GKILDALL.WS4

List of Gary Kildall texts compiled by Emmanuel ROCHE.

1968

- "Experiments in large-scale computer direct access storage manipulation"
  Thesis for Master of Science, University of Washington
  December 1968 (Thesis No.17341)
  (ROCHE> Retyped: GKMS.WS4)

1969

- "Experiments in large-scale computer direct access storage manipulation"
  Technical Report No.69-01, Computer Science Group
  University of Washington, 1969
  (ROCHE> Missing...)

1970

- "APL\B5500: The language and its implementation"
  Technical Report No.70-09-04, Computer Science Group
  University of Washington, September 1970
  (ROCHE> Retyped: GKAPL.WS4)

- "The ALGOL-E Programming System"
  Internal Report, Mathematics Department,
  Naval Postgraduate School, Monterey, California
  December 1970
  (ROCHE> Missing...)

1971

- "A Heathkit method for building data management programs"
  Gary Kildall & Earl Hunt
  ACM SIGIR Information Storage and Retrieval Symposium
  1971, pp.117-131
  (ROCHE> Retyped: GKEH.WS4)

1972

- "A code synthesis filter for basic block optimization"
  Technical Report No.72-01-01, Computer Science Group

University of Washington
January 1972
(ROCHE> Missing...)

- "ALGOL-E: An experimental approach to the study of programming languages"
Naval Postgraduate School, Monterey, California
NPS Report NPS-53KG72 11A
January 1972
(ROCHE> Missing...)

- "ALGOL-E: An experimental approach to the study of programming languages"
Gary Kildall & Alan Roberts
ACM SIGCSE Bulletin Vol.4, No.1 (March 1972), pp.127-135
(ROCHE> Retyped: GKALG.WS4)

- "Global expression optimization during compilation"
Thesis for Ph.D. in Computer Science, University of Washington
May 1972 (Thesis No.20506)
(ROCHE> I have it, but not yet retyped)

- "Global expression optimization during compilation"
Technical Report No.72-06-02, Computer Science Group
University of Washington, June 1972
(ROCHE> Missing...)

1973

- "A unified approach to global program optimization"
ACM First Symposium on Principles Of Programming Languages (POPL)
Boston, Massachussetts, October 1973, pp.194-206
(ROCHE> I have it, but not yet retyped) (Lots of Maths symbols...)

1974

- "High-level language simplifies microcomputer programming"
"Electronics", June 27, 1974, p.103
(ROCHE> Retyped: GKHLL.WS4)

- "System languages: management's key to controlled software evolution"
Proceedings of the 1974 western electronics show and convention
(WESCON), September 1974
(ROCHE> Retyped: GKSL.WS4)

1975

- "CP/M: A disk control program for microcomputer system development"
"Journal of Microcomputer Applications", June 1975

(ROCHE> Missing...)

- "Microcomputer Software Design: A Checkpoint"
  Proceedings of the Fall Joint Computer Conference
  Spartan Books, New-York, 1975
  (ROCHE> Retyped: GKCHK.WS4)

1976

1977

1978

- "A simple technique for static relocation of absolute machine code"
  DDJ, #22, Vol.3, No.2, February 1978, pp.10-13
  (ROCHE> Retyped: GKPRL.WS4)

1979

1980

- "The evolution of an industry: One person's viewpoint"
  DDJ, #41, Vol.5, No.1, January 1980, pp.6-7
  (ROCHE> Retyped: GKEI.WS4)

1981

- "CP/M: A Family of 8- and 16-bit Operating Systems"
  BYTE, June 1981, p.216
  (ROCHE> Retyped: GKBYTE.WS4)

1982

- "PL/I for limited resource computers"
  "Microsystems", Jan/Feb 1982, pp.28-29
  (ROCHE> Retyped: GKLRC.WS4)

- "PL/I-80"
  "Interface Age", June 1982, p.71
  (ROCHE> Retyped: GKPLI.WS4)

- "Running 8-bit software on dual-processor computers"
  "Electronic Design", September 16, 1982, p.157

(ROCHE> Retyped: GKED.WS4)

1983

1984

1985

1986

- "The compact disk ROM: applications software"
  Tim Oren & Gary Kildall
  "IEEE Spectrum", Vol.23, No.4, April 1986, pp.49-54

EOF

CSLABUW.WS4 (= Computer Science LABoratory at University of Washington)

- "The Computer Science teaching laboratory at the University of Washington"
  Earl Hunt
  ACM "SIGCSE Bulletin", No.2, 1970, pp.30-33

  (ACM = Association for Computing Machinery)
  (SIGCSE = Special Interest Group on Computer Science Education)

(Retyped by Emmanuel ROCHE.)

The purpose of this paper is to describe the Computer Science Laboratory at the University of Washington, explain how it is run, and to examine both its benefits and costs to the University. In order to do so, it is necessary to discuss briefly the University of Washington itself. The University of Washington is a very large, single-campus, state university. In a typical quarter, approximately 33,000 students are enrolled. There is a faculty of about 2,000. By contrast, the computer science program is a small interdisciplinary program confined strictly to graduate education. The program was begun in 1967, and granted its first Ph.D. in 1968. There are at present somewhat more than 40 graduate students, and a full-time equivalent faculty of about seven. Prior to the development of the program, there were, of course, teaching courses in Electrical Engineering and other departments, and a few courses in mathematics, but the advanced computer science stamp for the University of Washington had not yet been established.

The equipment on campus is also important to our story. The University Computer Center operates a Control Data 6400 and a Burroughs B5500. These machines are available to the general user, although the B5500 gives priority to administrative data processing. Use of the machines is charged either to research accounts, or to special departmental accounts established to support computing. That is to say, the Computer Center does not decide who runs on the computer. Rather, this is levied as a charge against departmental budgets. Like any other department, the Computer Science Group also has a budget for computing on the Center's equipment. In addition, there are a variety of special computers on campus. For example, the Physics Department contains a PDP-10 and an XDS-930. There is a very large Raytheon computer complex in Biophysics, and there are a number of 1130 and 1800 installations on campus. All these computers are associated with specific research projects and, therefore, are available only to students who are working with the faculty on these projects. In addition, the University has provision for favorable rates on a number of commercial time-sharing installations. Departments do not receive funds for off-campus computing, so that extra-mural funds must be sought by anyone who wishes to make use of a conversational computing capability.

The Computer Science Laboratory contains a Xerox Data Systems Sigma 5 which is funded by a special departmental account, so that no student is ever charged for use of this machine. However, only a favored few students and faculty, to wit, those associated in some way with the Computer Science program, may have access to the machine at all.

The laboratory system

The Sigma 5 is a 32-bit machine with an 850-nanosecond cycle-time and 16 general registers. In a word, it looks very much like a 360-44, a machine to which it is frequently compared. The basic Washington configuration consists of the Sigma 5 itself with 24,000 words of core, a 6-megabyte disc, and 2 low-speed tape drives. This is slightly larger than the initial configuration, which was installed in September, 1969, and consisted of 16,000 words of core and a .75-megabyte disc. The computer contains an external interface which is a multiplexing device for tying foreign equipment to the computer, and an EIA interface which provides for four outlets. Three of these outlets are now used for a standard telephone line attached to 103A data sets. The fourth port is used to attach a 1200-baud line to a Computer Displays, Inc. ARDS unit. This is a keyboard, plus a "memory scope" CRT, plus a graphic input device known as a mouse. We are now attaching an IMLAC Corp. PDS-1 Display Unit, which is a small computer used to control a high-resolution Visual Display System. We have a floating-point hardware, and two general register blocks of 16 registers. The latter may seem somewhat unusual in a machine that only has 24K of actual core, and obviously is not doing very much multi-programming.

In summary, we have a large number of peripherals, but not enough core or secondary memory space to do any really useful work. This is intentional, since the configuration is not intended to serve users. The rationale for the laboratory is that computer science students should be able to do anything at all in computing on an EXPERIMENTAL basis, but that the University cannot afford to let all its 30,000 potential users run wild with any application whatsoever. In some aspects of Computer Science, it is obvious that the Computer Science student or faculty member does not appear, to a machine, to be any different from any other user. Examples are studies in numerical analysis, compiler testing, and much of the work in artificial intelligence and pattern recognition. These projects need a very big machine, and we have one, the University's CDC 6400. We compete for resources to use this machine on the same basis as anybody else. We, at one time, did limited interactive and large information retrieval application studies on the Burroughs B5500, but this had to drop out as administrative data processing took over that machine. It is really only for this reason that we have a disc as large as we do.

Having said what the computer science facility is not for, let us consider what it is for. Within our configuration, we can set up a prototype system of considerable complexity. We have all the problems introduced by multiple I/O devices, allocation of secondary memory and primary memory, tape drives, paper-tape card reading, and teletype work. We can prove that a system would work, but cannot operate it economically. This is all right, because we have shown that, when a programming idea works, then the computer science research is finished, and we should turn our product over to somebody else. Therefore, the Computer Science facility has been designed to make it easy to bring up and tear down operating systems. Similarly, we can do graphics projects on an experimental basis. Of course, we are not unique in this. There are even a few fortunate schools in which the general user can do a graphics project. However, they are very few, and the University of Washington is simply not one

of them. We hope our limited graphics capability will serve as a prototype to guide other graphics users, as well as opening a new research avenue for our own students. A similar sort of argument holds for studies of interacting computing and conversational computing systems. Formerly, we used the B5500 to study conversational computing and, in fact, produced some very interesting systems, such as the Burroughs B5500 APL system, using that machine. However, it is now no longer available to us on a reliable basis. We can use our own machine to run a conversational system. Granted, it can have at most 3 teletypes, but to go from zero to one is a giant step in this field; to go from one to three is, we feel, a substantial step, and from three to n a much smaller one.

Another very important class of research topics that we can attack with this facility, and that we certainly could not touch in any general university facility, consists of those projects which attach hardware to the computer. For good reason, no production manager wants a strange signal flowing into his system. Since we run on a hands-on basis, all we have to assure is that the strange systems do not physically damage our equipment. We are helped here eagerly by the Xerox Data Systems staff. Projects which involve hardware modification and measurement have been done. If these were done on the University's production machine, the effect on throughput would be intolerable. It appears to us that the only way the University can do any sort of research in these fields is by having equipment dedicated to computer science research.

Operations

We have a hands-on operation. There are no paid operators. Instead, we give a driver's license examination to each prospective user, who then operates the machine for himself. This is obviously not enough for systems work. We do have a staff of about four system programmers and a small secretarial staff, supervised by a faculty director. The staff size varies from time to time. During summers, we try to employ students who have interesting projects, but who are not normally associated with the lab, to give them the experience, and to give us the benefit of developing a particular project.

A very important point is that the Director (a faculty member) and the secretary are the only non-students involved. There is a designated head systems programmer, who has a university rating and salary as a systems programmer but, in fact, is an advanced graduate student who can qualify for this position. The remaining systems programmers are also students in the program. The point is that all the staff identify themselves as students, and the laboratory as a student-faculty operation to be run together, rather than something that is run by old bogey "university administration".

The systems programming jobs are excellent as training positions. We try to rotate these frequently, rather than keeping one person in any of them during his graduate career. The ideal is that a person will spend one year fairly early in his graduate career as a learner working with the system, and then, perhaps two years later, during a period in which he has completed substantially all his classwork, but has not yet seized up a research idea, he will take some supervisory responsibility as a senior systems programmer. This

gives him valuable experience while he is waiting for the idea that will occupy him full-time on his Ph.D. research. We do not believe that it is a good idea to let a man have substantial systems programming responsibility after he has identified his Ph.D. topic.

The idea of heavy student involvement has been characteristic of the laboratory from its inception. The selection of the equipment was done by students, with faculty supervision. I want to stress that this was a very serious business. Selected students participated in the preparation of a request for bid and an evaluation of proposals, as part of a seminar. Some students who were particularly interested worked throughout the summer on all the problems involved in actually getting the machinery here. There was no behind-the-back direction, the graduate students were made fully conversant with all stages of negotiation and evaluation, both with the University administration, whom in our case represented management, and with the various vendors, at all times. At present, the laboratory staff and interested students participate in the planning of expansions, preparation of requests for bid for additional equipment, and the evaluation of proposals by vendors. We feel that this is extremely important practical training for Computer Science students, including those students who will become faculty members and who, while they will have primary responsibilities for the normal academic work, will live in continual danger of being appointed to the Computer Center Policy Committee, the Committee to Buy the New Computer or, if exceedingly unlucky, become the Computer Science Director.

Results

Now, I want to ask the question of whether, or not, we have accomplished the job the University desired. There are three ways of asking this: "What are the advantages?", "What are the disadvantages?", and "How do we sum them up?"

First of all, what have we accomplished? One way of evaluating this is to look at the sort of projects that have been done in the year that the laboratory has been in operation. Our major project was the development of a paging system. We now quite routinely run jobs which require up to 128,000 words of core, using a paging system which is transparent to the user. This system came up and became a useful tool, even though not completely debugged, in a very short time. It was developed by one faculty member with help of two others, and four or five interested students, not all of whom are on the laboratory staff. In work associated directly with class work, another professor, who is charged with the advanced software courses, has been able to give realistic assignments and term projects. These include such things as the construction of text editors, interactive desk calculators systems, input/output drivers, assemblers and compilers, and the development of a graphic language. Of course, some of these projects could have been done on the University main computers, but we doubt that very many of them would have been done, or would have been carried out to the extent that they were carried out, or with the appreciation of the hardware and operating system interactions that the students had to gain, in order to do the work on our primitive machine.

A number of research projects have begun, which would have been impossible without hands-on access to equipment. Some of the most interesting are in the

area of measurements where, as I have indicated, we can actually attach physical devices and do hardware measurements, ignoring the cost of substantial interference with the execution of programs. If the hardware interface causes a crash, or if the software measurement takes excessive time, that is simply not a problem. In a straight software field, we have begun a fairly large study of interactive problem solving, using both the graphics device manipulation and the information retrieval capability of the system. Another project has developed a flying spot scanner for photographic image analysis, which is hardwired into the machine for studies of pattern recognition from filmed data.

Personnel training is, of course, a major purpose of a university. By far, the greatest amount of training is that of the laboratory personnel themselves. By maintaining steady student rotation, we are able to give students experience in the "nitty-gritty" aspects of Computer Science which tend to be lost in the maze of Artificial Intelligence and automata theory. This sort of training is not limited just to the laboratory staff, since we welcome all students in the various Computer Science courses who are willing to work. The spirit in the lab is that we do not want to shut anyone out, unless he is lazy. This does not pose a disciplinary problem for the faculty, incidentally, as the students handle problems nicely themselves.

There is one way in which we have not begun to fulfill the role which the University desired for us. It was hoped that this facility would serve as a prototype for other departmental research facilities. In particular, it was hoped that we would be able to work with other departments, to provide them with some sort of guide to the sort of equipment they really needed, using us as a guinea pig to set up the system which they wanted, and then seeing how much of our equipment they actually used. This would have been a great help in letting them decide what course to follow in acquiring a computer capability, and in preparing requests for proposals from vendors. As yet, however, we have only begun to get into such work, in a small way. We hope that, in the next few years, as we become better known on campus, this will become more and more one of our tasks.

In every life, a little rain must fall. There are obvious disadvantages to the approach we have taken. The major one is cost. This is an expensive system to run for a forty-graduate student program. The monthly budget for equipment is approximately $7,000, and for personnel (not including the faculty director) $2,500. We also have a rather large space requirement in terms of the amount of space usually devoted to a departmental activity. However, in terms of the amount of space for a computer facility, our quarters in the Computer Center are uncomfortably crowded, and not particularly luxurious. In fact, we have been featured in "Computer World" as the computer that was drowned by the laboratory upstairs, when they forgot to close off one of their water spouts. Again, the students were involved in mopping up the computer, and presumably they learned something from that about the reliability of various pieces of hardware.

A more subtle cost is the diversion of faculty effort. Because of the high turnover of personnel, our faculty have to devote considerable time to breaking in new students on the system. A professor cannot rely upon the manuals, and consultants and old-time programmers, that would be available if he were using the Computer Center for his course work; he must guide the

students himself.

There is a hidden educational cost to running this machine. The laboratory can become a tail wagging a dog. It is clear that, at the University of Washington, that "where the action is" is in the Computer Science Lab. This means that the automata theorist, if he is rather a sensitive soul, may feel left out. We cannot evaluate how much this system acts as a magnet which diverts student attention from other quite worthy aspects of computer science. Even within the research fields of operating systems and interactive work, where this facility should contribute most to the research effort, there is a problem when students are trapped into a continual cycle of learning, and practicing new techniques without ever taking a step back to consider what the basic scientific problems are. This is always a nebulous problem in graduate education, and we do not know whether making good equipment available makes it more serious. I personally felt, when I was acting as faculty director, that I had to be extremely careful to watch for the student who runs from one fun project to another, without ever trying to develop new research questions of his own. The lab would be a good place to breed computer nuts, instead of computer scientists. The problem would be even more serious if we had an undergraduate program, because it would be very easy to get students diverted into the fun things, without them ever learning a great deal about the major problems. Our graduate students have, at least, had forced exposure to other fields.

Notwithstanding these problems, we feel there can be no doubt that the Computer Science Research facility is a fine thing for the educational program. Whether it justifies the expense to the University is something of which we are perhaps not the best judges. We think that, in a very short time, the Computer Science program at the University of Washington has moved from non-existence to a small, reasonably good program which stresses the more practical aspects of the computing field. We do not want to rest on our laurels (if any); we hope that we will get better.

EOF

GKALG.WS4

- "ALGOL-E: An Experimental Approach to The Study of Programming Languages"
  Gary Kildall & Alan B. Roberts
  ACM "SIGCSE Bulletin", Vol.4, No.1, March 1972, pp.127-135
  (ACM SIGCSE Second Symposium on Education in Computer Science)

  (ACM = Association for Computing Machinery)
  (SIGCSE = Special Interest Group on Computer Science Education)

(Retyped by Emmanuel ROCHE.)

A common approach to the teaching of Programming Languages (Course I2, Curriculum 68) has been to teach several languages, each demonstrating a feature deemed significant, such as ALGOL, LISP, SNOBOL, and COBOL [References 3 & 7]. The problem that exists with this method is that far too much time is spent learning the details necessary to use the languages, leaving time for only a few trivial programs in each language. A popular alternative to this approach is to teach the course using a single general-purpose language which has a broad repertoire of language features, such as PL/I. While this method successfully avoids much of the detail which characterizes the former, it too seems to have a serious drawback. The student can become quite talented at programming in the language, and still have very little feel for the implications of the higher level language structures at the machine level. Moreover, these languages typically provide no means by which the student can readily investigate these implications. Hence, ALGOL-E is proposed as a programming language system which provides such a capacity.

ALGOL-E is a programming language based on ALGOL-60, defined within the framework of a complete system designed with the teaching of programming language concepts in mind [Ref. 4]. A basic design criterion for the language was that it be simple and easy to use, while not severely compromising the language constructions available. The language is constructed such that it can be implemented using a single execution stack [Ref. 6]. The system is defined by three programs, each corresponding to a distinct level of formal definition. The first of these programs is a stack machine simulator which accepts zero-address machine code. The machine operators of the simulated machine are defined formally in terms of primitive register operations. The second is an assembler which provides a convenient means of symbolic programming at the machine level, rather than using absolute machine code. The syntax of the assembly language is given in the Backus notation, and the associated semantic actions are defined using the machine operators. Finally, the third program is a compiler for the ALGOL-E language. As with the assembler, syntactic structure is given in Backus-Naur Form (BNF); however, the semantic interpretation of the high-level language is formally defined using the assembly language.

The notation used in the description of register actions defining the machine operations is essentially that used by Burroughs on the B5500 [Ref. 2]. Memory is considered as a vector M, with the S register (rS) pointing to the top of the stack. The C register (rC) is the instruction counter controlling

instruction sequencing. Figure 1 below gives descriptions of some of the machine operators. The notation is fairly straightforward; note, however, that rA and rB are abbreviations for the top two stack locations, M[rS] and M[rS-1], respectively. The add operation, for example, indicates that the top two elements of the stack are summed, and the result replaces those two elements on the stack. The instruction counter is then incremented to the following instruction. These operator definitions serve as a basis for the definition of the Assembler.

OpCode	Operation	Action of Operator
129	addition (ADD)	rB:=rB+rA; rS:=rS-1; rC:=rC+1
138	less or equal (LEQ)	rB:= if rB <= rA then 1, otherwise 0, rS:=rS-1; rC:=rC+1
148	load (LOD)	rA:=M[rA]; rC:=rC+1
149	store (STO)	M[rB]:=rA; rB:=rA; rS:=rS-1; rC:=rC+1
177	store and destruct stack (STD)	M[rB]:=rA; rS:=rS-2; rC:=rC+1
155	exchange (XCH)	M[rS+1]:=rB; rB:=rA; rA:=M[rS+1]; rC:=rC+1
161	branch to segment (BRS)	rC:=rA; rS:=rS-1
162	branch to segment conditional (BSC)	rC:= if rB = 0 then rA, otherwise rC+1; rS:=rS-2
0, 1, ..., 127	literal call (LIT)	rS:=rS+1; rA:=C[rC]*; rC:=rC+1
146	immediate one syllable (IM1)	rC:=rC+1; rS:=rS+1; rA:=C[rc]; rC:=rC+1
147	immediate two syllables (IM2)	rC:=rC+1; rS:=rS+1; rA:=256*C[rC]+C[rC+1]; rC:=rC+2

*C represents the code area of memory, addressed here as a vector.

Figure 1

The Assembler is designed to achieve readability and facility at the machine level. The simple structure of the assembly language is defined in BNF, to provide an introduction to the notation prior to considering the more complex structure of the high level language. Pseudo-instructions, CON, VEC, and STR, are provided for symbolically defining integer constants and variables, integer vectors, and print strings, respectively. Symbolic labels are also allowed. The assembly language LIT, or literal call syllable, is used to place a value onto the stack. This value might be a data value to be used with an arithmetic operation, a variable address to be used with a load or store operation, or a label address to be used with a branch operation. The assembler LIT translates into one of several machine instructions, LIT, IM1, or IM2, depending on the size of the value. Assembly language mnemonics

associated with the machine instructions are given in Figure 1 above.

Neither the machine nor the assembly language described above constitutes an end in itself, but rather they exist for the purpose of illustrating structures and concepts in the high-level language. The features of interest are many and varied. Significant among these is block structure. All too often, block structure and scope of variables is learned by example, with the student gaining very little feel for its implementation. Closely related to the understanding of block-structured declarations is the understanding of memory allocation and, in particular, the difference between compile-time allocation and dynamic allocation, and their respective implementations. Along with the idea of dynamic allocations for such things as arrays comes very naturally the problem of array subscripting. An understanding of the mechanism by which subscripting is accomplished in a particular implementation is certainly requisite to reasonable and efficient use of subscripted variables. Another area of significant importance is subprograms. Included here are the differences between functions and subroutines, and the handling of their parameter lists. Certainly, a large source of chagrin for students in this area is recursion, and the handling of parameters and local variables during recursive calls. Not only does the understanding of the implementation of recursion serve to demonstrate the overhead involved in its use, but also it seems that, for many students, it serves to shed light on the concept itself. Finally, there is the implementation of the basic statements of the language. The parsing and transformation of generalized language statements to the machine level are not intuitively obvious. Among these would be assignment statements, iterative statements, and conditionals.

Given that the concepts described above are things which merit investigation in a programming language course, it is worth noting that the investigation of them involves scrutiny of the compilation process, as well as execution. This is the reason that an "understanding gap" from high-level language to machine exists, because compilation is precisely the step which is obscured most from the user. ALGOL-E is designed to allow investigation of the compilation process without too large an investment of time in the details of the compiler writer's job.

The compiler's symbol table is one of the first areas to come under study. Block structure is implemented by association of address with symbol in a table built as a stack. A vector of "block level" pointers mark the block levels within the table, as shown in Figure 2 below. When the end of a block is scanned by the compiler, the table is cut back to the level indicated by the last entry in the block level vector. A degree of efficiency in storage use can be obtained by resetting the storage location counter when the symbol table is cut back. This causes the storage locations to be reassigned in the next block, and has the effect of sharing storage locations between variables declared in parallel blocks of a program. A problem is created in this area, however, when subprograms are considered. Even though a block within a subprogram is ended, the subprogram can be referenced anywhere within the block in which it is defined, and hence the storage locations within the subprogram cannot be reassigned until this outer block is closed. To avoid this problem, an additional vector is used in conjunction with the symbol table which records the maximum storage location assigned within a block, so that they can be protected if the block is recognized as part of a subprogram.

```
BEGIN LOCAL U, GAMMA;
    BEGIN LOCAL A,
    X, B;
        BEGIN
        LOCAL R, S;
        ...
        END
    END
END
```

```
  BLOCK       SYMBOL    LOCATION
  +-----+     +-------+  +-----+
.|    |    .|     | |    |
.|    |    .|     | |    |
.|    |    .|     | |    |
.|    |    7|  S  | |  6 |
.|    |    6|  R  | |  5 |
5|    | +-> 5|  B  | |  4 |
4|    ||   4|  X  | |  3 |
3|  5 +-+   3|  A  | |  2 |
2|  2 +---> 2| GAMMA | |  1 |
1|  0 |     1|  U  | |  0 |
  +-----+     +-------+  +-----+
```

Figure 2

Two program options augment investigation of the compilation process. One of these is a trace of the parsing reduction as applied in the recognition of program syntactic structure (The parsing algorithm is the mixed strategy precedence algorithm of McKeeman [Ref. 5].). The other is a trace of the code generated during compilation, to demonstrate the association of semantics with each of the reductions. This code is printed in the assembly language mnemonics for readability. With these options, a number of important things become readily visible. The code which accomplishes dynamic allocation for such things as arrays and recursion is available, and the code corresponding to the mapping of array subscripts and subprogram parameters can also be explored. In addition, the implementation of the various language statements is easily investigated. Code tracing during compilation is, of course, not unique to the ALGOL-E compiler. The advantage here, however, is that the ALGOL-E machine is conceptually simple, and the semantic actions are well-defined. Hence, the student can readily understand the ALGOL-E code trace.

Figure 3 below shows a program listing demonstrating the parse and code tracing facilities.

```
CARD | BL | SYL |
  1 |  0 |   0 | BEGIN LOCAL X,I;
$PARSE
  3 |  1 |   0 |     X:=I;
                  <BEGIN> <DECLARATION SET> ;
                  <IDENTIFIER>
                  <IDENTIFIER>
                  <VARIABLE>
                  <PRIMARY ELEMENT>
```

```
            <PRIMARY>
            <TERM>
            <ARITHMETIC EXPRESSION>
            : = <EXPRESSION>
            <VARIABLE> <RIGHT PART>
            <ASSIGNMENT STATEMENT>
            <SIMPLE STATEMENT>
            <STATEMENT>
$PARSE
$CODE
  6 |  1 |   4 |   FOR I:= 1 STEP 1 UNTIL 10 DO
           4 |     001
           5 |     001
           6 | STO 149
           7 | IM2 147
           8 |     000
           9 |     000
          10 | BRS 161
          11 |     001
          12 |     001
          13 | LOD 148
          14 | ADD 129
          15 |     001
          16 | XCH 155
          17 | STO 149
        (BACKSTUFF 8,9: 18)
          18 |     010
          19 | LEQ 138
          20 | IM2 147
          21 |     000
          22 |     000
          23 | BSC 162
  7 |  1 |   24 |       X:= I;
          24 |     000
          25 |     001
          26 | LOD 148
          27 | STO 149
          27 | STD 177
          28 | IM2 147
          29 |     000
          30 |     000
          31 | BRS 161
        (BACKSTUFF 21,22: 32)
        (BACKSTUFF 29,30: 11)
$CODE
  9 |  1 |  32 | END
 10 |  1 |  32 | EOF
PRT=2, DATA=0, CODE=9 (WORDS).
```

Figure 3

Along with the compilation options available, there are built-in procedures which aid both in language investigation and in debugging of programs. These include an execution trace feature, and a snapshot dump capability. The trace

feature prints one line per instruction, indicating the contents of machine registers and the top of the stack. The snapshot dump facility allows the programmer to obtain a memory dump at any point in the program. The dump obtained, however, is not the usual octal or hexadecimal dump available on most machines. Rather, it is decoded for readability, with the memory areas flagged, the code area appearing in the assembler mnemonics, and the stack contents appearing in decimal form, as shown in Figure 4 below. In this form, the dump feature becomes a viable tool for the investigation of memory changes for such things as successive levels of recursion.

```
CARD | BL | SYL |
$STACK
  2 |  0 |   0 |
  3 |  0 |   0 | BEGIN FUNCTION FACTORIAL (N);
  4 |  2 |  10 |       IF N EQL 0 THEN
  5 |  2 |  14 |          BEGIN DUMP; FACTORIAL:=1 END
  6 |  3 |  25 |       ELSE FACTORIAL:=N * FACTORIAL(N-1);
  7 |  1 |  48 |
  8 |  1 |  48 |       WRITE (FACTORIAL (3) )
  9 |  1 |  52 | END
 10 |  1 |  53 | EOF
PRT=3, DATA=0, CODE=14 (WORDS).
CODE FILE WRITTEN
END OF COMPILATION DECEMBER 28, 1971.  CLOCK TIME = 15:35:0.72


A L G O L - E   M E M O R Y   D U M P
-------------------------------------

MACHINE REGISTERS
          RA: 2
          RB: 24
          RS: 31
          RC: 31 (SYLLABLE=19)
          RG: 8188

INPUT BUFFER:
||

OUTPUT BUFFER:
||

PROGRAM REFERENCE TABLE (PRT)
          MEMORY ADDR    FIXED FORMAT
            0            0
            2            28
```

(Note: 28 is the memory location associated with function call
(factorial=0, n=0, pointer=28) See execution stack below.)

```
FIXED DATA AREA (FDA)
          MEMORY ADDR    FIXED FORMAT   CHARACTER FORMAT

CODE AREA
```

```
MEMORY ADDR   SYLLABLE LOC      SYLLABLES
  3           0          IM2 000 045 BRS
  4           4          001 IM2 000 000
  5           8          000 SAV 001 LOD
  6           12          000 EQL IM2 000
  7           16          029 BSC 002 BIF
  8           20          DEL 001 000 XCH
  9           24          STD IM2 000 041
  10           28          BRS 001 LOD 001
  11           32          LOD 001 MIN 004
  12           36          PRO MUL 000 XCH
  13           40          STD 001 000 UNS
  14           44          RTN 002 000 STD
  15           48          DMP 003 004 PRO
  16           52          WRV XIT XIT XIT

EXECUTION STACK
        MEMORY ADDR   FIXED FORMAT
          17          0
          19          3
          20          0---+
          21          3   |
          22          1---+
```

(Note: Function values associated with original call
(factorial=0, n=3, pointer=1))

```
          23          2
          24          0---+
          25          2   |
          26          20--+
```

(Note: Function values associates with first recursive call
factorial=0, n=2, pointer=20))

```
          27          1
        --> 28          0---+
          29          1   |
          30          24--+
```

(Note: Function values associated with second recursive call
(factorial=0, n=1, pointer=24))

```
          31          2
          32          29
          33          0
          8188          49--+
          8191          64--+
```

(Note: Return addresses (saved from top of execution stack, toward bottom).)

END OF DUMP.

Figure 4

The definition of the system on several levels lends itself to the possibility of formally proving the correctness of various aspects of the compilation process. An example is given in Appendix A. In the example, it is shown that a FOR statement of the form

FOR I:=1 STEP 1 UNTIL N DO <simple statement>

terminates, as long as the <simple statement> terminates and does not alter the values of I and N.

To date, ALGOL-E has been used twice in the teaching of a programming languages course. Because of its limitations as far as data types, allowing only integers, and its lack of such features as string manipulation and list processing, it is not intended as the subject of the entire course. It has been found that approximately six weeks of an eleven week quarter is a reasonable time to study ALGOL-E, and gain a feel for the language and its structure. From there, a transition is made to ALGOL-W [Ref. 1] to study string manipulation and list processing. The similarities between the two languages are such that the transition is quite easy. Relative to list processing, a second version of the algol-E machine exists, which maintains a "Free Storage Area" separate from the stack. This version can be used to demonstrate a linked-list approach to storage management suitable for the more general dynamic allocations necessary for such things as list processing.

A convenient side effect of the ALGOL-E system worth noting is that the emphasis placed upon the compilation process as a key to understanding programming language features establishes continuity with Compiler Construction (Course I5, Curriculum 68). The ALGOL-E programs were constructed using the XPL Compiler Generator System [Ref. 5], and they can serve as an excellent example for the teaching of the compiler writing course.

In summary, the emphasis in the ALGOL-E system is placed upon providing the student tools whereby he can gain understanding of high level language concepts through direct investigation. It is designed to replace exposition by an instructor with experimentation by the student, and to provide concrete examples in the place of vague generalizations. The ALGOL-E programming system will operate on any computer capable of running the XPL Compiler Generator System, typically an IBM S/360 Model 50, or larger. The system is available in its entirety, along with documentation, from the W. R. Church Computer Facility, Naval Postgraduate School, Monterey, California, 93940.

Appendix A

Consider the following FOR statement in ALGOL-E:

FOR I:=1 STEP 1 UNTIL N DO <simple statement>

The ALGOL-E semantics for this statement are:

<for statement> ::= <for clause> <step expression> <until clause>
<do statement>

<for clause>; <step expression>; <until clause>; <do statement>;
END:

<for clause> ::= FOR <assignment statement>
{save identifier V in assignment statement}
<assignment statement>; LIT UNTIL; BRS

<step expression> ::= STEP <expression>
STEP: <expression>; LIT V; LOD; ADD; LIT V; XCH; STO

<until clause> ::= UNTIL <expression>
UNTIL: <expression>; LEQ; LIT END; BSC

<do statement> ::= DO <simple statement>
<simple statement>; LIT STEP; BRS

where <assignment statement>, <expression>, and <simple statement> are further expanded to their respective representatives.

A machine execution is determined by a vector M representing the program variables and the execution stack, a code vector C, and registers rC and rS. A machine execution sequence consists of an initial configuration <M0, rC0, rS0> and a sequence (not necessarily finite) of successor configurations <M1, rC1, rS1>, <M2, rC2,rS2>, ..., <Mk, rCk, rSk>, where <Mi+1, rCi+1, rSi+1> is derived from <Mi, rCi, rSi> by application of the definition of operator C[rCi]. A machine execution sequence terminal at t is a finite machine execution sequence <M0, rC0, rS0>, <M1, rC1, rS1>, ..., <Mk, rCk, rSk> such that rCk=t and rCi<>t whatever i, 0 <= i < k.

Suppose the variables I and N in the above FOR statement are elements 0 and 1 of M, respectively. Expanding the semantics given above, C is:

```
C[1]    LIT I
C[2]    LIT 1
C[3]    STO
C[4]    LIT UNTIL           (13)
C[5]    BRS
C[6]    LIT 1
C[7]    LIT I
C[8]    LOD
C[9]    ADD
C[10]   LIT I
C[11]   XCH
C[12]   STO
C[13]   LIT N
C[14]   LOD
C[15]   LEQ
C[16]   LIT END             (k+2)
C[17]   BSC
C[18]   <simple statement>
C[k]    LIT STEP            (6)
C[k+1]  BRS
C[K+2]  ...
```

Let <simple statement> be such that all machine execution sequences beginning with initial configuration <Mi, 18, rS> are terminal at k with configuration <Mj, k, rSj> rSi=rSj, Mip=Mjp whatever p <= rSi, 18 <= rCq <= k whatever q it exists i <= q <= j. In other words, the simple statement must leave the stack pointer in the same position, with no changes below that point in the stack, and no changes to I or N. It must also not branch out of the range of the simple statement.

Theorem: All machine execution sequences of the above FOR statement with initial configuration <(lambda, n, lambda, ..., lambda),1,1> are terminal at k+2 whatever n belonging to Z, where lambda is an indeterminate value.

Proof:
P(x): Consider a machine execution sequence of the above FOR statement with initial configuration <(lambda, n, lambda, ..., lambda),1,1> which does not contain a configuration with rC=k+2 such that configurations with rC=17 occur in this sequence at least x times. If <M, 17, rS> is the xth occurrence of such a configuration, then M0=x.

The proposition above states that, if, in the execution of a FOR statement, the test is reached for the xth time without having terminated the loop, then the value of the loop index I is X.

To show P(1): Consider the following enumeration of an execution sequence beginning with initial configuration

<(lambda, n),1,1>* apply rS:=rS+1; rA:=C[rC]; rC:=rC+1

*M is of fixed length, and is assumed to be "large enough" for the problem. Only the pertinent portion of M is shown here.

<(lambda, n, 0),2,2> apply rS:=rS+1; rA:=C[rC]; rC:=rC+1

<(lambda, n, 0),3,3> apply M[rB]:=rA; rB:=rA; rS:=rS-1; rC:=rC+1

<(1, n, 1),4,2> apply rS:=rS+1; rA:=C[rC]; rC:=rC+1

<(1, n, 1, 13),5,3> apply rC:=rA; rS:=rS-1

<(1, n, 1),13,2> apply rS:=rS+1; rA:=C[rC]; rC:=rC+1

<(1, n, 1, 1),14,3> apply rA:=M[rA]; rC:=rC+1

<(1, n, 1, n),15,3> apply rB:=1 if rB <= rA otherwise 0;
rS:=rS-1; rC:=rC+1

<(1, n, s),16,2> apply rS:=rS+1; rA:=C[rC]; rC:=rC+1

<(1, n, s, k+2),17,3>

Note that the only element of a configuration to this point which is dependent on the value of n is the value of s. It follows that all machine execution sequences begin with this sequence, hence at the first occurrence of a configuration with rC=17, it is always the case that M0=1.

To show P(q) implies P(q+1): Let S be a machine execution sequence such that configurations with rC=17 occur at least q+1 times. In the qth occurrence of <M, 17, rS>, M0=q by the inductive hypothesis. Consider the machine execution sequence with initial configuration <(q, n, s, k+2),17,3>. C[17] is BSC which is defined as:

rC:= rA if rB=0 otherwise rC+1; rS:=rS-2

Hence, there are two possible successor configurations:

(a) <M, k+2, 1>
and
(b) <M, 18, 1> ,

but case (a) cannot occur, since it causes S to be terminal at k+2, which contradicts the fact that <M, 17, rS> occurs at least q+1 times. Hence, the successor configuration is <M, 18, 1>. Consider a machine execution sequence S' with initial configuration <(q, n), 18, 1>. By the conditions on the <simple statement>, S' is terminal at k with configuration <(q,n), k, 1>. Since 18 <= rC <= k for all configurations of the <simple statement>, the configuration <M, 17, rS> cannot have occurred in S'.

Consider a machine execution sequence S" with initial configuration <(q, n), k+1>. Enumerating S":

<(q,n),k,1>
<(q,n,6),k+1,2>
<(q,n),6,1>
<(q,n,1),7,2>
<(q,n,1,0),8,3>
<(q,n,1,q),9,3>
<(q,n,q+1),10,2>
<(q,n,q+1,0),11,3>
<(q,n,0,q+1),12,3>
<(q+1,n,q+1),13,2>
<(q+1,n,q+1,1),14,3>
<(q+1,n,q+1,n),15,3>
<(q+1,n,t),16,2>
* <(q+1,n,t,k+2),17,3>

Extending the sequence S by S' followed by S", it is evident that * is the q+1 occurrence of a configuration with rC=17, hence P(q+1). Therefore, by induction, it is true that, for every positive integer x, the value of I at the xth test in the FOR loop is x. The proof can then be completed in the following manner:

(1) Consider the case n <= 0 and demonstrate that the branch is taken to the terminal configuration rC=k+2 at the first test.
(I=1>N)

(2) Show by induction that, for any positive value of n, the branch to the terminal configuration is taken at the n+1 test.
(I=N+1>N)

The logic above can be extended to show such corollaries as the fact that the simple statement is executed exactly n times for all values n > 0, and the fact that stack underflow does not occur. More precisely, the stack pointer at the termination of a FOR statement is in same position as in the initial configuration.

