages. High-level application languages have been widely accepted because they apply to the predominant category of megacomputer programming activity and because high-level languages naturally satisfy the broad needs of this category.

High-level system languages are intended to provide algebraic notation and high-level control structures for such things as process scheduling and resource management, while retaining both efficiency and the capability for basic machine operations. The conflict of these high-level and low-level objectives accounts for the limited acceptance of system languages compared to the impact of application languages. The predominant use of microcomputers clearly falls into the domain of system programming even though a significant amount of application programming is emerging.

As with megacomputers, most of this system programming activity has been done in assembly language. The traditional assortment of application language translators (Fortran, Basic, Cobol, and PL/M,[4] a dialect of PL/I) has been implemented for microprocessors for a variety of reasons: 1) regardless of how appropriate they may or may not be for a given problem, these languages are familiar; (2) the techniques for compiling or interpreting these languages are well established; and 3) a few problems being solved with microprocessors are, in fact, application problems. Also, as with megacomputers, some microprocessor system problems have been solved with limited success using the application languages which are available.

System language. It is interesting that so little microprocessor programming has been done using established system languages such as BCPL, C, Simula, or Bliss. This is probably a result of their limited circulation in the computer science milieu, their requirement for large, complex compliers, the limited efficiency of the code they produce, and the size of the run-time package necessary during program execution. All of this could have been said 15 years ago about high-level languges and computers in general.

Objectives

A system programming language designed especially for microcomputers should have the following characteristics:

Reinforce good programming practices. Both in form (syntax) and in meaning (semantics), a high-level language can facilitate the programming process by being readable, clearly defined, and natural for the representation of algorithms. A language whose syntax is complicated by excessive, illogical, or irregular notation is difficult to learn and leads to repeated compilation errors. A language whose semantics are unclear can lead to obscure logical errors. A language whose primitive operations are not suitable for representing the solution to a problem can introduce errors in mapping from the known solution to a computer program.

Manage computing resources. The details of resource management (register and memory allocation in particular), both during the creation and the execution of a program, are a critical aspect of the programming process. By managing these details, a programming language can free the programmer to think more about the problem to be solved and less about the state of the machine. On the other hand, if the management of resources is either inappropriate or inefficient for data structures, the language can interfere with the programming process. Ideally, the programmer should be able to control resource management to the degree justified by the circumstances.

Allow access to the architecture of the machine. Most microprocessor applications require precise control of the machine and sometimes require its full operational capability. Forcing these precision requirements through the filter imposed by high-level-language constructs can be awkward and prohibitive. All of the primitive elements and operations of the machine which are available through assembly language must be accessible. Otherwise, the barrier created by the language can prevent a viable solution from being achieved.

Produce efficient code. While the costs associated with computer memory continue to drop dramatically, memory costs remain one of the critical items in determining the economic feasibility of a microprocessor application due to the multiplier effect applied to these costs when the system is replicated. It is often said that microprocessor manufacturers supply CPU's in order to promote memory sales. By knowing the efficiency of a particular language translator, and by quantifying the expense of the required memory versus the overall program development costs, it is possible to determine the cross-over point at which it is advantageous to use a high-level language instead of an assembly language.[5] In general, the fewer times a system is to be replicated, the more likely that a high-level language is appropriate. By improving translation efficiency, this cross-over point occurs at a higher replication factor, thus extending the viability of high-level-language programming to more and more applications.

Be relatively easy to compile. Certain characteristics of the translation process are critically important in the microprocessor environment. First of all, the compiler should run on the target microprocessor. Otherwise, the user is confronted by the expense and complexity of first running the compiler on a host computer and then transferring the results to the target machine. Second, the speed of the translation process directly and indirectly affects program development: Directly, the time it takes to correct problems in a program is influenced by compilation time. Indirectly, if the translation turnaround time is excessive, the programmer is inhibited from using the compiler to its maximum benefit and may resort to debugging strategies which offset the advantages of using a high-level language.

Application language design for megacomputers has principally emphasized potent programming constructs because they offer potential savings in the most costly aspect of program development—the human factor. Access to the architecture of the machine is usually unnecessary and, in fact, undesirable. Efficient code is certainly important in megacomputer programs, but it is far more critical in the microcomputer world. The code which microcomputer compilers produce is being scrutinized much more than that of their megacomputer counterparts. While a formidable theory and practice of code optimization exists, it trades generated code efficiency for compiler complexity. This may be an acceptable tradeoff in the megacomputer domain, but not for smaller and slower microcomputers. Here the size and performance of the compiler represent constraints equally as stringent as code efficiency.