Appendix B: A complete ALGOL-E program (105 colums wide)

```
 2 |  0 |   0 | BEGIN
 3 |  0 |   0 | COMMENT -- ALGOL-E program to compute the shortest route between cities.
 4 |  0 |   0 |
 5 |  0 |   0 |    INPUT DATA:
 6 |  0 |   0 |         city1 dist city2
 7 |  0 |   0 |
 8 |  0 |   0 | Where CITY1 and CITY2 are the integers corresponding to the cities involved
 9 |  0 |   0 | (see the "PRINTCITY" procedure below). DIST is a non-negative integer which
10 |  0 |   0 |     (1) indicates the distance between the two cities if greater than zero,
11 |  0 |   0 |     (2) indicates that the shortest route between the two cities is desired
12 |  0 |   0 |         if equal to zero.
13 |  0 |   0 |
14 |  0 |   0 | The end-of-data is indicated by CITY1 equal to HALT;
15 |  0 |   0 |
16 |  0 |   0 | LOCAL n, halt; n:=10; halt:=99999;
17 |  1 |   8 |     BEGIN ARRAY dist, route, flag <1:n>; ARRAY citylist <1:n, 1:n>;
18 |  2 |  35 |
19 |  2 |  35 |     PROCEDURE printcity (k, c);
20 |  3 |  45 |         IF k GTR n OR k LEQ 0 THEN WRITEON (TAB c, k) ELSE
21 |  3 |  65 |         CASE k OF
22 |  3 |  71 |            BEGIN k:=k;
23 |  5 |  87 |              WRITEON (TAB c, "Seattle"       );
24 |  5 |  97 |              WRITEON (TAB c, "Boise"         );
25 |  5 | 107 |              WRITEON (TAB c, "Modesto"       );
26 |  5 | 117 |              WRITEON (TAB c, "Sacramento"    );
27 |  5 | 127 |              WRITEON (TAB c, "San Francisco" );
28 |  5 | 137 |              WRITEON (TAB c, "Monterey"      );
29 |  5 | 147 |              WRITEON (TAB c, "Las Vegas"     );
30 |  5 | 157 |              WRITEON (TAB c, "Los Angeles"   );
31 |  5 | 167 |              WRITEON (TAB c, "Bakersfield"   );
32 |  5 | 177 |              WRITEON (TAB c, "Tijuana"       )
33 |  5 | 180 |            END;
34 |  2 | 238 |
35 |  2 | 238 |     PROCEDURE addcity (c1, d, c2);
36 |  3 | 248 |         BEGIN LOCAL i; SKIP (1);
37 |  4 | 252 |         printcity (c1, 10); WRITEON (TAB (i:=APPEND + 1), "is", TAB i+3, d,
38 |  4 | 275 |         TAB (i:=APPEND + 1), "miles from"); printcity (c2, APPEND + 1);
39 |  4 | 295 |         citylist <c1, c2> := citylist <c2, c1> := d;
40 |  4 | 313 |         WRITEON (TAB APPEND, ",");
41 |  4 | 319 |         END;
42 |  2 | 326 |
43 |  2 | 326 |     LOCAL r;
44 |  2 | 326 |
```

```
45 |  2 | 326 |    FUNCTION path (city, dest);
46 |  3 | 336 |       IF city EQL dest THEN path:=0 ELSE
47 |  3 | 349 |       IF flag <city> EQL 1 THEN path:=halt ELSE
48 |  3 | 370 |       BEGIN LOCAL i, q, rt, j, t; ARRAY dst, rte <r+1:n>;
49 |  4 | 389 |       LOCAL bestdist, bestcity, bestroute;
50 |  4 | 389 |       flag <city>:=1; bestdist:=halt; bestcity:=bestroute:=0; rt:=r;
51 |  4 | 409 |       FOR i:=1 STEP 1 UNTIL n DO
52 |  4 | 430 |          BEGIN q:=citylist <city, i>;
53 |  5 | 440 |          IF q NEQ halt THEN
54 |  5 | 445 |            BEGIN t:=path (i, dest);
55 |  6 | 459 |            IF t+q LSS bestdist THEN
56 |  6 | 467 |              BEGIN
57 |  6 | 471 |              bestroute:=r; bestcity:=i; bestdist:=t+q;
58 |  7 | 486 |              FOR j:=rt+1 STEP 1 UNTIL r DO
59 |  7 | 510 |                 BEGIN rte <j>:=route <j>; dst <j>:=dist <j>
60 |  8 | 529 |                 END
61 |  8 | 534 |              END;
62 |  6 | 538 |            r:=rt
63 |  6 | 539 |            END;
64 |  5 | 542 |          END;
65 |  4 | 546 |          COMMENT -- Arrive here after all connections have been examined;
66 |  4 | 546 |          IF bestcity NEQ 0 THEN
67 |  4 | 550 |            BEGIN
68 |  4 | 554 |            FOR j:=rt+1 STEP 1 UNTIL bestroute DO
69 |  5 | 578 |               BEGIN route <j>:=rte <j>; dist <j>:=dst <k>
70 |  6 | 597 |               END;
71 |  5 | 606 |            r:=bestroute+1; dist <r>:=citylist <city, bestcity>;
72 |  5 | 626 |            route <r>:=bestcity
73 |  5 | 631 |            END;
74 |  4 | 634 |          flag <city>:=0; path:=bestdist
75 |  4 | 641 |          END;
76 |  2 | 656 |
77 |  2 | 656 |    PROCEDURE routefinder (c1, d, c2);
78 |  3 | 666 |       BEGIN LOCAL i;
79 |  4 | 666 |       IF (i:=APPEND) GTR 1 THEN
80 |  4 | 672 |         BEGIN WRITEON (TAB i-1, "."); SKIP (1)
81 |  5 | 685 |         END;
82 |  4 | 688 |       SKIP (1); WRITE ("The shortest route from"); printcity (c1, APPEND+1);
83 |  4 | 705 |       WRITEON (TAB APPEND+1, "to"); printcity (c2, APPEND+1); r:=0;
84 |  4 | 725 |       WRITEON (TAB (i:=APPEND+1), "is", TAB i+3, (i:=path (c1, c2) ), TAB APPEND+1,
85 |  4 | 756 |       "miles");
86 |  4 | 759 |       IF i EQL halt THEN WRITE (TAB 10, "(The cities are not connected)") ELSE
87 |  4 | 774 |       IF i EQL 0 THEN WRITE (TAB 10, "(Don't go anywhere)") ELSE
88 |  4 | 792 |       WRITEON (TAB APPEND+1, "via:");
89 |  4 | 804 |       SKIP (1);
90 |  4 | 808 |       FOR i:=1 STEP 1 UNTIL r DO
91 |  4 | 827 |          BEGIN WRITE (TAB 15, dist <r-i+1>, TAB APPEND+1, "miles to");
92 |  5 | 852 |          printcity (route <r-i+1>, APPEND+1); WRITEON (TAB APPEND, ",");
93 |  5 | 876 |          END;
94 |  4 | 880 |       IF (i:=APPEND) GTR 1 THEN WRITEON (TAB i-1, "."); SKIP (1);
95 |  4 | 902 |       END;
96 |  2 | 909 |
97 |  2 | 909 |    LOCAL city1, city2, distance;
98 |  2 | 909 |
```

```
 99 |  2 | 909 |     FOR city1:=1 STEP 1 UNTIL n DO
100 |  2 | 930 |         BEGIN flag <city1>:=0; route <city1>:=dist <city1>:=0;
101 |  3 | 950 |         FOR city2:=1 STEP 1 UNTIL n DO citylist <city1, city2>:=halt;
102 |  3 | 985 |         END;
103 |  2 | 989 |     READ (city1);
104 |  2 | 992 |     WHILE city1 NEQ halt DO
105 |  2 |1001 |           BEGIN READ (distance, city2);
106 |  3 |1007 |           IF distance EQL 0 THEN
107 |  3 |1011 |             routefinder (city1, distance, city2) ELSE
108 |  3 |1026 |           addcity (city1, distance, city2); READ (city1)
109 |  3 |1041 |           END;
110 |  2 |1047 |     END;
111 |  1 |1050 | SKIP (3); WRITE ("End Of Run.");
112 |  1 |1058 | END
113 |  1 |1058 | EOF
PRT=40, DATA=67, CODE=265 (WORDS).
CODE FILE WRITTEN
END OF COMPILATION DECEMBER 28, 1971.  CLOCK TIME = 15:36:52.20.

113 CARDS WERE READ.
NO ERRORS WERE DETECTED.

SET UP TIME             0:0:0.57.
ACTUAL COMPILATION TIME  0:0:10.03.
CLEAN-UP TIME AT END     0:0:0.47.
TOTAL TIME IN COMPILER   0:0:11.07.
COMPILATION RATE :675 CARDS PER MINUTE.

PRT=40, DATA=67, CODE=265


      Seattle is 150 miles from Boise,
      Boise is 300 miles from Modesto,
      Seattle is 400 miles from Modesto,
      Modesto is 150 miles from Monterey,
      Modesto is 50 miles from San Francisco,
      San Francisco is 200 miles from Las Vegas,
      Las Vegas is 350 miles from Monterey,
      Los Angeles is 400 miles from Las Vegas,
      Bakersfield is 300 miles from Monterey,
      Bakersfiled is 250 miles from Las Vegas,
      Los Angeles is 450 miles from Tijuana,
      Tijuana is 700 miles from Las Vegas,
      Las Vegas is 920 miles from Boise.

  The shortest route from Seattle to Monterey is 550 miles via:

      400 miles to Modesto,
      150 miles to Monterey.

  The shortest route from Seattle to Seattle is 0 miles
      (Don't go anywhere)

  The shortest route from Boise to Tijuana is 1250 miles via:
```

300 miles to Modesto,
50 miles to San Francisco,
200 miles to Las Vegas,
700 miles to Tijuana.

End Of Run.

Bibliography
---

1. "ALGOL-W (Revised)"
   Bauer, Becker, Graham, and Satterthwaite
   Stanford University, 1969

2. "Burroughs B5500 Information Processing Systems Reference Manual"
   Burroughs Corporation, 1964

3. "A Comparative Study of Programming Languages"
   Bryan Higman
   American Elsevier Publishing Company, Inc., 1967

4. "The ALGOL-E Programming System"
   Gary Kildall
   Naval Postgraduate School, Monterey, California, 1971

5. "A Compiler Generator"
   McKeeman, Horning, and Wortman
   Prentice-Hall, Inc., 1970

6. "ALGOL-60 Implementation"
   Randell, and Russell
   Academic Press, 1964

7. "Programming Systems and Languages"
   Saul Rosen
   McGraw-Hill Book Company, 1967

EOF

GKAPL.WS4 (= Gary Kildall APL technical report)

APL\B5500: the language and its implementation

---

by Gary Arlen Kildall

(Retyped by Emmanuel ROCHE.)

Technical Report No.70-09-04,
Computer Science Group

University of Washington,
Seattle,
Washington 98105

September 1970

Abstract

APL\B5500 is a multiple-user interactive system for a conversational programming language implemented on the Burroughs B5500 computer at the University of Washington. The language is patterned after APL\360, which is an implementation of "Iverson Notation". This paper describes the differences between the APL\360 and APL\B5500 languages. In addition, the algorithms and data structures used in the implementation of APL\B5500 are given.

Acknowledgements

The APL\B5500 programming system was developed by members of the Computer Science Group at the University of Washington. The system was implemented by Leroy Smith, Sally Swedine, Mary Zosel, and the author, under the guidance of Dr. Hellmut Golde. The author is grateful to Dr. Golde and the co-implementors of APL\B5500 for numerous helpful suggestions regarding the preparation of this manuscript.

Table of contents

List of figures

List of tables

---

1. APL\360 and APL\B5500 transliteration
2. APL\B5500 monitor commands
3. APL\B5500 function editor commands
4. Priviledged monitor commands
5. Infix to Reverse Inverted Polish transformations

Section 1: Introduction

APL\B5500 is a multiple-user interpretive system for a conversational programming language implemented on the Burroughs B5500 computer at the University of Washington. The language is patterned after APL\360 [see Reference 1], which is an implementation of "Iverson Notation" [Ref.2]. The APL\B5500 system provides line-by-line evaluation of APL statements as input by a programmer at a remote teletype station. The system provides both an "immediate execution mode" and a "stored program facility". The basic data elements of APL are numeric and character constants. Identifiers, however, can be used to name numeric and character data for later reference. In addition, the data elements are in the form of scalars, vectors, and arrays. A large number of special-purpose operators operate on the data elements, allowing concise expression of mathematical or manipulative constructs.

A comprehensive set of commands allows communication with the APL system monitor, providing a number of facilities useful in a conversational programming environment.

The conciseness of APL statement expression, along with APL monitor functions, makes APL\B5500 an excellent interactive programming system.

A full discussion of the capabilities of APL\B5500 are given elsewhere [Ref.3]; the purpose here is to provide a reasonably complete discussion of the internal structure of the system. It is useful, however, to provide an introduction to the language, as well as to point out major differences between the APL\360 and APL\B5500 languages.

The structure of APL statements is most easily shown with a simple example. Consider the following ALGOL 60 program:

```
begin integer n; real t;
read (n);
   begin real array a[1:n]; integer i;
   for i:=1 step 1 until n do
      begin read (a[i]); t:=t+a[i]
      end;
   t:=t/n
   end;
write (t)
end;
```

which calculates the average value from a set of values stored in a dynamically allocated array A (This ALGOL example includes the use of "read"

and "write" procedure calls, which produce the obvious effect. Clearly, the computation could be performed without the array A; it is included here in order that the dynamic storage allocation can be compared.). With input data

5, 5, 4, 4, 8, 16

the ALGOL program produces the ouput

7

An APL statement pair which performs the same computation is as follows:

```
X := 5 5 4 4 8 16
(+/X) % RHO X
```

Dynamic storage allocation occurs on the first line, where a vector constant is assigned to the variable X. The second line performs the required computation, and causes the numeric scalar result to be printed.

The second APL statement contains three operators which require explanation: the reduction operator (/), the divide operator (%), and the RHO operator.

The reduction operator applies the operator occurring on its left to the vector to its right, by "placing" the operator between each element of the vector. Thus, since

X = 5 5 4 4 8 16

then

(+/X)

is equivalent to

5 + 5 + 4 + 4 + 8 + 16

In this case, the divide operator divides the scalar to its left by the scalar value occurring on its right. The divide operator, as used here, is said to be "dyadic", since it occurs between two operands (i.e., it operates on two operands, resulting in a single operand).

The RHO operator is "monadic" in this example, since it operates on only one operand (the one which occurs on its right). The RHO operator is used here to extract the "dimensionality" of the vector X. RHO X results in the scalar value 6, since X represents a vector with six elements. Thus, the second APL expression

(+/X) % RHO X

reduces to a final scalar result through the following steps:

```
(+/X) % RHO X
(+/X) % 6
```

```
(+/(5 5 4 4 8 16)) % 6
(42) % 6
7
```

It should also be noted that there is no hierarchy of operator evaluation in an APL statement. All operators are applied from right to left in the statement: the order of evaluation can be controlled, however, by properly parenthesizing sub-expressions. Hence, the APL statement

```
+/X % RHO X
```

reduces to (approximately) the same result as above, through the steps:

```
+/X % RHO X
+/X % 6
+/(5 5 4 4 8 16) % 6
+/(5%6 5%6 4%6 4%6 8%6 16%6)
((5%6)+((5%6)+((4%6)+((4%6)+((8%6)+(16%6))))))
7
```

Note that, in this case, the divide operator divides the vector on its left by the scalar on its right, resulting in a vector. The results may actually differ, of course, due to truncation errors.

There are approximately forty-five special-purpose operators in APL\B5500. Most of these operators can be taken as monadic or dyadic, depending upon the context in which they are used.

As mentioned previously, APL statements can be grouped together in stored programs or "functions". The APL programmer defines a function by entering "function definition mode". This is accomplished by typing a $ ("del") at the teletype, followed by an identifier which names the function (Although all APL programs are called "functions, it is not necessary that the program return a value in the usual functional sense.).

An APL function defined for the purpose of calculating the average of a set of numbers follows:

```
      $AVERAGE
[1]    X:=[]
[2]    (+/X) % RHO X
      $
```

The line numbers enclosed in square brackets on the left are typed out automatically by APL\B5500 when the user is operating in function definition mode. In addition to automatic line numbering, an extensive set of commands is provided for displaying defined functions, deleting function lines, and altering previously typed lines.

The above function is invoked by typing

```
AVERAGE
```

at the teletype. The function begins execution at line one, and immediately

encounters the [] ("quad"). Executing the quad causes the teletype to be opened for input with the prefix

```
[]:
```

indicating that APL requires input data from the terminal station in order to continue execution. The APL programmer may then type input data, such as:

```
5 5 4 4 8 16
```

and the result

```
7
```

is printed at the teletype.

As mentioned above, APL\B5500 provides the APL user with a set of monitor communication commands. These monitor commands allow the user to sign onto the APL system, interrogate APL regarding the contents of a work area, maintain separate work areas, and provide APL execution parameters for his programs.

All APL monitor commands are prefixed with a ")" by the user, in order to distinguish them from other APL statements. Most of the APL\B5500 commands correspond exactly to APL\360 commands [Ref.1]. A more complete discussion of APL monitor commands is found in a later section; the sample terminal session given in Figure 1 below, however, includes a number of monitor command examples ("<--" indicates lines typed by the APL user).

```
XXXXXXXXXXXXX
 D. NIXON LOGGED IN WEDNESDAY 10-21-70  10:42
      X:=5 5 4 4 8 16<--
      X<--
5   5   4   4   8   16

      $AVERAGE[[]]$<--

      AVERAGE
[1]    X:=[]
[2]    (+/X) % RHO X


      AVERAGE<--
[]:
      5 5 4 4 8 16<--
7

      )VARS<--
AVERAGE(F) X        Y
      )FNS<--
AVERAGE
      )DIGITS<--
9
```

```
      )DIGITS 3<--
      1 % 3<--
0.333
      )DIGITS 5<--
      1 % 3<--
0.33333
      )WIDTH<--
72
      )LOGGED<--
(1) IS D. NIXON
(2) IS P. NIXON
      )ORIGIN<--
1
      )FUZZ<--
1@-11
      )SEED<--
59823125
      )CLEAR<--
      )VARS<--
NULL.
      )OFF<--
      END OF RUN
```

Figure 1: Sample APL\B5500 terminal session

1.1 Hardware configuration required for APL\B5500

APL\B5500 is implemented on a Burroughs B5500 computer system. The machine used in the implementation is a single processor system with 32,768 words of 48-bit central memory. Messages to and from remote teletypes are buffered in a single Burroughs B487 Data Transmission Terminal Unit (DTTU). The B487 DTTU is interfaced with model 33 or model 35 teletypes through line adaptors and Western Electric 103A2 (dial-up) data sets. The equipment required for remote operation of APL\B5500 is a model 33 or model 35 teletype with attached acoustic coupler or data set. The remote teletypes must operate in half-duplex mode. In addition, teletypes may be directly connected to the B487 DTTU through line adaptors.

The virtual memory of the APL system requires access to at least one B475 Disk File Storage Module (9.6 million character capacity).

No line printers, tape drives, or card readers are required for normal APL operation.

A complete description of the B5500 hardware components is given in the B5500 hardware reference manual [Ref.4].

1.2 Software environment required by APL\B5500

APL\B5500 is designed to operate concurrently with other batch and

conversational programs under control of the B5500 multiprogramming Master Control Program (MCP). APL is coded entirely in B5500 Extended ALGOL, except for a few statements which allow APL to directly communicate with the MCP. For the most part, APL runs under the same conditions as many B5500 user program, and thus enjoys the protection and facilities (e.g., dynamic storage allocation, automatic overlay, and disk-file Input/Output facilities) provided by the MCP.

A primary design objective in the organization of APL was that the resulting system operation interfer as little as possible with normal B5500 user program processing. In light of this objective, APL central memory requirements are approximately 3000 words of resident (non-overlayable) storage, with an additional 7000 words of transient (overlayable) storage. Resident and transient requirements can be altered at APL system compilation time, with a corresponding trade-off in system response time.

The current version of APL\B5500 also requires the services of a separate priviledged program called the "remote handler". The remote handler interrogates the B487 DTTU, and passes messages between the B487 and APL. APL has been coded in such a way as to allow the remote handler to be removed, and its functions taken over by APL, with a small amount of re-coding.

In many ways, APL\B5500 can be considered a time-sharing submonitor and language processor under the B5500 MCP, since it:

(1) handles its own virtual memory,
(2) handles its own terminal Input/Output processing,
(3) handles execution of APL statements and functions,
(4) schedules APL user tasks and APL monitor tasks for execution,
(5) maintains back-up storage for APL work areas, and
(6) provides an APL-oriented command language for user control of APL monitor functions.

After initial connection of a user terminal to the B5500, the terminal is under control of the APL\B5500 system.

A complete description of the B5500 software is given in the "Narrative Description of the B5500 Master Control Program" [Ref.5].

Section 2: APL\B5500 language description

The APL\B5500 statement and monitor command syntax is, for the most part, structurally equivalent to AL\360, with a transliteration of the APL\360 character set. The correspondence between the two languages is maintained as much as possible, in order that an APL programmer can easily make the transition from one language to the other. In addition to the usual APL\360 operators, the proposed epsilon operator ("execute string") is implemented in APL\B5500, as shown in Table 1 below. The monadic epsilon operator operats on a vector character string containing an APL statement. The result of the operation is the result of the evaluation of the APL statement contained in

the character string. Thus,

```
EPS "2+3"
```

results in the scalar 5. If the APL statement is invalid, an appropriate error message is printed.

Table 1: APL\360 and APL\B5500 transliteration

(Note: some APL\360 characters are not shown below.)

APL\360	APL\B5500	Monadic form	Dyadic form
+	+	identity	addition
-	-	additive inverse	subtraction
x	&	sign	multiplication
	%	mult inverse	division
*	*	exponential	exponentiation
	LOG	natural logarithm	logarithm
	CEIL or MAX	ceiling	maximum
	FLR or MIN	floor	minimum
|	ABS or RESD	absolute value	residue
!	FACT or COMB	factorial	combinatorial
?	RNDM	random number	random deal
~	NOT	negation	
o	CIRCLE	circular	circular
<	LSS		less than
<=	LEQ		less or equal
=	=		equals
=/	NEQ		not equal
>=	GEQ		greater or equal
>	GTR		greater than
	AND		and
	OR		or
	NAND		nand
	NOR		nor
	IOTA	index generator	indexing
	RHO	dimension vector	restructuring
,	,	ravel	catenation
	TRANS	transpose	
	BASVAL	base-2 value	base value
	REP		representation
	EPS	execute	membership
	TAKE		take
	DROP		drop
;	;		heterogeneous output
/	/		compression
\	\	scan	expansion
	PHI	reversal	rotation
	SORTUP	sorting up	
	SORTDN	sorting down	

APL\360	APL\B5500	Usage

RETURN key	<--	End of message signal
[]		Input, or display
["]		Character input
=: or GO		Transfer control
:=		Assignment
[..;..;..]	[..;..;..;..]	Subscripts
-	#	Minus sign
E	@ or E	Power of ten
$		Function definition
'string'	"string"	String quotes

APL\B5500 monitor commands are summarized in Table 2 below. The command structure is similar to that of APL\360, except for the "SYN", "NOSYN", "STORE", "ABORT", and "line edit" commands.

Table 2: APL\B5500 monitor commands

Monitor command	Monitor function
)SAVE <name>	All variables and functions in the active work area are stored in a disk file library. The library is labeled with the user's B5500 <job number> and the identifier specified by <name>.
)SAVE <name> LOCK	This command performs the same function as above, except that all other APL users are prevented from accessing the library.
)LOAD <name>	The library labeled <job number> and <name> is activated for the user. All library variables and functions are accessible after the LOAD operation.
)LOAD <job number>,<name>	This command allows access to saved libraries of other APL users when the library was originally saved without the lock option. The <job number> corresponds to the user who originally saved the library.
)COPY <name>,<function>	This command adds the function named by <function> to the active work area for the user from the library labeled by <name>.
)COPY <job number>,<name> <function>	This command has the same function as the copy command above, except that another APL user's library may be referenced.

)CLEAR <name>

This command removes the referenced library from the disk.

)CLEAR

This command causes all variables and functions in the active work area to be erased.

)ERASE <name>

This command selectively erases variables or functions named by <name> from the active work area.

)FNS

This command provides the user with a list of all defined functions in the active work area.

)VARS

This command lists all variables and function names in the active work area. Functions are identified by a following "(F)".

)SI

This command lists the names of all suspended functions in the active work area.

)SIV

This command lists the names of local variables in suspended functions, as well as the function name.

)ABORT

This command terminates all suspended functions.

)STORE

This command stores variables into the active work area which are global to suspended functions, and which have been altered during function execution. If the ABORT command is issued before the STORE in suspended mode, global variables are left in their original state for re-execution at a later time.

)"<search string>"<insert string>"<search string>"

This is the Line Edit command. The command is used to alter the most recently typed line. This command is described fully in the text.

)ORIGIN <integer>

The origin (first subscript) of arrays is assumed to be that specified by the

integer value <integer>.

)WIDTH <integer>

This command changes the width of the output line to <integer> characters.

)DIGITS <integer>

This command changes the number of digits printed after the decimal point in output to <integer> digits.

)SEED <integer>

This command changes the base of the random number generator.

)SYN

This command causes APL to check each line typed by the user in function definition mode for syntactic correctness.

)NOSYN

This command reverses the action of the SYN command above.

)LOGGED

This command lists the terminal number and user identification of each active APL user.

)MSG <integer> <message>

The MSG command allows active APL users to communicate. The <integer> is the terminal number of the station which is to receive the character string <message>.

NOTE: If the <integer> in any of the commands ORIGIN, WIDTH, DIGITS, or SEED is omitted, then the current value assumed by APL is printed.

The line edit command is particularly useful when only a slight error has been made in a line typed by the user. The form of the Line Edit is:

)"<search string>"<insert string>"<search string>"
or
)"<search string>"<insert string>"

In either case, the last message typed by the APL user is edited according to the Line Edit command, and resubmitted for processing by APL. The action of the Line Edit is as follows: the first <search string> is located by APL in the last line typed by the user; when it is found, the <insert string> is placed into the line, and all characters up to the occurrence of the second <search string> are deleted. If the first <search string> is not found, then no changes are made. If the second <search string> is not found, then all characters after the <insert string> are deleted. Finally, if the second

<search string> is not specified, then no characters are deleted. The null string is found immediately in all cases. Thus, if the user first types:

```
((+/(X-AVE)*2/% N-1)*.5
```

he will receive an error message (unbalanced parenthesis: "*2/" should have been typed as "*2)"). The line can be altered by typing:

```
)"2")"%"
```

and APL will respond with:

```
((+/X-AVE)*2)% N-1)*.5
```

The statement is then resubmitted for execution. This command is extremely useful when a long expression has been typed which needs simple alteration. A similar command is available in function definition mode, allowing alteration of all or part of a function definition.

APL\B5500 also differs from APL\360 in the method of handling global variables when executing functions. Functions which contain errors (syntactic or semantic) are "suspended" at the point where the error occurs. Suspended functions may have operated upon global variables to produce new values for the global variables. The altered values are not permanently entered into the active work area until the function has successfully completed, or until the user has issued the STORE command while the function is suspended. This feature allows re-execution of the corrected function, without re-initialization of the global data.

The APL\B5500 function editor differs somewhat from the APL\360 editor. The APL editor is invoked whenever the APL user types a $ ("del") followed by a function "header" while in calculator mode. The simplest form of a function header is an APL identifier. Hence, if the user types:

```
$F
```

APL will enter function definition mode, and (assuming F is a new function) will respond with:

```
[1]
```

and await the first line of the function F by opening the teletype for input. As subsequent lines of text are entered, the line counter is incremented by one. Thus, the user could enter the three lines:

```
[1]  A
[2]  B
[3]  C
```

with the line numbers to the left supplied automatically by APL. Although APL will "prompt" the user for the fourth line, it is possible to insert lines elsewhere in the function. The user could, for example, insert a line between lines one and two by replying to the prompt with:

[4] [1.1]D

overriding the line prompt. APL will then take the increment last used by the APL programmer, and prompt with:

[1.2]

The smallest increment possible is .0001 between lines. The largest line number possible is 9999.9999.

In general, any line prefixed by a "[" while in function definition mode is taken to be an editor command. Table 3 below provides a complete listing of APL\B5500 editor command.

Table 3: APL\B5500 function editor commands

APL editor command	Command function
[[]]	
This command causes the currently active function to be displayed at the terminal.	
[<line reference>[]]	
This command causes the line specified by the <line reference> to be displayed at the terminal.	
[<line reference>[]<line reference>]	
This command causes all lines from the first through the second <line reference> to be displayed at the terminal.	
[<line reference>]<statement>	
This command inserts the APL statement specified by <statement> in the current function at the time denoted by <line reference>. The current line and the increment are changed in most cases.	
[<line reference>]	
This command deletes the function line corresponding to <line reference>.	
[<line reference>][<line reference>]	
This command deletes all lines from the first through the second <line reference>.	
[IOTA]	
This command causes the current function to be completely renumbered, starting at one with an increment of one.	

[["]]<line edit>

This command causes the APL editor to alter all lines of the current function, according to the rule given in the <line edit>. The <line edit> is the same as the Line Edit command described under APL monitor commands (in Table 2).

[<line reference>["]]<line edit>]

This command is similar to the above edit command, except that only the line referred to by <line reference> is altered.

[<line reference>["]<line reference>]<line edit>

This command applies the <line edit> from the line specified by the first <line reference> through the line specified by the second <line reference>.

The <line reference> is a basic constituent in almost all editor commands. In the simplest case, the <line reference> is an integer value corresponding to a line of a function. Thus (referring to the delete command of Table 3 above), the user could delete the first three lines of the above function by typing (after the APL prompt):

```
[1.2] [1][2]
```

APL deletes the lines, and returns the prompt:

```
[1.2]
```

opening the terminal for input. Note that the function F now contains:

```
[3]    C
```

Another type of <line reference> is an APL statement label. Statements are labeled by placing identifiers separated by colons before the APL statement. Thus, the APL user may continue definition of F by typing (with prompting by APL):

```
[1.2]  [4]D
[5]    E: F+G
[6]    L1:L2:H+I
[7]
```

where "E", "L1", and "L2" are all statement labels. Although statement labels are used primarily for transfer of control at function execution time, they can be used as <line reference>s when in function definition mode. The line with <line reference< 5 can be deleted by typing either of the following commands:

```
[5]
[E]
```

A <line reference> may also involve a numeric offset on either side of

the statement label. Line 5 can be displayed by typing:

```
[L1-1[]]
```

and APL will respond:

```
[5]    E:F+G
```

Further, an entire set of lines around statement five may be displayed by typing:

```
[E-1[]E+1]
```

resulting in the response from APL:

```
[4]    D
[5]    E: F+G
[6]    L1:L2:H+I
```

APL allows statement labels to be edited as well. The statement at line six can be edited by typing:

```
[L2["]]:L"3"
```

APL searches the line labeled L2 for an occurrence of ":L", inserts a "3" immediately after the occurrence, and deletes characters up to the following ":". Hence, the command:

```
[L2[]]
```

results in an error message (the label L2 no longer exists), but the command:

```
[L3[]]
```

results in the display:

```
[6]    L1:L3:H+I
```

A last point which should be made is that labels within functions are treated as local variables, but are initialized to their respective line numbers. The line number value of a label may be altered during function execution. Further examples of function definition are given in the discussion of the APL\B5500 function editor implementation.

Appendix A shows a sample APL\B5500 terminal session including examples of APL operators, APL monitor commands, and APL function editor usage.

A formal definition of the syntax of APL\B5500 is included in Appendix B.

Section 3: APL\B5500 implementation

The internal data structures and program organization of APL\B5500 are given in the following sections. The time-sharing facilities of APL are explained, along with a description of monitor command execution and APL "machine" organization. APL\B5500 functions are logically divided into components parts, as shown in Figure 2 below:

1. APL Resource management. Central memory and central processor resources are allocated by the Resource Management component of APL\B5500.

2. Terminal Input/Output handler. Terminal message bufferring and dispatching, along with primitive Input/Output facilities, are provided by the Terminal Input/Output handler.

3. Virtual memory management. The Virtual Memory Management section provides an APL controlled extension of the B5500 central memory resources.

4. APL function editor. Terminal messages issued while the user is in function definition mode are processed by the APL Function Editor.

5. Monitor command handler. All terminal input messages which are prefixed with a ")" (i.e., APL monitor commands) are processed by the Monitor Command Handler.

6. APL statement compiler. The APL Statement Compiler checks the syntax of APL statements submitted for execution by the user. In most cases, "pseudo-code" is generated, corresponding to the APL statement.

7. APL "machine". The APL "machine" is a software simulation of a computing machine oriented toward execution of APL statements.

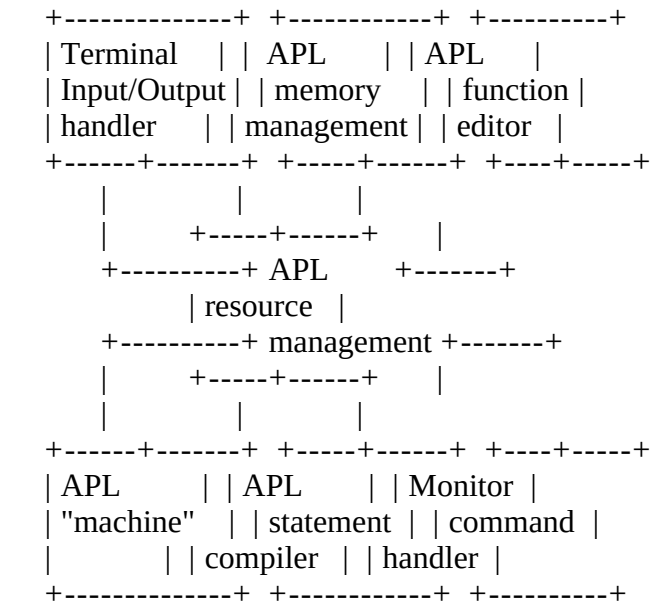


Figure 2: APL\B5500 software components

3.1 APL resource management

APL Resource Management is responsible for allocation of work to the other components of the APL system. In addition, the needs of the various terminal users are monitored constantly.

The current "state" of each active APL user is maintained in a table called the User State Table, shown in Figure 3 below. Each element of the User State Table, called a User State Register, corresponds to exactly one APL user. The field width of each element in the User State Register varies according to the maximum data size.

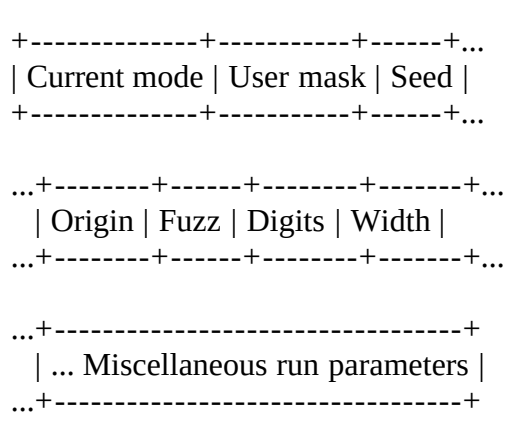


Figure 3a: User State Register

User State Register 1
User State Register 2
...
User State Register i <---- Current user
...
User State Register n

Figure 3b: User State Table

APL schedules tasks for execution based on a simple two-queue algorithm [Ref.6], with tasks which have not required a full time-slice in a FIFO (first-in first-out) queue for immediate processing. A production queue lists all tasks which require central processor resources, and which have used at least one time-slice. Users without a task in the immediate queue or production queue are considered to be in an "idle" queue.

The "current mode" field of the User State Register indicates the present status of the corresponding user's APL run. The current mode of a particular user can be:

1. Caculator mode. The user is in an idle state, and is not using the APL Function Editor. Further, the user is not executing APL statements. APL is awaiting input from the user's terminal.

2. Execution mode. The user is in the process of executing an APL statement.

The APL statement may or may not have invoked functions.

3. Function definition mode. The user is currently defining an APL function. All messages, except those prefixed by ")", are directed to the APL Function Editor.

4. Input mode. A user in execution mode is changed to input mode when his APL program requests input from the terminal (by encountering a "quad" or "quote quad"). The user is restored to execution mode when input is completed.

5. Error mode. A user is put into error mode when his program encounters an error during execution. Messages are sent to the terminal, and corrective action is taken before changing the user's current mode.

Concurrency of APL tasks is thus maintained by retaining the status of each user, in order that his task may be started and stopped in various states of execution. The CURRENT USER is set by the Resource Manager, before execution of an APL task is initiated. The current user is indicated by an index to the corresponding User State Register, as shown in Figure 3b above.

When control is given to another APL component, such as the Function Editor, a small increment of processing is done for the current user, with control returning to the Resource Manager almost immediately. Since the parameters required by each component are maintained in the User State Registers, it is possible for one user to define one line of an APL function and, immediately after, another user can use the function editor to define a line of his APL function.

This notion of concurrency is, of course, fundamental to the operation of any operating system, including time-sharing. Each component of the system must be coded in such a way as to return control to the Resource Management component as soon as possible, in order that other system users do not notice any delay. This notion is referred to here as "functional concurrency".

The "user mask" field of the User State Register shown in Figure 3a above contains a set of binary "switches" associated with the user's APL run. The bit positions of the user mask include:

1. The master mode bit. The master mode bit is set if the User State Register belongs to the APL system supervisor. The system supervisor's user code is compiled into the APL system, and thus is the only INITIAL valid user. A user operating with the master mode bit set (i.e., the system supervisor) is provided with additional monitor commands which allow user codes and user phrases to be added to or deleted from the APL system.

2. The debug bit. The debug bit can be set by a user operating in master mode. When this bit is set, various system diagnostic information is provided at the user's terminal.

3. The nosyntax bit. The nosyntax bit of the User State Register can be set with the "NOSYN" monitor command. Input lines typed by the user in function definition mode are not checked by APL for syntactic correctness when the nosyntax bit is set.

4. The suspension bit. The suspension bit is set when the user's APL execution encounters an error. The function in error is "suspended", and may be re-activated at a later time.

The remaining fields of the User State Register contain values of run time parameters, along with variable information used by the various components of the APL system. The actual contentof the "miscellaneous run parameters" field, shown in Figure 3a above, is discussed in detail when the individual components are considered.

It should be noted that the system components, other than the Resource Manager, need not be concerned with keeping track of the active system users. Once the Resource Manager selects a user for execution, the other system components refer to the User State Register indicated by the current user index in the User State Table. Thus, the individual components act upon data either located in the current User State Register, or upon data addressed by fields within the current User State Register.

A simplified state diagram, given in Figure 4 (ROCHE> Too difficult to translate into "ASCII graphics". Maybe a GIF of JPEG file will be needed to display it?), shows the action of the Resource Manager.

3.2 The terminal Input/Output handler

The Terminal Input/Output Handler provides an interface between the APL system components and the terminal users. This interface includes message queueing facilities, I/O interrupt handling, input message scanning and translation, and output formatting capabilities. In addition, the Terminal Input/Output Handler adds items to the immediate queue for processing at the appropriate times. The I/O functions are grouped into the following types:

1. Message queue and table maintenance. Information is maintained describing the status of each terminal port. I/O buffers and queues are also kept in order by this component.

2. Input message scanner. The scanner provides a common facility for extracting lexical elements from the input messages corresponding to the current user.

3. Output formatting routines. All preparation of output messages from the APL system components is handled by the Output Formatting Routines.

The message queue and table maintenance component of the I/O Handler maintains the status of each terminal port in the Station Table, shown along with the relevant fields of a Station Element in Figure 5 below. The Terminal Input/Output Handler maintains the Station Table in order that it may determine for each terminal port:

1. if the station is physically connected to the B5500 system (physical connection bit),

2. if the terminal has an input message or messages waiting to be processed by

APL (read ready bit),

3. if the terminal is set-up to accept a message (write ready bit),

4. if a message has been sent to a station but transmission has not been completed (output finish wait bit),

5. if some components of the APL system has requested that a message be readied for processing (input request bit),

6. if an APL component has passed a message through the I/O Handler to be written on the terminal (output request bit),

7. if the "break key" at the terminal has been depressed by the APL user (break key depression bit),

8. if an APL system component has acknowledged that the break key has been depressed, has taken the appropriate action, and has requested that the break key depression bit be reset (break key reset bit),

9. the number of messages from the terminal which have not yet been processed by APL (input queue size),

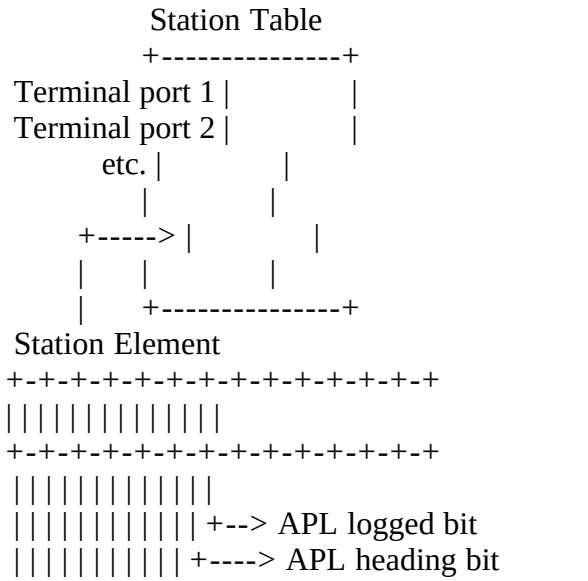
10. the number of messages produced by APL which have not yet been sent to the terminal because the station is not write ready (output queue size),

11. if the maximum number of messages in the input queue or in the output queue has been reached (input or output queue size exceeded bit),

12. if the APL\B5500 heading has been printed at the station (APL heading bit), and

13. if the user has successfully signed-on to the APL system (APL logged bit).

Thus, since there is a Station Element for each terminal port, the I/O Handler can immediately determine the I/O status of any terminal.



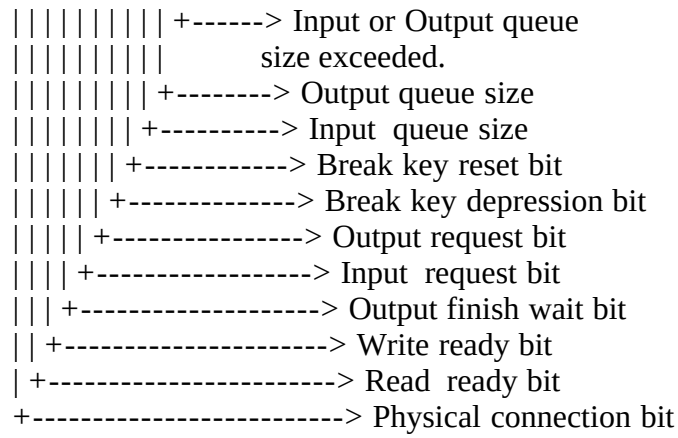
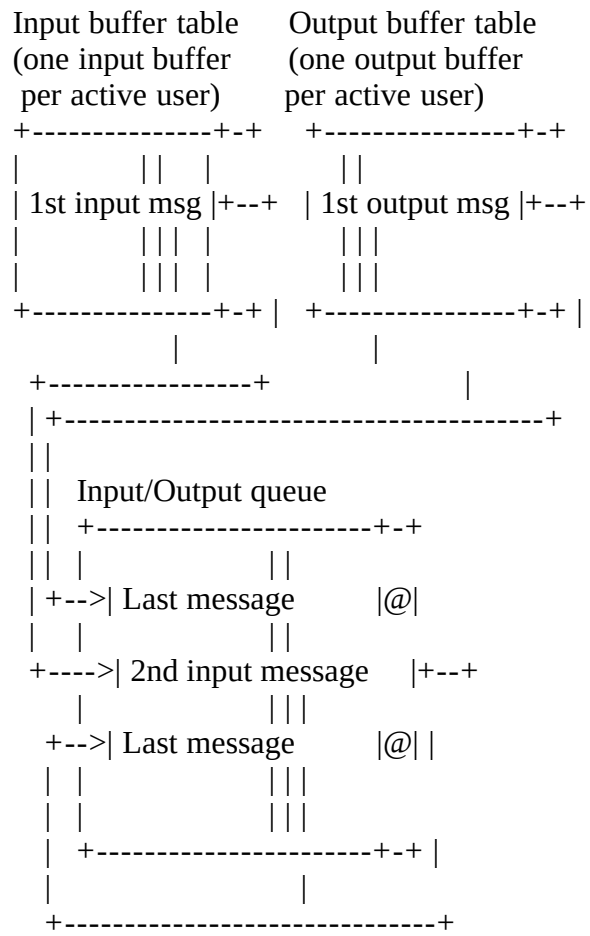


Figure 5: station table with corresponding station element

Input/Output buffers are maintained for active APL users, as shown in Figure 6 below.



Note: The end of list is denoted by "@".

Figure 6: Input/Output buffers and queues

An I/O queue is maintained on back-up storage with forward pointers (starting at the input or output buffer) connecting all elements of the queue

for a particular user. The disk I/O queue may, at a particular point in time, contain both input and output messages in transit to or from the APL system.

The I/O queue are, of course, maintained on a first-in first-out basis, except when a time-slice "jiggle" is sent by an APL component. The jiggle is a null message which rattles the teletype typing mechanism, letting the system user know that APL processing is in progress. The jiggle message goes to the front of the output queue for a particular user.

Since the terminal user normally waits for a response from APL for each line of input, the input queue will rarely contain more than one message.

The Terminal Input/Output Handler has access to the scheduling queues which are searched by the Resource Manager. Thus, when a particular user sends a message for APL processing while in the idle queue, the I/O Handler places the user in the immediate queue for processing. The Resource Manager allocates processor time to the user when the user gets to the fron of the immediate queue.

The state diagram of Figure 7 (ROCHE> Too difficult to translate into "ASCII graphics". Maybe a GIF or JPEG file will be needed?) shows the logic of the I/O Handler in processing terminal messages.

The scanning and formatting routines provide APL system component interface with the I/O Handler. The input message scanner provides lexical analysis of input messages upon request by APL components. The terminal input buffer referenced by the scanner is always that of the current user, as defined by the Resource Manager. The scanner provides translation from external symbols to internal coded form, identifies and converts positive and negative integer and real numbers, as well as numbers in power-of-ten notation. In addition, APL variable, function, and command identifiers are isolated by the scanner. Except for string constants, all input to the APL components is processed by the input scanner.

Conversely, all output from APL components directed to terminal users is funneled through the output formatting routines. The formatting routines provide the APL system with primitive formatting capabilities; character strings are appended to the output buffer belonging to the current user according to the following output controls:

1. append characters to those already in the current user's output buffer, but do not send the message to the station (there is more to come),

2. append characters to the characters in the current user's output buffer, and queue the message for output,

3. first, send the contents of the current user's output buffer; place characters in the output buffer, but do not send the message,

4. first, send the contents of the current user's output buffer to the terminal, place characters in the output buffer, and send this second message also.

In conclusion, the Terminal Input/Output Handler processes all

terminal messages sent from other APL components, or sent to APL components from the user's terminals. Functional concurrency is maintained by initiating as many terminal message transfers as possible without causing unnecessary delays before returning to the Resource Manager.

3.3 Virtual memory management

As mentioned previously, a fundamental design criterion was that APL\B5500 interfere as little as possible with normal B5500 operations. In particular, the central memory requirements for APL must be minimized, without causing an excessive increase in overall response time. One solution to the storage problem might be to use the automatic overlay feature of the B5500 MCP. Automatic overlay, however, cannot be directly controlled by APL\B5500. Thus, the data areas used by APL are extended beyond the central memory areas, through the use of an APL-suited virtual memory structure.

The APL Virtual Memory is a completely independent component of the APL\B5500 system, but is used in conjunction with a central memory data area called the "scratch pad". The scratch pad data area make use of the automatic overlay features of the MCP, while the virtual memory is directly controlled by APL.

Although the physical structure of the virtual memory is described elsewhere [Ref.7], enough detail is given here, in order that one may understand its use by the various APL system components.

The virtual memory is PHYSICALLY structured using simple demand paging techniques [Ref.8]. A file on back-up storage is divided into "pages" (the page size is determined at compile time) with an index to these pages residing in central memory. In addition, a number of central memory "page frames" are maintained in central memory to hold most-recently accessed pages. The number of page frames can be altered dynamically by other APL components, depending upon APL storage requirements and the number of active APL users.