While system programming languages for megacomputers have been concerned with these same objectives, they have been achieved in terms inappropriate for microcomputers. Because of the storage capacity and processor speed of megacomputers, code efficiency, as stated, has been justifiably attained at the expense of compiler complexity.[6]

The PLZ family

The PLZ family of languages is an attempt to appropriately balance the language objectives listed above for the express purpose of microcomputer system software development. Each language in the PLZ family is based on the kernel grammar described in Table 1. Two members of the family (PLZ/ASM and PLZ/SYS) are briefly discussed here as examples. Programming style and resource management are governed by conventional high-level language constructs. Procedure oriented, the PLZ family has a syntactic and semantic style which blends elements of Algol, Pascal, and Mesa to form coherent, easy-to-learn languages. Direct access to the architecture of the machine is possible by selecting the PLZ language (PLZ/ASM) which includes assembly instructions. PLZ compilers can produce efficient code, and the translation process is relatively simple because the kernel grammar has been carefully designed to contain high-level statements in such a form that they can be translated easily and mapped into efficient machine-language sequences.

Modular basis. A PLZ program is a set of modules; a module is the basic unit of translation.

Table 1. PLZ kernel grammar.

module	=>	module__identifier
		MODULE
		declaration*
		END module__identifier
declaration	=>	externals
	=>	globals
	=>	internals
externals	=>	EXTERNAL
		identifier+
globals	=>	GLOBAL
		identifier__declaration+
internals	=>	INTERNAL
		identifier__declaration+
identifier__declaration	=>	identifier+ type
		[':=' '('constant__expression+')']
	=>	procedure
type	=>	simple__type
	=>	ARRAY '['constant__expression type']'
	=>	RECORD '['identifier__declaration+']'
procedure	=>	procedure__identifier
		PROCEDURE
		local__declaration*
		[ENTRY
		statement*]
		END procedure__identifier
local__declaration	=>	LOCAL
		identifier__declaration+
statement	=>	if__statement
	=>	select__statement
	=>	do__statement
	=>	exit__statement
	=>	repeat__statement
	=>	procedure__statement
	=>	return__statement
if__statement	=>	IF condition
		THEN statement*
		[ELSE statement*]
		FI
select__statement	=>	IF selector
		select__element*
		[ELSE statement*]
		FI
select__element	=>	CASE constant__expression+ THEN
		statement*
do__statement	=>	[label__identifier]
		DO
		statement*
		OD
exit__statement	=>	EXIT [FROM label__identifier)
repeat__statement	=>	REPEAT [FROM label__identifier]
return__statement	=>	RETURN

A module consists of data declaration and units of execution called procedures. While inter-modular communication is allowed, references across module boundaries are intended to be less frequent than those within a module. This serves to localize the scope of attention of the programmer and reinforces "information hiding" as suggested by Parnas.[7] Furthermore, it is intended that a module (code and data) serve as a unit of overlay for programs that need not be wholly present in main memory during execution. Isolating high-frequency references within a module reduces the potential for transfers between secondary and primary memory.

Symbol classes. Symbol declarations for modules separate into three categories (Figure 1). *External* declarations refer to identifiers which are defined in other modules. *Global* declarations refer to variables or procedures which reside in the module in which they are declared and are accessible by other modules. *Internal* declarations refer to variables or procedures which reside in the module in which they are declared and are private to that module.

Procedures. A procedure consists of the declaration of local variables and a sequence of executable statements. Conditional or selective execution is controlled by the *if* statement and the *select* statement. They are analogous to conventional conditional and case statements.

The framework provided for repetitive statements is the *do* statement. Statements between the symbols DO and OD are executed repeatedly until control is diverted through an *exit*, *repeat*, or *return* statement. The *exit* statement causes execution to continue at the first statement following the DO...OD block which contains the *exit* statement, whereas the *repeat* statement causes execution to continue at the first statement of the DO...OD block which contains the *repeat* statement. Furthermore, the *exit* and *repeat* statements may be qualified by a label indicating a specific block to which, or from which, execution continues.

The kernel contains no constructs for the manipulation of data; thus, it is not itself a programming language. It is by the addition of data manipulation to the kernel that a PLZ language is defined. Thus far, two PLZ languages, PLZ/ASM and PLZ/SYS, have been implemented.