Virtual memory access routines provide the interface between APL system components and the APL Virtual Memory. The virtual memory access routines give the virtual memory a CONCEPTUAL structure, which is quite different from the physical structure.

Conceptually, the APL virtual memory is divided into "storage units". The storage units can be created dynamically by APL components, and are of no predetermined size (except for the maximum extent of the disk file).

The storage units, in turn, can be of two types: "ordered" storage, or "sequential" storage. Ordered storage units contain data in tables appearing to the APL components as contiguous alphabetically arranged data with fixed field lengths. Sequential storage units contain data elements of variable length, but are only accessible through a fixed address within the storage unit. The maximum number of storage units which can be active at any given time is 512.

Figure 8 below shows the conceptual structure of virtual memory.

Ordered and sequential storage units are often related through a right-most field in ordered storage unit elements. This field might contain the address of a data element in an associated sequential storage unit, although this assumption is not made by the virtual memory routines.

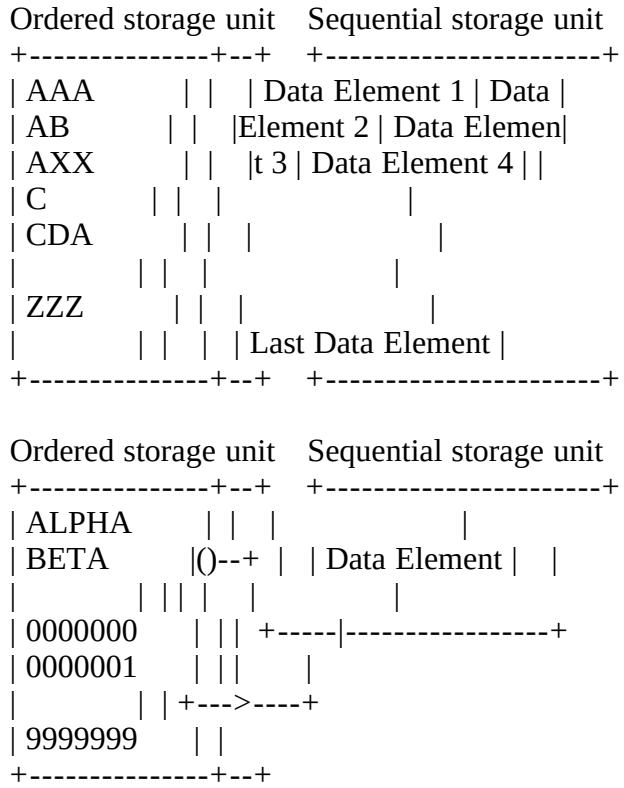


Figure 8: Conceptual structure of virtual memory

APL initializes with storage units one and two reserved for user codes and associated user phrases, as shown in Figure 9 below. Initially, storage unit one contains only the system supervisor's user code, and unit two contains his user phrase. Units one and two increase in size as the system supervisor adds more user codes and user phrases.

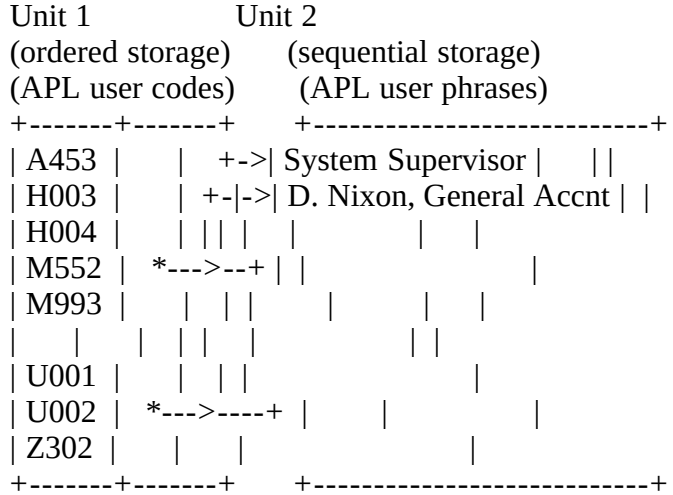


Figure 9: User codes and user phrases

The services provided for APL components by the virtual memory access routines can be categorized as follows:

3.3.1 Storage maintenance
---

a. A particular direct access disk file can be named for use as a virtual memory back-up storage unit.

b. Particular storage units can be created and destroyed.

c. The number of active page frames can be increased or decreased.

d. APL components may designate that vital information, necessary for proper system recovery in case of failure, be written onto back-up storage.

3.3.2 Storage interrogation and alteration
---

a. Variable-length data can be stored in a specified sequential storage unit.

b. Ordered storage units may be searched for a particular data item on demand by an APL component.

c. Information can be inserted into a specified ordered storage unit.

d. The contents of a particular address in either ordered or sequential storage can be retrieved.

e. Elements in either ordered or sequential storage units can be deleted.

f. Entire storage units can be erased with the corresponding data areas added to free storage.

3.3.3 Storage utility functions
---

a. The number corresponding to the next available storage unit can be obtained by an APL component. The unit can then be used for storage.

b. The size (number of data elements) in a specific storage unit can be requested by an APL component.

c. The mode of a particular unit can be determined (i.e., whether the unit has been designated as an ordered or a sequential storage unit).

The work area of an APL user may consist of several ordered and sequential storage units. At sign-on time, the user is assigned an ordered

storage unit, called "names", and a sequential storage unit, called "data". The "names" unit contains variable and function names, along with additional information about the names. The "data" storage unit holds numeric and character array results computed during the APL run. As shown in Figure 10 below, the "data" storage unit also contains a "recent" copy of the user's User State Register. This recent copy of the User State Register allows a user to restart an APL session with very little loss of work in the case of a hardware or software failure.

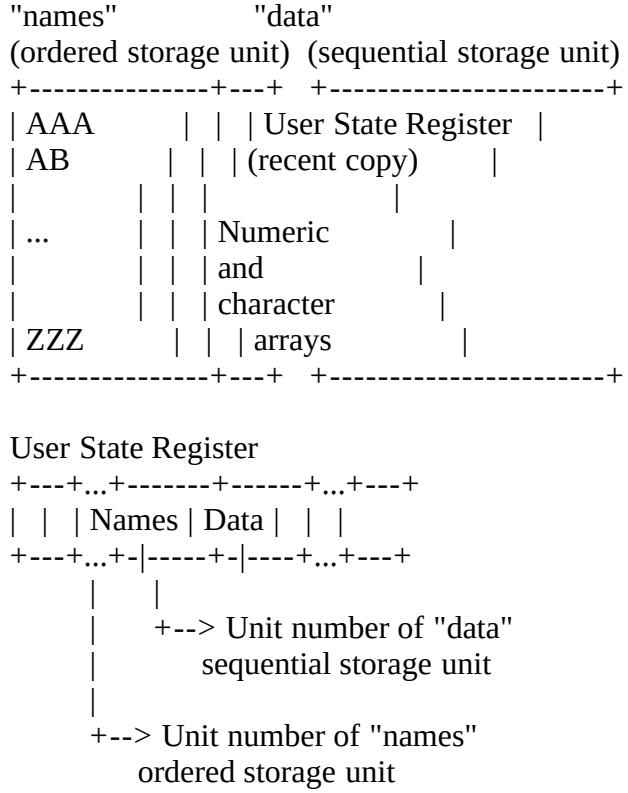


Figure 10: Storage units for names and data

The important concept here is that at any point in the APL run, a portion of the user's work area, consisting of ordered and sequential storage units, is located in central memory page frames, while the remainder is located on back-up storage. The portion in central memory is based entirely on data access activity. This, of course, is a fundamental concept in any implementation of demand paging. The scheme does, however, allow the storage units to become much larger than would be possible if all tables and data were to remain in central memory.

It will become evident in later sections that the ordered and sequential storage units, along with the virtual memory access routines are well-suited to the needs of the APL system.

As a final note, the APL Virtual Memory Manager, like other APL components, maintains functional concurrency. When the virtual memory manager has back-up storage maintenance to perform, it does so in small increments each time it is called. Thus, control returns to the Resource Manager as soon as possible.

3.4 The APL function editor

The APL Function Editor component of the APL\B5500 system provides function definition and editing capabilities for APL users. The Function Editor handles the syntax of the function header, and creates internal data structures from the header to pass to the other APL components. The Editor relies upon the virtual memory access routines in implementing the editing functions.

Every function defined by the APL user causes two units of storage to be allocated: an ordered storage unit called a "function label unit", and a sequential storage unit called a "function text unit". The function label unit contains entries corresponding to the line numbers of the function, along with addresses of lines of function text in the corresponding function text unit.

Figure 11 below shows the interconnection of the function label unit and the function text unit. The left-most field of each entry of the function label unit contains the line numbers in full character form (without a decimal point). The right-most field contains the address of the corresponding line in the function text unit. Note that the function header is assumed to be line zero of the function.

Addition and deletions of text and line numbers is accomplsihed by using the corresponding virtual memory access routines.

```
      Function definition
      -------------------
      $STDEV
[1]    AVE := (+/X)% N := RHO X
[2]    ( (+/(X-AVE)*2)%N-1 )*.5
[3]    $
```

```
      Storage unit structure
      ----------------------
Function label unit     Function text unit
(ordered)               (sequential)
+------------+---+      +-------+---------------+
| 00000000   | *-|-->---| STDEV | AVE := (+/X)% |
+------------+---+      +-------+-|-------------+
| 00010000   | *-|-->------->-----+    +----<-----+
+------------+---+      +------------+-|-------+| |
| 00020000   | *-|-->-+ | N := RHO X | ( (+/(X- | |
+------------+---+    | +------------+-----+----+ |
                 | | AVE)*2)%N-1 )*.5 |    | |
                 | +------------------+----+ |
                 +-------->------------>-----+
```

Figure 11: Function storage units

The Function Editor also keeps track of local variables and labels in functions. As shown in Figure 12 below, a number of cases occur:

1. the local variables are all marked with a right-most field in the function label unit which is less than or equal to zero:

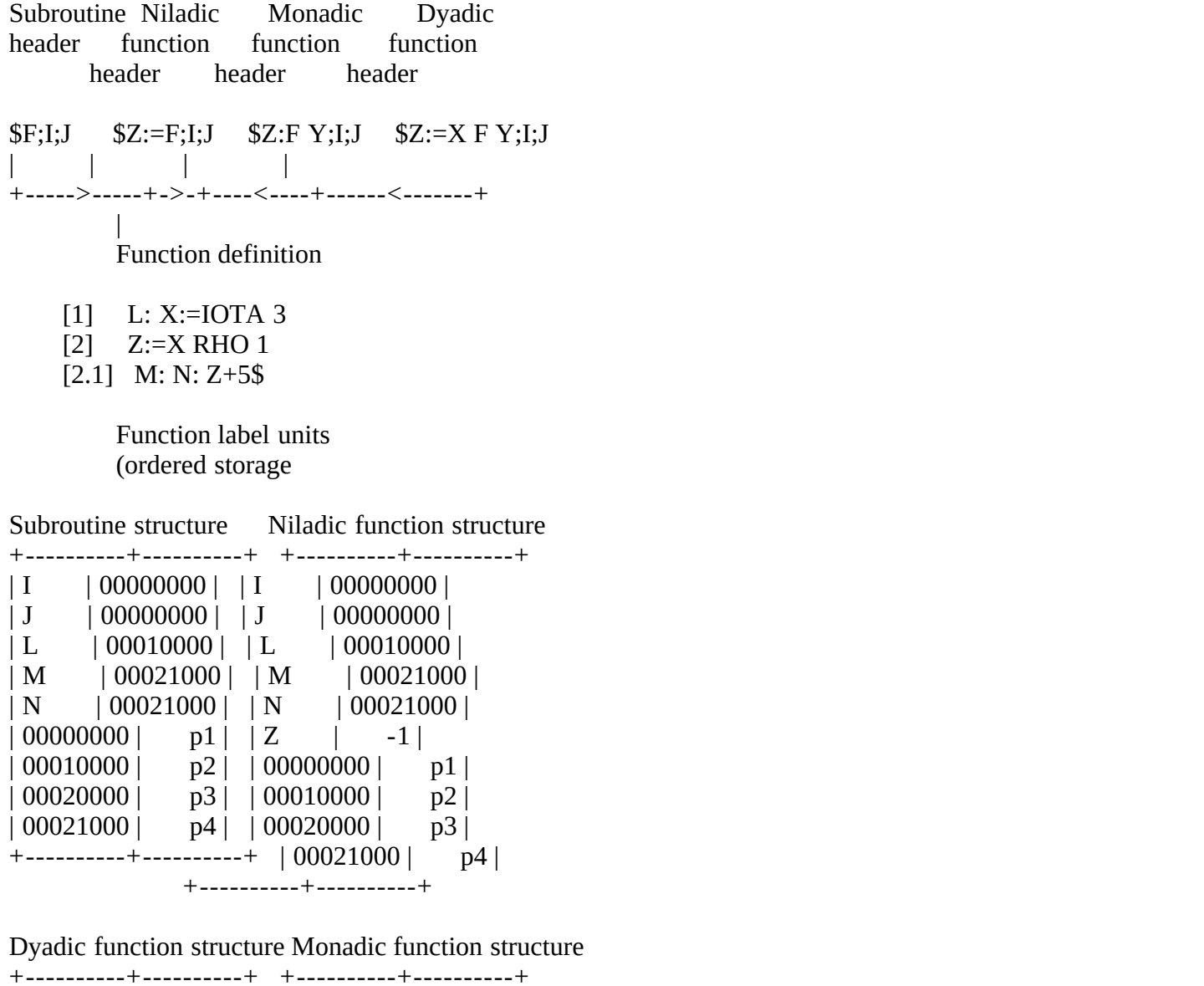
a. the local variable which contains the value to be returned at the end of function execution is marked with a negative one,

b. the arguments (formal parameters) are marked with a minus two and minus three,

c. all other local variables are marked with zeroes.

2. labels are marked with the full character representation of their corresponding line numbers.

Case (2) above allows access to lines of text through the line labels.



```
| I       | 00000000 |   | I       | 00000000 |
| J       | 00000000 |   | J       | 00000000 |
| L       | 00010000 |   | L       | 00010000 |
| M       | 00021000 |   | M       | 00021000 |
| N       | 00021000 |   | N       | 00021000 |
| X       |      -3 |   | Y       |      -2 |
| Y       |      -2 |   | Z       |      -1 |
| Z       |      -1 |   | 00000000 |      p1 |
| 00000000 |      p1 |   | 00010000 |      p2 |
| 00010000 |      p2 |   | 00020000 |      p3 |
| 00020000 |      p3 |   | 00021000 |      p4 |
| 00021000 |      p4 |   +----------+----------+
+----------+----------+
```

NOTE: F is the function name, I and J are local variables, X and Y are formal parameters, and Z denotes the value returned by F. The values p1,p2,p3,p4 represent the addresses of the corresponding lines of text in the function text unit.

Figure 12: Function label unit structure

Like all other APL system components, the Function Editor must maintain functional concurrency. Clearly, there are many situations where functional concurrency becomes a problem (e.g., displaying lines of text). Thus, the Function Editor maintains a number of variables in the User State Register, as shown in Figure 13 below. The Function Editor fields are only defined, when the user is in function definition mode, when they contain:

1. the number of the ordered storage unit assigned as the function label unit,

2. the number of the sequential storage unit assigned as the function text unit,

3. the name of the function being edited,

4. the current line of the function being defined,

5. the current line increment for this user,

6. the editing "submode" (i.e., deleting text, editing, or displaying lines of text),

7. the editing submode boundaries (e.g., the starting and ending line numbers for the display command).

User State Register

```
+------------+
| .......... | User must be in function definition mode
+------------+
| Function   | Ordered storage unit corresponding to
| label unit | function label unit
+------------+
```

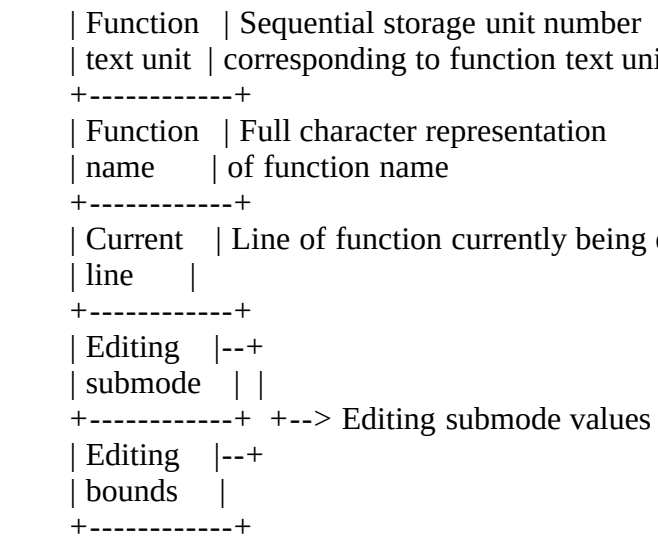


Figure 13: User state register entries for the function editor

A corresponding entry is made in the "names" ordered storage unit for this user as soon as the function definition mode is closed. The entry for a function consists of the name of the function in the left-most field, and the numbers corresponding to the function label unit and the function text unit in the right-most field.

Note also that the Function Editor examines the nosyntax bit of the user mask whenever a new line is inserted or an old line is edited in a function. If the nosyntax bit is reset, then the Editor passes the APL statement to the APL Statement Compiler for a syntax check. The user is notified if errors are detected.

The state diagram of Figure 14 (ROCHE> Too difficult to translate to "ASCII graphics. Maybe a GIF or JPEG file will be needed?) below shows the basic logic of the Function Editor.

Section 5: The APL monitor command handler

After a terminal user has initially signed-on to the APL\B5500 system, the Resource Manager passes all messages which begin with a ")" to the Monitor Command Handler for processing. The Monitor Command Handler processes the monitor commands shown in Table 2 above (in "APL\B5500 language description"), along with the system supervisor commands listed in Table 4 below.

Table 4: Priviledged monitor commands

Monitor command	Command function
)ASSIGN <user code> <user phrase>	

This command assigns a new user code to be recognized by APL. The <user code>

goes into the user code ordered storage unit. The <user phrase> goes into the user phrase sequential storage unit, and serves to identify the user to other APL users.

)DELETE <user code>

This command removes a user code and associated user phrase from the APL system.

)LIST CODES

This command provides a listing at the system supervisor's terminal of all assigned user codes.

)LIST USERS

This command provides a complete listing of all assigned user codes and user phrases.

)DEBUG MEMORY <integer>

This command specifies that a trace of APL virtual memory activity be given. The <integer> specifies trace options.

)DEBUG POLISH

This command causes the APL statement compiler to print a trace of the code produced for each APL statement executed by the system supervisor.

The entire set of monitor commands can be categorized as:

5.1 System maintenance commands

The system maintenance commands allow the APL system supervisor to add, delete, and alter user codes and user phrases. In addition, the supervisor can set system diagnostic flags. These commands are recognized only when the master mode bit is set in the user's User State Register.

5.2 Work area maintenance commands

Work area maintenance commands allow the APL user to add or delete items from his associated work area. The user may also save the work area in a separate file, and later reactivate the work area.

5.3 APL run parameter specification

Variables which affect the APL run for a user can be displayed and altered through APL monitor commands (e.g., "WIDTH" and "ORIGIN").

5.4 Line edit command

The last line entered by each user can be altered and re-submitted, as discussed previously using the Line Edit command.

5.5 Function suspension commands

The function suspension commands allow the user to control function execution when functions have been suspended due to errors.

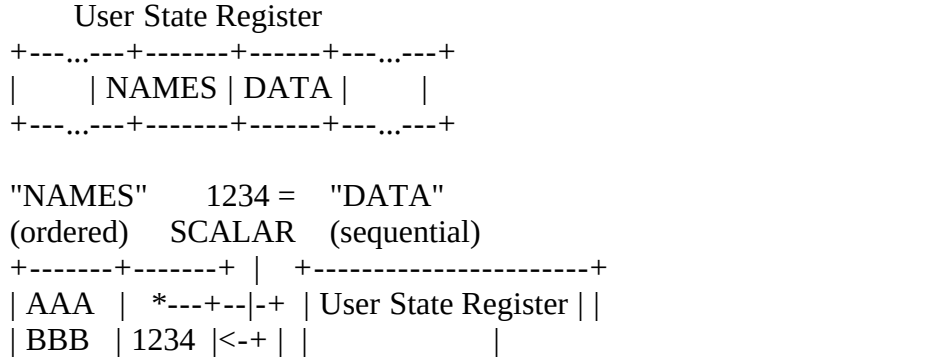
5.6 Run termination commands

The APL user may terminate the APL run using a number of different options.

The implementation of most of the monitor communication algorithms is straightforward. It is useful, however, to examine the data structures involved in these operations.

If the monitor command to be executed is a system maintenance command, the master bit of the User State Register for the current user is examined. If this bit is reset, then the user is issued an error message. Otherwise, the Monitor Command Handler uses the virtual memory access routines to examine, add to, or delete from, the user code ordered storage unit and the user phrase sequential storage unit.

The work area maintenance commands access the "names" ordered storage unit. The variables and functions can be listed and deleted by application of the appropriate virtual memory access routines. In addition, the total content of the work area may be copied to an external library for later use. This operation involves accessing and copying all ordered and sequential storage units allocated for the user's work area. The "names" ordered storage unit provides an entry point for referencing all variables and functions. The library is constructed by first constructing a dictionary, as shown in Figure 15 below.



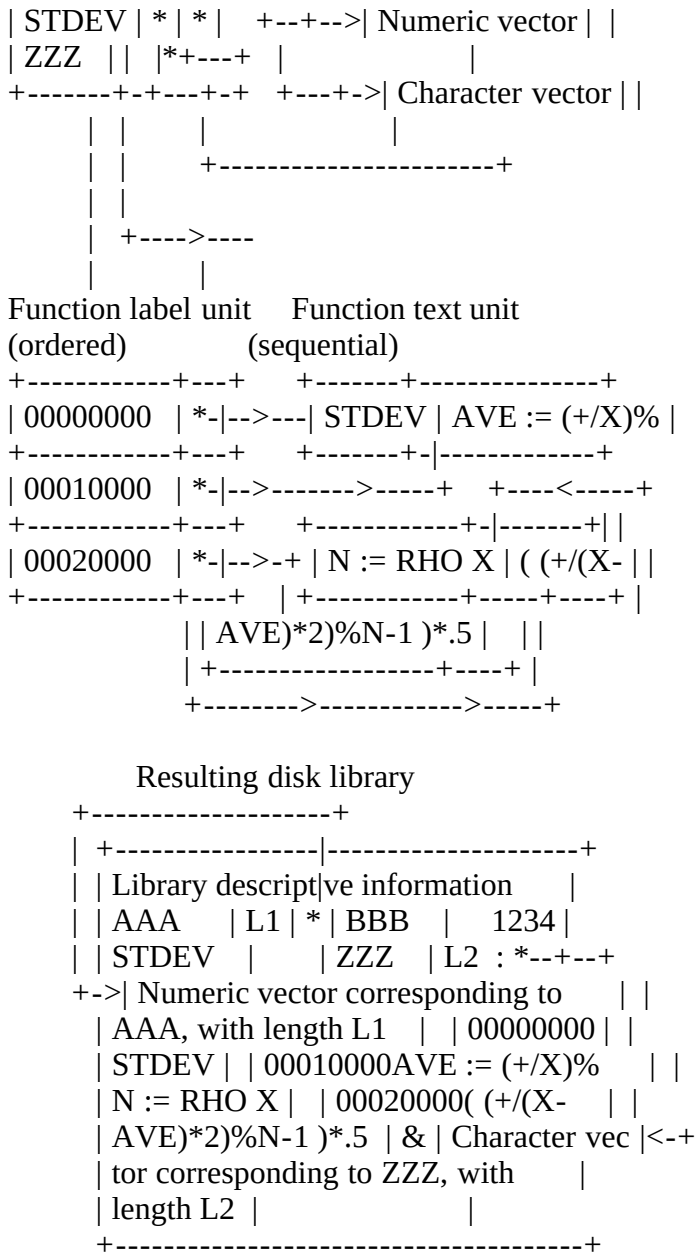


Figure 15: The format of a library

All non-scalar data is copied into the library from the "data" sequential storage unit, with appropriate addresses in the library dictionary. Whenever functions are encountered in the "names" unit, the corresponding function label unit and function text unit are accessed through the unit numbers in the right-most field of the function entry. The line label, along with the function text for each line, is forward-chained for each function. The dictionary entry for the function addresses the head of this chain.

Library load and copy operations reference the directory of a particular library to obtain addresses and data lengths in the library. The load and copy operations occur in just the opposite order from the save operation. The situation arises, however, when copying functions into an active work area, where the function name being copied is identical to a variable name occurring in the "names" unit. In this case, the variable, along with the corresponding data, is removed from the work area before copying the

function.

All of the above operations make use of the virtual memory access routines in searching and altering storage units.

The APL run parameter specification commands are easily implemented. Display is accomplished by referencing the corresponding field of the User State Register (e.g., the "digits" field). Similarly, the fields may be altered directly on command by the user. Thus, if the command is "DIGITS", the "digits" field is retrieved from the User State Register, and displayed. If the message typed by the user is "DIGITS 3", then a new value of three is inserted in the user's User State Register.

The Line Edit command is implemented by retaining a copy of the last message typed by the user in calculator mode (initially, the null message). The Line Edit is processed according to the rules given earlier, and a "simulated" teletype input is performed with the new edited line. The simulated input, however, goes to the beginning of the input queue for the user. The Line Edit command does not replace the last message typed by the user; hence, it is possible to edit the same line several times.

User to user communication is made possible with two monitor commands: the "LOGGED" command, and the "MSG" command. The first command displays the user phrases corresponding to each active APL user, along with the user's station number. The Monitor Command Handler refers to the user phrase storage unit to obtain this information.

The users may communicate as shown in Figure 16 below. The user specifies the station number with the "MSG" command of the user which is to receive the message. The message is extracted from the originator's input, and placed (with the proper prefix) at the beginning of the output queue of the station receiving the message.

```
      FROM (2): ARE YOU GOING TO BE WORKING LATE TONIGHT...
      )MSG 2 I THINK I WILL QUIT ABOUT MIDNIGHT<--
      X<--
3
      FACT X*2<--
362880
      LOG 473<--
6.1591
      FROM (2): HAVE YOU FINISHED THE NUMERICAL ANALYSIS ASSIGNMENT...
      )MSG 2 I ALMOST HAVE THE BIG ANSWER<--
      L0G 474<--
SYNTAX ERROR AT 474
      LOG 474<--
6.16121
      (LOG 3)+(LOG 4)<--
2.48491
      LOG 12<--
2.48491
      FROM (2): DO YOU HAVE A SAVED COPY THAT I CAN COMPARE WITH.
      )MSG 2 YES I SAVED ONE ABOUT 1 HOUR AGO<--
```

```
      LOG 362880
12.80183
      FROM (2): IT IS UNLOCKED... WHAT IS THE NAME OF THE LIBRARY...
      )MSG 2 IT IS INTERPOLLY... READY TO GO<--
      )VARS<--
INTERP (F) STRING  X  Y
      STRING<--
A VERY FAT CAT

      INTERP<--
      FROM (2): OK... I AM GOING TO LOAD IT...
      INTERPOLATION PROBLEM C1

INPUT X VALUES

[]:
      V:= 2 4 6 8 10 12 15 20<--
INPUT Y VALUES

[]:
      LOG V<--
INPUT VALUE TO INTERPOLATE

[]:
      13<--
INTERPOLATED VALUE IS 2.56564

      LOG 13<--
2.56495
      )OFF<--
      END OF RUN
```

Figure 16: User/user communication

The last command to consider is the "OFF" command. This command informs the APL system that the user wishes to discontinue the APL session. Two options are available:

1. "OFF", and
2. "OFF DISCARD".

In case (1), APL assumes the user wishes to be physically disconnected from the system with the active work area saved under the library name "CONTINUE". The appropriate bits are reset in the station table entry for the port, and the library is constructed. A termination message is then printed, followed by deallocation of data areas (storage units, buffers, and registers.

Case (2) is similar to the first, except that a library is not constructed. In either case, assumes that some user wishes to sign-on again after a short period. The teminal is not physically disconnected, and the buffers are retained for this port until a fixed time has elapsed without a sign-on at the terminal.

The monitor command handler is distinct from the other APL components, but provides a command language and command facilities which are useful in the APL environment.

In conclusion, it can be easily seen that the Monitor Command Handler makes use of the virtual memory access routines in the implementation of nearly all the commands.

Section 6: Storage and representation of APL data structures

The methods used in data storage and representation are fundamental in the understanding of the two APL components remaining to be discussed: the APL Statement Compiler, and the APL "machine".

The fast-access data area mentioned earlier, called the scratch pad, contains data which is "active". Further, each data item residing in the scratch pad has an associated "descriptor", which gives the characteristics of the data. The organization of the scratch pad, data layout, and descriptor formats are the subjects of this section.

The scratch pad may be considered the memory of the simulated APL machine. The scratch pad is, in fact, an array which increases and decreases in size as the requirements for working storage increase and decrease. Space is allocated within the scratch pad using a variation of simple segmenting [Ref.9].

All APL data in a particular user's work area can be considered "active" or "passive". Data can be active for a user only when the user is executing an APL statement or function, and the data has been referenced during the execution. Passive data is that data which can be referenced through the "names" ordered storage unit assigned to the user. References to passive data may occur when the user is executing an APL statement. In this case, a copy of the passive data is brought into the scratch pad during the computation. Active data in the scratch pad may replace passive data in the user's work area at the end of execution (i.e., the user returns to calculator mode from execution mode). In addition, new results may have been computed during execution, causing additions to the "names" and "data" storage units.

At any point in the execution of several user's APL programs, the scratch pad contains active data for all of these users. The passive data, however, is kept distinct in the individual "names" and "data" storage units referenced through the corresponding User State Registers.

Data which is active in the scratch pad is identified through the use of "descriptors". The descriptors, shown in Figure 17 below, identify data by providing the following information:

1. Descriptor identification bit. The descriptor identification bit is set if the descriptor refers to APL data.

2. Data presence bit. The data presence bit is set when the data corresponding to the descriptor is present in scratch pad memory.

3. Named bit. The named bit is set in a descriptor when the data associated with a descriptor is not a temporary result.

4. Scalar bit. The scalar bit is set in descriptors which reference scalar data.

5. Character bit. The character bit is set in a descriptor when the descriptor refers to a character array rather than numeric data.

6. Back pointer field. The back pointer field is primarily used to identify the origin of the descriptor in scratch pad memory.

7. Rank field. The rank field of a descriptor contains the number of dimensions in the data associated with the descriptor.

8. Scratch pad field. The scratch pad field holds the actual scratch pad address of the data associated with the descriptor.

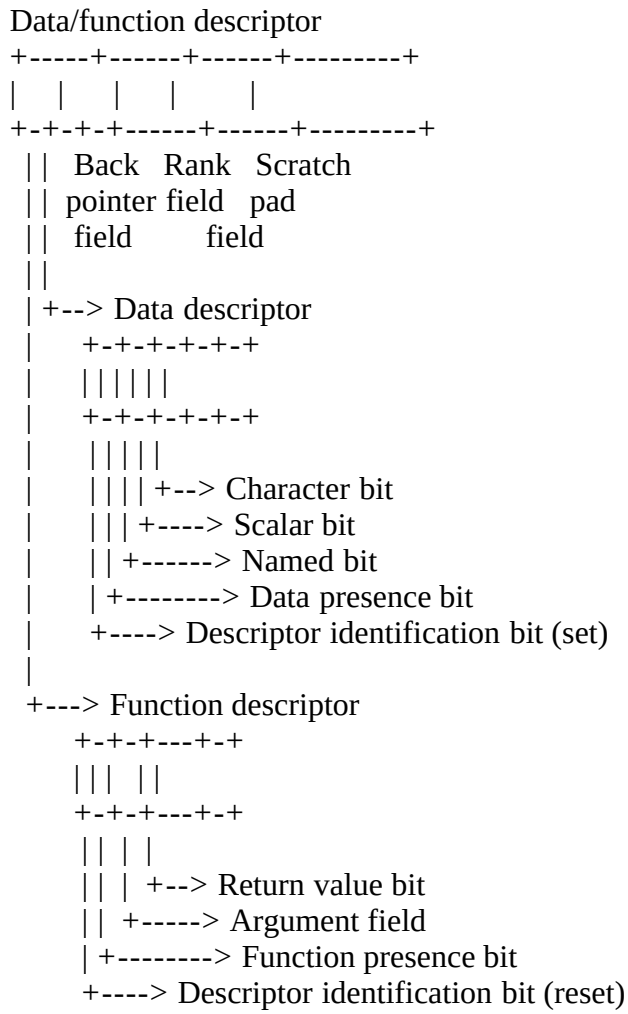


Figure 17: Data and function descriptors

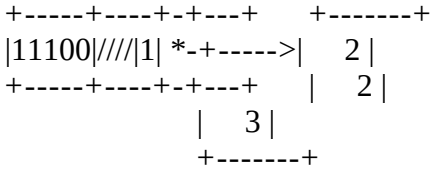
The size of each field depends upon the maximum value that can be assumed in each case.

Data in array form is stored in row-major order, with the dimensionality of the array in the first few locations, as shown in Figure 18 below.

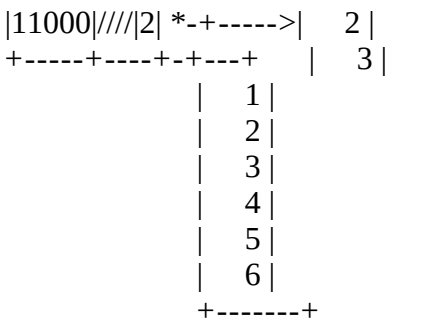
APL statement:
2 3 RHO IOTA 6

Scratch pad representation:

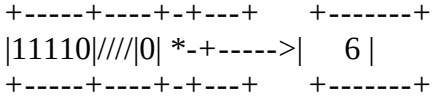
Data descriptor for vector (2 3)



Data descriptor resulting from execution



Data descriptor for constant 6



Data descriptor for IOTA 6

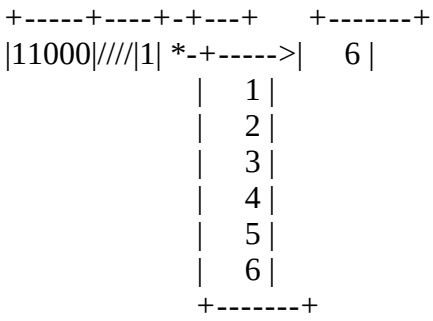


Figure 18: Scratch pad data representation

The use of descriptors allows execution-time determination of the complete meaning of a particular operator. Thus, the meaning of the statement

X+Y

cannot be exactly determined at compile-time, since the "+" could represent a scalar-scalar, scalar-array, or array-array operation. The exact operation is determined at execution time, by examining the data descriptors involved in the operation.

The use of descriptors is also extended to APL functions. Referring again to Figure 17 above, function descriptors contain the following information:

1. Descriptor identification bit. The descriptor identification bit is reset for function descriptors.

2. Argument field. The argument field contains the number of arguments (parameters) required for function execution.

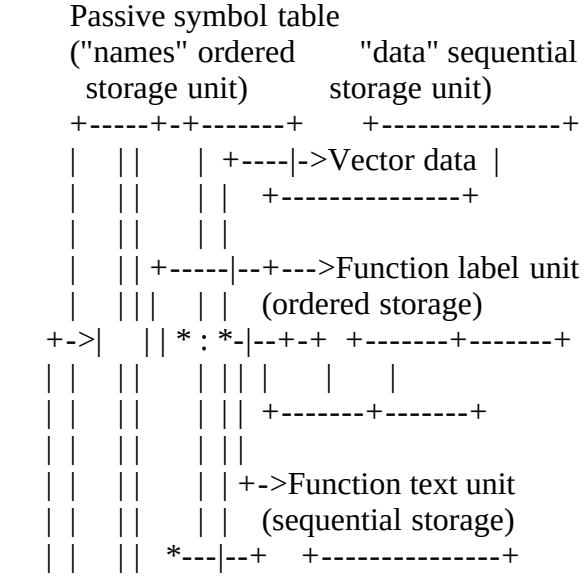
3. Return value bit. The return value bit is set for function descriptors corresponding to functions which return a value from execution.

The presence bit, back pointer field, and scratch pad field are used in the same manner as in the data descriptor.

Descriptor access is accomplished through the symbol tables described in the following section.

Section 7: Active and passive symbol tables

Corresponding to active and passive data and functions in APL\B5500, there are active and passive symbol tables. The passive symbol table is just the "names" ordered storage unit shown in Figure 10 above. The details of the passive symbol table entries are shown in Figure 19 below. The contents of the right-most field of a passive symbol table entry depends upon the type of entry. In particular, a non-present (presence bit reset) data or function descriptor may appear with the name, or simply a scalar value will appear if the name represents a scalar.



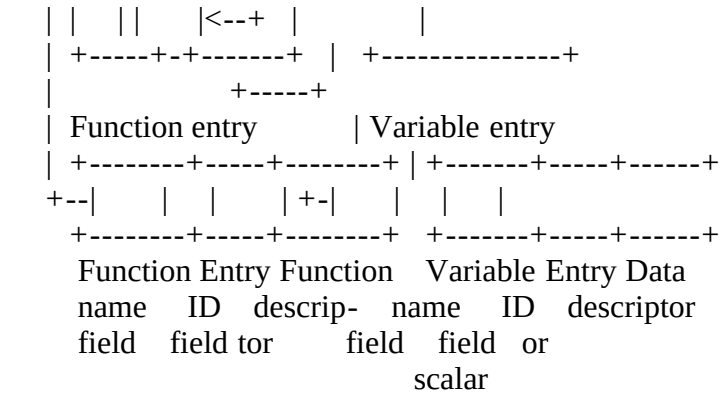


Figure 19: Passive symbol table

Each passive symbol table entry is identified by the entry identification field. The entry identification field may take on one of the following values:

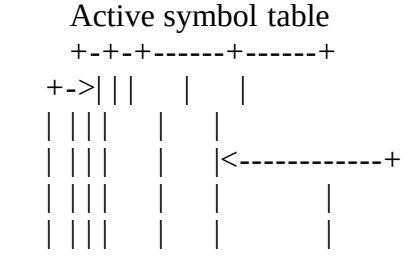
1. Scalar. The name corresponding to the entry is a scalar. The scalar vale is contained in the right-most field of the passive symbol table entry.

2. Array. The entry corresponds to an array variable. The right-most field contains a non-present data descriptor. The scratch pad field contains an address in the "data" sequential storage unit, where the corresponding data can be found. The data is loaded into the scratch pad when the variable becomes active and is accessed.

3. Function. The entry represents a defined function. The right-most field contains a non-present descriptor. The back pointer field, however, contains the unit number of the function label unit, and the scratch pad field contains the number of the function text unit corresponding to the function.

The passive symbol table is always searched using the virtual memory access routines.

The active symbol table exists for a particular user only during execution of a statement or function. The active symbol table, shown in Figure 20 below, is located in the scrtach pad, and is addressed through the symbol base field of the user's User State Register. At any given time, there may be several active symbol tables in the scratch pad; one for each user of APL in the process of executing APL statements. The active symbol table contains entries for the active data, not including constants, temporary results, or local variables. The descriptors in the active symbol table may or may not have their presence bits set.



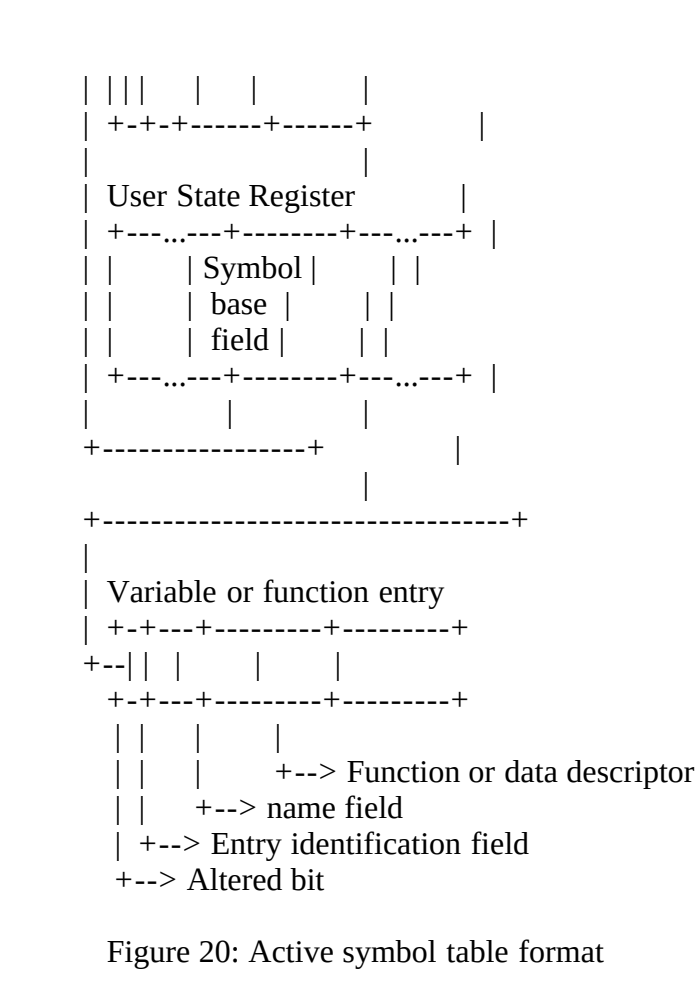
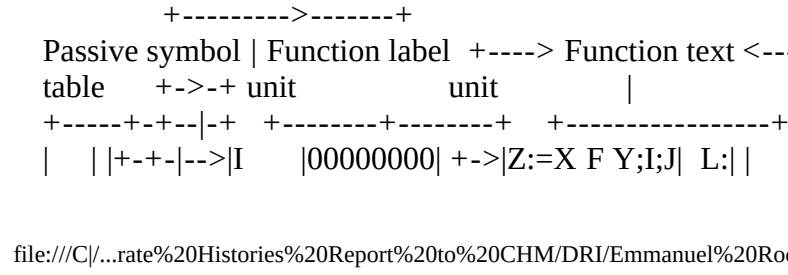


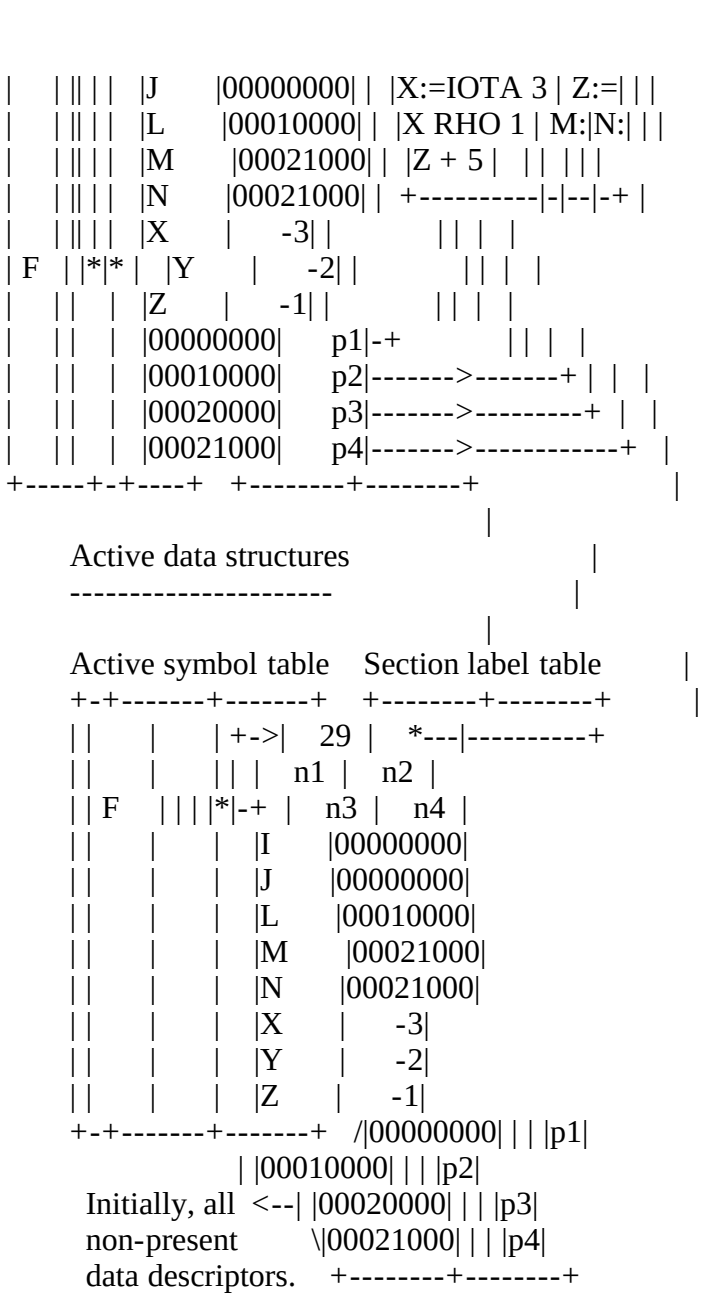
Figure 20: Active symbol table format

The entry identification field (switched to the front of the name for machine-dependent reasons) has an additional bit position, called the "altered bit", in the active symbol table. The altered bit indicates whether or not changes have been made to data which is active, and thus needs to be changed in the "data" storage unit. In addition, the altered bit is set for variables which are created during execution and do not yet exist in the passive symbol table. All variables with their altered bit set are changed or entered in the passive symbol table when the user returns to calculator mode from execution.

Another important symbol table used in compilation and execution of APL statements is called the function label table, shown in Figure 21 below. The function label table is essentially an extension of the active symbol table in the scratch pad. A function label table is constructed whenever an APL statement references a function with the presence bit reset in the corresponding function descriptor. The function descriptor is replaced by a data descriptor referencing the function label table as soon as the table is constructed. The information found in the function label table is derived from the function's corresponding function label unit.

Passive data structures





NOTE: n1 is the relative location of the first numeric label, n2 is the relative location of the first argument, n3 is the relative location of the second argument, and n4 is the relative location of the result.

Figure 21: Function label table structure

The function label table is logically an APL numeric vector referenced by the (now present) data descriptor in the active symbol table. The right-most fields of the numeric labels are initially all non-present data descriptors with the scratch pad fields referencing the corresponding lines of text in the function text unit.

Thus, it is possible to address all variables and functions through the passive symbol table for a particular user. In addition, all active data and functions, along with function labels and local variables, are accessible through the active symbol table and function label table.

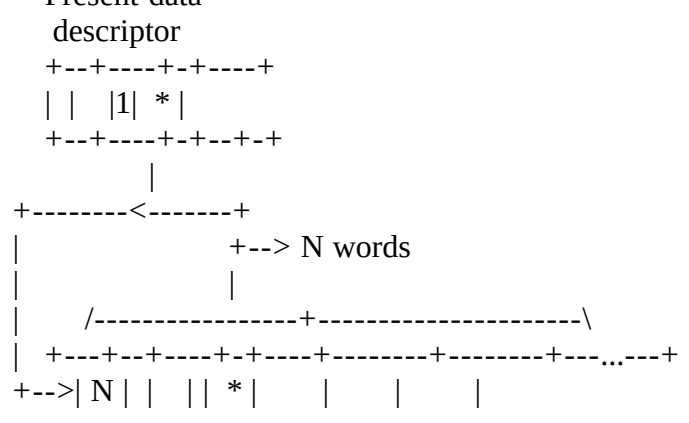
The reasons for maintaining the active and passive areas can be stated as follows: the amount of data and the number of functions in a work area may be voluminous. Further, during the execution of a calculator mode statement (which places the user in execution mode), it may happen that only a small fraction of the work area is actually referenced. Although passive data and functions are readily accessible through the virtual memory, it happens that little used areas remain on back-up storage (because of the demand paging). Active data and functions, however, are not paged out of central memory, since they reside in the scratch pad. The assumption, of course, is that the accessed data has the highest probability of being accessed again, before the user returns to calculator mode.

It should also be noted that lines of text in an active function are compiled only on demand. That is, there must be an attempt by the user's APL program to execute a particular line of an APL function before that line is compiled. Once the line is compiled, it remains in the scratch pad in a compiled form until the user returns to calculator mode. Again, the assumption is that function lines which have been executed have the highest probability of being re-executed. Further discussion of demand paging is found in the sections which follow.

Section 8: The APL statement compiler

The APL Statement Compiler generates an internal pseudo-code string corresponding to APL statements submitted for compilation. The APL Statement Compiler can be called while the current user (as defined by the Resource Manager) is in function definition mode, or in execution mode. If the user is in function definition mode, no code is generated, nor is any scratch pad memory allocated.

In execution mode, the compiler returns a present data descriptor addressing an APL numeric vector in the scratch pad. This vector has, as its first element, a present data descriptor addressing an APL character vector containing the original APL statement, as shown in Figure 22 below. The original statement is maintained with the compiled code, in order that error messages may be printed during execution. In any case, the APL Statement Compiler is called upon to compile only one statement at a time, thus maintaining functional concurrency.



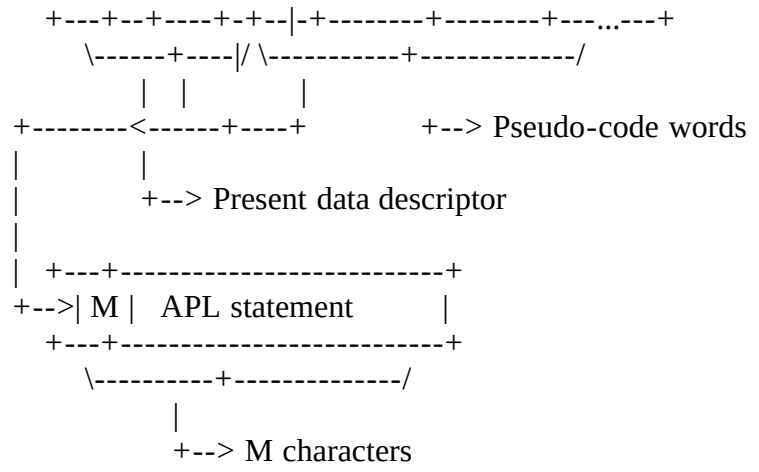
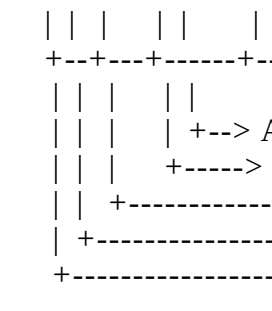


Figure 22: Code string format

APL pseudo-code words, shown in Figure 23 below, are interpreted by the simulated APL machine during execution of the statement.

Pseudo-code words



Operator type field	Type field	Location field
Niladic	Operand	Decriptor
Monadic	Constant	address
Dyadic	Operator	Operator
Triadic	Function	code

Rank field	Address field
Value returned or not returned Operator subscript	Location of lexical item which generated the code word (used for error reporting).

Figure 23: Pseudo-code word format

The type field of the code word indicates whether the code word represents an operand or constant fetch, or an APL operation or defined function call. If the code word represents an operand or constant, then the fields are defined as follows:

1. Operator type. The operator type is not used for operands and constants.

2. Type field. The type field indicates whether the code word represents an operand or a constant.

3. Location field. The location field contains the scratch pad address of the data descriptor corresponding to the code word (i.e., the descriptor for the operand or constant).

4. Rank field. The rank field is not used.

5. Address field. The address field contains the location of the operand or constant within the original APL statement. This location allows exact error reporting during execution.

If the code word represents a function or operator, then the fields are:

1. Operator type. The operator type indicates the number of operands involved in the operation or function.

2. Type field. The type field indicates whether the code word represents a function or operand.

3. Location field. IF the code word represents an APL operator, then the location field contains an integer number assigned to this operator. If the code word describes a function, then the location field contains the address of the descriptor for this function in the active symbol table.

4. Rank field. The rank field contains an operator "subscript" (see the following explanation of the "[" operator) if the code word represents an operator. If the code word represents a function, then the rank field indicates whether or not the function returns a value.

The final code string for an APL statement is generated in two passes: a forward pass, called the lexical pass, and a backward pass, called the code generation pass. The two pass approach is taken in order to arrange the code words in the proper order for a right-to-left execution of the statement.

The lexical pass involves the identification of each lexical item in the infix expression (operators, constants, variables, and functions). During this pass, shown in Figure 24 below, a push-down stack, called Infix, is loaded with code words corresponding to the lexical items. All table look-ups in the function label table, active symbol table, and passive symbol table occur during the lexical pass. In addition, all scalars, along with numeric and character vector constants, are placed into the scratch pad.

Figure 24: Lexical pass state diagram (ROCHE> Too difficult to translate to "ASCII graphics". Maybe a GIF or JPEG file will be needed?)

During the code generation pass, the Infix stack is examined, and a form of suffix notation is generated by rearrangement of the code words. This

form will be termed "reverse inverted Polish", since it involves not only suffix form, but also a rearrangement of the operands (this form can also be thought of as direct Polish written backwards). The reverse inverted Polish form is suitable for execution by the simulated APL machine, and is sufficient for the proper right-to-left execution. The fundamental transformations from infix to reverse inverted Polish form are shown in Table 5 below. Note the "subscript computation" operator in Table 5, denoted by "[[n". This operator is "subscripted" by n; that is, the number of operands involved in the subscript computation is denoted by n.

Table 5: Infix to reverse inverted Polish transformations

Infix	Reverse inverted Polish
m X	X m
X d Y	Y X d
M X	X M
X D Y	Y X D
X[s1;s2;...;Sn]	Sn Sn-1 ... S2 S1 X[[n

NOTE: m denotes a monadic operator, d denotes a dyadic operator, M denotes a monadic function, D denotes a dyadic function, and X, Y, S1, S2 through Sn denote valid APL expressions.

One might think that the usual reverse Polish form would be sufficient for proper APL statement execution. However, statements such as:

A + A:=5

where the variable "A" initially has a value other than five is evaluated incorrectly if reverse Polish form is used.

The logic of the code generation pass is shown in Figure 25 below in simplified form.

Figure 25: Code generation pass (ROCHE> Too difficult to translate to "ASCII graphics". Maybe a GIF or JPEG file will be needed?)

There are a number of special cases not covered by the diagram in Figure 25 above, such as the occurence of the quad or quote quad; however, the diagram does cover most cases. Error conditions are not shown in the state diagram for the code generator, but may be detected in a number of ways, including:

1. attempting to mark an operator or function as dyadic when it is defined as a monadic operator or function, or vice-versa;

2. the absence of an anticipated code word on the operator stack (e.g., delete ")" from the top of the operators stack anticipates the occurrence of the ")"

);

3. examination of the operators stack for extraneous symbols at the end of the transformation.

Figure 26 below shows the steps occurring in the transformation of an APL statement into reverse inverted Polish form. For purposes of illustration, the symbolic equivalent of each code word in the Infix stack is used, rather than the code word itself. Note also that the stacks extend to the right for readability. Operators in the final code string which have been recognized as monadic are marked with a prime (').

```
APL expression:
(D is a dyadic function, M is a monadic function)
A[ABS B+3;C[3]-B:=M 5;(3 D 4)-8]

(1)  I   A[ABS B+3;C[3]-B:=M 5;(3 D 4)-8 ]
 n   O   1]<----------------------------+
 1-->C-->+

(2)  I   A[ABS B+3;C[3]-B:=M 5;(3 D 4)-8
 n   O   1]                      |
 1   C   8<----------------------------+

(3)  I   A[ABS B+3;C[3]-B:=M 5;(3 D 4)-
 n   O   1] -<-----------------------+
 1   C   8

(4)  I   A[ABS B+3;C[3]-B:=M 5;(3 D 4)
 n   O   1]- )<----------------------+
 1   C   8

(5)  I   A[ABS B+3;C[3]-B:=M 5;(3 D 4
 n   O   1]-)                   |
 1   C   8 4<-----------------------+

(6)  I   A[ABS B+3;C[3]-B:=M 5;(3 D
 n   O   1]-) D<------------------+
 1   C   8 4

(7)  I   A[ABS B+3;C[3]-B:=M 5;(3
 n   O   1]-) D------+          |
 1   C   8 4  3  D<--+          |
           +-----------------+

(8)  I   A[ABS B+3;C[3]-B:=M 5;(
 n   O   1]- *--)----+
 1   C   8 4 3 D -<--+

(9)  I   A[ABS B+3;C[3]-B:=M 5;
 n   O   1]
 2   C   8 4 3 D -
```

```
(10) I   A[ABS B+3;C[3]-B:=M 5
 n   O   1]              |
 2   C   8 4 3 D - 5<--------+

(11) I   A[ABS B+3;C[3]-B:=M
 n   O   1] M<-------------+
 2   C   8 4 3 D - 5

(12) I   A[ABS B+3;C[3]-B:=--+
 n   O   1] M---->---+  :=<--+
 2   C   8 4 3 D - 5 M'

(13) I   A[ABS B+3;C[3]-B
 n   O   1] :=----------+-------+
 2   C   8 4 3 D - 5 M' B  :=<--+

(14) I   A[ABS B+3;C[3]-
 n   O   1] -<---------+
 2   C   8 4 3 D - 5 M' B :=

(15) I   A[ABS B+3;C[3]
 n   O   1]- 2 ]<-----+
 1   C   8 4 3 D - 5 M' B :=

(16) I   A[ABS B+3;C[3--->---+
 n   O   1]-2[             |
 1   C   8 4 3 D - 5 M' B := 3

(17) I   A[ABS B+3;C[1--+
 n   O   1]- 2 ] [1<----+
 2   C   8 4 3 D - 5 M' B := 3

(18) I   A[ABS B+3;C----------+
 n   O   1] -*--[1-------------|----+
 2   C   8 4 3 D | - 5 M' B := C [1 -
             +---------------+

(19) I   A[ABS B+3;
 n   O   1]
 3   C   8 4 3 D - 5 M' B := 3 C [1 -

(20) I   A[ABS B+3-------->----------+
 n   O   1]                        |
 3   C   8 4 3 D - 5 M' B := 3 C [1 - 3

(21) I   A[ABS B+
 n   O   1] +<--+
 3   C   8 4 3 D - 5 M' B := 3 C [1 - 3

(22) I   A[ABS B--->----+
 n   O   1] + ----------|->---------------+
 3   C   8 4 3 D - 5 M' B := 3 C [1 - 3 B +

(23) I   A[ABS----+
```

```
 n   O   1] ABS<--+
 3   C   8 4 3 D - 5 M' B := 3 C [1 - 3 B +

(24) I   A[3-+
 n   O   1] [3 ABS------------>------------+
 1   C   8 4 3 D - 5 M' B := 3 C [1 - 3 B + ABS'

(25) I   A-------------------------------------+
     O   [3-----------------------------------|-+
     C   8 4 3 D - 5 M' B := 3 C [1 - 3 B + ABS' A [3
```

Resulting reverse inverted Polish:
8 4 3 D - 5 M' B := 3 C [1 - 3 B + ABS' A [3

NOTE: "I" represents the Infix stack (in symbolic form), "O" represents the Operator stack, and "C" represents the Code string. Each step shows the effects of one circuit through the diagram of Figure 25 above.

Figure 26: Transformation of an APL statement

Figure 27 below shows the evaluation of the resulting reverse inverted Polish taken from Figure 26 above. Rectangles enclose operands and operators which result in a single operand.

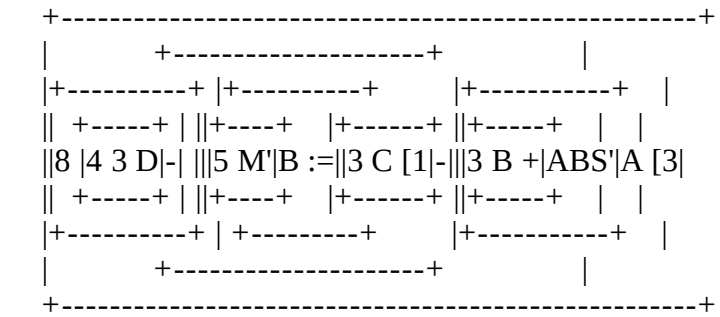
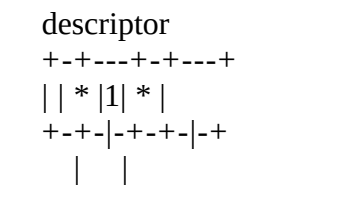


Figure 27: Evaluation of reverse inverted Polish

All constants created during the lexical pass are attached to the code string through the data descriptor returned by the statement compiler, as shown in Figure 28 below. Thus, the data descriptor returned by the statement compiler provides access to the constants associated with the APL statement, the APL statement itself, and the pseudo-code words corresponding to the original statement.



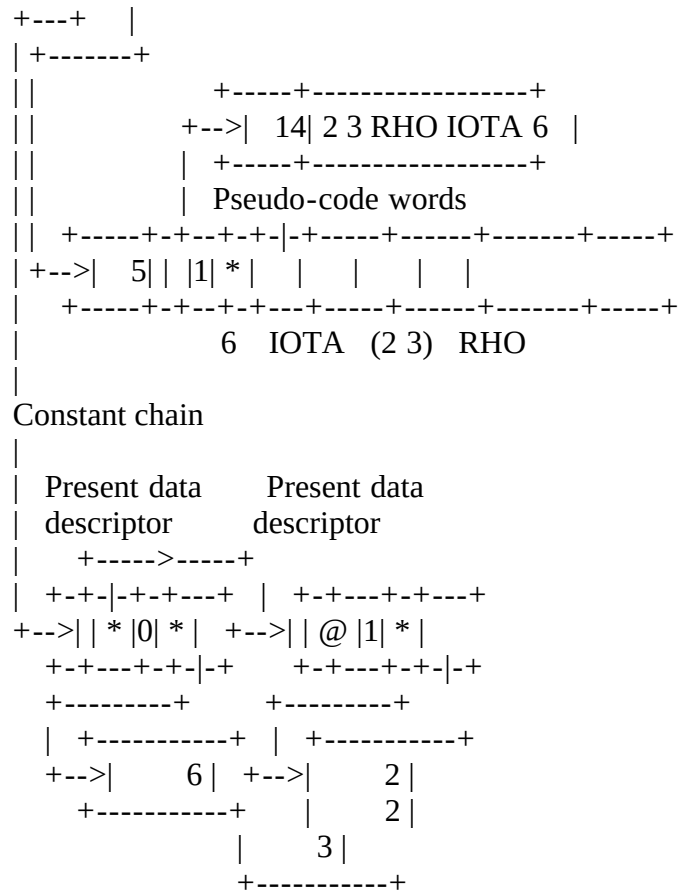


Figure 28: Data structure resulting from statement compilation

The connection between the APL statement compiler and the simulated APL machine is given in the following section.

Section 9: The APL "machine"

The APL "machine" component of APL\B5500 is a software simulation of a fictitious APL processor. The architecture of this "machine" is similar to that of the B5500 in that it is a stack-oriented, descriptor-based processor. The simulated machine, however, executes an order-code which is suitable for APL statement execution. Thus, the simulated machine does not directly use the B5500 hardware stack mechanism. The Machine is capable of interpreting APL statements when expressed in reverse inverted Polish form. In addition, the Machine provides control functions including transfer-of-control within APL functions, and execution interruption facilities necessary for maintaining functional concurrency within the component.

Each APL user in execution mode is provided with an execution stack located in the scratch pad, and addressed through the stack base field of the User State Register. The execution stack, shown in Figure 29 below, has the top-of-stack index as its first element. The remaining elements of the execution stack consist of either descriptors or execution control words (described below).

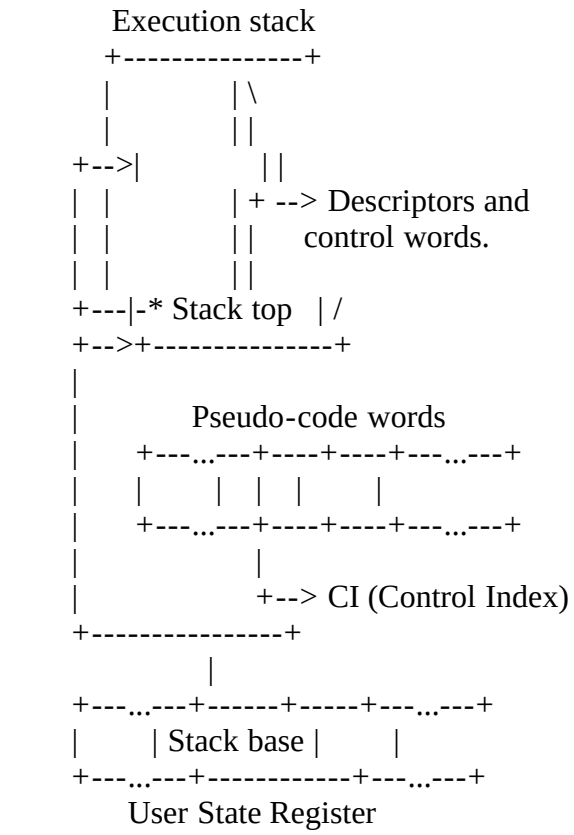
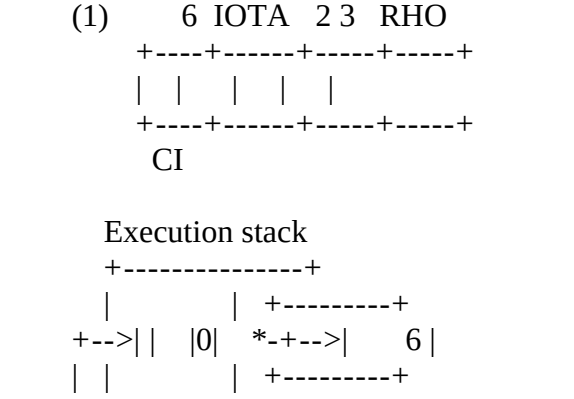
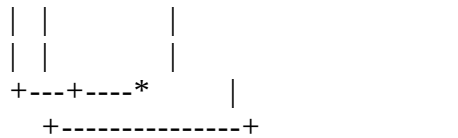


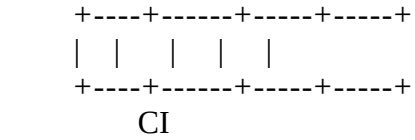
Figure 29: Execution stack and control index

In addition to the execution stack, a control index (CI) is maintained, which points to the current pseudo-code word being processed for the user. As the CI moves through a code string, code words representing operands cause the corresponding descriptor to be loaded onto the execution stack. Code words which represent operators or user-defined functions, however, cause the corresponding operation or function to be applied to the top descriptors. The descriptor resulting from the operation of function call replaces those descriptors involved in the operation or function call. Data descriptors with a reset named bit (temporary data) cause the corresponding data to be removed from the scratch pad when "unstacked". Figure 30 below shows the steps involved in the execution of the simple APL statement of Figure 28 above. This method of APL statement interpretation is, of course, both natural and straightforward.



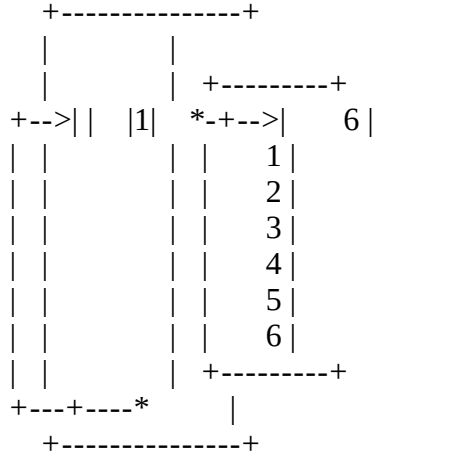


(2) 6 IOTA 2 3 RHO

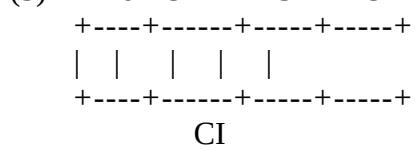


CI

Execution stack

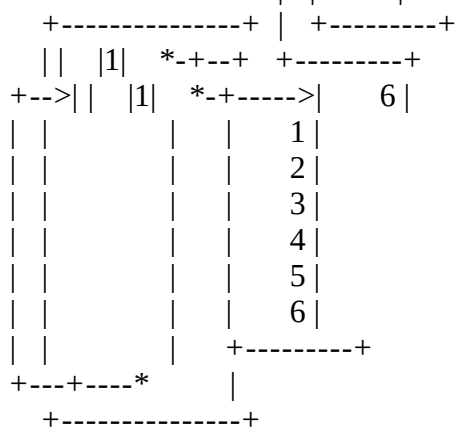


(3) 6 IOTA 2 3 RHO

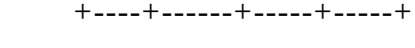


CI

Execution stack



(4) 6 IOTA 2 3 RHO



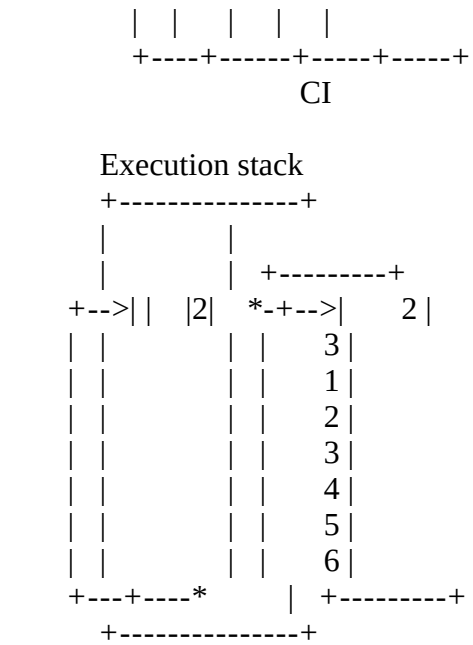


Figure 30: Interpretation of APL code strings

Control words mark various positions in the execution stack. The control words appear in a number of forms, as shown in Figure 31 below.

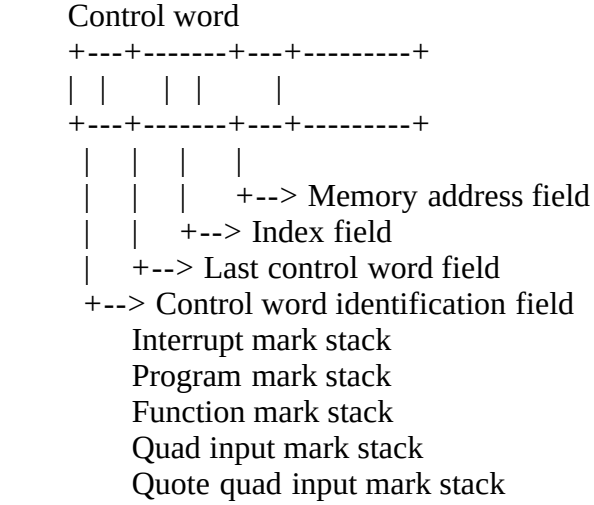


Figure 31: Control word format

The control words have the following functions:

1. Interrupt mark stack control word (IMS). The interrupt mark stack control word is placed at the top of the execution stack at the end of the user's execution period. Information in this control word allows later recovery of additional information for restarting execution.

2. Program mark stack control word (PMS). The program mark stack control word contains information leading to the code string in execution directly above the control word in the stack.

3. Function mark stack control word (FMS). The function mark stack control word is inserted into the execution stack whenever defined functions are invoked during execution.

4. Quad input and quote quad input mark stack control words (QMS and QQMS). The quad input and quote quad input mark stack control words are inserted into the execution stack whenever the user's APL program requests quad or quote quad input from the terminal.

All control words are linked together in the execution stack through the "last control word" field of each control word. The use of the "index" field and the "memory address" field depends on the type of control word.

The execution stack is initialized with a program mark stack upon entry to execution mode from calculator mode. The program mark stack addresses the data descriptor corresponding to the compiled calculator mode statement, as shown in Figure 32a below.

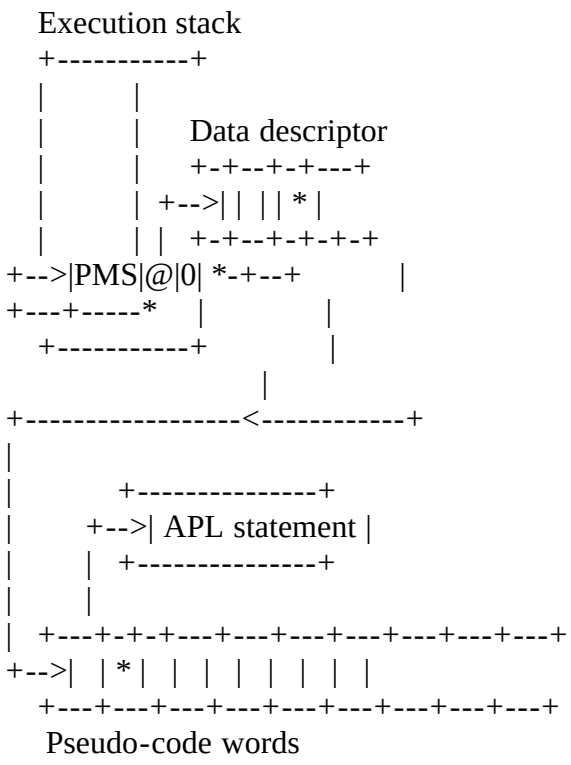
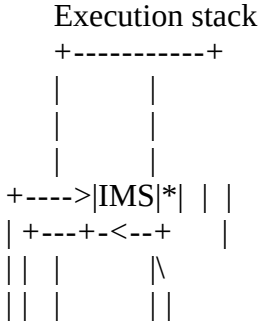


Figure 32a: Initial execution stack contents



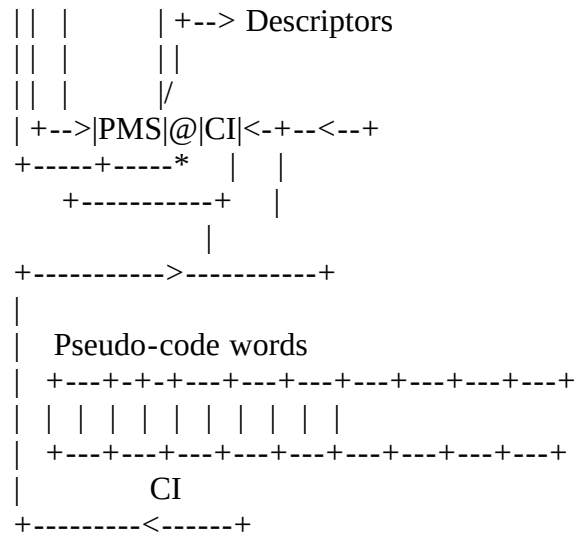


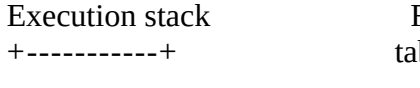
Figure 32b: Execution stack after interruption

An interrupt mark stack control word is inserted at the top of the execution stack whenever time-slice interruption of execution occurs, as shown in Figure 32b above. At the time of the interruption, the control index (CI) is placed into the index field of the program mark stack.

A number of actions take place in the case that a calculator mode statement invokes a function, or a function invokes another function. The arguments to a function are at the top of the execution stack at the time of the call, because of the form of the reverse inverted Polish. The function descriptor is examined in the active symbol table (addressed directly by the pseudo-code word) and, if not present, the function label table is constructed as shown in Figure 21 above. The function mark stack control word is inserted into the execution stack, followed by the descriptors for each argument and local variable. In order to obtain call-by-value parameters, all data described by data descriptors with a set named bit cause a copy operation on the data items before passing the new descriptor to the function.

The descriptors corresponding to local variables, including labels, are kept in the stack area above the function mark stack. Local variables which do not correspond to formal parameters are initially set to "null" by placing a null vector (rank field zero) data descriptor into the stack. Labels are treated as any other local variable, except that the descriptor in the stack is initially a present scalar data descriptor addressing a numeric scalar corresponding to the line on which the label appears.

The simulated APL "machine" also keeps track of the current line being executed by a user when the user is executing a function. The current line is called the line index (LI), and is essentially an index into the corresponding function label table. A program mark stack control word is inserted into the execution stack after the parameters and local variables, in order to start the function, as shown in Figure 33 below.



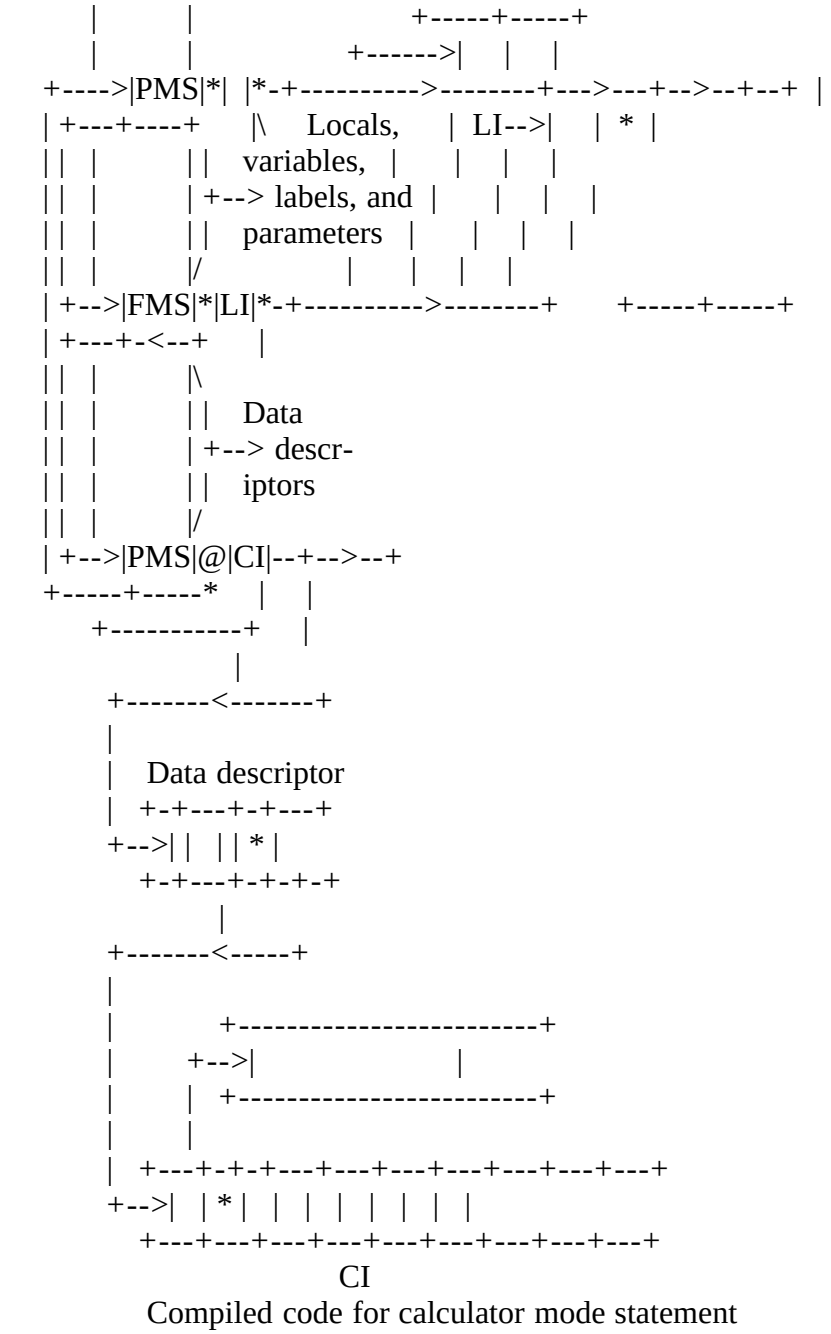


Figure 33: Stack organization for function execution

Two points should be made about function execution. First, because of the index field in each of the control words, functions may be invoked at any point in the execution of a function. The CI is saved in the previous program mark stack, and the LI is saved in the previous function mark stack (if it exists). Upon return from the function execution, the CI and LI can be recovered, and control is returned to the pseudo-code word which follows the function call. Secondly, since the descriptors for parameters, local variables, and labels are maintained in the execution stack, and since the pseudo-code strings are "pure" (i.e., they are not self-modifying), recursive function invocations is permitted.

At the end of function execution, the function mark stack is deleted. If the function returns a value, the descriptor representing the value is

placed at the top of the execution stack. The LI and CI are then recovered from the control words which are "lower" in the execution stack.

Note also that the data descriptors in the function label table are initially marked non-present (refer to Figure 21 above). Any reference to a non-present data descriptor causes the APL "machine" to make the data present (i.e., in the case of data, the "data" sequential storage unit is referenced with the corresponding data brought into the scratch pad). In the case of data descriptors in the function label table, the corresponding APL statement is retrieved from the function text unit, and the APL Statement Compiler is called to compile the line. The resulting data descriptor replaces the previously non-present data descriptor in the function label table. The compiled form of the statement then remains in the scratch pad, until the user returns to calculator mode.

This "demand compilation" avoids unnecessary compilation of statements which are never executed. In addition, functional concurrency is more easily attained, since the compilation is incremental.

When the user returns to calculator mode from execution, the Resource Manager calls upon the APL "machine" to make active data into passive data. The active symbol table is examined for variables which have the altered bit set. Entries are then made into the "names" and "data" storage units for these variables. Thus, the passive data retains its original form until the completion of execution. Passive data is not altered if an error is encountered during the execution of the APL program, unless the STORE monitor command is issued by the user.

If an execution error is encountered, the user is notified, and the execution is suspended. During suspension, the user may examine the active symbol table, the stack locations corresponding to the local variables of the most-recently executing function, and the local variables of any other suspended functions. The user may alter these variables, and continue function execution, or abort the execution. If the function is aborted before a STORE command is issued, then the active symbol table values are destroyed, and the passive symbol table values are retained. Thus, the function can be restarted without reinitialization of global variables.

Additional functions of the simulated APL "machine" include:

1. deallocation of all scratch pad memory cells (returning the storage areas to the B5500 MCP) when no users are in execution mode, and

2. deallocation of areas reserved for a particular user returning to calculator mode from execution mode.

Although the above discussion is a simplification of the functions of the simulated APL "machine", it does provide an outline of the operations and data structures involved. The state diagram given in Figure 34 below shows the logic of the APL "machine".

Figure 34: APL "machine" logic (ROCHE> Too difficult to translate to "ASCII graphics". Maybe a GIF or JPEG file will be needed?)

A detailed discussion of efficient APL "machine" organization and data representation, along with an extensive bibliography concerning APL-related topics, is given by ABRAMS [Ref.10].

Conclusion

The APL\B5500 system is a self-contained time-sharing submonitor for the Burroughs B5500 computer providing full APL\360 processing capabilities. Although the design of APL\B5500 was affected by limited computer resources, such as central memory, the overall design is thought to be sufficiently general to be applicable to other APL implementations.

The APL\B5500 system is presently in a stable condition: no major modifications in design are foreseen. It is necessary, however, to measure the effectiveness of the various APL components in an attempt to make minor modifications and adjustements to tune the system for best performance.

References

1. Iverson, K.E., and Falkoff, A.D., "APL\360: User's Manual", IBM Corp., 1968.

2. Iverson, K.E., "A Programming Language", Wiley, 1962.

3. Kildall, G., Smith, L., Swedine, S., and Zosel, M., "Preliminary APL\B5500 Manual", University of Washington Computer Center, 1970.

4. "B5500 Information Processing Systems Reference Manual", Burroughs Corporation, Detroit, Michigan.

5. "A Narrative Description of the Burroughs B5500 Disk File Master Control Program", Burroughs Corporation, Detroit, Michigan.

6. Stimler, S., "Some criteria for time-sharing system performance", CACM, Vol.12, No.1 (January 1969), pp.41-53.

7. Kildall, G., "Experiments in large scale computer direct access storage manipulation", Technical Report No.69-1-01, Computer Science Group, University of Washington, Seattle, Washington, January 1969.

8. Kuehner, C., and Randell, B., "Demand paging in perspective", AFIPS conference proceedings, 33, Part 2, 1968, pp.1011-1018.

9. Randell, B., "A note on storage fragmentation and program segmentation", CACM, Vol.12, No.7 (July 1969), pp.365-372.

10. Abrams, P., "An APL machine", SLAC Report No.114, Stanford Linear Accelerator Center, Stanford University, Stanford, California, February 1970.