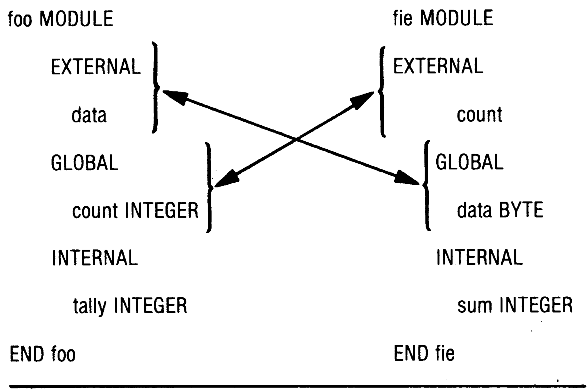


Figure 1. Module declarations.

PLZ/ASM

PLZ/ASM is a low-level system programming language similar in concept to PL/360,[8] built by adding assembly language instructions to the kernel framework. The assembly language is that of the Zilog Z80 microprocessor.[9] The control statements in the kernel primarily generate test and branch instructions which do not interfere with the programmer's control of register values or condition codes. The ideal use of PLZ/ASM avoids the assembly versions of branches altogether. The program in Figure 2 is written in PLZ/ASM.

This blending of high-level data declaration and control structure with low-level assembly instructions creates a balance between desirable programming practices and machine-dependent operations. Furthermore, the compatibility of PLZ/ASM modules with other members of the PLZ family provides convenient and efficient access to low-level processes without reducing entire programming tasks to this level.

PLZ/SYS

PLZ/SYS is a high-level system programming language with no direct access to machine operations.[10] The principal construct added is an assignment statement with conventional algebraic notation. The procedure statement is a special form of the assignment statement which causes the execution of the procedure denoted by the procedure identifier and the assignment of any returned values. The declaration of the procedure may include a RETURNS list, in which case the number of variables in the list must be equal to the number of variables on the left-hand side of the assignment operator. An assignment operator is not used in the case of a procedure which returns no values. The procedure statement may contain a list of actual parameters which are assigned or bound to the corresponding formal parameters declared in the procedure declaration. Parameters are passed to the procedure by value only—i.e., the formal parameter is treated as a local declaration of a variable whose value is assigned from the actual parameter list upon entry to the procedure. The program in Figure 3 is written in PLZ/SYS.

Strong type checking in the style of Mesa[11] is enforced with data types being either one of the five predefined simple types (BYTE, WORD, INTEGER, SHORT_INTEGER, and pointer), one of the two structured types (ARRAY and RECORD), or the name of a user-defined type. Pointer types and user-defined types are modeled after their counterparts in Pascal.[12] Data allocation is static except for local data requirements which are dynamically allocated to enable reentrant procedures.

Conclusion

PLZ/ASM and PLZ/SYS are examples of a low-level and a high-level system language, respectively, from the PLZ family. A task can be partitioned into PLZ/ASM and PLZ/SYS modules depending on its low-level versus high-level requirements. Once individual modules (written in different languages of the PLZ family) have been translated into relocatable machine code, they can be linked together into a single program. The static and dynamic linkage conventions between modules are facilitated by the syntactic similarity of data declarations throughout the family.

```
bubble__sort MODULE
  CONSTANT
    flag: = 0             ! bit position of switch indicator !
  INTERNAL
    y ARRAY [10 BYTE]
      := (33  10 200  40  41   3   3  33  50 199)
    sort PROCEDURE
      ! This procedure uses a standard exchange (bubble) sort
        algorithm to sort the elements of array 'y' in ascending
        order. At entry, register b contains 'n', the number
        of elements to be sorted !
      ENTRY
        DO
          res    flag,c  ! switched := false !
          ld     ix,y    ! y[i] address with i=0 !
          ld     d,b
          dec    d       ! limit := n-1 !
          DO
            IF ZERO      ! test i > limit !
              THEN EXIT
            FI
            ld    e,(ix)
            ld    a,(ix+1)
            cp    e                ! test y[i] > y[i+1] !
            IF CARRY
              THEN
                ld      (ix),a     ! interchange !
                ld      (ix+1),e   ! y[i] and y[i+1] !
                set     flag,c     ! switched := true !
            FI
            inc   ix     ! i := i+1 !
            dec   d
          OD
          bit    flag,c
          IF ZERO        ! test if switched !
            THEN RETURN
          FI
        OD
    END sort
    main PROCEDURE
      ENTRY
        ld      b,10     ! set n !
        sort
        RETURN
    END main
END bubble__sort
```