Appendix A: Sample terminal session

```
XXXXXXXXXXXXXXX
 MARY LOGGED IN THURSDAY 10-22-70  09:27
     )VARS<--
 INTERP (F) NEWTON (F) STRING  X       X0
     )FNS<--
 INTERP  NEWTON
     )ERASE STRING<--
     )VARS<--
 INTERP (F) NEWTON (F) X       X0
     2+2<--
4
     2-2<--
0
     -2<--
-2
     #2<--
-2
     2 #2<--
2  -2
     2#2<--
2  -2
     2&3+4<--
14
     (2&3)+4<--
10
     )")"-"4<--
(2&3)-4
     2
     3.4 MAX 4.5<--
4.5
     )DIGITS<--
3
     )DIGITS 9<--
     4 & 3 MAX 5.1<--
20.4
     (4&3) MAX 5.1<--
12
     CIRCLE 1<--
3.141592654
     CIRCLE 1 2<--
3.141592654  6.283185307
     1 CIRCLE 1<--
0.841470985
     IOTA 4<--
1  2  3  4
     CIRCLE IOTA 2<--
3.141592654  6.283185307
```

```
    $G:=M GCD N<--
[1]   G:=M<--
[2]   M:=M RESD N<--
[3]   =:(M NEQ 0)/XIT<--
[4]   [3["]]/"CONT"T<--
[4]   [3[]]<--

[3]   =:(M NEQ 0)/CONT

[4]   N:=G<--
[5]   [4["]]"CONT:"<--
[5]   =:1<--
[6]   [[]]<--

    G:=M GCD N
[1]   G:=M
[2]   M:=M RESD N
[3]   =:(M NEQ 0)/CONT
[4]   CONT:N:=G
[5]   =:1

[6]   [CONT[]]<--

[4]   CONT:N:=G

[6]   [2[]4]<--

[2]   M:=M RESD N
[3]   =:(M NEQ 0)/CONT
[4]   CONT:N:=G

[6]   [CONT-2[]CONT+1]<--

[2]   M:=M RESD N
[3]   =:(M NEQ 0)/CONT
[4]   CONT:N:=G
[5]   =:1

[6]   $<--
    2 GCD 2<--

    )SI<--
    GCD     S
    )SIV<--
    GCD     S  CONT    G      M       N
    CONT<--
4
    G,M,N<--
0  2  2

    =:0<--
    $GCD<--
[6]   [3.1]=:0<--
[3.2] [[]]<--
```

```
    G:=M GCD N
[1]   G:=M
[2]   M:=M RESD N
[3]   =:(M NEQ 0)/CONT
[3.1] =:0
[4]   CONT:N:=G
[5]   =:1

[3.2] $<--
    2 GCD 2<--
2
    36 GCD 64<--
4
    $GCD<--
[6]   [3][3.1]<--
[6]   [[]]<--

    G:=M GCD N
[1]   G:=M
[2]   M:=M RESD N
[3]   =:L& M NEQ 0
[4]   CONT:N:=G
[5]   =:1

[6]   [CONT["]]""N<--
[6]   [4[]]<--

[4]   NT:N:=G

[6]   [4["]]""N:<--
[6]   [3]=:L& M NEQ 0<--
[4]   [4["]]"L:"<--
[4]   [4[]]<--

[4]   L:N:=G

[4]   [[]]$<--

    G:=M GCD N
[1]   G:=M
[2]   M:=M RESD N
[3]   =:L& M NEQ 0
[4]   L:N:=G
[5]   =:1

    36 GCD 64<--
4

    $Z:=FIB N<--
[1]   =:N+2 MIN 4<--
[2]   =:Z:=0<--
[3]   =:1-Z:=L<1<--
[4]   =:Z:=(FIB N-1)+FIB N-2<--
```

```
[5]   [[]]$<--

      Z:=FIB N
[1]   =:N+2 MIN 4
[2]   =:Z:=0
[3]   =:1-Z:=1
[4]   =:Z:=(FIB N-1)+FIB N-2

      FIB 0<--
0
      FIB 1<--
1
      FIB 2<--

      )SI<--
      FIB     S
      )SIV<--
      FIB     S  N       Z
      N<--
2
      Z<--
1
      )ABORT<--
      $FIB[4[""]]""Z$<--
[5]   [4[]]<--

[4]

[5]   [4]=:Z:=(FIB N-1)+FIB N-2<--

[5]   [4[]]<--

[4]   Z:=(FIB N-1)+FIB N-2

[5]   $<--
      FIB 2<--
1
      FIB 4<--
      FIB 5<--
      )SIV<--
NULL.
      $FIB[[]]$<--

      Z:=FIB N
[1]   =:N+2 MIN 4
[2]   =:Z:=0
[3]   =:1-Z:=1
[4]   Z:=(FIB N-1)+FIB N-2

      $FIB[1[""]]:"(N+2)" M<--
[5]   [1[]]<--

[1]   =:(N+2) MIN 4
```

```
[5]  $<--
     FIB 2<--
1
     FIB 4<--
3
     FIB 6<--
8
     FIB 8<--
21
     $INTERP[[]]$<--

     INTERP;X;Y;Z;D;N
[1]  "INTERPOLATION PROBLEM C1"
[2]  =:(0=&/(IOTA N:=RHO X)=X IOTA X:=[])/UNIQERR,0 RHO []:="INPUT X VA
LUES"
[3]  =:(N NEQ RHO Y:=[])/DIMERR,0 RHO []:="INPUT Y VALUES"
[3.5] =:(N GEQ D:=X IOTA Z:=[])/FOUNDZ,0 RHO []:="INPUT VALUE TO INTERP
OLATE"
[4]   =: 0,0 RHO []:="INTERPOLATED VALUE IS";+/(Y&(&/D)%D:=Z-X)%&/(N,N-
1)RHO((N*2)RHO 0,N RHO 1)/,X CIRCLE . -X
[5]  FOUNDZ: =:0,0 RHO []:="INTERPOLATED VALUE IS";Y[D]
[6]  UNIQERR: =:0,0 RHO []:="X VALUES NOT UNIQUE  ERROR"
[7]  DIMERR: "DIMENSIONS DO NOT MATCH   ERROR"

     $INTERP[IOTA]<--
[9]  [FOUNDZ-1[]FOUNDZ+1]$<--

[5]  =: 0,0 RHO []:="INTERPOLATED VALUE IS";+/(Y&(&/D)%D:=Z-X=%&/(N,N-
1)RHO((N*2)RHO 0,N RHO 1)/,X CIRCLE . -X
[6]  FOUNDZ: =:0,0 RHO []:="INTERPOLATED VALUE IS";Y[D]
[7]  UNIQERR: =:0,0 RHO []:="X VALUES NOT UNIQUE  ERROR"

     INTERP<--
INTERPOLATION PROBLEM C1

INPUT X VALUES

[]:
     1 4 6 10<--
INPUT Y VALUES
     3 8 12 40<--
INPUT VALUE TO INTERPOLATE

[]:
     5<--
INTERPOLATED VALUE IS 9.592592593

     $X:=FX NEWTON DFX; ERR<--
LABEL ERROR AT  X:=FX NEWT

     )FNS<--
FIB     GCD     INTERP  NEWTON
     $NEWTON[[]]$<--
```

```
    X:=FX NEWTON DFX;ERR
[1]   X:=X0
[2]   =:((ABS ERR) GEQ @-6)/2, 0 RHO X:=X-ERR:=EPS FX, "%", DFX

    "((X*2)-2)" NEWTON "2&X"<--
1.144123562
    "((X*2-2" NEWTON "2&X"<--
SYNTAX ERROR AT  (X*2-2%2&X

NEWTON
[2]   SYNTAX ERROR AT  EPS FX, "%

    )"-2")"<--
"((X*2-2)" NEWTON "2&X"
    SYNTAX ERROR AT  (X*2-2)%2&

NEWTON
[2]   SYNTAX ERROR AT  EPS FX, "%

    )SIV<--
    NEWTON  S  DFX    ERR     FX     X
    NEWTON  S  DFX    ERR     FX     X
    =:0<--
1
    )ABORT<--
    )SIV<--
NULL.
    )"2")<--
=:0
    SYNTAX ERROR AT  0

    "WALLA WALLA WASH"<--
WALLA WALLA WASH

    (" " NEQ STRING)/STRING:=[]<--
[]:
    WALLA WALLA WASH<--
SYNTAX ERROR AT  WASH
SYNTAX ERROR

    (" " NEQ STRING)/STRING:=    []<--
[]:
    "WALLA WALLA WASH"<--
WALLAWALLAWASH

    STR:=['']<--
A FAT CAT<--
    STR<--
A FAT CAT

    STRING[ 2 10 RHO 6 + IOTA 10]<--
WALLA WASH
WALLA WASH
```

```
    2 10 RHO 6 DROP STRING<--
WALLA WASH
WALLA WASH

    )WIDTH 30<--
    $NEWTON[[]]$<--

    X:=FX NEWTON DFX;ERR
[1]   X:=X0
[2]   =:((ABS ERR) GEQ @-6)/2,
 0 RHO X:=X-ERR:=EPS FX,"%",
DFX

    )DIGITS<--
9
    1%3O<--
SYNTAX ERROR

    1%30<--
0.033333333
    )DIGITS 3<--
    1%30<--
0.033
    )WIDTH 72<--
    13 RNDM 52<--
49  43  27  14  26  21  44  9  16  29  6  30  32

    )OFF
    END OF RUN
```

Appendix B: Syntax

(Nota Bene: " <-- " means: Press the RETURN/ENTER key"...)

```
<apl program> ::= )<login> <-- <statement set> <-- )<logout> <--
<login> ::= <user code>
<user code> ::= <identifier>
<logout> ::= OFF<off option>
<off option> ::= DICARD|<empty>
<statement set> ::= <statement>|<statement set> <-- <statement>
<statement> ::= <monitor command>|<apl statement>|<empty>
<monitor command> ::= )<command>
<command> ::= <library maintenance>|CLEAR|ERASE<identifier list>|FNS|
            VARS|SI|SIV|ABORT|STORE|<buffer edit>|<run parameter>|
            LOGGED|<message>
<library maintenance> ::= LOAD<library name>|<copy>|<clear>|<save>
<library name> ::= <library prefix><library suffix>
<library prefix> ::= <job number>,|<empty>
<job number> ::= {user account number}
<library suffix> ::= <identifier>
<copy> ::= COPY<library name><copy name>
<copy name> ::= <stored program name>|<variable name>
```

<stored program name> ::= <identifier>
<variable name> ::= <identifier>
<clear> ::= CLEAR<library suffix>
<save> ::= SAVE<library suffix><lock option>
<lock option> ::= LOCK|<empty>
<identifier list> ::= <identifier>|<identifier list><space><identifier>
<buffer edit> ::= "<line edit>
<line edit> ::= <search string>"<insert string><quote option>
<search string> ::= <proper string>|<empty>
<insert string> ::= <proper string>|<empty>
<quote option> ::= "<search string>|<empty>
<run parameter> ::= <parameter type><number>|SYN|NOSYN|<parameter type>
<parameter type> ::= ORIGIN|WIDTH|DIGITS|SEED|FUZZ
<message> ::= MSG<station><improper string>
<station> ::= <unsigned integer>
<improper string> ::= <improper string element>|<improper string>
<improper string element>
<improper string element> ::= <visible string character>|"|<space>
<apl statement> ::= <stored program definition>|<basic statement>
<stored program definition> ::= $<definition entry><stored program body>$
<definition entry> ::= <stored program name>|<header>
<header> ::= <stored program options><local variables> <--
<stored program options> ::= <function specifier><parameter options>
<function specifier> ::= <variable name> := |<empty>
<parameter options> ::= <niladic name>|<monadic name><formal
parameter>|<formal parameter><dyadic name>
<formal parameter>
<niladic name> ::= <niladic subroutine name>|<niladic function name>
<dyadic name> ::= <dyadic subroutine name>|<dyadic function name>
<monadic name> ::= <monadic subroutine name>|<monadic function name>
<niladic subroutine name> ::= <identifier>
<niladic function name> ::= <identifier>
<dyadic subroutine name> ::= <identifier>
<dyadic function name> ::= <identifier>
<monadic subroutine name> ::= <identifier>
<monadic function name> ::= <identifier>
<formal parameter> ::= <identifier>
<local variables> ::= <local set>|<empty>
<local set> ::=; <identifier>|<local set>;<identifier>
<stored program body> ::= <stored program statement>|<stored program
body> <-- <stored program statement>
<stored program statement> ::= <edit>|<compound statement>|<empty>
<edit> ::= [<edit command>
<edit command> ::= <display>|<insertion>|<change>|<delete>
<display> ::= <line option>[]<line option>
<line option> ::= <line reference>|<empty>
<line reference> ::= <label expression>|<number>
<label expression> ::= <identifier><relative location>
<relative location> ::= <direction><number>|<empty>
<direction> ::= +|-
<insertion> ::= <line reference>]<compound statement>
<change> ::= <line option>["]<line option>]<line edit>
<delete> ::= <line reference>]<delete option>
<delete option> ::= [<line reference>]|<empty>

<compound statement> ::= <label set><basic statement>
<label set> ::= <label>|<label set><label>|<empty>
<label> ::= <identifier>:
<basic statement> ::= <expression>|<subroutine call>|<transfer statement>
<expression> ::= <operand>|<assignment statement>|<left part><expression>
<operand> ::= <constant>|<identifier><subscript option>|
(<expression<)|[]|['']|<niladic function name>
<subscript option> ::= [<subscript list>]|<empty>
<subscript list> ::= <subscript>|<subscript list>;<subscript>
<subscript> ::= <expression>|<empty>
<assignment statement> ::= <assign operand>:=<expression>
<assign operand> ::= <identifier><subscript option>
<left part> ::= <monadic operator>|<operand><dyadic operator>
<monadic operator> ::= <monadic function name>|<monadic scalar
operator>|<monadic mixed operator>|
<monadic suboperator>
<monadic scalar operator> ::= +|-|&|%|*|LOG|CEIL|FLR|ABS|FACT|RNDM|NOT|CIRCLE
<monadic mixed operator> ::= ,|RHO|IOTA|BASVAL|TRANS|EPS
<monadic suboperator> ::= <monadic suboperator type><dimension part>
<monadic suboperator type> ::= <reduction type operator>|PHI|SORTUP|SORTDN
<reduction type operator> ::= <dyadic scalar operator>/|<dyadic
scalar operator>\
<dimension part> ::= [<expression>]|<empty>
<dyadic operator> ::= <dyadic function name>|<dyadic scalar operator>|
<dyadic mixed operator>|<dot operator>|<dyadic

--------> Missing line in my photocopy ! <----------

<dyadic scalar operator> ::= +|-|&|*|LOG|MAX|MIN|%|RESD|COMB|AND|OR|
NAND|NOR|LSS|LEQ|=|GEQ|GTR|NEQ|CIRCLE
<dyadic mixed operator> ::= ,|EPS|RHO|IOTA|BASVAL|REP|RNDM|TAKE|DROP
<dot operator> ::= <dyadic scalar operator>.<dyadic scalar operator>
<dyadic suboperator> ::= <dyadic suboperator type><dimension part>
<dyadic suboperator type> ::= PHI|/|\
<subroutine call> ::= <operand><dyadic subroutine name><expression>|
<monadic subroutine name><expression>|<niladic
subroutine name>
<transfer statement> ::= =:<expression>

APL SYNTAX -- Constants and Identifiers

<data element> ::= <identifier>|<constant>
<identifier> ::= <letter>|<identifier><letter>|<identifier><digit>
<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<constant> ::= <number>|<string>
<number> ::= <decimal number><exponent part>|<decimal number>|
<exponent part>
<decimal number> ::= <integer><decimal fraction>|<integer>|<decimal fraction>
<integer> ::= <unsigned integer>|+<unsigned integer>|#<unsigned integer>
<unsigned integer> ::= <digit>|<unsigned integer><digit>
<decimal fraction> ::= .<unsigned integer>

<exponent part> ::= <exponent symbol><exponent sign><unsigned integer>
<exponent symbol> ::= @|E
<exponent sign> ::= #|-|+|<empty>
<empty> ::= {the null string of symbols}
<string> ::= "<proper string>"
<proper string> ::= <string element>|<proper string><string element>
<string element> ::= <string character>|""
<string character> ::= <visible string character>|<space>
<visible string character> ::= <letter>|<digit>|<special symbol>
<special symbol> ::= .|(|)|,|&|$|*|+|;|:|#|%|=|@|/|\|[|]|-
<space> ::= <single space>|<space><single space>
<single space> ::= {a single unit of horizontal spacing which is blank}

EOF

GKCD.WS4 (= Gary Kildall CD-ROM article)

- "The compact disk ROM: applications software"
Tim Oren & Gary Kildall
"IEEE Spectrum", Vol.23, No.4, April 1986, pp.49-54

(Retyped by Emmanuel ROCHE.)

It optimizes access time, is compatible with various operating systems, and has the potential for multimedia use

Given the specialty of the compact-disk, read-only memory -- putting massive databases on a user's desk inexpensively -- many engineers will likely find themselves perfecting the technology and expanding its uses for years to come. A spinoff of compact-disk audio technology, the optically read CD-ROM presents various challenges to computer software and systems engineers:

- Overcoming a relatively slow rate of data access.
- Integrating audio, video, and graphics.
- Providing multiuser access to data.
- Accommodating a lack of standards for CD-ROM data structures.
- Updating the CD-ROM database.

In addition, systems designers and managers must consider:

- Hardware requirements for workstations using the CD-ROM.
- Costs of converting data and producing ROMs.
- Rapid changes in a new technology.

Solutions to these concerns combine hardware, software, and systems approaches -- which are themselves still evolving to a remarkable degree.

All of the challenges, and some of the approaches to their solutions, arise naturally from the pathway that information follows on its way from publisher to user via the CD-ROM. First, the data are scanned from printed pages or converted from magnetic media to a form suitable for the CD-ROM. At this data preparation stage, some indexing of information and optimization for CD-ROM storage is usually performed. The transformed data are stored on magnetic tape, which then goes to a premastering facility for error-correction coding and data interleaving and scrambling. The premaster tape is used to create a glass master disk for verification. Mass production follows, with the pressing process embossing the data onto plastic disks for end users. The final step is playback in a CD-ROM drive, typically attached to a desktop computer.

It is here that the hardware concerns -- primarily the performance of the drive unit and the capacity of the medium -- become evident. The prime software goal is to create data structures and accessing strategies optimized for the drive's characteristics. Economic concerns arise at each stage of the process, with one-time costs predominating on the data preparation and mastering side, and unit costs in the production and retrieval steps.

CD-ROM drives: wait before hurrying

CD-ROM drives that provides access up to 600 megabytes of data on each 120-millimeter disk have been available for under $1000 since March (1986). Although this combination of cost and capacity is unprecedented, the drives suffer from slow access speed. The laser pickup head typically requires 0.5 second to access information on the innermost tracks, and 1.5 seconds to reach the outer tracks.

By contrast, hard-disk units can access data in tens of milliseconds. The CD-ROM is slowed by the mass of its pickup head, which contains a focusing system with several lenses and which must be positioned with extreme accuracy. A high-torque stepping motor would move the head more quickly but would increase the cost and mass of the drive beyond practicality. To help compensate for the long access time, most CD-ROM drives have a small tipping mirror that rapidly directs the laser beam to nearby tracks without any movement of the entire lens assembly. Once the head has moved to the desired spot on the disk, the sequential reading speed of the CD-ROM is quite good: 1.2 megabits per second.

The slow access rate is a severe drawback for the most common forms of information stored on CD-ROMs: sequential data files, conventional databases, and textual information bases. In any of these cases, data structures designed for magnetic disks will produce unacceptable results on the CD-ROM.

In a typical magnetic disk file and directory structure, each link from one block of a directory or file to another will cause a movement of the pickup head. The primary purpose of such links is the expansion of existing files and directories when information is added.

But on a CD-ROM there is no possibility of expanding the information once it is stored. Because the number and size of the data files are known when the data are prepared, a much simpler disk structure may be used. Finding a file will require at most two head motions, the first to reach the directory and the second to move to the file. The data blocks are contiguous within each file and within the disk directory, taking advantage of the drive's sequential read rate and tipping mirror.

In a relational database, one of the most common types, stored information is considered to consist of tables. For instance, an auto insurance database might include a table with one column containing the names of policy holders, another column for policy numbers, and additional columns for renewal dates and descriptions of automobiles owned. Each row of the table would consist of the data for a single customer. The columns in another table in the same database would designate the auto maker, model and year, estimated value, accident rate, and theft rate. Each row in this table would refer to a single type of car insured by the firm using the database.

Information in such a database is retrieved through indexes that store the order and location of each row as sorted by a key column. One such index might allow insurance policies to be accessed in order of renewal date. On a magnetic disk, the indexes are constructed as a "tree" of links to information. Each branching point, or node, in the tree contains links to

further nodes, leading eventually to the data itself. Finding a particular record requires traversing several levels of the tree, and following each of the links requires a movement of the pickup head.

Such a relational database will perform very poorly on a CD-ROM unless it is modified. The data structure must be adapted to the characteristics of the drive. In particular, the number of levels in the tree, and thus the number of head seeks, can be reduced by making each node much larger -- increasing the number of its links. This also takes advantage of the CD-ROM's high sequential read speed.

The large data capacity of the CD-ROM can also be used to advantage in the "join" operation that is commonly performed to merge two tables. For instance, there might be a need to selectively combine the two auto insurance tables to yield yet another one, consisting of data on policy number, auto value, and loss rate only. Because a join operation requires a great deal of access to the storage unit, it saves time to compute such operations in the data preparation stage and then store the redundant tables on the CD-ROM.

Searching the whole text on a desktop

Full-text searching, which examines an entire database for the occurrence of a single word or a complex combination of terms, is becoming possible for the personal computer for the first time, thanks to CD-ROM technology. With a complete full-text search package, a user will be able to specify the desired proximity of multiple-search terms; to use Boolean AND, OR and NOT operators in the specification; and to specify a non-unique search term with "wild-card" characters.

The data structures needed for full-text search are generated by the inversion method. The entire database is scanned to find and count the occurrences of each word. At this point, words that occur so frequently as to be meaningless, like "of" and "to", are discarded. In material about data processing, for example, terms such as "computer" might also fall into this category.

The database is re-read once to record the position of every occurrence of each word. The resulting large table is the word index, or full-text inversion, of the database. A magazine article on semiconductor technology, for example, might use the word "micron" four times. The inversion of the article would contain an entry for "micron", along with the four locations where the word occurred.

On a magnetic disk system such an index is usually stored in a multilevel data structure similar to that of a relational database. The strategy for conversion to CD-ROM also follows the database pattern: the levels of the data structure are reduced in number, so the index for each letter of the alphabet can be stored in one undivided area, and read in a single operation. The pointers to text locations would appear in another table nearby.

A search for each paragraph containing the two words "micron" and "leakage" would require four seek operations on the CD-ROM. One motion is required to read the index for the letter "m" and verify the existence of "micron". A

second motion reads the locations of the word. The same two operations are performed for "leakage". The various text locations, obtained from the index, are then compared to see if any paragraph contains both terms.

Of course there is a price to pay for such extensive indexing. A full-text inversion can occupy as much storage space on the CD-ROM as the text itself does. If complete indexing exceeds the storage capacity of the ROM, it may be possible to discard more terms during the first phase of inversion. On the other hand, storing only key words or an index for abstracts of papers would occupy only a few percent of the total capacity.

When these expedients are unacceptable, compressing the data and indexes may be possible. Optimal encoding of text information can double the effective storage, but at the cost of retrieval efficiency. Some compression of the inversion can be achieved by storing the distance between successive locations of a word, rather than storing the actual location of every occurrence. Again, the penalty is a slower search.

Because the CD-ROM pickup movement is slow, it is a poor device for timesharing by several users or tasks. If several simultaneous searches are competing, the pickup head will be moved frequently as each task takes its turn. This "thrashing" destroys carefully optimized sequences of head motion, and performance quickly degrades.

The reproducible nature of the CD-ROM disk offers some solutions on a systems level. For example, identical CD-ROMs may be clustered in multiple drives under an intelligent controller to serve several users simultaneously. The controller can switch read requests to the available drive that has its pickup head closest to the requested location on the disk. Systems designers must carefully assess the load factor per user to determine the number of identical drives needed to prevent thrashing and give all users adequate access times.

If multiple drives are too costly, other users may be locked out when a search is in progress. The penalties in response time and bottlenecking of tasks are obvious, but they may not be excessive when there are only a few users.

Multiple media are coming

Currently CD-ROM is a closed system with no capability for audio or video information, but this may change soon. On February 24, (1986) Sony and Philips announced their intent to create a specification for interactive audio and video applications on CD-ROM -- the "CD Interactive Media" (CD-I). The specification would be a complete format for interactive use of CD-ROM, including speech; natural still and animated pictures; and computer graphics, files, and programs as well as audio and video. The CD-I standard will tentatively include specifications for a low-cost player based on the Motorola 68000 microprocessor, with the first player expected to be introduced sometime next year.

In the interim, multimedia presentations based on CD-ROM will require an additional peripheral player. In such cases, the CD-ROM would hold search-key information for a remotely controlled audio CD or a videodisk player.

If cost prohibits the use of two drives, inexpensive stopgap measures are available. Sound sequences may be digitized and recorded on the CD-ROM. Similarly, printed images may be captured with scanners, or video images with frame grabbers, and the results placed on the disk. However, both the audio and the video techniques may yield inferior reproduction, and they are not portable from one "player" to another. For example, the playback of digitized audio may differ on computers with different hardware clock rates and may not be feasible at all on timeshared systems. Scanned images may not match the resolution, aspect ration, or color capability of a display.

One technique for overcoming graphics contraints on the CD-ROM is to include standard graphics metafiles, such as the one based on the Graphical Kernel System (GKS). A metafile captures the sequence of output operations, such as lines, pie segments, or text, that make up a picture. Given the availability of GKS driver software for a display, a metafile may be played back with no loss of resolution. However, picture libraries in this form are not generally available, so the technique is limited to applications where graphics can be generated from scratch.

Looking for software standards

Like other peripherals, the CD-ROM must contend with a cluttered computer marketplace. While designers of dedicated CD-ROM workstations may to some degree choose their own environment, the developer of general-purpose CD-ROM hardware or software faces a variety of competing processor and bus architectures, operating systems, and user interfaces.

Compounding this difficulty is one of the most controversial issues now facing CD-ROM technology: the lack of standards for formatting data on the disk. While the CD-ROM hardware standard guarantees that a given disk may be read in any drive, the lack of standards for file or index structures on the disk means that a disk prepared by one CD-ROM company cannot be read by the software of another vendor. Thus, the purchaser of several different CD-ROM databases would have to use different software to access each one, with no convenient way to integrate the data extracted.

A standard called Unifile was proposed last year by Digital Equipment Corp. This proposal and others, ranging from descriptions of bands on the disk to detailed file directory and index standards, are under consideration by an informal industry association called the High Sierra Group.

Choosing single standards for directories, file organization, and indexing structures may be premature at this time. Mimicking magnetic-disk data structures on current CD-ROM drives has severe performance penalties. Furthermore, it would be equally shortsighted to adopt these expedients as standards without further experience with current systems, not to mention the improvements that may occur in CD-ROM drives.

Two other factors argue against a standard now. First, information providers such as news organizations, book publishers, and database proprietors want their CD-ROM products to be protected from piracy. They have requested that

the developers of CD-ROM systems scramble the indexes and encrypt the data to protect their investments. Second, CD-ROM technology and the techniques for efficient handling of very large masses of data are too immature to judge what types of information may be found on CD-ROMs within, say, five years, or to predict what accessing structures may be appropriate for this information.

How, then, is bedlam to be averted? The solution may lie in discarding the false analogy of the CD-ROM retrieval system and a standardized record player. Unless the CD-I player comes to dominate the business environment, there will be an excess of architectures for the foreseeable future. Further, the record player analogy ignores the adaptability of a general-purpose computer. The capacity of a CD-ROM is ample for storage of unique versions of the retrieval software for dozens of different information delivery systems. A CD-ROM standard need include only a disk header that allows the delivery system to identify and retrieve software, making a standardized data architecture less urgent.

A simple CD-ROM disk design should include program storage in the format of each intended delivery system. The programs would include retrieval software, video-display drivers when necessary, and any other desirable software. A single band on the disk would be allocated to each combination of processor and operating system. For instances, one band might be allocated to Intel microprocessors operating under the MS-DOS standard, a second band to VMS running on VAX machines, and a third to Motorola 68000-based Unix systems.

The database itself would reside in one or more bands occupying most of the disk and would include the accessing structures appropriate to the application. This approach is flexible enough to reserve disk areas for future audio and video storage.

Direct updates of a CD-ROM database are, of course, impossible, so the entire disk must be replaced if changes are made in the data. Given current lead times for mastering and production, a quarterly update cycle is the most frequent that can be reliably maintained. If more frequent changes are necessary, updates must be distributed onto magnetic media or transferred from online sources to magnetic disk by the user.

However, since even a 1 percent change in a full CD-ROM represents over 5 megabytes of material, a few such changes can easily fill in a hard disk. Another consideration is the method of integrating the new and the old data, which must be done smoothly to avoid undue annoyance to the user -- say an engineer searching an updated database for articles on a new type of MOSFET not found in the original CD-ROM database. The search software must direct the query to the new material on the magnetic disk without explicit instructions from the user.

Toward this end, the software that controls access to the data must be able to recognize queries to updated portions of the database, and to redirect them transparently to the magnetic disk files. The various indexes on the CD-ROM would also have to be checked against the updated information, disabling entries for obsolete data and finding those for new data. New indexing information could be distributed on the magnetic disk as part of the update, or it could be created by the user's computer, based on a version of the indexing software used in the original data preparation.

The mastering and production costs and the production schedules for CD-ROMs depend heavily on the availability of manufacturing facilities, which are essentially the same as those for producing compact audio disks. Most production facilities are dedicated to the audio CD market, which leads CD-ROMs by orders of magnitude.

Since March (1986) the production phases from premastering to the first mass-produced disks have typically required four to six weeks. This rate would permit quarterly updates of CD-ROMs for legal, medical, and financial data, as well as other types of databases in which timely information is crucial. New production capacity is coming on-stream rapidly, however, and a monthly update cycle could be practical by the end of this year.

Costs for premastering and mastering are now around $3000 for each master disk, and $4 to $5 per CD in lots of 10,000. These rates will fall as the production capacity grows.

Justified for multimegabyte storage

Not all databases belong on CD-ROMs, of course. If the information does not exceed 10 megabytes, there is little reason to put it on a CD-ROM, unless it is merely part of a broader, general program. At present, a standard CD-ROM can store about 550 megabytes, which may include anywhere from 300 to 450 megabytes of raw information, depending on the density of indexing.

Databases that exceed this limit may be accompanied with multiple drives, or "jukebox" changers -- but at a greater cost, naturally. It may be possible to break up some large databases into chunks, each of them residing on a single disk. The chunks ought to be sufficiently autonomous, however, to allow a search to continue without disruption through a change of one or more disks.

The markets with the greatest immediate potential for CD-ROMs include lawyers, doctors, and engineers and other manufacturing professionals who regularly retrieve specifications. Many of these markets are already served by dedicated workstations and online data utilities. For workstations, CD-ROM offers an immediate cost savings over magnetic disk. A workstation that is based on CD-ROMs may help a user recover its cost in as little as a few months by substantially curtailing the bill for online data services.

A vendor who wants to deliver CD-ROM technology for the office or home personal computer, however, faces a problem. With no currently installed base of CD-ROM drives, publishers are reluctant to convert information from print to the CD-ROM format. And without many disk titles available, the consumer is hesitant to buy a CD-ROM drive. This impasse may be broken in the next year as the price of drives continues to fall, and two or three CD-ROM titles of general interest -- like encyclopedias, dictionaries, movies, sports features, and reference sets of classical and religious literature -- appear on the market. As the first set of consumers enter the market place, publishers, in turn, will be encouraged to jump into the business.

The CD-ROM is likely to have the greatest impact when it penetrates schools

and municipal libraries. To save space and money, such institutions might choose the CD-ROM format for storing references that are not frequently accessed.

Refinements are imminent

Current technical developments indicate that solutions are imminent for the most severe applications problems of the CD-ROM. Lighter pickups and erasable optical media are now under development and could reach the mass market within two years.

Even when these major problems are overcome, however, that will still leave the far-reaching question of how to deal effectively with massive amounts of information. Users such as educators and businessmen unaccustomed to complex information retrieval systems may be overwhelmed by a deluge of unstructured data, two orders of magnitude greater than that seen up to now.

A CD-ROM system that can deliver the equivalent of 400 volumes of text must include powerful but transparent accessing methods if the information is to be useful. Current retrieval systems, limited to full-text search, must give way to workstations for research and writing with optical databases at the core.

CD-ROM databases will continue to offer document title indexing and full-text search, but in the future they will also link related documents in what is becoming known as a "hypertext" system.

Such a system provides direct links to cited articles at the point of references and, through a full indexing system, it can show all the citations of a document under study. With this capability, the researcher can move quickly through networks of related information, saving the browse path as a personalized index.

Database word indexes are likely to be augmented with a thesaurus of synonymous terms, so the search system can suggest extensions and refinements to the searches made on it. The use of hypertext links as a data flow graph will enable searches to be restricted to "nearby" information -- information about closely related or similar topic.

When indexing information becomes available to the user during a search, the retrieval software can be reconfigured to the research task at hand. An offshoot will be such secondary products as encyclopedia study guides, case law studies, and medical tutorials, to allow the reader to examine the underlying source documents immediately.

Text and indexing information may be extracted from the database or entered by the user to form new documents that will be added transparently to the existing data bases. A single document might be viewed as contiguous text, as a skeletal outline, or as a collection of notecards. Document editing tools are likely to include notecard and outline processors for organizing ideas, and text and image editors to add character and picture information. Database references in a newly formed document will remain linked to the underlying information, and may also be linked to other documents.

Progress in CD-ROM technology could make such workstations a reality soon. With these tools, existing records can then be converted into structures of related knowledge. When mixed-mode drives become available, video and audio information can be incorporated directly into these archives, and animations and simulations will be added to encyclopedias as personal computers grow ever more powerful.

EOF

GKCHK.WS4

- "Microcomputer software design -- A checkpoint"
  Gary KILDALL
  Proceedings of the Fall Joint Computer Conference, 1975

(Retyped by Emmanuel ROCHE.)

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 [Ref. 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 to say, 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 time-sharing 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.

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

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 special 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.

System 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 system languages, are traditionally used to implement the lowest level system functions, to avoid the rigidities of assembly language coding. In the larger computer environment, system 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 [Ref. 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 [Refs. 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 [Refs. 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:

1) compute partial pressures of nitrogen for several controlling tissues,
2) monitor external parameters such as elapsed time and current dive depth,
3) drive simple displays with the current and safe ascent depths, and
4) 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 the 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.

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

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 7x9 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 illustrate the range of applicability of PL/M, however, additional projects from more traditional computer areas are considered.

Figure 3. A disk-based microcomputer development system

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 whch 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 to say, 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 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 [Ref. 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 to say, 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. "Microcomputer Software Makes its Debut"
   Falk, H.
   IEEE Spectrum, Vol.10, No.11, October 1974

2. "A Guide to PL/M Programming"
   Intel Corporation

3. "Software Engineering Techniques"
   Buxton, J.
   NATO Science Committee, April 1970

4. "Structured Programming"
   Dahl, et al.
   Academic Press, 1972

5. "The Elements of Programming Style"
   Kernighan, B., et al.
   McGraw-Hill, 1974

6. "Advanced Programming Techniques Volume 1: Program Structure and Design
   Yourdon
   Yourdon Inc., 1974

7. "Processors and Profits: How Microprocessors Boost Them"
   Davidow, W.
   "Electronics", July 11, 1974

8. "Managing a Programming Project"
   Metzger
   Prentice Hall, 1973

9. "Systems Languages: Management's Key to Controlled Software Evolution"
   Kildall, G.
   Proceedings of the 1974 Western Electronics Show and Convention

(WESCon), September 1974

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

```
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 (0A Hexadecimal) */
00019  1
00020  1      /* Teletype output subroutines */
00021  1
00022  1   PrintChar: PROCEDURE (Char);
00023  1      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 <> '$';
00053  2        CALL PrintChar (Message);
00054  3        A = A + 1;
00055  3      END;
00056  2
00057  2    END Print;
00058  1
00059  1    /* End of Teletype output subroutines */
00060  1
00061  1    /* FRACS holds the number of 1/60ths of a second which
00062  1    have elapsed in the last partial second, while
00063  1    SECS, MINS, and HRS hold the elapsed time counts */
00064  1
00065  1    DECLARE (Fracs, Secs, Mins, Hrs) BYTE;
00066  1
00067  1  TimeKeeper: PROCEDURE INTERRUPT 2;
00068  2    /* The TimeKeeper procedure is called through an external
00069  2    interrupt (RST 2) every 1/60th of a second. The procedure
00070  2    updates the values of HRS, MINS, and SECS so that the total
00071  2    elapsed time since system start-up is maintained in
00072  2    BCD-pair form */
00073  2
00074  2    IF (Fracs := Fracs + 1) >= 60H THEN  /* One full second */
00075  2       DO;
00076  2       Fracs = 0;
00077  3
00078  3       IF (Secs := DEC (Secs + 1)) = 60H THEN  /* One minute */
00079  3          DO;
00080  3          Secs = 0;
00081  4
00082  4          IF (Mins := DEC (Mins + 1)) = 60H THEN  /* Hour */
00083  4             DO;
00084  4             Mins = 0;
00085  5             IF (Hrs := DEC (Hrs + 1)) = 24H THEN
00086  5                /* One day elapsed */ Hrs = 0;
00087  5             END;
00088  4          END;
00089  3       END;
00090  2    END TimeKeeper;
00091  1
00092  1  /* Set counters to zero */
00093  1  Fracs, Secs, Mins, Hrs = 0;
00094  1
00095  1  /* Start counting time */
00096  1  ENABLE;
00097  1
00098  1  /* Write initial message */
00099  1  CALL CRLF; CALL CRLF;
00100  1  CALL Print (.'** Elapsed Time Counter **$');
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 = 0 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
```

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 Hours 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 "interupt 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 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 Hours 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 Hours 01 Mins
    00 Hours 02 Mins
    00 Hours 03 Mins

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

EOF

GKED.WS4

-"Running 8-bit software on dual-processor computers"
Gary A. Kildall
"Electronic Design", September 16, 1982, p.157

(Retyped by Emmanuel ROCHE.)

Abstract: New desktop computers with both 8-bit and 16-bit microprocessors can take advantage of the many application programs written for 8-bit processors.

New generation 16-bit microprocessor chips are appearing in increasing numbers in desktop computers. However, new computers traditionally suffer from a lack of application software when first introduced, and the new 16-bit computers are no different. Recognizing that there are thousands of application programs for today's 8-bit computers, many desktop computer manufacturers are producing dual-processor computers that run both 8-bit and 16-bit software without intervention from the operator.

For the CP/M software user, this means that all present 8-bit application software can operate side by side with the new 16-bit software. From a programmer's standpoint, 8-bit processors have advantages in simplicity and program density that make them preferable in many applications. The dual-processor computers let a programmer select either processor, depending upon the intended use.

In contrast to CP/M-86, most other 16-bit operating systems do not use the same file format or data structures as their comparable 8-bit systems. They, therefore, require two resident operating systems that maintain two environments with different command lines, displays, and diskette formats. As a result, the 8-bit and 16-bit environments in a dual-processor system must be treated differently by an operator. In addition, data files must be "imported" or "exported" from one operating system to another, using utility programs. Further, the import and export operations must use distinct diskettes or hard-disk data areas, to keep the operating systems from interfering with one another. That prevents effective disk-space allocation, and can be confusing for even the most experienced computer operators.

The 8080, Z-80, and 8085 are the mainstays of the 8-bit micros, but only the 8080 instructions are common to all three chips. As a result, nearly all 8-bit application software that operates with CP/M uses the 8080 instruction set. This instruction set has proved sufficient for many advanced commercial, scientific, and educational uses.

8080 is the base

The 8080 instruction set is largely contained with the 8086, but there are incompatibilities that require 8080 programs to be recoded, to properly use

the 8086 instruction set. The 8086 has one major advantage over the 8-bit microprocessor chips: it lets application programs access sixteen times more directly addressable memory. This means that new or evolving application programs for 16-bit processors can use this additional memory to implement features that could not be included in their 8-bit counterparts.

CP/M-based software depends upon three components, when moving to the 16-bit world: file formats, data formats, and machine instructions. File formats define the layout of data on a backup storage device, such as a floppy or hard disk. Data formats define the layout of data in main memory -- the arrangement of bits, bytes, words, and characters in a specific data structure. Machine instructions define the manner in which data files and data structures are processed. Under CP/M, only the machine instructions need change when software moves to the 8086.

File format compatibility means that a diskette can be used for both CP/M and CP/M-86 without change. Thus, a word processing program with CP/M, for example, allows text files to be read using CP/M-86. Without such compatibility, a programmer would have to choose either the 8-bit or 16-bit processor and file format, but not both, as the target environment. File format compatibility means that all removable and fixed storage devices are shared between both processors.

While data format compatibility may not be critical to a user, it is very important to a software vendor. Application software contains data area definitions that follow the conventions of a particular operating system. These conventions are used when the software package calls upon the operating system for services, such as reading data from a file or keyboard. Data formats are identical for CP/M and CP/M-86, so that a software vendor can easily convert from one operating system to another. Converted software contains fewer errors, because the changes required are minimized. This helps bring converted application packages to market quickly. Application writers also benefit, because the well-understood 8-bit interface is used in their new 16-bit programs. Data format compatibility means that CP/M-86 completely replaces CP/M, so that only one operating system is necessary to support both processors.

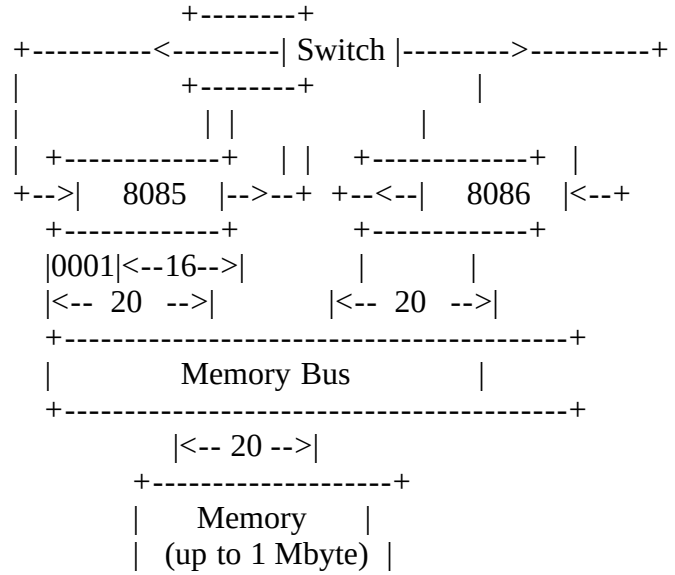


Figure 1. This dual-processor computer allows the 8-bit 8085 and 16-bit 8088 to share memory. The switch controls the processors through their hold and output lines.

Dual-processor hardware

A dual-processor version of CP/M does not require complex hardware. For example, an 8-bit 8085 can share memory with a 16-bit 8088 processor (Figure 1 above). In this example, the memory ranges from 128 Kilobytes to 1 Megabyte. The 8085 has only 16 address lines that access 64 Kilobytes of memory. The 8088, on the other hand, adds a "segment number" that produces a 20-bit address to access all of memory. Each 8085 memory reference has a high-order 1 bit that translates the reference to a location with the second 64 Kilobytes of memory. The first 8085 address, 0000, becomes 8088 hexadecimal 10000, and the last 8085 address, FFFF, is translated to 8088 memory location 1FFFF.

Both processors access memory through a common bus containing the address lines, data lines, and read and write signals. Only one processor can access memory at any instant, so a simple electronic switch must control each processor. Initially, this switch activates the Hold signal for the 8085, and then releases the 8088, so that it can read or change data throughout the entire memory area.

A program running on the 8088 releases control of memory to the 8085 by executing an output instruction to the output port connected to the electronic switch. The switch suspends the 8088 by enabling its Hold signal, and then initiates the 8085 by releasing the signal. The 8085 returns control to the 8088 in exactly the same way. An area in the second 64 Kilobytes of memory, common to both the 8085 and 8088, is used to exchange data values between processors.

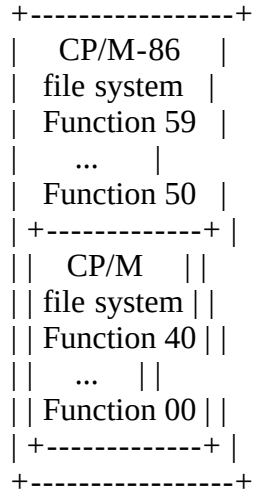


Figure 2. All CP/M functions are contained within CP/M-86, which contains ten function calls for 16-bit operation.

CP/M's upward compatibility

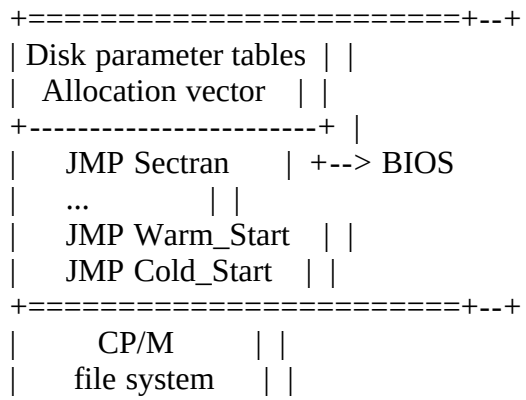
Upward compatibility from CP/M to CP/M-86 simplifies dual-processor operation. Application programs that operate under CP/M use a common set of system function calls, numbered from 0 through 40. These function calls let programs communicate with the operator console, and transfer data to and from disk files. CP/M-86 contains all CP/M function calls, and adds new calls, numbered 50 through 59, for 8088 memory management (Figure 2 above). Because CP/M-86 includes all CP/M function calls, only CP/M-86 need be included in the dual-processor computer. CP/M-86 intercepts all 8-bit CP/M calls, and runs them on the 8088.

Two significant side effects occur when using CP/M-86 to process all 8-bit CP/M calls. First, all operating system functions are run on the faster 8088 microprocessor. Existing 8-bit programs benefit from this speed increase when they perform CP/M calls. A second, more important, side effect is that the available transient program area (TPA) is substantially increased. Only a simple software interface need be stored in the 8085 memory area, leaving considerable memory space for the TPA. A 62 Kilobyte TPA thus is feasible in most implementations.

In this dual-processor system, CP/M-86 controls the operation of the computer by performing the usual cold-start sequence when the computer is first powered up. CP/M-86 reads console commands, and loads and runs 8088 programs in the 16-bit environment.

If the operator types an 8-bit program name, CP/M-86 must first run a 16-bit program, RUN85, to initialize the 8-bit memory area. RUN85 reserves the 64 Kilobyte memory region at 10000 for 8-bit program execution, and then initializes data structures and program areas within this region that duplicate 8-bit CP/M. RUN85 loads the 8-bit program into the 64 Kilobyte region, transfers control to the 8085, and then waits to intercept any CP/M system calls. Each time the 8-bit program performs a system call, RUN85 regains control, and performs the required function with a direct call to the CP/M-86 system.

When RUN85 transfers to the 8085, the 8-bit memory region contains data areas that correspond to 8-bit CP/M, along with machine code that transfers control back to the 8088 when system calls are made.



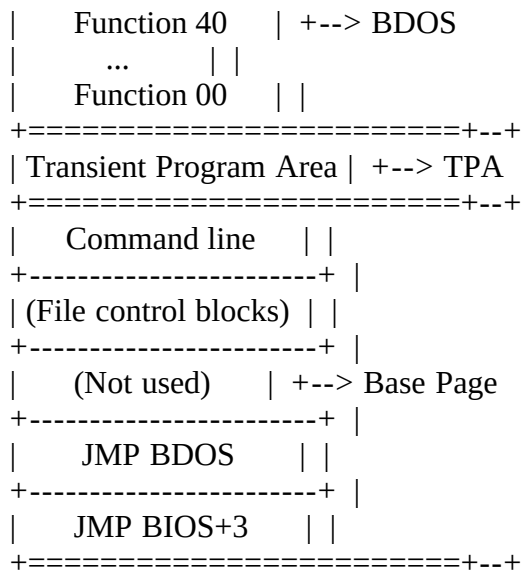


Figure 3. The 8-bit CP/M memory is basically organized into four blocks: Basic I/O System (BIOS), Basic Disk Operating System (BDOS), Transient Program Area (TPA), and the Base Page. The CP/M-86 program called "RUN85" initializes the BDOS, BIOS, and Base Page areas before an 8-bit program begins to run.

RUN85 initializes three memory areas before the 8-bit program is started (Figure 3 above): the Base Page, the Basic Disk Operating System (BDOS), and the Basic I/O System (BIOS). Most application programs use only the Base Page and BDOS, while a few also include direct BIOS calls. To be complete, the data and function of all three areas must be duplicated. The TPA holds the 8-bit program loaded by CP/M-86, and requires no special initialization.

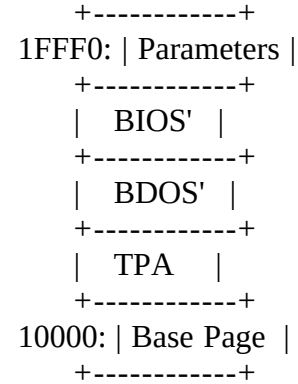


Figure 4. RUN85 creates an 8-bit memory image within the CP/M-86 memory area. The image contains the Parameters (16 bytes to hold data values shared by the 8085 and 8088); BDOS' and BIOS' interfaces with the BDOS and BIOS running on the 8088, as well as the TPA and RUN85 Base Page.

Base Page initialization is done by RUN85. RUN85 first copies its own base page, prepared by CP/M-86, to 8085 memory at 10000 (Figure 4 above). RUN85 inserts two jump instructions in low memory, that lead to the CP/M-86

interface code shown in Fig. 4 as BDOS' and BIOS'.

The BDOS' program module

The BDOS' module interfaces the 8085 program with CP/M-86 running on the 8088, while the BIOS' module intercepts all direct BIOS calls from the 8085, and passes the calls to the CP/M-86 BIOS. The last 16 bytes of memory, labeled "Parameters" in Figure 4 above, are reserved to pass data between the 8085 and 8088.

The BDOS' module is a simple interface, with the real BDOS running on the 8088. Each 8-bit BDOS call enters through the JMP BDOS call in the Base Page. Upon entry, the 8085 registers contain the following values:

Register C: function code, 0 to 40
Register DE: entry value, 0 to 65,535

Upon exit from the BDOS' module, 8-bit CP/M programs expect the following returned values:

Register A: byte return value from CP/M
Register HL: word return value from CP/M
Register B: same as H

Normally, programs use register A for a byte value, and register pair HL when a word value is brought back from CP/M. Register B contains the high-order byte of a word result, to be compatible with early versions of CP/M. The Parameters region contains reserved areas, to pass these parameters between processors (Figure 5 below).

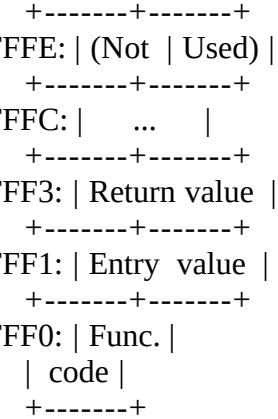


Figure 5. The Parameters region of the CP/M-86 memory area of Figure 4 above holds the CP/M function code (0 to 40), as well as the entry and return values for the 8085 CP/M program.

The BDOS' module stores the function code and entry value into the Parameters, and switches to the 8088 to perform the CP/M function. Upon return from the 8088, the BDOS' program recalls the return value, and goes back to the calling

program. Figure 6 below gives detailed machine code for the BDOS' program (absolute addresses are shown for simplicity in each example).

```
; BDOS' Program Module
;
MOV     A,C           ; Get Function Code
STA     0FFF0H        ; Save in "Func"
XCHG                  ; Entry Value to HL
SHLD    0FFF1H        ; Save in "Entry Value"
OUT     Switch        ; Switch to 8088, Wait
;
; 8085 Now in "Hold" State, Stopped Until
; CP/M-86 Returns from 8088 Processor.
;
LHLD    0FFF3H        ; Return Value to HL
MOV     A,L           ; Byte Return Value
MOV     B,H           ; Copy of H to B
RET                   ; Return to Caller
```

Figure 6. The BDOS' program module sets up the 8085 registers, and switches operations to the 8088 until the function is performed, when it returns to the 8085.

Recall that RUN85 suspends execution when control transfers to the 8085. The OUT instruction in the BDOS' interface module again reverses roles, by transferring back to the 8088, to let RUN85 continue its operation. RUN85 performs a CP/M-86 function call, using the function code (Func) and entry value deposited by BDOS' in the Parameters region of common memory. RUN85 sends these values to CP/M-86, using the following registers:

Register CL: function code, 0 to 59
Register DX: entry value, 0 to 65,535

Return values from CP/M-86 appear in the following registers:

Register AL: byte return value from CP/M-86
Register AX: word return value from CP/M-86
Register BX: (Same as AX)

RUN85 takes the 16-bit return value in AX, and stores it into the return value portion of common memory. Figure 7 below shows the RUN85 program segment that intercepts and processes CP/M calls.

```
Goto85: OUT     Switch        ; Transfer to 8085
        ;
        ; 8088 Now in "Hold" State. Wait for 8085
        ; to Execute "OUT Switch" For Next CP/M Call.
        ; (Assume DS = 10000H on Entry.)
        ;
        MOV     CL,0FFF0H     ; Function Number 0 to 59
        MOV     BX,0FFF1H     ; Entry Value
```

```
INT     224          ; Calls CP/M-86
MOV     0FFF3H,AX    ; Save Return Value
JMP     Goto85       ; Go Back for Another Call
```

Figure 7. RUN85 intercepts and processes calls with an intercept module running on the 8088. When the 8085 makes a CP/M call, RUN85 retrieves the function code and entry value, and calls CP/M-86 to perform the function.

This program segment follows the 8085 memory initialization described above. The OUT instruction relinquishes control to the 8085, to begin running the 8-bit application program. The BDOS' module returns to this program segment, following the OUT instruction, when a CP/M call is made from the 8085. As shown, RUN85 retrieves the function code and entry value from the Parameters memory segment, and calls CP/M-86 to perform the function. When CP/M-86 completes the operation, RUN85 stores the returned value back to the Parameters segment of common memory. RUN85 then loops to the OUT instruction, to send control back to the 8085, so that it can continue to run the 8-bit program.

The 8085 and 8088 continually pass control back and forth, until the program ends by calling CP/M function 0 (System Reset). This function clears RUN85, and returns memory allocated for 8085 processing.

The BIOS' program module

The BIOS' module captures all direct BIOS calls from the 8085. Entry to the BIOS' is through a sequence of seventeen 8085 jump (JMP) instructions located at the beginning of the BIOS' module. When a BIOS call takes place from an 8085 application program, the BIOS' module translates it into a direct call to the 8088 BIOS that is a part of CP/M-86. In general, a program enters the BIOS with active data in register pairs BC and DE. Single byte values return in register A, and word values are brought back in register pair HL.

The BIOS' module works with RUN85 to transfer the data in 8085 registers BC and DE to 8088 registers CX and DX, respectively. The BIOS' program stores BC and DE into the Parameters area of common memory, and relinquishes control to RUN85. RUN85 retrieves the parameters in the same manner as a BDOS' call, transfers to the 8088 BIOS, and stores the result parameters on return.

A direct BIOS call is simple under CP/M-86, because a specific function is included for that purpose. Function call 50 transfers program control through the 8088 BDOS directly to the 8088 BIOS. The DX register pair is assumed to address a five byte area of the form shown in Figure 8a below.

(a)

Function code 0-16	Content of BC	Content of DE

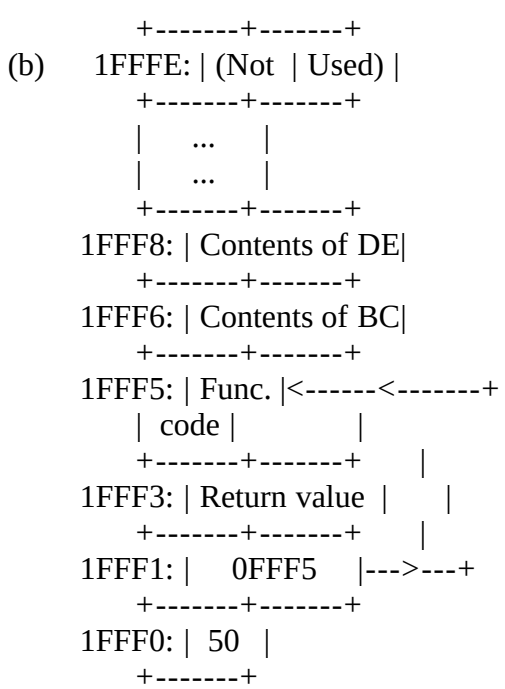


Figure 8. A direct BIOS call uses function 50 under CP/M-86 to transfer program control to the 8088 BIOS. The DX register pair addresses a five byte area, (a) comprising the function code and the contents of registers BC and DE. The Parameters region of the memory is augmented by two additional fields, to hold the register contents (b).

To accommodate this additional information, the Parameters region is augmented by two additional fields, labeled "Content of BC" and "Content of DE", as shown in Figure 8b above. In Figure 8b, "50" invokes the direct BIOS call from RUN85, while "Func" corresponds to one of the seventeen jump elements of the BIOS jump vector. Figures 9a and 9b below show the BIOS' module in outline form.

```
(a)   ; BIOS' Jump Vector
      ;
            JMP     f0
            JMP     f1
            ...
            JMP     f16

      ; Ready "Func" for Go88
      ;
      f0:    MVI     A,0
            JMP     Go88
      ;
      f1:    MVI     A,1
            JMP     Go88
      ;
            ...
      ;
      f16:   MVI     A,16
```

```
        JMP     Go88
```

(b)
```
    ; BDOS' Interface to RUN85
    ;
    Go88:   LXI     H,0FFF5H        ; Addr Func
            SHLD    0FFF1H          ;
            MOV     M,A             ; A --> Func
            INX     H               ; to FFF6
            MOV     M,C             ; Low of BC
            INX     H               ; to FFF7
            MOV     M,B             ; High of BC
            INX     H               ; to FFF8
            MOV     M,E             ; Low of DE
            INX     H               ; to FFF9
            MOV     M,D             ; High of DE
            MVI     A,50            ; Direct Call
            STA     0FFF0H          ; to 8088 BIOS
            OUT     Switch          ; go to 8088
            LHLD    0FFF3H          ; Word Return
            MOV     A,L             ; Byte Return
            RET                     ; to Caller
```

Figure 9. Each BIOS' jump is intercepted by the BIOS' jump module (a), which transfers operations to a common sequence (b), where RUN85 takes over.

The program segment in Figure 9a intercepts each BIOS jump, and transfers to common code in Figure 9b, where the 8085 relinquishes control to RUN85. The program in Figure 9b stores the BC and DE registers in common memory, and sets up CP/M-86 function call 50. The OUT instruction sends control to the 8088, where RUN85 is in a Hold state, waiting for a CP/M-86 call. Referring back to Figure 7, RUN85 gets control following its OUT Switch instruction, performs the direct BIOS call, and returns to the 8085. The 8085 executes the last instruction in Figure 9b, where control returns to the 8085 application program.

Three deficiencies of this model must be corrected in a working BIOS'. The first involves the 8085 startup sequence; the others involve the Disk Parameter Table and Allocation Vector data structures located within the BIOS.

On initial startup, the 8085 begins execution at location zero. This leads directly to the BIOS' warm start entry, which immediately terminates execution of RUN85 and the 8085 program. To prevent this, the first call to the warm start entry must instead transfer to the beginning of the TPA. The application program runs to completion before control again transfers to the BIOS' warm start. This time, control passes through to RUN85, to stop the 8-bit program.

Two additional BIOS' entry points return the addresses of two data structures: the disk parameter table and the allocation vector, shown in Figure 3 above. These data structures reside, however, in the BIOS that is a part of CP/M-86, and cannot be addressed by the 8085. Whenever these addresses are requested by the 8-bit application program, RUN85 must copy the values from the 8088 memory

space into a reserved area within the BIOS' module. These two data structures are rarely referred to by application programs, so that the additional overhead has little effect on overall execution time.

Appendices

The dual-processor approach

The dual-processor desktop computers that include both an 8-bit and a 16-bit microprocessor chip in the same cabinet offer the best of both worlds to their users. The low-cost microprocessors share expensive memory, power supplies, and peripheral equipment. All 8-bit application software is immediately accessible. Application writers choose either the 8-bit or 16-bit microprocessor and, in fact, can switch from 8- to 16-bit processors with no change in operator interaction. The target processor is simply of no consequence.

Several dual-processor microcomputers have been introduced recently. In April of this year (1982), Digital Equipment Corp. presented its "Rainbow" desktop computer, which includes Z-80 and 8088 microprocessor chips, along with CP/M-86.

Rainbow's operation demonstrates effective use of both processors. The operator begins by typing the name of a program following the usual CP/M prompt. If the operator types the name of an 8-bit program, the Rainbow operating system loads the machine code into memory, and starts the Z-80 microprocessor. If the user names a 16-bit program, the Rainbow loads the machine code into memory, and starts the 8088 microprocessor. Only the machine code differs for these two programs, so that the diskettes and hard-disks are accessed in the same way by both microprocessors. Most important, the operator need not be concerned which microprocessor chip is executing the program.

Compupro offers a more advanced desktop computer, that uses MP/M-8/16. MP/M-8/16 is Compupro's name for the combined multi-user operating system that employs MP/M-86 to perform single-user, single-stream CP/M functions, along with multi-user, multi-tasking MP/M operations. As in the Rainbow operating system, MP/M-8/16 automatically selects an 8085 or 8088 microprocessor chip to run the application program. Compupro's machine has all the features of the combined CP/M and CP/M-86 Rainbow computer, but adds the dimension of multiple taks operating from one or more consoles.

Another interesting dual-processor is Dynabyte's Monarch, which features a high-speed Z-80 coupled with an 8086 microprocessor. The Monarch operating system includes MP/M-II facilities using MP/M-86.

Extensions to concurrent operating systems

Dual-processor computers can also support concurrent operations, which let an operator run more than one program at a time. In a dual-processor environment,

one or more terminals can be connected to a computer that runs a mixture of 8-bit and 16-bit programs using both processors. Though the hardware and software systems for concurrent operation are more complex, they are simple extensions of the model given for the simpler case of CP/M.

The fundamental job of a concurrent operating system is to keep track of the status of each active program as they take turns using the processor. This is accomplished by maintaining, in memory, a machine state for each program -- the contents of the hardware registers internal to the microprocessor chip. A machine state is saved and exchanged whenever an active 8-bit or 16-bit program performs an input or output operation, or when a predetermined "slice" of time elapses.

The simplest way to extend the dual-processor example to concurrent operation is to ensure that the machine state is saved and exchanged only when the 8088 has control. This minimizes changes to MP/M-86 to support both processors, because only the 8088 machine state needs to be managed.

To ensure that the 8088 has control during a machine state change, consider the two possible ways to change that state: through a system call, or through an external asynchronous interrupt. A system call leads to the RUN85 program, which does operate under control of the 8088. The asynchronous interrupt, however, comes from an external device, such as a disk controller, keyboard, or timer, and can occur when either processor is operating. In this case, interrupt circuitry must be used, to allow the 8085 to discontinue processing and transfer to RUN85, before the interrupt is accepted.

Specifically, the circuit interrupts the 8085 if it has control, and holds the interrupt source until control is exchanged to the 8088. If the interrupt occurs when the 8085 is processing, the circuit forces a restart (RST) instruction onto the bus, to stop the 8085. The 8085 fields this interrupt, and transfers to RUN85. When RUN85 regains control, using the 8088, the interrupt circuit releases the interrupt source, and the machine state is changed by MP/M-86. This extended model for concurrent operation works properly, but has the restriction that only one 8085 program can participate in the multitasking environment at any given time. The restriction is due to the absolute 8085 memory assignment between hex locations 10000 and 1FFFF. To remove it, additional relocation hardware must be added to the 8088, to map the 64 Kilobyte 8085 memory to an arbitrary base location, usually to a "paragraph" address that is a multiple of 16 bytes.

With the relocation hardware, multiple 8085 programs can run under MP/M-86. Each 8085 program corresponds to one RUN85 program under MP/M-86. Each RUN85 program allocates its own 64 Kilobyte segment for the 8085 program, sets the relocation register whenever control transfers to the 8085. When the 8085 interrupt occurs, the 8085 interrupt handling code saves the 8085 machine state in its own 64 Kilobyte segment before transferring to RUN85. As long as the 8085 interrupt code and machine state are placed in the same relative location within each 8085 memory area, the relocation hardware automatically commences execution of the 8085 with the proper machine state. To save space and time, RUN85 is coded as a re-entrant resident system process under MP/M-86.

Concurrent CP/M provides similar features to MP/M-86, with the addition of so-

called "virtual screens", that let an operator switch between the interactions of a variety of programs. Concurrent CP/M requires the same model as MP/M-86 for operation with dual processors.

EOF

GKEH.WS4 (Gary Kildall and Earl Hunt ACM SIGIR article)

- "A Heathkit Method for Building Data Management Programs"
Earl Hunt, University of Washington, and
Gary Kildall, Naval Postgraduate School, Monterey, California
ACM SIGIR Information Storage and Retrieval Symposium, 1971, pp.117-131

(Retyped by Emmanuel ROCHE.)

[Portions of this research were supported by the National Science Foundation, Grant NSF B7-1438R, the United States Air Force Office of Scientific Research, Air Systems Command Grant AFOSR 70-1944 and by the University of Washington Institutional Research Fund. During the initial period in which the research was conducted, Gary Kildall held a National Science Foundation pre-doctoral fellowship at the University of Washington.]

Abstract

One of the difficulties faced in implementing information management and retrieval systems is that each case seems to present its own special complexities. As a result, information retrieval systems typically fall behind their programming schedule and have many bugs when delivered. In this paper, a set of basic operations on types of files are defined. These operations are intended to fulfill the same role for information retrieval systems programmers that functions such as LOG(X) fill for mathematical applications programmers... they should make the job very much easier. The file operations have been implemented as a run-time package written in FORTRAN IV and Burroughs Extended ALGOL. The approach has been used to develop three different information management systems; an APL interactive computing system, a generalized information retrieval system, and a specialized information retrieval system for map-oriented data. These systems are described.

Key words and phrases

Information retrieval, file management, information management, interactive programming, graphics, systems programming, paging

Programs to manage large amounts of intricately structured data are hard to write. They are delivered late and usually contain many bugs when delivered. Only indifferent success has been obtained from "generalized" data management systems. The usual way to write large data management programs is to obtain the service of several wise old (or bright, young) hands and hope that they will read Knuth [Ref. 7] and Lefkovitz [Ref. 8] before setting out to reinvent the wheel. Usually, this hope is not fulfilled. The resulting code is often so replete with programmer-specific or problem-specific tricks that it is hard to document, maintain, or modify.

We shall describe a method for the rapid construction of data management programs. The basic idea is that all data management is done through a set of subroutines, called a "kernel package", which we have found helpful. The programmer using or modifying a system designed with the kernel package need only grasp the few simple concepts and operations involved in the kernel. The approach is somewhat akin to the approach used to build complex gadgetry out of Heathkit or Digibit components, hence the name of the paper. Our approach has been applied in the construction of three substantial applications; a conversational version of the APL language, a generalized information retrieval system, and a system for graphics-based information retrieval. At least one other organization has also used our approach successfully. We shall first describe the technique, and then discuss the way it was used in each application.

Basic concepts

The term "Level 0" storage will refer to word-addressable, fast access store (usually core memory), and "Level 1" storage will refer to file-oriented random access store, normally a drum or disk. We assume a system which has both level 0 and level 1 store. The ideas could easily be generalized to a multi-level system. We also assume a user who writes a program to manipulate data in level 0 store, using the FORTRAN or ALGOL languages, but who has a very large amount of data that is to be held in level 1 store and requested on demand. The kernel package is used to relieve the user from worrying about either the management of level 1 store or the interplay between level 0 and level 1 store. The kernel also provides the user with commands for creating and manipulating very complicated data structures without ever being forced outside of the confines of FORTRAN or ALGOL.

Data structures

The user's data resides in a named file in level 1 storage. The kernel system divides the file in two ways. Physically, the file is organized into "pages", which are brought into level 0 as needed. The kernel system keeps track of the location of pages in both level 0 and level 1 store. Logically, the user's data is organized into "units" which he manipulates directly. Insofar as the user is concerned, a unit is a logically-connected set of records in one of two types of formats. If the unit is an "ordered storage unit", its entries are organized by fixed formats called "fields". These are established when the unit is created. One of the fields is designated as the "sort field". Within an ordered storage unit, records are kept sorted on the sort field. "Sequential storage units" contain variable-length character strings, placed in the unit wherever room for them can be found at the time they are inserted. Ordered storage units are typically used to provide indices by which particular strings may be found in sequential storage units. A simple example is shown in Figure 1.

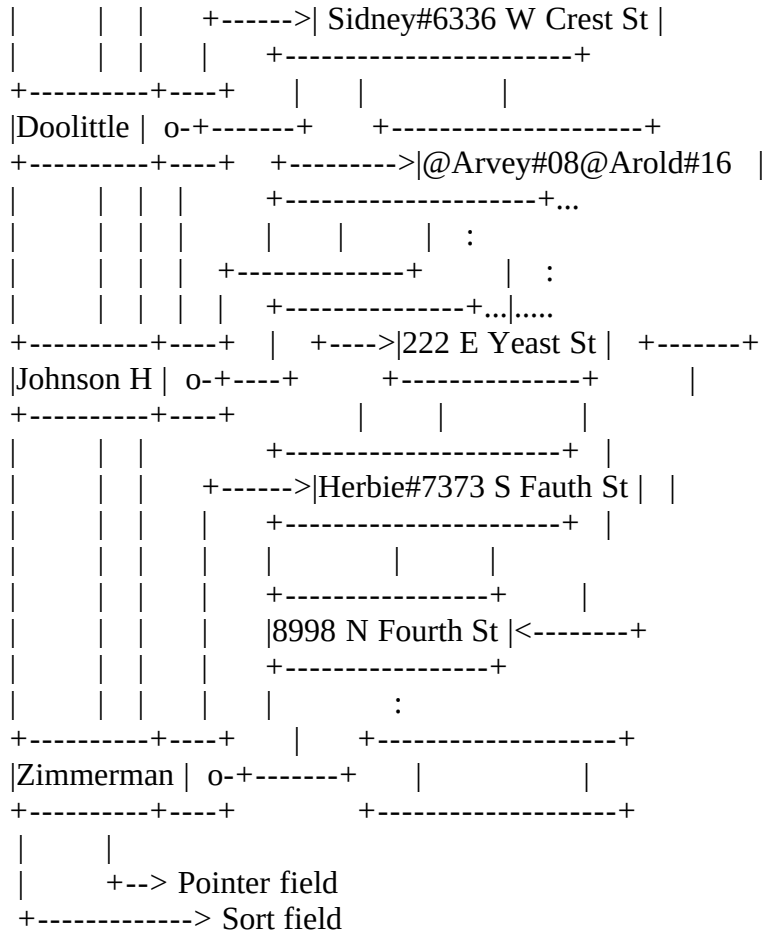


Figure 1. Ordered storage provides entry into sequential storage

Units are identified to the user's program by unit number. A unit is created (and space reserved for it) only when the user's program calls the appropriate kernel subroutine. Pages are created for units on demand. Page creation and manipulation is done entirely within the kernel system, so the user has no way of controlling or interfering with it.

Table 1. Kernel system subroutines

Format: Subroutine
Description

1) Storage maintenance subroutines

NAME
Identifies the user's level 1 file to the kernel subroutines.

SEQUENTIAL
Specifies that a particular unit is to be used as a sequential storage unit; unspecified units are treated as ordered storage units.

ALLOCATE
Indicates the amount of level 0 storage to be used for paging.

MAINTENANCE
Orders that the assignment of level 1 space be examined by the kernel routines to determine if more efficient balance can be achieved between the pages.

WRAPUP
Orders that all file maintenance be completed; it is used at the end of the user's program run.

2) Storage interrogation and alteration subroutines

STORESEQ
Stores a character string from the user's program into a specified sequential storage unit.

SEARCHSTORE
Searches a specific ordered storage unit for a programmer-provided character string, and then stores the string in proper sorted order in the unit.

SEARCHORD
Search an ordered storage unit for a specified character string; the location of the closest matching string is returned.

STOREORD
Stores a string of characters into an ordered storage unit in a specific location.

CONTENTS
Retrieves a character string from a specific programmer-provided location in a sequential or ordered storage unit.

DELETE
Removes entries from ordered or sequential storage units.

RELEASE UNIT
Orders the kernel subroutines to release all pages assigned to a particular unit to the garbage collector.

3) Storage utility subroutines

NEXTUNIT
Returns the number of the lowest unassigned unit to the user's program.

SIZE
Returns the number of entries in a particular storage unit.

UNITMODE
Returns the mode (sequential or ordered) of a particular unit.

System use

The subroutines within the kernel system are listed in Table 1. They fall into three major groups. "Storage maintenance" subroutines communicate between the user's program and the computer's operating system, and declare and release units. They also allow the user a small amount of control in balancing the use of physical resources. The user may specify the amount of level 0 memory to be reserved for pages (thus decreasing the number of references to level 1 storage to get data needed by the user program), and he may specify times at which page and unit assignments in level 1 store are to be re-examined to see if better storage utilization can be obtained by shuffling data between pages. The user does not control the process of examination.

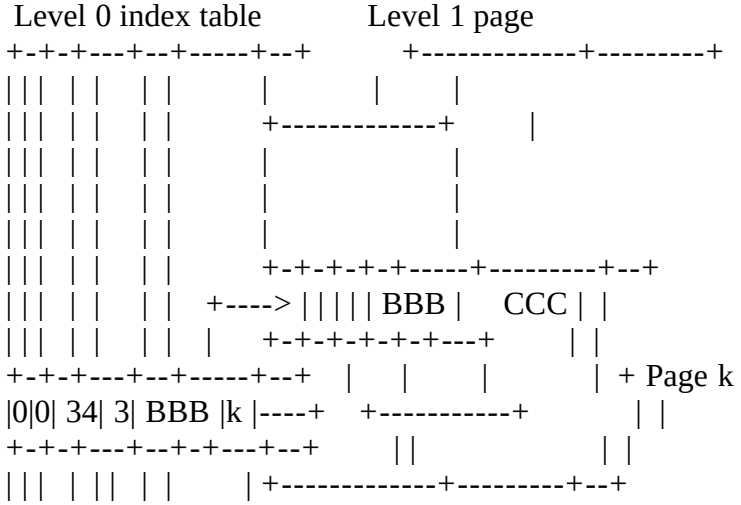
"Storage interrogation and alteration commands" allow the user to add, retrieve, or delete items to, from, or in units. The items are defined by naming character strings, simple variables, or arrays in the user's program. Finally, utility routines are provided to let the user determine the status of the data created by the kernel system.

The paging method

Let us shift from the user's view of the system as a package of subroutines to the systems program view, as a demand paging system for data.

When the user first initiates the system, he names a level 1 file. The kernel divides the file into pages. Each page consists of two parts; the "page descriptor" and the "page contents" (user information). The page descriptor indicates the physical layout of user information in the contents area, including the size of the free space area at the end of the page. The unit number, and type of the unit to which the page is assigned, is also in the page descriptor. Any number of pages may be assigned to a unit. Assignment is done by the kernel system in response to user needs.

The kernel system keeps track of page assignments through the use of an "index table" in level 0 store. The entries in this table for all pages assigned to a particular unit are stored sequentially. The table entries are descriptors of the page. The relation between the index table and the pages is shown in Figure 2.



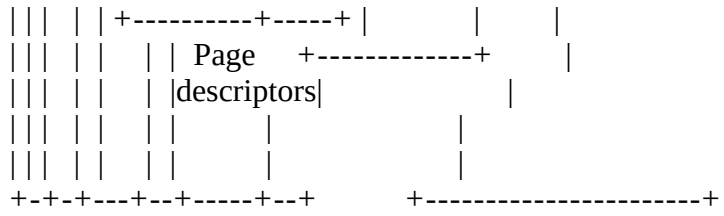


Figure 2. Relationship between page descriptors and index table

In addition to the index table, the kernel system maintains a "unit table" which specifies the first and last entries in the index table assigned to each unit. The unit table is also kept in level 0 store.

A basic principle of kernel system operation is that pages should be shuffled as little as possible. Therefore, when a page is assigned to a unit, a certain amount of blank "free space" is left at the end of the page. As more data is moved into the page, the free space is used as a buffer to avoid page overflow. If page overflows do occur, new pages are allocated. One of the major jobs of the MAINTENANCE subroutine is to examine pages, create new pages, and shuffle data between pages to avoid an imbalance between contents and free space that may have arisen as data was manipulated. This is done without moving data from level 1, by using descriptors in the index table. Having a free space area minimizes the number of times maintenance needs to be requested by the kernel system itself. Ideally, maintenance is limited to the post-run WRAPUP or to times indicated by the user's program. This is particularly useful in conversational application, since MAINTENANCE can be called while the program is awaiting input from a console. There is no chance of the system becoming unresponsive, for the MAINTENANCE subroutine (unlike WRAPUP) returns periodically to see if the program which called it wishes to go on to another task.

Copies of one or more pages may be held in level 0 storage at run time. The exact number of copies held is determined by the user, through the ALLOCATE command. The kernel system keeps track of the names of pages in level 0, so that there are no unnecessary accesses to level 1. In general, it pays to have at least two pages in core, so that one may have available portions of a sequential unit and portions of an ordered unit serving as an index to the sequential unit.

This concludes our brief description of the kernel program. Kildall [Ref. 5] describes it in much greater detail. The original kernel was written in Burroughs Extended ALGOL for the Burroughs B5500 and is inextricably tied to that machine. Two of the applications we will describe are B5500 programs. A second kernel has been written in FORTRAN IV. We have attempted to make it "machine independent" except where it interfaces with the operating system. For instance, the FORTRAN kernel must have the machine word and character size specified to it. The FORTRAN kernel has been tested and used for an application on the XEROX Data Systems Sigma 5, a 32-bit, byte-oriented machine. [We are aware of a second FORTRAN kernel used to implement a business information system on a Digital Equipment Corporation PDP-10. We have examined this kernel and, although it is written in FORTRAN, we feel that it uses so many (quite effective) machine-dependent tricks that it is solely a PDP-10 program.] Subsequently, we plan to test it on other machines.

Applications
==========

B5500 APL

A conversational APL interpreter has been written for the Burroughs B5500. The system is similar to APL/360 [Ref. 4] in its outward characteristics. It is a multi-user, conversational computing system operating as a user program under the Burroughs B5500 Master Control Program. B5500 APL is internally divided into several components; a resource management section which schedules work for the other components, a terminal message handler for input and output, a monitor command and function editor section through which the user defines, edits, and traces the execution of APL functions, and a compiler-simulator section which translates from APL code to the order code of an hypothetical APL computer and then simulates the action of that computer. These components are apparent to the user. Quite hidden from him, but of central concern to us, is the virtual memory management section, which controls the allocation of user space in level 1 store. This section is not specialized to APL. It could equally well be used for any conversational computing application where the user needed the illusion of having a very large memory.

When an APL user enters the system for the first time, he is assigned two storage units; an ordered unit called his "name table" and a sequential unit called his "data table". Since the system is designed for multiple users, the numbers of these units cannot be predicted in advance. From the viewpoint of the kernel system, then, the APL system program is the user and the APL user simply an input source for the user program. The APL system calls for units from the kernel as it needs them.

In addition to calling for units when a user enters the system, APL must call for units through its function editor, as the user builds his library of programs and data. The APL user can declare three types of names; scalar names, array names (for numeric or character arrays), and function names. All names, regardless of types, are given entries in the name table. If the name is the name of a scalar variable, its current value is also kept in the name table. If the name is an array name, the name table entry contains a pointer from the name table into the data table, where the string of characters defining the array value is located. When the name is the name of an APL function, the situation is more complicated. Upon user declaration of a function, two units are created for it: the "function label table" and the "function text table". The label table is an ordered unit containing two types of entries. There is one entry for each line of text in the definition of the APL function definition. The line entry contains pointers to the line definition, which is stored as a character string in the function text table.

Recall that, from the viewpoint of the kernel, the APL system itself is the user, and hence owns the file which the kernel is asked to organize. The result of the above process is that the kernel creates on the APL level 1 a virtual memory for the APL user, containing his APL program and the value for all his variables. Because the size of the units assigned to a user can be expanded as needed (so long as there is room on the APL file itself) by causing the kernel to create new pages, the APL user has the illusion of

having a very large machine. The way this machine is laid out in units is shown in Figure 3. The unit arrangement is transparent to the APL user, just as the page arrangement is transparent to the APL system programmer.

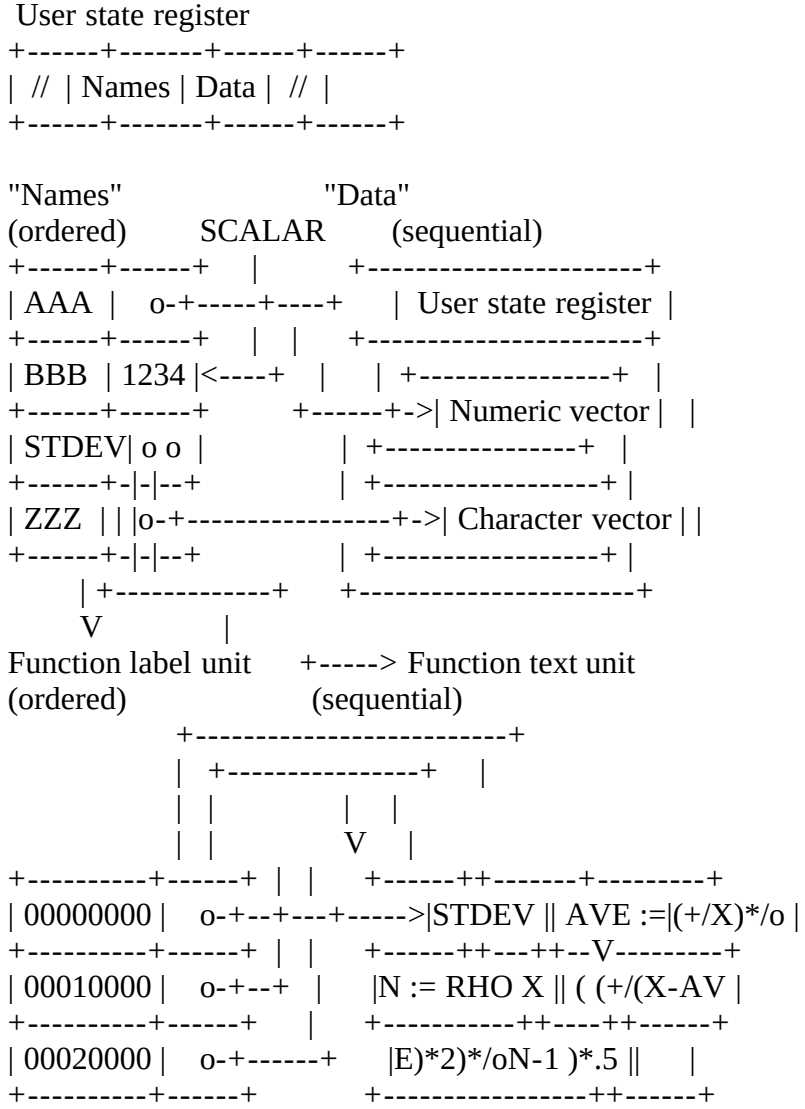


Figure 3. The format of user 'core' storage

To operate, the APL system must have an entry into the units assigned to each user. This is provided by a user state register (shown in Figure 3) which, among other things, contains the unit numbers of the user's name and data table. This is all the system needs to retrieve any piece of information about the status of the user's work area virtual memory. In practice, the user's APL program is executed by bringing appropriate bits and pieces of it from the level 1 file into a level 0 scratchapd area, where data is presented to the APL compiler and simulated APL computer. Note that, because of the elaborate entry system into user virtual memory provided by the kernel program, only a very small amount of APL program needs be brought into level 0 at any one time. For example, functions may be retrieved one line at a time. This greatly reduces the demand on costly level 0 memory without restricting the size of the program a user may write in APL.

A second interesting result of using the kernel system as a virtual memory for APL is that the system is very hard to disrupt due to computer system crashes.

The APL system keeps in its level 1 file the equivalent of a program counter for virtual memory. This program counter is updated whenever new data is moved to or from the scratchpad to level 1. When the machine crashes, the level 1 APL file will contain sufficient information to restart the user from the point of the last level 1 access before the crash. Typically, then, the amount of computing the user will lose is measured in milliseconds. This is no small advantage to any interactive computing system.

The B5500 APL system, and with it the Extended ALGOL kernel program, have now been in operation for over a year. We regard this as a stable application of the Heathkit technique. The details of the B5500 APL system are given elsewhere (Kildall [Ref. 6]).

A generalized information retrieval system

The B5500 Extended ALGOL kernel has also been used to construct a program called IRSYS, for defining and operating information retrieval systems, by Finke [Ref. 1 and 2]. Like the APL system, IRSYS is best thought of as a program by which the ultimate user defines his application, and not as an application program in itself. Again like the APL application, the IRSYS programmer is quite unaware of the kernel system, although the IRSYS program writer uses kernel continually.

Externally, IRSYS looks like any number of other information retrieval systems. Its basic user unit is the "data set". A data set consists of a reserved "data set symbol", followed by one or more "data set elements", and an "end symbol". A data set element consists of an "element symbol" and an "element value". The element value may be either a number or a character string. For example, a data set defining a book might be written

```
/DSET
$AUTHOR MEADOW C.T.
$DATE 1970
$PUBLISHER WILEY
$COMMENT TOPICAL TITLE
/END
```

The terms /DSET and /END are the data set symbol and end symbol, respectively. $AUTHOR, $DATE, etc. are element symbols, and the strings following them element values. A data set may have more than one element of the same type. IRSYS is used to define a file consisting of such items, and to retrieve items referenced by the values of different elements.

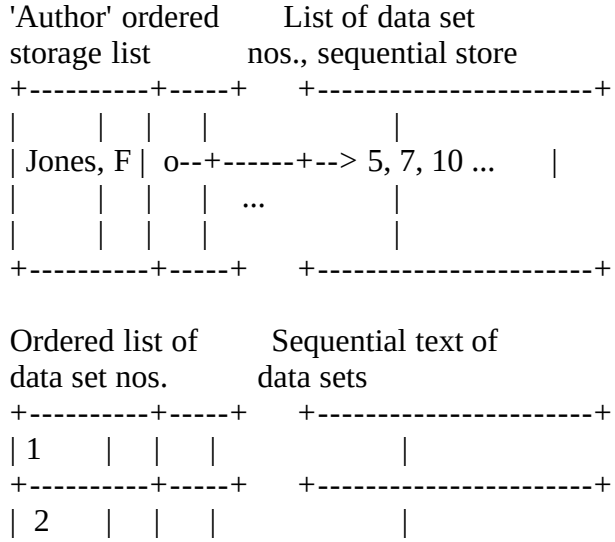
A user interacts with IRSYS in three ways. In "definition mode", the user states what the reserved symbols will be. Elements are defined to be retrievable with character string values, as $AUTHOR in the example above, numerically retrievable with numbers as values, as $DATE in the example, or miscellaneous, non-retrievable elements ($COMMENT above). IRSYS accepts these definitions and reserves the necessary tables for the user dictionary by calls to the kernel system. Unlike APL, IRSYS is a single user program, so the kernel is used to organize a separate IRSYS file for each user.

In "retrieval mode", the user writes queries about data sets by expressing Boolean combinations of statements about the values of retrievable elements. Relational statements may be used to reference values of numerically retrievable elements. Thus, the query for (($PUBLISHER WILEY) AND ($DATE .GT. 1960)) refers to all data sets with WILEY in the $PUBLISHER element and with a $DATE element greater than 1960. IRSYS will locate the data sets specified by a query and report their number. It will print these sets only on command, either on the line printer or on a console.

In "storage mode", the user stores data sets into his IRSYS file and edits data sets already in the file. Obviously, one definition session must precede any other session, and at least one storage session must precede the first retrieval session. Otherwise, there is no fixed order to the sequence of sessions in each mode. IRSYS can accept input either from a remote console, as an interactive IR system, or from the card reader in batch mode. A file established in batch mode may be interrogated from a console and vice versa.

Now, let us look at how IRSYS uses the kernel system. Only a few examples of the technique will be given, since there are many special uses to allow for "pathological" user definitions such as one name's being contained in another. Finke [Ref. 1] discusses all applications of the kernel with considerable detail and clarity.

Each retrievable element symbol has associated with it an ordered storage unit which is created when the retrievable element is defined. The value entry contains both the value name and a pointer to a second unit in which the strings are the internal numbers of data sets which have the appropriate element and value. Whenever a data set is entered in storage mode, it is assigned an internal number. Its elements are then examined. The element name table of each element in the data set is examined to see if the element value has appeared before. If it has, there will be an appropriate entry in the element table, which will point to a sequential unit. This unit contains a list of previous data sets which have the same element and value as the new data set. The internal number of the current data set is added to this list. If the element value has never appeared before, a new entry is made in the element name unit (in the proper place in the sequence) and a new, one entry list is created in the sequential storage unit. The result is a set of cross-index tables pointing to lists of data set numbers, as shown in Figure 4.



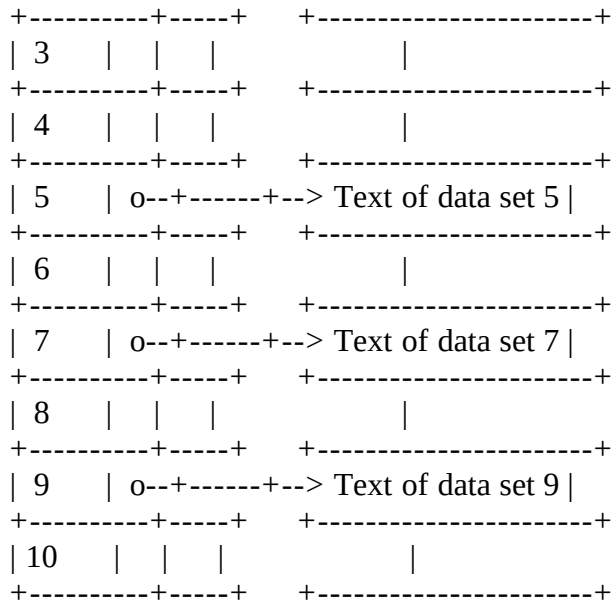


Figure 4. Organization of data set files in IR system

In addition, when the data set is entered, its entire text is converted to an internal form (from which the original form is recoverable), and is placed in a sequential unit called the DATA SET Table. An entry for the data set is also made, under its internal number, in the Data Set number table. The Data Set number table is an ordered storage unit whose entries point to the appropriate text in the Data Set Table. The result is shown in simplified form in Figure 4. Not shown is another ordered set in which IRSYS keeps a list of the element names and the number of their associated units. [The observant reader may have noted that, if this were all that IRSYS did, it would be unable to handle situations in which one term included another -- e.g. JOHN and JOHNSON, if both terms filled up one of the fixed fields of an ordered storage unit. Such problems are handled in IRSYS by a rather complex system of pointers from ordered to sequential units, the decription of which would add little to the current discussion.] IRSYS uses the kernel to keep this list and many others besides. By examining Figure 4, one can see that, given an element value, IRSYS has the capability of finding data sets containing that element.