Figure 2. Sample PLZ/ASM program.

```
bubble__sort MODULE
  CONSTANT
    true := 1
    false := 0
  INTERNAL
    y ARRAY [10 INTEGER]   ! unsorted array !
      := (33  10 2000 400 410   3   3  33 500 199)
    sort PROCEDURE (n BYTE)
      ! This procedure uses a standard exchange (bubble) sort
        algorithm to sort the elements of array 'y' into ascending
        order. The value of the parameter 'n' is the number of
        items to be sorted!
      LOCAL
        i j limit BYTE       ! indices for stepping through array !
        temp INTEGER
        switched BYTE        ! flag to indicate completion !
      ENTRY
        limit := n-2         ! stopping point for index !
        DO
          switched := false  ! initialize flag to indicate no !
                             ! changes !
          i := 0             ! start at first array element !
        DO
            IF i > limit     ! exit if pass through array complete !
              THEN EXIT
            FI
            j := i + 1          ! index of adjacent element !
            IF y[i] > y[j]      ! compare adjacent elements !
              THEN
                temp  := y[i]       ! interchange elements !
                y[i]  := y[j]
                y[j]  := temp
                switched := true    ! indicate switch !
            FI
            i := i + 1              ! advance to next pair !
          OD
          IF NOT switched           ! return if no changes !
            THEN RETURN
          FI
        OD
    END sort
    main PROCEDURE
      ENTRY
        sort (10)                   ! invoke sort procedure !
        RETURN
    END main
END bubble__sort
```

Figure 3. Sample PLZ/SYS program.

Other PLZ family members are being contemplated—one with complete dynamic allocation for list processing, one with an extensive math package, one for coordinating concurrent processes, one for graphics, and one for text processing. Having simple specialized languages similar in concept to the SIMPL family[13] is more desirable than collecting these features into a single language. ■

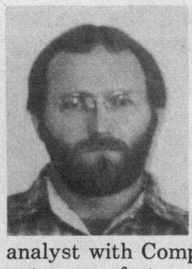
Acknowledgments

Several people have contributed to the PLZ family concept and the details of its members. Principal among these is William M. McKeeman, who has influenced virtually all aspects of the work. Tod Snook has coordinated the implementation effort

and contributed countless suggestions for improving the integrity of the design. Janet Roberts formulated a precise specification for PLZ/SYS from a somewhat chaotic collection of ideas. Bill Lane, Steve Meyers, and Armen Nahapetian have served as dependable implementers throughout the project.

References

1. D. Knuth, "Computer Programming as an Art," *CACM*, Vol. 17, No. 12, December 1974, pp. 667-673.
2. C. Bass and D. Brown, "A Perspective on Microcomputer Software," *Proc. IEEE*, Vol. 64, No. 6, June 1976, pp. 905-909.
3. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972, 220 pp.
4. G. A. Kildall, "High-Level Language Simplifies Microcomputer Programming," *Electronics*, June 24, 1974, pp. 103-109.
5. J. Gibbons, "When to Use Higher-Level Languages in Microcomputer-Based Systems," *Electronics*, August 7, 1975, pp. 107-111.
6. W. A. Wulf et al., *The Design of an Optimizing Compiler*, Elsevier, New York, 1976.
7. D. Parnas, "A Technique for Software Module Specification," *CACM*, Vol. 15, No. 5, May 1972, pp. 330-336.
8. N. Wirth, "PL/360, A Programming Language for the 360 Computers," *JACM*, January 1968.
9. *Z80-CPU Technical Manual*, Zilog, Inc., Cupertino, California, 1976.
10. "Report on the Programming Language PLZ/SYS," Zilog, Inc., Cupertino, California, 1977.
11. C. M. Geschke, J. H. Morris, Jr., and E. H. Satterthwaite, "Early Experience with Mesa," *CACM*, Vol. 20, No. 8, August 1977, pp. 540-553.
12. N. Wirth, "The Programming Language Pascal," *Acta Informatica*, Vol. 1, 1971, pp. 35-63.
13. V. R. Basili, "The SIMPL Family of Programming Languages and Compilers," Department of Computer Science, University of Maryland, 1976.



Charlie Bass is director of software at Zilog, Inc., and is engaged in the development of microcomputer systems. As a lecturer at Stanford University and recently a visting assistant professor at UC Santa Cruz, he teaches programming linguistics and compiler construction. Earlier at UC Berkeley he was a lecturer in data structures and operating systems. As a systems analyst with Computer Usage Co., he worked with computer manufacturers in the US and Japan on the construction of operating systems and compilers.

He received a BS in physics from the University of Florida in 1964, an MS from the University of Miami, and a PhD in electrical engineering from the University of Hawaii. His doctoral research led to the development of the University of Hawaii Time-Sharing System to support the Aloha System Computing Network.

Reader Service Number 4