In interactive information retrieval applications, the user must often sharpen his question before he locates the set of items he really wants. For example, suppose we were interrogating a file of cinema reviews. The question "$RATING INDECENT" might locate 4970 data sets, while the question "$RATING INDECENT AND $DATE .GT. 1965" might find only 100. [O tempora! O mores!] It would be more efficient if the questions were asked in series, and the search for "$DATE .GT. 1965" restricted to the set of data set numbers already retrieved in response to the first question. To allow for such situations, IRSYS permits the user to "nest" his questions, first asking a question which refers to a set of data sets, and then asking questions which are understood to refer only to that set. The kernel system is used to maintain the various temporary lists needed by IRSYS in this exchange.

IRSYS has now been in operation about nine months without maintenance. It has produced useful results for a number of users.

Graphic information retrieval

Our final example is different in a number of ways. The application involves the use of an XDS Sigma 5 to control a graphic display device known as an ARDS. [This is a Computer Display Incorporated Advanced Remote Display System (ARDS). The ARDS is essentially a storage type CRT, a keyboard, and a graphic input device known as a "mouse". By moving the mouse on a table, the user can manipulate a point or vector on the display face, which can then be located by the computer. In addition to the (hopefully) machine-independent kernel system, this application uses the systems software for controlling the mouse, ARDS, and communication equipment which is part of the operating system of the University of Washington Computer Science Teaching Laboratory (Hunt [Ref. 3]).]

A MAP MANIPULATOR program has been written in FORTRAN, using a FORTRAN IV kernel, to allow the user to draw a map (or portion of a map) on the face of the ARDS display. The map is a conventional street map, composed of streets, barriers, areas, and points of various types. Messages may be associated with any map element. Speed indicators are associated with areas and streets. The MAP MANIPULATOR program is not, itself, intended to do anything useful. It is intended to be a component in a larger system for displaying information in a "command and control" situation in which the user must observe and direct moving units, such as police patrol assignments or air traffic control.

Initially, the user "draws" the map on the ARDS display face using the mouse. At any time, he may edit the map or associate a message with an element of the map. The user indicates an element of interest either by "pointing" to it with the mouse, or by referencing to its internal number. (If the user does not know the internal number of an element, he may obtain it by pointing and asking.) Once the user has established a map, he may ask that different portions of it be displayed, at different magnifications, by zooming or windowing. Thus, he has at his command roughly the capabilities of a simplified SKETCHPAD system (Sutherland [Ref. 9]) specialized for map manipulation.

The basic information unit of MAP MANIPULATOR is the "map table". This is a sequential unit whose records are strings in a language for defining map elements. Each sentence in this language must conform to a BNF syntactical specifications. For example, the grammatical definition of a road is:

<road>::= 1 <roadname> <route> | <road> <route>
<roadname>::= {positive integer}
<route>::= <speed> <line>
<speed>::= {negative number}
<line>::= {coordinate pair} {coordinate pair}

The associated semantics are:

1 is an identification symbol to aid in interpretation. Remember that this "language" is for strings that will be read by a program, not a person.

The <roadname> integer is a pointer to an entry in the "road table",

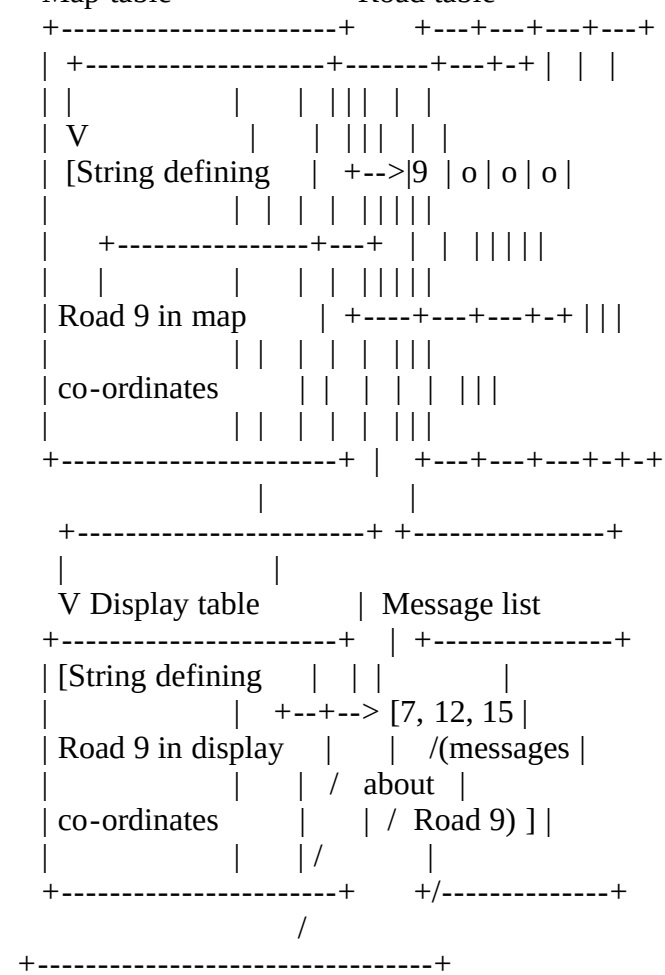
an ordered unit that will be described below.

The absolute value of <speed> indicates a multiple of the basic speed. This is interpreted as the value of speed of movement along the road. It is intended for use in answering questions about best routes from one point to another, or about the transit time along a particular route.

The coordinate pairs for the <line> are the endpoints of a vector. In the "map table", they are in map-coordinates, i.e. they refer to the scale of the map, which may be much more than can be displayed.

"Road table" is an ordered unit which contains one entry for each road in the map. This entry contains pointers back to the string defining the road in the "map table", and, if appropriate, to the string defining the road as displayed (in display face coordinates) on the sequential unit "display table". In addition, a "road table" entry contains a pointer to a "message list" table entry. "Message list" is a sequential unit whose records are lists of message numbers associated with a particular map element. Each message number names an entry in the ordered "message table" which, in turn, contains a pointer to the message text in the sequential "message text" table.

The overall structure of the data defining roads in MAP MANIPULATOR is shown in Figure 5. Similar figures could be constructed showing tables for barriers, areas, pointers and events.



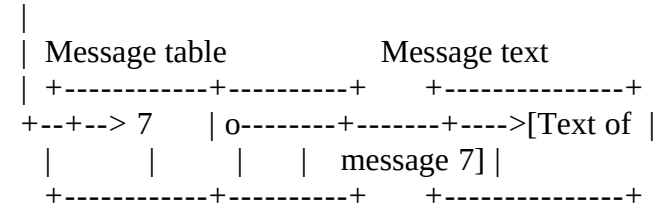


Figure 5. Structure of MAP MANIPULATOR

We have pointed out that "map table" defines elements in terms of map coordinates. A "display table" unit, similar in form to "map table", defines the elements actually being displayed at any one time in terms of display coordinates. This is necessary both for windowing and zooming, and to determine what element the user is pointing at with the mouse.

The structure of data in MAP MANIPULATOR gives us the potential for asking a number of questions about graphically-displayed data. For example, one can display a map, indicate an area by pointing to it, ask that the area be displayed at an appropriate magnification, and either insert or retrieve messages about the area. Note that these messages could be entries into an information retrieval system similar to one established by IRSYS. We have not actually made this connection as yet.

Summary

Nothing we have reported is terribly exciting or new. Building complex systems requires sophisticated tools. The Heathkit approach is an attempt to make it easier to program complex information management systems, and to produce programs that are easy to maintain and understand. Basically, the approach will work if many systems can be designed with the same tools. It will not work if every information management problem is unique. We think the Heathkit approach works well. While we certainly do not want to disparage any of our colleagues who have worked on the various applications we have described, we think it is correct to say that none of them came to these projects with any substantial background in system programming.

The timetable for the results was as follows:

> B5500 APL, which includes many things besides the virtual management memory package, was completed by four people, working at about a 25% effort for six months.
>
> IRSYS was completed by one person working half-time for six months.
>
> MAP MANIPULATOR was brought to a nearly workable stage by two people working quarter-time for three months. A substantial part of this time was spent checking the FORTRAN IV kernel system. The kernel system is far from simple, and obviously must work if anything else is to work.

We believe this is a good record. It may be, of course, that the resulting programs execute very inefficiently. It is hard to determine whether inefficiencies should be blamed on the Heathkit approach or on the fault in a

particular implementation. In fact, we are not all sure that inefficiency does follow from our approach. IRSYS has been used by a number of people who have not complained about the bills. IRSYS contains routines to monitor system operation, and this is currently being done. APL B5500 appears to be reasonably economic if the system on which it runs is not too heavily loaded. In any case, execution efficiency is not the only criterion for system programming. There is a good argument for bringing up a working system quickly and then replacing it with a highly efficient one at your leisure. How many microseconds are there in a month?

Acknowledgements

So many people have aided us at various stages that our acknowledgements must perforce read like movie credits. We list them alphabetically by project.

APL project: Hellmut Golde, LeRoy Smith, Sally Swedine, and Mary Zosel.

IRSYS: Jeanne Finke.

FORTRAN IV kernel: Thomas Kuffel

MAP MANIPULATOR: Jerry Jensen and Duane Sand.

And last, but not least, let us mention the first user who was willing to rely on a Heathkit-built system (IRSYS) to solve his problem, although he did not have the slightest interest in advances in computer science. Such a user always discovers the bugs in the perfectly checked out system. Our user, Professor David Bonsteel of the University of Washington School of Architecture, was no exception. May his buildings never crash.

References

1. "A Generalized File Management System"
Finke, Jeanne
University of Washinton Computer Science M.S. Thesis (1970a)

2. "A User's Guide to the Operation of an Information Storage and Retrieval System on the B5500 Computer"
Finke, Jeanne
University of Washington Computer Science Technical Report 70-1-3 (1970b)

3. "The Computer Science Teaching Laboratory at the University of Washington"
Hunt, Earl
ACM, SIGCSE Bull. 2, 1970, pp.30-33

4. "APL/360: User's Manual"
Iverson, Ken and Falkoff, A.D.
IBM Corp., 1968

5. "Experiments in Large Scale Computer Direct Access Storage Manipulation"

Kildall, Gary
University of Washington Computer Science Technical Report 69-01, 1969

6. "APL/B5500: The Language and Its Implementation"
Kildall, Gary
University of Washington Computer Science Technical Report 70-09-04, 1970

7. "The Art of Computer Programming, Vol. 1"
Knuth, Don
Addison-Wesley, 1969

8. "File Structures for Online Systems"
Lefkovitz, D.
Spartan Books, New-York, 1969

9. "SKETCHPAD, A Man-machine Graphical Communication System"
Sutherland, Ivan
Proceedings of the Spring Joint Computer Conference, 1963
Spartan Books, New-York, pp.329-346.

EOF

GKEI.WS4

- "The Evolution of an Industry: One Person's Viewpoint"
  Gary Kildall
  DDJ ("Dr. Dobb's Journal"), #41, Vol.5, No.1, January 1980, pp.6-7

(Retyped by Emmanuel ROCHE.)

1973...

I was sitting quietly at my desk when Masatoshi Shima hurried into my office at Intel and asked me to follow him to his laboratory down the hall. In the middle of his work bench, among the typical snaggle of jumpers, oscilloscopes and multi-meters, sat a binocular microscope with spider-leg probes, all of which were subjecting a minute piece of silicon to helpless investigation. I peered through the microscope at the enlarged regular patterns with particular interest. As a consultant, my job was to design and develop certain software tools for Intel. One was Interp/80, a program which simulated Intel's newly evolved 8080 microprocessor to be used by Intel customers on time-sharing systems. As I searched for something recognizable, I hoped my simulation resembled the operation of Shima's first 8080 chip which had finally come to life.

My proposal to Intel had been simple: I would provide them with a language, called PL/M, to replace serious systems programming in assembly language. The compiler would first be written in FORTRAN for operation on time-sharing computers and "cross-compiled" to the 8-bit processors. Next, we would write a PL/M compiler in PL/M and "boot-strap" from the time-sharing computer to a resident compiler operating on Intel's new Intellec-8 development system. The first part was complete, PL/M cross-compilers and Interp simulators were implemented for the now-best-forgotten 8008, as well as the 8080. Programs had been written and tested by Intel's software group, consisting of myself and two other people, and we were ready for the real machine. Things were going well: the resident compiler would be the next step.

Unfortunately, nearly all small computer systems in 1973 used paper tape as the backup storage device, with the ubiquitous model 33 Teletype serving as the nerve-shattering I/O device. It was readily apparent that resident development systems could not compete with time-sharing services when considering throughput, resources, and services. Still, the notion of a personal computer for software development interested everyone.

I became intrigued with a new device, called a floppy disk, which, though designed by IBM to replace punched cards, appeared to have much greater potential. The device was ideal: over 3,000 times the data rate of a Teletype, each $7 diskette could randomly access the equivalent of 2000 feet of paper tape. Best of all, the drive was priced at a low $500. Due to a slight problem of under-capitalization, I found this incredibly low price still a bit high.
At that time, a smallish company called Shugart Associates was in operation a few miles up the road from Intel. Dave SCOTT, then marketing manager at Shugart Associates, donated one of their 10.000-hour test drives to the cause,

complete with worn-out bearings and a bearing repair kit. It was only later, as I sat in my office at home, staring at the naked disk drive, that I realized I had no cabinet, no cables, no power supplies, no controller, and most distressing of all, no hardware design experience. To make matters worse, no controllers were commercially available, even if I could afford one.

After several abortive attempts at constructing an interface to my Intellec-8, it became readily apparent that my efforts would be better directed toward the software aspects. Between projects, I put together the first CP/M file system, designed to support a resident PL/M compiler. The time-sharing version of PL/M, along with the Interp simulator, allowed me to develop and checkout the various file operations to the level of primitive disk I/O. A simulation is, after all, just a simulation, and the inability to make that 10.000-hour drive work for just one more hour was frustrating.

Shortly thereafter, in the Fall of 1974, John Torode became interested in the project. I offered as much moral support as possible while John worked through the aberrations of the IBM standard to complete one of my aborted controllers. Our first controller was a beautiful rat's nest of wirewraps, boards and cables (well, at least it was beautiful to us!) which, by good fortune, often performed seeks, reads, and writes just as requested. For agonizing minutes, we loaded the CP/M machine code through the paper tape reader into the Intellec-8 memory. To our amazement, the disk system went through its initialization and printed the CP/M prompt at the Teletype.

Anyone who has brought-up CP/M on a homebuilt computer has felt this moment of elation. A myriad of connections are properly closed, bits are flying at lightning speeds over busses and through circuits and program logic to produce a single prompt. In comparison to our paper tape devices, we had the power of a S/370 at our fingertips. A few nervous tests confirmed that all was working properly, so we retired for the evening to take on the simpler task of emptying a jug of not-so-good red wine while reconstructing battles, and speculating on the future of our new software tool.

In the months that followed, CP/M evolved rather slowly. Intel was experiencing enormous growth, and all software development was halted while new management structures were instituted. Intel expressed no interest in CP/M, nor in continuing any resident compiler work. Nearly two years passed before Intel again took interest in resident software tools, with their introduction of the ISIS operating system and later, the resident PL/M-80 compiler.

Meanwhile, John Torode redesigned and refined our original controller and produced his first complete computer system, marketed under his company name, Digital Systems (which later became Digital Microsystems). The first commercial licensing of CP/M took place in 1975 with contracts between Digital Systems and Omron of America for use in their intelligent terminal, and with Lawrence Livermore Laboratories where CP/M was used to monitor programs in the Octopus network. Little was paid to CP/M for about a year. In my spare time, I worked to improve overall facilities, and added an editor, assembler, and debugger which were predecessors of the current ED, ASM, and DDT programs. By this time, CP/M had been adapted for four different controllers.

In 1976, Glenn Ewing approached me with a problem: IMSAI Incorporated, for

whom Glenn consulted, had shipped a large number of disk sub-systems with a promise that an operating system would follow. I was somewhat reluctant to adapt CP/M to yet another controller, and thus the notion of a separated Basic I/O System (BIOS) evolved. In principle, the hardware-dependent portions of CP/M were concentrated in the BIOS, thus allowing Glenn, or anyone else, to adapt CP/M to the IMSAI equipment. IMSAI was subsequently licensed to distribute CP/M version 1.3, which eventually evolved into an operating system called IMDOS.

By coincidence, Jim Warren and I were both consulting at Signetics Corporation during this time. Jim was then the editor of DDJ, and pushed for sale of CP/M to the general public. There was, at the time, a pervading paranoia among software vendors who felt that any and all loose software would be immediately "ripped-off" by this immoral group of computer junkies. Jim's faith in the industry, however, led me to introduce the CP/M 1.3 system for sale on the open market at $70 per copy. In the months that followed, the nature of the computer hobbyist became apparent. In most case he was, like myself, in the computer industry and merely wanted a personal computer for his own endeavors. CP/M gradually gained popularity through a "grassroots" effect and, to the amazement of the skeptics, the rip-off factor was practically nil. A new company called Digital Research was formed to support CP/M, develop new products, and provide administrative functions.

It has been nearly three years since CP/M's initial introduction, with several revisions and improvements. Although floppy disks maintain their popularity, CP/M 2.0 is now offerred to manage larger capacity disks which are becoming more readily available. Customer needs and demands have also led to the recent introduction of MP/M, a CP/M-compatible multi-terminal multi-programming system for more sophisticated applications.

More important, however, is that CP/M provides a manufacturer-independent basis for an evolving software market. We all know that software is expensive to develop and support, with numbers quoted in the hundreds of thousands of dollars over the product lifetime. In a classical computer market-place, these costs are amortized over a few installations, resulting in seemingly outrageous prices. Active CP/M users, however, number in the tens of thousands and can be reached through any number of popular magazines. Thus, marketers reduce their prices substantially to interest a much larger customer base. Software is sold profitably as an independent commodity by a large number of responsible companies, and the benefits to the consumer are clear. Competition forces low prices and quality control, with selection among a wide variety of software products. Currently, CP/M-compatible products range from word-processing programs through business systems to a variety of languages processors for BASIC, FORTRAN, COBOL, Pascal, and others. All are priced in the $100 to $700 range. The future is, without doubt, optimistic for producers and customers alike.

EOF

GKFS.WS4 (= Gary Kildall First Symposium)

- "A Unified Approach to Global Program Optimization"
  Gary A. Kildall
  ACM First Symposium on Principles of Programming Languages, 1973

(Retyped by Emmanuel ROCHE.)

Abstract

A technique is presented for global analysis of program structure, in order to perform compile-time optimization of object code generated for expressions. The global expression optimization presented includes constant propagation, common subexpression elimination, elimination of redundant register load operations, and live expression analysis. A general-purpose program flow analysis algorithm is developed, which depends upon the existence of an "optimizing function". The algorithm is defined formally using a directed graph model of program flow structure, and is shown to be correct. Several optimizing functions are defined which, when used in conjunction with the flow analysis algorithm, provide the various forms of code optimization. The flow analysis algorithm is sufficiently general that additional functions can easily be defined for other forms of global code optimization.

Section 1: Introduction

A number of techniques have evolved for the compile-time analysis of program structure, in order to locate redundant computations, perform constant computations, and reduce the number of store-load sequences between memory and high-speed registers. Some of these techniques provide analysis of only straight-line sequences of instructions [Ref.5,6,9,14,17,18,19,20,27,29,34,36, 38,39,43,45,46], while others take the program branching structure into account [Ref.2,3,4,10,11,12,13,15,23,30,32,33,35]. The purpose, here, is to describe a single program flow analysis algorithm which extends all of these straight-line optimizing techniques to include branching structure. The algorithm is presented formally, and is shown to be correct. Implementation of the flow analysis algorithm in a practical compiler is also discussed.

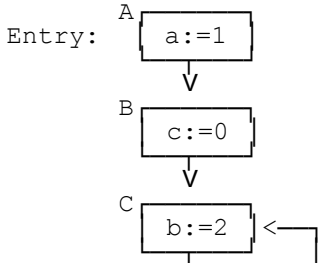
The methods used here are motivated in the section which follows.

Section 2: Constant propagation

A fairly simple case of program analysis and optimization occurs when constant computations are evaluated at compile-time. This process is referred to as "constant propagation", or "folding". Consider the following skeletal ALGOL-60 program:

```
begin integer i,a,b,c,d,e;
a:=1; c:=0; ...
for i:=1 step 1 until 10 do
    begin b:=2; ...
    d:=a+b; ...
    e:=b+c; ...
    c:=4; ...
    end
end
```

This program is represented by the directed graph shown in Figure 1 (ignoring calculations which control the FOR-loop). The nodes of the directed graph represent sequences of instructions containing no alternate program branches, while the edges of the graph represent program control flow possibilities between the nodes at execution-time.



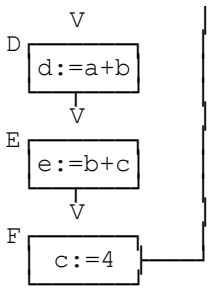


Figure 1. A program graph corresponding to an ALGOL-60 program containing one loop.

For purposes of constant propagation, it is convenient to associate a "pool" of propagated constants with each node in the graph. The pool is a set of ordered pairs which indicate variables which have constant values when the node is encountered. Thus, the pool of constants at node B, denoted by P_B, consists of the single element (a,1) since the assignment a:=1 at node A must occur before node B is encountered during execution of the program.

The fundamental global analysis problem is that of determining the pool of propagated constants for each node in an arbitrary program graph. By inspection of the graph of Figure 1, the pool of constants at each node is

$P_A = 0$
$P_B = \{(a,1)\}$
$P_C = \{(a,1)\}$
$P_D = \{(a,1),(b,2)\}$
$P_E = \{(a,1),(b,2),(d,3)\}$
$P_F = \{(a,1),(b,2),(d,3)\}$.

In the general case, P_N could be determined for each node N in the graph as follows. Consider each path $(A,p_1,p_2,\ldots,p_n,N)$ from the entry node A to the node N. Apply constant propagation throughout this path to obtain a set of propagated constants at node N for this path only. The intersection of the propagated constants determined for each path to N is then the set of constants which can be assumed for optimization purposes, since it is not known which of the paths will be taken at execution-time.

The pool of propagated constants at node D of Figure 1, for example, can be determined as follows. A path from the entry node A to the node D is (A,B,C,D). Considering only this path, the "first approximation" to P_D is

$$P_D^1 = \{(a,1),(b,2),(c,0)\}$$

A longer path from A to D is (A,B,C,D,E,F,C,D) which results in the pool

$$P_D^2 = \{(a,1),(b,2),(c,4),(d,3),(e,2)\}$$

corresponding to this particular path to D. Successively longer paths from A to D can be evaluated, resulting in P_D^3, P_D^4, ..., P_D^n for arbitrarily large n. The pool of propagated constants which can be assumed, no matter which flow of control occurs, is the set of constants common to all P_D^i; that is,

$$P_D = \bigcap_i P_D^i$$

This procedure, however, is not effective since the number of such paths may have no finite bound in the case of an arbitrary directed graph. Hence, the procedure would not necessarily halt. The purpose of the algorithm of the following section is to compute this intersection in a finite number of steps.

Section 3: A global analysis algorithm

The analysis of the program graph of Figure 1 suggests a solution to the global constant propagation problem. Considering node C, the first approximation to P_C is given by propagating constants along the path (A,B,C), resulting in

$$P_C^1 = \{(a,1),c,0)\}.$$

Based upon this approximate pool, the first approximations to subsequent nodes can be determined:

$P_D^1 = \{(a,1),(c,0),(b,2)\}$
$P_E^1 = \{(a,1),(c,0),(b,2),(d,3)\}$
$P_F^1 = \{(a,1),(c,0),(b,2),(d,3),(e,2)\}$.

Using P_F^1, the constant pool resulting from node F entering node C is

$P = \{(a,1),(b,2),(d,3),(e,2),(c,4)\}$.

Note, however, that since

$$P_C = \bigcap_i P_C^i$$

it follows that $P_C \_ P_C^1 \cap P_C^2$. Thus, rather than assuming $P_C^2 = P$, the second approximation to P_C is taken as

$$P_C^2 = P_C^1 \cap P = P_C^1 \cap \{(a,1),(b,2),(d,3),(e,2),(c,4)\} = \{(a,1)\}.$$

Using P_C^2, the circuit through the loop past C is traced once again. The next approximation at subsequent nodes can then be determined, based upon P_C^2:

$P_D^2 = P_D^1 \cap \{(a,1),(b,2)\} = \{(a,1),(b,2)\}$,
$P_E^2 = P_E^1 \cap \{(a,1),(b,2),(d,3)\} = \{(a,1),(b,2),(d,3)\}$,
$P_F^2 = P_F^1 \cap \{(a,1),(b,2),(d,3)\} = \{(a,1),(b,2),(d,3)\}$.

Continuing around the loop once again from node F to node C, the third approximate pool P_C^3 is determined as

$$P_C^3 = P_C^2 \cap \{(a,1),(b,2),(d,3)\} = \{(a,1)\}.$$

Clearly, no changes to subsequent approximate pools will occur if the circuit is traversed again since $P_C^3 = P_C^2$, and the effect of P_C^2 on the pools in the circuit has alredy been investigated. Thus, the analysis stops, and the last approximate pools at each node are taken as the final constant pools. Note that these last approximations correspond to the constant pools determined earlier by inspection.

Based upon these observations, it is possible to informally state a global analysis algorithm.

a. Start with an entry node in the program graph, along with a given entry pool corresponding to this entry node. Normally, there is only one entry node, and the entry pool is empty.

b. Process the entry node, and produce optimizing information (in this case, a set of propagated constants) which is sent to all immediate successors of the entry node.

c. Intersect the incoming optimizing pools with that already established at the successor nodes (if this is the first time the node is encountered, assume the incoming pool as the first approximation, and continue processing).

d. Considering each successor node, if the amount of optimizing information is reduced by this intersection (or if the node has been encountered for the first time) then process the successor in the same manner as the initial entry node (the order in which the successor nodes are processed is unimportant).

In order to generalize the above notions, it is useful to define an "optimizing function" f which maps an "input" pool, along with a particular node, to a new "output" pool. Given a particular set of propagated constants, for example, it is possible to examine the operation at a particular node and determine the set of propagated constants which can be assumed after the node is executed. In the case of constant propagation, the function can be informally stated as follows. Let V be a set of variables, let C be a set of constants, and let $\underline{N}$ be the set of nodes in the graph being analyzed. The set $U = V \times C$ represents ordered pairs which may appear in any constant pool. In fact, all constant pools are elements of the power set of U (i.e., the set of

all subsets of U), denoted by P(U). Thus,

$f: N \times P(U) \rightarrow P(U)$,

where (v,c) f(N,P) <=>

a. (v,c) P and the operation at node N does not assign a new value to the variable v,

or

b. the operation at node N assigns an expression to the variable v, and the expression evaluates to the constant c, based upon the constants in P.

Consider the graph of Figure 1, for example. The optimizing function can be applied to node A with an empty constant pool, resulting in

$f(A,0) = \{(a,1)\}$.

Similarly, the function f can be applied to node B with {(a,1)} as a constant pool yielding

$f(B, \{(a,1)\}) = \{(a,1),c,0)\}$.

Note that given a particular path from the entry node A to an arbitrary node N N, the optimizing pool which can be assumed for this path is determined by composing the function f up to the last node of the path. Given the path (A,B,C,D), for example,

$f(C,f(B,f(A,0))) = \{(a,1),(c,0),(b,2)\}$

is the constant pool at D for this path.

The pool of optimizing information which can be assumed at an arbitrary node N in the graph being analyzed, independent of the path taken at execution time, can now be stated formally as

$P_N = \cap x$
$x \ F_N$

where

$F_N = \{f(p_n,f(p_{n-1},...f(p_1,P))...) \mid (p_1,p_2,...,p_n,N)$ is a path from an entry node P_1 with corresponding entry pool P to the node N}.

Before formally stating the global analysis algorithm, it is necessary to clarify the fundamental notions.

A finite directed graph G = <N,E> is an arbitrary finite set of "nodes" N and "edges" E _ N × N. A "path" from A to B in G, for A,B N, is a sequence of nodes $(p_1,p_2,...,p_k)$ P_1 = A and P_k = B, where (p_i,p_{i+1}) E i, $1 \leq i < k$. The "length" of a path $(p_1,p_2,...,p_k)$ is k-1.

A "program graph" is a finite directed graph G along with a non-empty set of "entry nodes" _ N such that given N N a path $(p_1,...,p_n)$ P_1 and P_n = N (i.e., there is a path to every node in the graph from an entry node).

The set of "immediate successors" of a node N is given by

$I(N) = \{N' \in N \mid (N,N') \in E\}$.

Similarly, the set of "immediate predecessors" of N is given by

$I^{-1}(N) = \{N' \in N \mid (N',N) \in E\}$.

Let the finite set P be the set of all possible optimizing pools for a given application (e.g., P = P(U) in the constant propagation case, where U = V × C), and ^ be a "meet" operation with the properties

^: P × P → P,
x ^ y = y ^ x (commutative),
x ^ (y ^ z) = (x ^ y) ^ z (associative),

where x, y, z P. The set P and the ^ operation define a finite meet-semilattice.

The ^ operation defines a partial ordering on P given by

$x \leq y <=> x \wedge y = x$ x,y P.

Similarly,

$$x < y \iff x \leq y \text{ and } x \neq y.$$

Given $X \subseteq \underline{P}$, the generalized meet operation

$$\bigwedge_{x \in X} x$$

is defined simply as the pairwise application of ^ to the elements of X. $\underline{P}$ is assumed to contain a "zero element" $\underline{0}$ $\underline{0} \leq x$ $x \in \underline{P}$. An augmented set $\underline{P}'$ is constructed from $\underline{P}$ by adding a "unit element" $\underline{1}$ with the properties $\underline{1} \in \underline{P}$ and $\underline{1} \wedge x = x$ $x \in \underline{P}$; $\underline{P}' = \underline{P} \cup \{\underline{1}\}$. It follows that $x < \underline{1}$ $x \in \underline{P}$.

An "optimizing function" f is defined

$$f: \underline{N} \times \underline{P} \rightarrow \underline{P}$$

and must have the homomorphism property:

$$f(N, x \wedge y) = f(N,x) \wedge f(N,y), \quad N \in \underline{N}, \; x,y \in \underline{P}.$$

Note that $f(N,x) < \underline{1}$ $N \in \underline{N}$ and $x \in \underline{P}$.

The global analysis algorithm is now stated:

Algorithm A

Analysis of each particular program graph G depends upon an "entry pool set" $\subseteq \underline{N} \times \underline{P}$, where $(e,x) \in \subseteq$ if e is an entry node with corresponding entry optimizing pool $x \in \underline{P}$.

A1[initialize]	$L \leftarrow \subseteq$.
A2[terminate?]	If $L = 0$ then halt.
A3[select node]	Let $L' \in L$, $L' = (N,P_i)$ for some $N \in \underline{N}$ and $P_i \in \underline{P}$, $L \leftarrow L - \{L'\}$.
A4[traverse?]	Let P_N be the current approximate pool of the optimizing information associated with the node N (initially, $P_N = \underline{1}$).
A5[set pool]	$P_N \leftarrow P_N \wedge P_i$, $L \leftarrow L \cup \{(N', f(N,P_N)) \mid N' \in I(N)\}$.
A6[loop]	Go to step A2.

For purposes of constant propagation, $\underline{P} = \mathcal{P}(U)$, where $U = V \times C$, as before. The meet operation is $\cap$, and the less-than-or-equal relation is $\subseteq$. Note that the zero element in this case is $0 = \mathcal{P}(U)$. The unit element in $\mathcal{P}(U)$ is U itself. The algorithm requires a new unit element, however, which is not in $\mathcal{P}(U)$. The new unit element is constructed as follows: let δ be a symbol not in U, and let $\underline{U} = U \cup \{\delta\}$. It follows that $\underline{U} \cap x = x$ $x \in \mathcal{P}(U)$ and $\underline{U} \in \mathcal{P}(U)$. Thus, $\underline{P}' = \underline{P} \cup \{\underline{U}\}$ is obtained from $\underline{P}$ by adding a unit element $\underline{U}$. As demonstrated in the proof in Theorem 2, the addition of the symbol δ to U causes the algorithm A to consider each node in the program graph at least once.

Appendix A shows the analysis of the program graph of Figure 1 using the entry pool set $\subseteq = \{(A,0)\}$.

Theorem 1

The algorithm A is finite.

Proof

The algorithm A terminates when $L = 0$. Each evaluation of step A3 removes an element from L, and elements are added to L only in step A5. Thus, A is finite if the number of evaluations of step A5 is finite. Informally, each evaluation of step A5 reduces the "size" of the pool P_N at some node N. Since the size cannot be less than $\underline{0}$, the process must be finite. Formally, step A5 is performed only when $P_N \neq P_N \wedge P_i$. But $(P_N \wedge P_i) \wedge P_N = P_N \wedge P_i \Rightarrow P_N \wedge P_i \leq P_N$, and $P_N \wedge P_i \neq P_N \Rightarrow P_N \wedge P_i < P_N$. Thus, the approximate pool P_N at node N can be reduced at most to $\underline{0}$ since $P_N \leftarrow P_N \wedge P_i$. Further, since the first approximation to P_N is $\underline{1}$ and the lattice is finite, it follows that step A5 can be performed only a finite number of times. Thus A is finite.

An upper bound on the number of steps in the algorithm A can easily be determined. Let n be the cardinality of $\underline{N}$ and h($\underline{P}$') be a function of $\underline{P}$' (which, in turn, may be a function of n) providing the maximum length of any chain between $\underline{1}$ and $\underline{0}$ in $\underline{P}$'. Step A5 can be executed a maximum of h($\underline{P}$') times for any given node. Since there are n nodes in the program graph, step A5 can be performed no more than n · h($\underline{P}$') times.

In the case of constant propagation, for example, let u be the cardinality of U. The size of U varies directly with the number of nodes n. In addition, the maximum length of any chain $u_1,u_2,...,u_k$ such that u_1 = U and u_k = 0, where u_1 u_2 u_3 ... u_k is u. Thus, h((U)) = u; and the theoretical bound is n · u. Since u varies directly with n, it follows that the order of the algorithm A is no worse than n^2.

The correctness of the algorithm A is guaranteed by the following theorem.

Theorem 2

Let F_N = {f(p_n,f(p_{n-1},..., f(p_1,P))...) | (p_1,...,p_n,N) is a path from an entry node p_1 with corresponding entry pool P to the node N}. Further, let

$$X_N = \begin{matrix} x \\ x \quad F_N \end{matrix}$$

corresponding to a particular program graph G, set $\underline{P}$', and optimizing function f, which satisfy the conditions of the algorithm A. If P_N is the final approximate pool associated with node N when A halts, then $P_N = X_N$ N N.

Theorem 2 thus relates the final output of the algorithm to the intuitive results which were developed earlier. The proof of Theorem 2 is given in Appendix B.

An interesting side-effect of Theorem 2 is that the order of choice of elements from L in step A3 is arbitrary, as given in the following corollary.

Corollary 1

The final pool P_N associated with each node N $\underline{N}$ upon termination of the algorithm A is uniquely determined, independent of the order of choice of L' from L in step A3.

Proof

This corollary follows immediately, since the proof of Theorem 2 in Appendix B is independent of the choice of L'.

Since the choice of L' from L in step A3 is arbitrary, it is interesting to investigate the effects of the selection criteria upon the algorithm. The number of steps to the final solution is clearly affected by this choice. No selection method has been established, however, to maximize this convergence rate. One might also notice that, by treating accesses to L as critical sections in steps A3 and A5, the elements of L can be processed in parallel. That is, independent processes can be started in step A3 to analyze all elements of L.

It is important to note, at this point, that the algorithm A allows one to ignore the global analysis, and concentrate upon development of straight-line code optimizing functions. That is, if an optimizing function f can be constructed for optimizing a sequence of code containing no alternative branches, then the algorithm A can be invoked to perform the branch analysis, as long as f satisfies the conditions of the algorithm.

Section 4: Common subexpression elimination

Global common subexpression elimination involves the analysis of a program's structures in order to detect and eliminate calculations of redundant expressions. A fundamental assumption is that it requires less execution time

to store the result of a previous computation and load this value, when the redundant expression is encountered.

As an example, consider the simple sequence of expressions:

... r:=a+b; ... r+x ... (a+b)+x ...

which could occur as a part of an ALGOL-60 program.

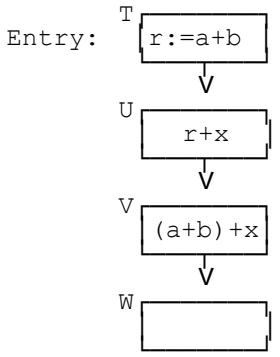


Figure 2. An acyclic program graph representing a simple computation sequence.

Figure 2 shows this sequence written as a directed graph. Note that the redundant expression (a+b) at node V is easily recognized. The entire expression (a+b)+x at node V is redundant, however, since r has the same value as a+b at node U, and r+x is computed at node U, ahead of node V. It is only necessary to describe an optimizing function f which detects this situation for straight-line code; the algorithm A will make the function globally applicable.

A convenient representation for the optimizing pool, in the case of common subexpression elimination, is a partition of a set of expressions. The expressions in the partition at a particular node are those which occur before the node is encountered at execution-time.

The optimizing function for common subexpression elimination manipulates the equivalence classes of the partition. Two expressions are placed into the same class of the partition if they are known to have equivalent values. Considering Figure 2, for example, the set of expressions which are evaluated before node T is encountered is empty; thus, $P_T = 0$. The expressions evaluated before node U are exactly those which occur at node T, including partial computations. The set of (partial) computations at node T is {a,b,a+b,r}. Since r takes the value of a+b at node T, r is known to be equivalent to a+b. Thus, P_U = {a|b|a+b,r}, where "|" separates the equivalence classes of the pool. Similarly, P_V = {a|b|a+b,r|x|r+x} and P_W = {a|b|a+b,r|x|r+x|(a+b)+x}. The expression a+b at node V is redundant, since a+b is in the pool P_V.

Note, however, that the redundant expression (a+b)+x at node V is not readily detected. This is due to the fact that r+x was computed at node U and, as noted above, the evaluation of r+x is the same as evaluation of (a+b)+x at node U. In order to account for this in the output optimizing pool, (a+b)+x is added to the same class as r+x. Thus, P_V becomes

{a|b|a+b,r|x|r+x,(a+b)+x}.

This process is called "structuring" an optimizing pool. Structuring consists of adding any expressions to the partition which have operands equivalent to the one which occurs at the node being considered. The entire expression (a+b)+x at node V is then found to be redundant, since the structured pool P_V contains a class with (a+b)+x.

An optimizing function $f_1(N,P)$ for common subexpression elimination can now be informally stated.

1. Consider each partial computation e in the expression at node N N.
2. If the computation e is in a class of P then e is redundant; otherwise
3. create a new class in P containing e and add all (partial) computations which occur in the program graph and which have operands equivalent to e (i.e., structure the pool P).
4. If N contains an assignment d:=e, remove from P all expressions

containing d as a subexpression. For each expression e' in P containing e as a subexpression, create e'' with d substituted for e, and place e'' in the class of e'.

The meet operation ^ of the algorithm A must be defined for common subexpression elimination. Since the optimizing pools in $\underline{P}'$ are partitions of expressions, the natural interpretation is as intersection by classes, denoted by . That is, given $P_1, P_2 \quad \underline{P}'$, $P = P_1 \quad P_2$ is defined as follows.

Let C = p n p
p P_1 p P_2

and $P(c) = P_1(c) \cap P_2(c) \quad c \quad C$.

C is the set of expressions common to both P_1 and P_2, while $P_1(c)$ and $P_2(c)$ are the classes of c in P_1 and P_2, respectively. Thus, the class of each c C in the new partition P is derived from P_1 and P_2 by intersecting the classes $P_1(c)$ and $P_2(c)$. For example, if $P_1 = \{a,b|d,e,f\}$ and $P_2 = \{a,c|d,f,g\}$ then C = {a,d,f} and $P_1 \quad P_2 = \{a|d,f\}$.

It is easily shown that has the properties required of the meet operation; hence, a "refinement" relation is defined:

$$P_1 \_ P_2 \iff P_1 \quad P_2 = P_1.$$

That is, $P_1 \_ P_2$ if and only if P_1 is a refinement of P_2. The refinement relation provides the ordering required on the set $\underline{P}'$ for the algorithm A.

The function f_1 can be stated formally, and shown to have the homomorphism property required by the global analysis algorithm [Ref.33]:

$$f_1(N, P_1 \quad P_2) = f_1(N, P_1) \quad f_1(N, P_2).$$

Before considering an example of the use of f_1 with the algorithm A, the function f_1 is extended to combine constant propagation with common subexpression elimination.

Section 5: Constant propagation and common subexpression elimination

The common subexpression elimination optimizing function f_1 of Section 4 can easily be extended to include constant propagation. Consider, for example, the following segment of an ALGOL-60 program:

```
... u:=20; ... v:=30; ... u+v ... x:=10;
... y:=40; ... x+y ... y-x ...
```

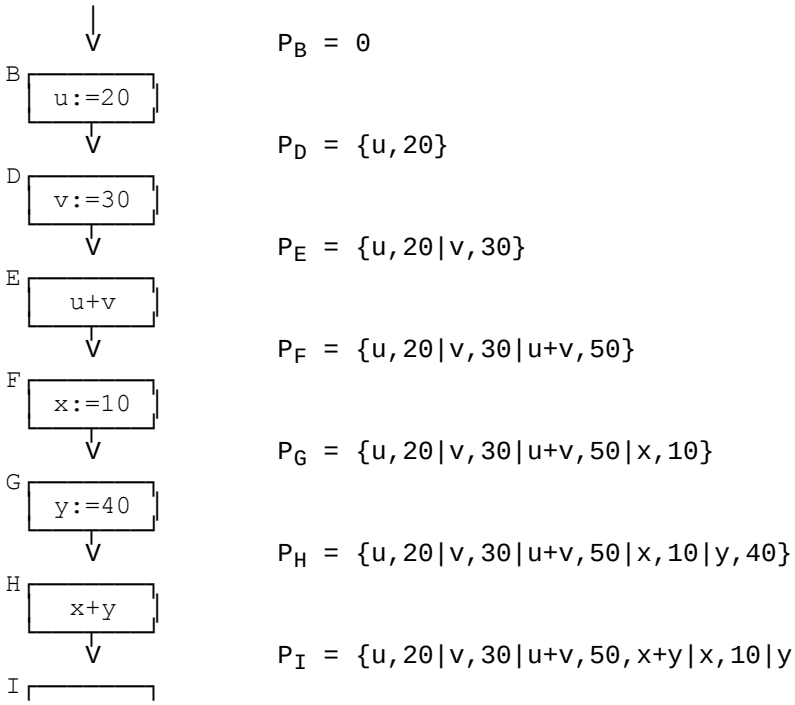


Figure 3. A program graph demonstrating the effects of constant propagation.

Figure 3 shows a program graph representing this segment. Assume the entry pool is empty; i.e., $P_B = 0$. The analysis proceeds up to node E as before, resulting in

$P_E = \{u,20|v,30\}$.

Note that u and v are both propagated constants in P_E since they are both in classes containing constants. If the expression u+v at node E is processed as in f_1, the output pool is

{u,20|v,30|u+v}.

Noting that u and v are in classes with constants, then u+v must be the propagated constant 20+30 = 50. Hence, the constant 50 is placed into the class of u+v in the resulting partition. Thus,

$P_F = \{u,20|v,30|u+v,50\}$.

The analysis continues as before up to node H, resulting in

$P_H = \{u,20|v,30|u+v,50|x,10|y,40\}$.

In the case of the f_1 optimizing function, the expression x+y at node H is placed into a distinct class. The operands x and y, however, are propagated constants since they are equivalent to 10 and 40, respectively. The expression x+y is equivalent to 50 which is already in the partition. Thus, x+y is added to the class of 50, resulting in

$P_I = \{u,20|v,30|u+v,50,x+y|x,10|y,40\}$.

Similarly, the output pool from node I is

{u,20|v,30,y-x|u+v,50,x+y|x,10|y,40}.

The analysis above depends upon the ability to recognize certain expressions as constants, and the ability to compute the constant value of an expression when the operands are all propagated constants. It is also implicit that no two differing constants are in the same class.

An optimizing function f_2 which combines constant propagation with common subexpression elimination can be constructed from f_1 by altering step (3) as follows:

3a. Create a new class in P containing e and add all (partial) computations which occur in the program graph, and which have operands equivalent to those of e (structure the pool as before).

3b. If e does not evaluate to a constant value based upon propagated constant operands, then no further processing is required (same as setp (3) of f_1); otherwise, let z be the constant value of e. If z is already in the partition P, then combine the class of z with the class of e in the resulting partition. If z is not in the partition P, then add z to the class of e. The expression e becomes a propagated constant in either case.

The function f_2 is stated formally and its properties are investigated elsewhere [Ref.33].

Section 6: Expression elimination

Expression optimization, as defined earlier, includes common subexpression elimination, constant propagation, and register optimization. The first two forms of optimization are covered by the f_2 optimizing function; only register optimization needs to be considered. It will be shown below that f_2 also provides a simple form of register optimization.

In general, global register optimization involves the assignment of high-speed registers (accumulators and index registers) throughout a program in such a manner that the number of store-fetch sequences between the high-speed

registers and central memory is minimized. The store-fetch sequences arise in two ways. The first form involves redundant fetches from memory. Consider the sequence of expressions

a:=b+c; d:=a+e;

for example. A straight-forward translation of these statements for a machine with multiple general-purpose registers might be

r_1:=b; r_2:=c; r_1:=r_1+r_2; a:=r_1;
r_1:=a; r_2:=e; r_1:=r_1+r_2; d:=r_1.

Note, however, that the operation r_1:=a is not necessary since r_1 contains the value of the variable a before the operation. McKeeman [Ref.38] discusses a technique called "peephole optimization" which eliminates these redundant fetches within a basic block.

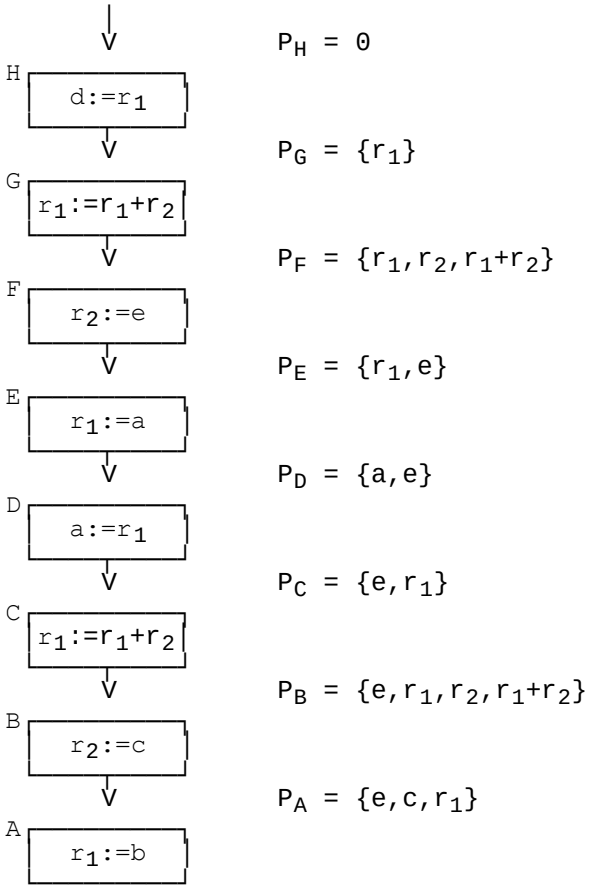


Figure 4. Elimination of redundant register load operations.

Figure 4 shows a program corresponding to the register operations above. The f_2 optimizing function is applied to each successive node in the graph, resulting in the optimizing pools shown in the Figure. In particular, note that

$P_E = \{a, r_1 | b | r_2, c\}$.

The operation at node E assigns the variable a to the register r_1. Since a is already in the class of r_1, however, the operation is redundant and can be eliminated. Hence, the f_2 optimizing function can be used to generalize peephole optimization. Further, the algorithm A extends f_2 to allow global elimination of redundant register load operations.

The second source of store-fetch sequences arises when registers are in use and must be released temporarily for another purpose. The contents of the busy register is stored into a central memory location and restored again at a later point in the program. An optimal register allocation scheme would minimize the number of temporary stores. This form of register optimization has been treated on a local basis, including algorithms which arrange arithmetic computations in order to reduce the total number of registers required in the evaluation [Ref.5,27,36,39,43,45,46]. Global register

allocation has also been formulated as an integer programming problem by Day [Ref.14], given that register interference and cost of data displacement from registers is known. No complete solution to the global register allocation problem is known by the author at this time.

A solution to the global register allocation problem will be aided by the analysis of "live" and "dead" variables at each node in the program graph. A variable v is live at a node N if v could possibly be referenced in an expression subsequent to node N. The variable v is dead otherwise. Recent work has been done by Kennedy [Ref.32] using interval analysis techniques to detect live and dead variables on a global basis.

An optimizing function f_3 can be constructed which produces a set of live expressions at each node in the graph. The detection of live expressions requires the analysis to proceed from the end of the program toward the beginning.

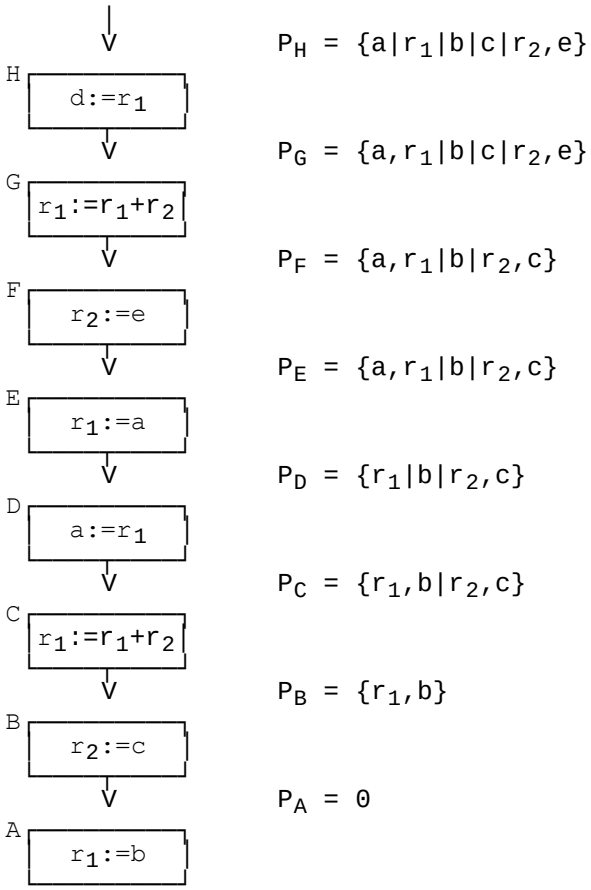


Figure 5. Detection of live expressions in a reversed program graph.

Figure 5 shows the graph of Figure 4 with the direction of the edges reversed. The live expressions at the beginning of the graph correspond to the live expressions at the end of program execution; hence, $P_H = 0$ (there are no live expressions at the end of execution). The expression $d:=r_1$ at node H refers to the expression r_1. Thus, r_1 is live ahead of node H. This fact is recorded by including r_1 in P_G.

$$P_G = \{r_1\}.$$

Since r_1 is assigned a new value at node G, it becomes a dead expression, but, since r_1 is also involved in the expression r_1+r_2, it immediately becomes a live expression again. Thus,

$$P_F = \{r_1, r_2, r_1+r_2\}.$$

The analysis continues, producing the optimizing pools associated with each node in Figure 5. The expressions which are live at node C, for example, are

$$P_B = \{e, r_1, r_2, r_1+r_2\}.$$

The optimizing function $f_3(N,P)$ which provides live expression analysis can be

informally stated as follows:

1. If the expression at node N involves an assignment to a variable, let d be the destination of the assignment; set P ← P - {e|d is a subexpression in e} (d and all expressions containing d become dead expressions).

2. Consider each partial computation e at node N. Set P ← P {e} (e becomes a live expression). The value of $f_3(N,P)$ is the altered value of P.

The algorithm A can then be applied to the reversed program graph using the optimizing function f_3. The exit nodes of the original graph become the entry nodes of the reversed graph. In addition, the meet operation of the algorithm A is the set union operation . The union operation induces the partial ordering given by

$$P_1 \leq P_2 \iff P_1 \quad P_2 = P_1 \iff P_1 \_ P_2, \quad p_1,P_2 \quad \underline{P}',$$

where $\underline{P}$ is the set of (partial) computations which occur in the program graph. Note that $\underline{0} = \underline{P}'$ and $\underline{1} = 0$ in this case. Thus, all initial approximate pools in the algorithm A are set to 0.

There is a simple generalization of detection of live expressions to "minimum distance analysis" where each live expression is accompanied by the minimum distance to an occurrence of the expression. The optimizing pools in this case are sets of ordered pairs (e,d), where e is a live expression and d is the minimum distance (in program steps) to an occurrence of e. The optimizing function extends live expression analysis by tabulating a distance measure as the live expression analysis proceeds. In addition, the meet operation consists of both set union and a comparison of the distances corresponding to each live expression. This minimum distance information can then be used in the register replacement decision: whenever all registers are busy and contain live expressions, the register containing the live expression with the largest distance to its occurrence is displaced.

Examples are given in the section which follows, demonstrating the f_2 and f_3 optimizing functions when used in conjunction with the algorithm A.

Section 7: A tabular form for the algorithm A

Table I

Step	N	$P_N \leftarrow P_N \wedge P_i$	$f(N,P_N)$	L
1			0	/A
2	(A)	0	a,1	/B
3	(B)	a,1	a,1|c,0	/0
4	C	a,1|c,0	a,1|c,0|b,2	/D
5	D	a,1|c,0|b,2	a,1|c,0|b,2|d,a+b,3	/E
6	E	a,1|c,0|b,2|d,a+b,3	a,1|c,0|b,2,e,b+c|d,a+b,3	/F
7	F	a,1|c,0|b,2,e,b+c|d,a+b,3	a,1|b,2,e|d,a+b,3|c,4	/0
8	(C)	a,1	a,1|b,2	/D
9	(D)	a,1|b,2	a,1|b,2	/E
10	(E)	a,1|b,2|d,a+b,3	a,1|b,2|d,a+b,3|b+c,e	/F
11	(F)	a,1|b,2	a,1|b,2|c,4	/0

The processing of the algorithm A can be expressed in a tabular form. The tabular form allows presentation of a number of examples, and provides an intuitive basis for implementing the optimizing techniques. In particular, this form allows representation of the approximate optimizing pools at each node, the elements of L, and the node traversing decision. As shown in Table I, the column labeled "N" contains the current node being processed (i.e., the N in $L' = (N,P_i)$ in step A5). The column labeled "$P_N \leftarrow P_N \wedge P_i$" shows the change in the approximate pool at node N when the node is traversed in step A5. The column marked "$f(N,P_N)$" contains the output optimizing pool produced by traversing the node N (the set braces are omitted for convenience of notation). The last column, marked "L", represents the set of nodes remaining to be processed (the set L of the algorithm A).

Paraphrasing the algorithm A, the tabular form is processed as follows.

1. List all entry nodes and entry pools vertically in the right-hand columns, with entry node e_i in column L, and associated entry pool x_i in column $f(N,P_N)$. Normally, there is only one entry node, with the

null set as an entry pool.

2. Select an L' from L as follows. Choose any node from column L, say node N. If there are no elements remaining in L then the algorithm halts. The line where N was added to L contains the associated output pool P_i in the column $f(N,P_N)$. Eliminate L' from L by crossing out N from column L.

3. Using L' = (N,P_i) from step 2, scan the table from the bottom upward to the first occurrence of node N in column N. The current approximate pool P_N is adjacent in the column $P_N \leftarrow P_N \wedge P_i$. If node N has not appeared in column N, then assume the first approximation to $P_N = \underline{1}$ (and hence, $P_N \leftarrow \underline{1} \wedge P_i = P_i$).

4. If $P_N \leq P_i$ then go to step 2. Otherwise, write the node name N in column N, and the value of the new approximate pool determined by $P_N \wedge P_i$ in the column marked $P_N \leftarrow P_N \wedge P_i$. Compute the output pool based upon the new approximate pool P_N in the column $f(N,P_N)$, and write the names of the immediate successors of N in column L. Go back to step 2.

Upon termination of this algorithm, the table is scanned from bottom to top; the first occurrence of each node N N is circled (ROCHE> Surrounded with "( )". Example: "(A)".). The pool associated with each circled node in column $P_N \leftarrow P_N \wedge P_i$ is the final pool for that node. Any nodes of N which do not appear in column N cannot be reached from an entry node, and can be eliminated from the program graph.

Table I shows the analysis of the program graph given in Figure 1, using the f_2 optimizing function. The entry node set for this analysis is _ = {(A,0)}, as before. L is treated as a stack; elements are removed from the lower right position of column L in step 2. After processing the graph, the final pools at each node are listed in the table opposite the circled nodes. The final pool at node E, for example, is

$P_E = \{a,1|b,2|d,a+b,3\}$.

The final pools determined by the algorithm correspond to those determined previously in Section 2.

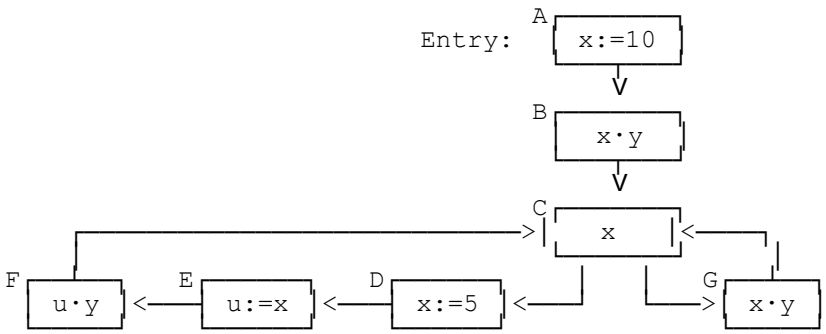


Figure 6. A program graph with two parallel feedback loops.

Table II

Step	N	$P_N \leftarrow P_N \wedge P_i$	$f(N,P_N)$	L
1			0	/A
2	(A)	0	x,10	/B
3	(B)	x,10	x,10|y|x·y	/C
4	C	x,10|y|x·y	x,10|y|x·y	/D,/G
5	G	x,10|y|x·y	x,10|y|x·y	/C
6	D	x,10|y|x·y	10|y|x·y|x,5	/E
7	(E)	10|y|x,5	10|y|x,5,u	/F
8	(F)	10|y|x,5,u	10|y|x,5,u|u·y,x·y	/C
9	(C)	x|10|y|x·y	x|10|y|x·y	/D,/G
10	(G)	x|10|y|x·y	x|10|y|x·y	/C
11	(D)	x|10|y|x·y	x,5|10|y	/E

Figure 6 shows a program graph with two parallel feedback loops. The analysis of this program is given in Table II, using the f_2 optimizing function. Note that in step 8,

P_F = {10|y|x,5,u}.

Applying $f_2(F,P_F)$, the resulting output pool is

{10|y|x,5,u|u·y,x·y}.

The expression x·y is placed into the class of u·y when the partition is structured. That is, x·y is an expression which occurs in the program, and x·y is operand equivalent to u·y. Thus, x·y must be added to the class of u·y in the output pool. The redundant expression x·y is detected at node G since the final pool P_G contains x·y.

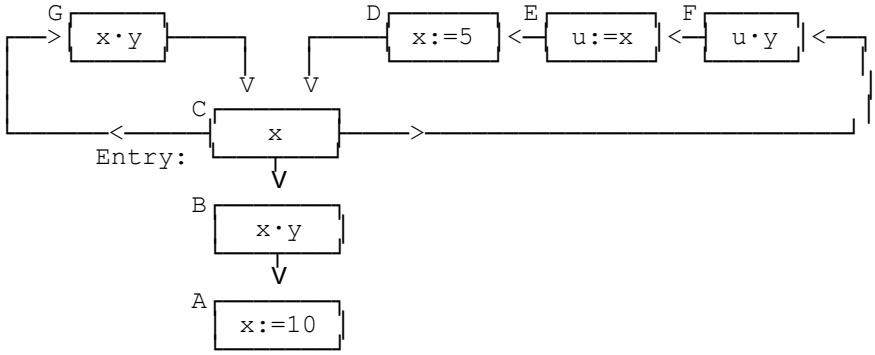


Figure 7. The reversed graph corresponding to the program graph of Figure 6.

Table III

Step	N	$P_N \leftarrow P_N \wedge P_i$	$f(N,P_N)$	L
1			0	/C
2	C	0	x	/B,/F,/G
3	G	x	x,y,x·y	/C
4	(C)	x,y,x·y	x,y,x·y	/B,/F,/G
5	(G)	x,y,x·y	x,y,x·y	/C
6	(F)	x,y,x·y	x,y,x·y,u,u·y	/E
7	(E)	x,y,x·y,u,u·y	x,y,x·y	/D
8	(D)	x,y,x·y	y	/C
9	(B)	x,y,x·y	x,y,x·y	/A
10	(A)	x,y,x·y	y,10	

Global live expressions analysis can be performed on the program graph of Figure 6 by reversing the graph, as shown in Figure 7. Given that node C is the exit node of the original graph, node C becomes the entry node of the reversed graph. Thus, _ = {(C,0)} in the analysis shown in Table III, using the f_3 optimizing function. For example, the final pool

P_A = {x,y,x·y}

indicates that the expression x, y, and x·y are live immediately following node A in the original graph.

This tabular form can be used for processing any program graph using an optimizing function which satisfies the conditions of the algorithm A.

Section 8: Implementation notes

Implementation of the above optimizing techniques in a practical compiler is considered below. In particular, the optimizer operates upon an intermediate form of the program, such as tree structures or Polish [Ref.24], augmented by branching information. The control flow analyzer accepts the intermediate form and calls the various optimizing functions to process each basic block, roughly paralleling the tabular form given previously. A single stack can be used to list uninvestigated basic blocks, corresponding to "L" of the tabular form. Pool information must be maintained for each basic block corresponding to the "$P_N \leftarrow P_N \wedge P_i$" column, but may be discarded and replaced if the node is encountered again in the analysis (i.e., the node reappears in column "N"). The output optimizing pools found in columns "$f(N,P_N)$", however, can be intersected with all immediate successors as they are produced, and thus need not be maintained during analysis. The final optimizing pools (determined by "scanning" the tabular form) are simply the current pools attached to each

basic block.

The optimizing functions and corresponding meet operations are generally simple to implement using bit strings for sets, and lists for ordered pairs. Common subexpression elimination, however, requires further consideration since direct representation and manipulation of structured partitions is particularly unwieldy.

One approach to handling structured partitions allows direct representation of the classes, but limits the number of expressions which appear. A list of all (sub)expressions is constructed by prescanning the program (an optimizing function which always returns 0 is useful for this scan). When a partition is structured, only those expressions which occur in the expression list are included. The set of eligible expressions can be further reduced by first performing live expression analysis. The expressions which appear in a partition are limited to the live expressions at the point the partition is generated. The use of live expression analysis before common subexpression elimination will generally reduce partition size and improve the convergence rate of the analysis algorithm.

A second approach to representation of structured partitions involves the assignment of "value numbers" to the expressions in the optimizing pools [Ref.13,24,33,34]. A value number is a unique integer assigned to all elements of the same class. The sequence of statements

```
a:=b+c; d:=b; e:=a;
```

results in the structured partition

P_1 = {b,d|c|b+c,d+c,a,e}.

Next, assign the value numbers 1, 2, and 3 to the three classes, and replace the expressions b+c and d+c by (1)+(2), representing the addition of elements of class (1) and class (2). P_1 can now be written as

```
P2 = { b,d| c |(1)+(2),a,e }.
       (1) (2)     (3)
```

Similarly, the sequence of assignments

```
a:=d; b:=c; e:=b+c;
```

produces the structured partition represented by

```
P2 = { a,d| b,c |(5)+(5),e }.
       (4)  (5)     (6)
```

which expands to

P_2 = {a,d|b,c|b+c,b+b,c+b,c+c,e}.

Thus, the assignment of value numbers provides a data structure whose size is linear in the number of expressions in the basic block. In addition, the value number representation is particularly easy to construct and use in the detection of common subexpressions.

Given two partitions P_1 and P_2 in value number form, the meet operation P = P_1 P_2 can be iteratively computed. The computation proceeds as follows. Construct a list C consisting of the number of occurrences of each value number in P_1. The elements of C thus provide a count of the number of elements in each class of P_1. This count is decremented whenever an element of the class is processed, until the count goes to zero indicating the entire class is exhausted.

A list R is also maintained, which gives a mapping of the class numbers in P_1 and P_2 to the resulting class numbers in P. The elements of R are of the form $r(r_1,r_2)$, indicating that value number r_1 from P_1 and value number r_2 from P_2 map to value number r in the resulting partition P. R is built during the construction of P.

The elements of P_1 are scanned and processed until the classes of P_1 are exhausted. Suppose q is an identifier in P_1 with value number v_1. The count corresponding to v_1 in the list C is first decremented. If q does not occur in P_2 then the next element of P_1 is selected. Otherwise, let v_2 be the value number corresponding to q in P_2. R is scanned for an element $v(v_1,v_2)$; if not found, a new value number v is assigned, and $v(v_1,v_2)$ is added to R. The

identifier q is placed into P with value number v.

If the element selected from P_1 is not an identifier, then it is an expression of the form $(n_1)\ \Theta\ (m_1)$ with value number v_1, where n_1 and m_1 are value numbers in P_1 (assuming all operations Θ are binary). If the count of either class (n_1) or (m_1) is non-zero in C, defray the processing of this expression; otherwise, decrement the count for class (v_1) in C, as above. Examine R for pairs of elements $n(n_1,n_2)$ and $m(m_1,m_2)$ where n_2 and m_2 are value numbers in P_2. For each such pair, search P_2 for an entry $(n_2)\ \Theta\ (m_2)$. If found, let v_2 be the value number of this matched expression. Scan R for an element of the form $v(v_1,v_2)$, and make a new entry if not found, as above. The expression $(n)\ \Theta\ (m)$ with value number v is then placed into the intersection P.

As an example, consider the class intersection of the partitions P_1 and P_2 given previously. These partitions are represented by the value number tables

P_1		P_2	
exp	val#	exp	val#
b	(1)	a	(4)
d	(1)	d	(4)
c	(2)	b	(5)
(1)+(2)	(3)	c	(5)
a	(3)	(5)+(5)	(6)
e	(3)	e	(6)

The class count list C for the partition P_1 is initially

val#	count
(1)	2
(2)	1
(3)	3

The identifiers b, d, and c are processed first, reducing the class counts for (1) and (2) to zero in C. The class mapping list at this point is

R = {7(1,5),8(1,4),9(2,5)}.

The identifiers b, d, and c are placed into P with value numbers 7, 8, and 9, respectively. The expression (1)+(2) with value number (3) is then processed from P_1, since the class counts for both (1) and (2) are zero. Based upon the mappings in R, P_2 is searched for an occurrence of (5)+(5) or (4)+(5). Since (5)+(5) occurs in P_2 with value number (6), R is scanned for an element of the form v(3,6), and, since no such element is found, 10(3,6) is added to R. The expression (7)+(9) with value number (10) is included in P. The identifier a is then processed, resulting in another mapping 11(3,4) in R; a is added to P with value number (11). Finally, the identifier e from P_1 with value number (3) is processed. A match is found in P_2 with value number (6). Since the element 10(3,6) is already in R, e is added to P with value number (10). The final value of the class list is

R = {7(1,5),8(1,4),9(2,5),10(3,6),11(3,4)}

which can now be discarded. The value of the resulting partition P is

exp	val#
b	(7)
d	(8)
c	(9)
(7)+(9)	(9)
a	(11)
e	(10)

which represents the structured partition

{b|d|c|b+c,e|a}.

Note that the predicate $P_2 \geq P_1$ is easily computed during this process.

The control flow analysis algorithm has been implemented as a general-purpose optimizing module, including several optimizing functions. The implementation is described in some detail elsewhere [Ref.33].

Section 9: Conclusions

An algorithm has been presented which, in conjunction with various optimizing functions, provides global program optimization. Optimizing functions have been described which provide constant propagation, common subexpression elimination, and a degree of register optimization.

The functions which have been given by no means exhaust those which are useful for optimization. Simplifying formal identities such as 0+x = 0+x = x can be incorporated to further coalesce equivalence classes at each application of the f_2 optimizing function. In addition, it may be possible to develop functions which extend live expression analysis to completely solve the global register allocation problem.

References

1
Aho, A., Sethi, R., and Ullman, J.
"A formal approach to code optimization"
Proceedings of a Symposium on Compiler Optimization
University of Illinois at Urbana-Champaign, July, 1970.

2
Allen, F.
"Program optimization"
in "Annual Review in Automatic Programming"
Pergamon Press, No.5 (1969), pp.239-307.

3
Allen, F.
"A basis for program optimization"
IFIP Congress 71, Ljubljana, August 1971, pp.64-68.

4
Allen, f.
"Control flow analysis"
Proceedings of a Symposium on Compiler Optimization
University of Illinois at Urbana-Champaign, July, 1970.

5
Anderson, J.
"A note on some compiling algorithms"
Comm. ACM, Vol.7, No.3 (March 1964), pp.149-150.

6
Arden, B., Galler, B., and Graham, R.
"An algorithm for translating boolean expressions"
Jour. ACM, Vol.9, No.2 (April 1962), pp.222-239.

7
Bachmann, P.
"A contribution to the problem of the optimization of programs"
IFIP Congress 71, Ljubljana, August 1971, pp.74-78.

8
Ballard, A., and Tsichritzis, D.
"Transformations of programs"
IFIP Congress 71, Ljubljana, August 1971, pp.89-93.

9
Breuer, M.
"Generation of optimal code for expressions via factorization"
Comm. ACM, Vol.12, No.6 (June 1970), pp.333-340.

10
Busam, V., and Englund, D.
"Optimization of expressions in FORTRAN"
Comm. ACM, Vol.12, No.12 (December 1969), pp.666-674.

11
Cocke, J.
"Global common subexpression elimination"
Proceedings of a Symposium on Compiler Optimization
University of Illinois at Urbana-Champaign, July, 1970.

12
Cocke, J., and Miller, R.

"Some analysis techniques for optimizing computer programs"
Proceedings of the Second International Conference of Systems Sciences,
Hawaii, January 1969, pp.143-146.

13
Cocke, J., and Schwartz, J.
"Programming languages and their compilers: preliminary notes"
Courant Institute of Mathematics Sciences,
New York University, 1970.

14
Day, W.
"Compiler assignment of data items to registers"
IBM Systems Journal, Vol.8, No.4 (1970), pp.281-317.

15
Earnest, C., Blake, K., and Anderson, J.
"Analysis of graphs by ordering nodes"
Jour. ACM, Vol.19, No.1 (January 1972), pp.23-42.

16
Elson, M., and Rake, S.
"Code generation technique for large language compilers"
IBM Systems Journal, Vol.8, No.3 (1970), pp.166-188.

17
Fateman, R.
"Optimal code for serial and parallel computation"
Comm. ACM, Vol.12, No.12 (December 1969), pp.694-695.

18
Finkelstein, M.
"A compiler optimization technique"
The Computer Review, February 1968, pp.22-25.

19
Floyd, R.
"An algorithm for coding efficient arithmetic operations"
Comm. ACM, Vol.4, No.1 (January 1961), pp.42-51.

20
Frailey, D.
"Expression optimization using unary complement operators"
Proceedings of a Symposium on Compiler Optimization
University of Illinois at Urbana-Champaign, July, 1970.

21
Frailey, D.
"A study of optimization using a general-purpose optimizer"
(Ph.D. Thesis), Purdue University, Lafayette, Ind., January 1971.

22
Freiburghouse, R.
"The MULTICS PL/I Compiler"
AFIPS Conf. Proc. FJCC, (1969), pp.187-199.

23
Gear, C.
"High-speed compilation of efficient object code"
Comm. ACM, Vol.8, No.8 (August 1965), pp.483-488.

24
Gries, D.
"Compiler construction for digital computers"
John Wiley and Sons, Inc., New York, 1971.

25
Hill, V., Langmaack, H, Schwartz, H., and Seegumuller, G.
"Efficient handling of subscripted variables in ALGOL-60 compilers"
Proc. Symbolic Languages in Data Processing,
Gordon and Breach, New York, 1962, pp.331-340.

26
Hopkins, M.
"An optimizing compiler design"
IFIP Congress 71, Ljubljana, August 1971, pp.69-73.

27
Horowitz, L., Karp, R., Miller, R., and Winograd, S.
"Index register allocation"
Jour. ACM, Vol.13, No.1 (January 1966), pp.43-61.

28
Huskey, H., and Wattenberg, W.
"Compiling techniques for boolean expressions and conditional statements in ALGOL-60"
Comm. ACM, Vol.4, No.1 (Janauary 1961), pp.70-75.

29
Huskey, H.
"Compiling techniques for algebraic expressions"
Computer Journal, Vol.4, No.4 (April 1961), pp.10-19.

30
Huxtable, D.
"On writing an optimizing translator for ALGOL-60"
in "Introduction to System Programming"
Academic Press, Inc., New York, 1964.

31
IBM System/360 Operating System, FORTRAN IV (G and H) Programmer's Guide
C28-6817-1, International Business Machines, 1967, pp.174-179.

32
Kennedy, K.
"A global flow analysis algorithm"
International Journal of Computer Mathematics,
Section A, Vol.3, 1971, pp.5-15.

33
Kildall, G.
"Global expression optimization during compilation"
Technical Report No.TR# 72-06-02,
University of Washington Computer Science Group,
University of Washington, Seattle, Washington, June 1972.

34
Kildall, G.
"A code synthesis filter for basic block optimization"
Technical Report No. TR# 72-01-01,
University of Washington Computer Science Group,
University of Washington, Seattle, Washington, January 1972.

35
Lowry, E., and Medlock, C.
"Object code optimization"
Comm. ACM, Vol.12, No.1 (January 1969), pp.13-22.

36
Luccio, F.
"A comment on index register allocation"
Comm. ACM, Vol.10, No.9 (September 1967), pp.572-574.

37
Maurer, W.
"Programming: An introduction to computer language technique"
Holden-Day, San Francisco, 1968, pp.202-203.

38
McKeeman, W.
"Peephole optimization"
Comm. ACM, Vol.8, No.7 (July 1965), pp.443-444.

39
Nakata, I.
"On compiling algorithms for arithmetic expressions"
Comm. ACM, Vol.19, No.8 (August 1967), pp.492-494.

40
Nievergelt, J.
"On the automatic simplification of computer programs"
Comm. ACM, Vol.8, No.6 (June 1965), pp.366-370.

41
Painter, J.
"Compiler effectiveness"
Proceedings of a Symposium on Compiler Optimization
University of Illinois at Urbana-Champaign, July, 1970.

42
Randell, B., and Russell, L.
"ALGOL-60 Implementation"

Academic Press, Inc., New York, 1964.

43
Redziejowski, R.
"On arithmetic expressions and trees"
Comm. ACM, Vol.12, No.2 (February 1969), pp.81-84.

44
Ryan, J.
"A direction-independent algorithm for determining the forward and backward
compute points for a term or subscript during compilation"
Computer Journal, Vol.9, No.2 (August 1966), pp.157-160.

45
Schnieder, V.
"On the number of registers needed to evaluate arithmetic expressions"
BIT, No.11, (1971), pp.84-93.

46
Sethi, R., and Ullman, J.
"The generation of optimal code for arithmetic expressions"
Jour. ACM, Vol.17, No.4 (October 1970), pp.715-728.

47
Wagner, R.
"Some techniques for algebraic optimization with application to matrix
arithmetic expressions"
Thesis, Carnegie-Mellon University, June 1968.

48
Yershov, A.
"On programming of arithmetic operations"
Comm. ACM, Vol.1, No.8 (August 1958), pp.3-6.

49
Yershov, A.
"ALPHA: An automatic programming system of high efficiency"
Jour. ACM, Vol.13, No.1 (January 1966), pp.17-24.

Appendix A: Analysis

1 A1: $L = \{(A,0)\}$

2 A3: $L' = (A,0)$, $L = 0$

3 A4: $P_N = P_A = \underline{1}$, $P_i = 0$, $P_A \quad P_i$, $P_A \leftarrow P_A \wedge P_i = P_i = 0$

4 A5: $P_A = 0$, $L = \{(B,\{(a,1)\})\}$

5 A3: $L' = (B,\{(a,1)\})$, $L = 0$

6 A5: $P_B = \{(a,1)\}$, $L = \{(C,\{(a,1),(c,0)\})\}$

7 A3: $L' = (C,\{(a,1),(c,0)\})$, $L = 0$

8 A5: $P_C = \{(a,1),(c,0)\}$, $L = \{(D,\{(a,1),(c,0),(b,2)\})\}$

9 A3: $L' = (D,\{(a,1),(c,0),b,2)\})$, $L = 0$

10 A5: $P_D = \{(a,1),(c,0),(b,2)\}$, $L = \{(E,\{(a,1),(c,0),(b,2),(d,3)\})\}$

11 A3: $L' = (E,\{(a,1),(c,0),(b,2),(d,3)\})$, $L = 0$

12 A5: $P_E = \{(a,1),(c,0),(b,2),(d,3)\}$,
$L = \{(F,\{(a,1),(c,0),(b,2),(d,3),(e,2)\})\}$

13 A3: $L' = (F,\{(a,1),(c,0),(b,2),(d,3),(e,2)\})$, $L = 0$

14 A5: $P_F = \{(a,1),(c,0),(b,2),(d,3),(e,2)\}$,
$L = \{(C,\{(a,1),(c,4),(b,2),(d,3),(e,2)\})\}$

15 A3: $L' = (C,\{(a,1),(c,4),(b,2),(d,3),(e,2)\})$, $L = 0$

16 A5: $P_C = \{(a,1)$, $L = \{(D,\{(a,1),(b,2)\})\}$

17 A3: $L' = (D,\{(a,1),(b,2)])\}$, $L = 0$

18 A5: P_D = {(a,1),(b,2)}, L = {(E,{(a,1),(b,2),(d,3)})}

19 A3: L' = (E,{(a,1),(b,2),(d,3)}), L = 0

20 A5: P_E = {(a,1),(b,2),(d,3)}, L = {(F,{(a,1),(b,2),(d,3)})}

21 A3: L' = (F,{(a,1),(b,2),(d,3)}), L = 0

22 A5: P_F = {(a,1),(b,2),(d,3)}, L = {(C,{(a,1),(b,2),(d,3),(c,4)})}

23 A3: L' = (C,{(a,1),(b,2),(d,3),(c,4)}), halt.

Appendix B: Proof

The proof of Theorem 2 is given below. First note that, given a program graph G with multiple entry nodes, an augmented graph G' can be constructed with only one entry node with entry pool $\underline{0}$. The construction is as follows. Let = $\{e_1,e_2,...,e_k\}$ be the entry node set and _ = $\{(e_1,x_1),(e_2,x_2),...,(e_k,x_k)\}$ be the entry pool set corresponding to a particular analysis. Consider the augmented graph G' = <$\underline{N}$',$\underline{E}$'> where $\underline{N}$' = $\underline{N}$ $\{v,v_1,...,v_k\}$ v,v_i $\underline{N}$ i, 1 ≤ i ≤ k, and $\underline{E}$' = $\underline{E}$ $\{(v,v_1),(v,v_2),...,(v,v_k),(v_1,e_1),...(v_k,e_k)\}$.

The augmented graph G' has a single entry node v and entry node set ' = {v}. The functional value of f is defined for these nodes as

$f(v,P) = \underline{0}$ P $\underline{P}$,

and

$f(v_i,P) = x_i$ P $\underline{P}$, 1 ≤ i ≤ k.

Hence, the analysis proceeds as if there is only a single entry node with entry pool $\underline{0}$; i.e., _' = {(v,$\underline{0}$)}.

Lemma 1

If $f(N,P_1 \wedge P_2) = f(N,P_1) \wedge f(N,P_2)$ then $P_1 \leq P_2 \Rightarrow f(N,P_1) \leq f(N,P_2)$, N $\underline{N}$, P_1,P_2 $\underline{P}$.

Proof

The proof is immediate since $P_1 \leq P_2 \Rightarrow f(N,P_1 \wedge P_2) = (f(N,P_1) \wedge f(N,P_2)) \Rightarrow f(N,P_1) \leq f(N,P_2)$.

Lemma 2

Let X _ $\underline{P}$, if $f(N,P_1 \wedge P_2) = f(N,P_1) \wedge f(N,P_2)$ N $\underline{N}$, P_1,P_2 $\underline{P}$ then

$$f(N,/\backslash x) = /\backslash f(N,x).$$
$$x\ X \qquad x\ X$$

Proof

The proof proceeds by induction on the cardinality of X, denoted by Ć (X). If Ć(X) = 1 then $f(N,/\backslash x) = f(N,x)$ and the lemma is trivially true.
$x\ X$
If Ć(X) = k, k > 1, assume lemma is true for all X' Ć(X') < k. Let y X and X' = X - {y}.

$$f(N,/\backslash x) = f(N,y \wedge (/\backslash x)) = f(N,y) \wedge f(N,/\backslash x) =$$
$$x\ X \qquad x\ X \qquad x\ X$$

$$f(N,y) \wedge (/\backslash,f(N,x)) = /\backslash f(N,x)$$
$$x\ X \qquad x\ X$$

Proof of Theorem 2

It will first be shown by induction on the path length that

$P_N \leq X_N$ N $\underline{N}$.

Consider the following proposition on n:

$P_N \leq f(p_n,f(p_{n-1},...,f(p_1,\underline{0}))...)$ for all final pools P_N and paths of length n from the entry node p_1 with entry pool $\underline{0}$ to node N, N $\underline{N}$.

The trivial case is easily proved. The only node which can be reached by a path of length 0 from the entry node p_1 is p_1 itself. Hence, it is only necessary to show that $p_{p1} \leq \underline{0}$. This is immediate, however, since $(p_1,\underline{0})$ is initially placed into L in step A1, and extracted in step A3 as $L' = (p_1,\underline{0})$. But, p_{p1} is initially $\underline{1}$, and hence p_{p1} $p_i = \underline{0}$ in step A4. Thus, $p_{p1} \leftarrow p_{p1}$ ^ $\underline{0} = \underline{0}$ in step A5. Thus, it follows that $p_{p1} = \underline{0} \leq \underline{0}$.

Suppose the proposition is true for all $n < k$, for $k > 0$. That is, $P_N \leq f(p_n,...,f(p_1,\underline{0}))...)$ for all paths of length less than k from p_1 to node N, for each node N $\underline{N}$.

Let K $\underline{N}$ a path $(P_1,...,p_k,K)$ of length k. It will be shown that $P_K \leq f(p_k,f(p_{k-1},...,f(p_1,\underline{0}))...)$.

Consider each immediate predecessor in $I^{-1}(K)$. Let p_k denote one such predecessor, and let $T = f(p_{k-1},...,f(p_1,\underline{0}))...)$. By inductive hypothesis, $p_{pk} \leq T$. It will be shown that P_K $f(p_k,T)$.

Since P_{pk} is the final approximation to the pool at p_k, $(K,f(p_k,P_{pk}))$ must have been added to L in step A5. But, $P_{pk} \leq T \Rightarrow f(p_k,P_{pk}) \leq f(p_k,T)$ by Lemma 1. The pair $(K,f(p_k,P_{pk}))$ must be processed in step A3 before the algorithm halts. Thus, either $P_K \leq f(p_k,P_{pk})$ in step A4, or $P_K \leftarrow P_K \wedge f(p_k,P_{pk})$.

In either case, $P_K \leq f(p_k,P_{PK})$. But

$P_K \leq f(p_k,P_{pk}) \leq f(p_k,T) \Rightarrow P_K \leq f(p_k,T)$

$\Rightarrow P_K \leq f(p_k,f(p_{k-1},...,f(p_1,\underline{0}))...)$.

Thus, since the proposition holds for paths of length k, it follows by induction that the proposition is true for all paths from the entry node to node N, for all N $\underline{N}$.

The following claim will be proved in order to show that $X_N \leq P_N$ for all N $\underline{N}$: at any point in the processing of G by the algorithm A, either N has not been encountered in step A5, or $X_N \leq P_N$, where P_N is the current approximate pool associated with node N, for all N $\underline{N}$. The proof proceeds by induction on the number of times step A5 has been executed. Suppose step A5 has been executed only once. Then $L' = (p_1,\underline{0})$ and the only node encountered in step A5 is the entry node p_1. The entry pool $\underline{0}$ corresponds to a path of length zero from p_1 to p_1. Thus, $\underline{0}$ $F_{p1} \Rightarrow X_{p1} = \underline{0}$ and the proposition is trivially true since $X_{p1} = \underline{0} \leq P_{p1} = \underline{0}$.

Suppose that either N has not been encountered in step A5, or $X_N \leq P_N$ N $\underline{N}$ when step A5 has been executed $n < k$ times, $k > 1$. Consider the k^{th} execution of step A5. Let $L' = (N,T)$ where $T = f(N',P_{N'})$ for some N' $I^{-1}(N)$. The pair (N,T) was added to L when the node N' was processed in the n^{th} execution of step A5, for $n < k$. Hence, $X_{N'} \leq P_{N'}$, by inductive hypothesis. But, using Lemma 2,

$$X_N \leq \bigwedge_{\substack{\text{paths} \\ (p_1,...,p_t,N',N)}} f(N',f(p_t,...,f(p_1,\underline{0}))...) =$$

$$f(N',\bigwedge_{\substack{\text{paths} \\ (p_1,...,p_t,N',N)}} (p_t,f(p_{t-1},...,f(p_1,\underline{0}))...) =$$

$f(N',X_{N'})$.

$X_{N'} \leq P_{N'}$, and thus $X_N \leq f(N',X_{N'}) \Rightarrow X_N \leq f(N',P_{N'}) = T$, using Lemma 1.

If this step is the first occurrence of node N in A5, then $P_N \leftarrow \underline{1} \wedge T = T$ since $f(N',P) \neq \underline{1}$ for any $N' \in \underline{N}$, $P \in \underline{P}$. In this case, $X_N \leq P_N = T$ after step A5. Otherwise, suppose this is not the first occurrence of node N in step A5. $X_N \leq P_N$ and $X_N \leq T \Rightarrow X_N \leq P_N \wedge T \Rightarrow X_N \leq P_N \leftarrow P_N \wedge T$ after step A5 is executed. Hence, the proposition holds for each execution of step A5. In particular, $X_N \leq P_N$ $N \in \underline{N}$ upon termination of the algorithm A. Hence, the theorem is proved since

$$P_N \leq X_N \text{ and } X_N \leq P_N \Rightarrow X_N = P_N \quad N \in \underline{N}.$$

EOF

GKHLL.WS4

- "High-level language simplifies microcomputer programming"
  Gary Kildall
  "Electronics", June 27, 1974, pp.103-109

(Retyped by Emmanuel ROCHE.)

Abstract: Just as FORTRAN and BASIC sharply reduce the time and effort required to program large computers, so Intel's PL/M eases the programming of systems based on LSI microprocessors; here are step-by-step directions.

The microcomputer is being applied to more and more tasks that are not economically feasible for a minicomputer, with its larger instruction set and higher speed and cost. Although the microprocessor is slower than the central processor of a minicomputer, it can easily perform many tasks that are complex enough to require extensive digital processing.

What's more, microprocessors, which serve as central processors of microcomputers and are generally made with MOS large-scale integration, are constantly attaining higher speeds and higher circuit density per chip. As the capabilities of microcomputers are being ever extended, programming aids are being developed to simplify their use, while minimizing design and development time. These aids sometimes require use of a larger computer; when this is the case, they can be used either on commercial time-sharing networks or on a user's own large in-house computer.

The microcomputer may be viewed as a ROM-driven LSI logic chip because the microcomputer can execute complicated sequences of instructions stored in an external memory. Thus, the microcomputer chip connected to a read-only memory containing the proper data can appear to be a single custom chip. In this way, the system designer can substitute microcomputer programming for traditional hard-wired logic design or custom chip fabrication, gaining advantages in reduced development time, ease of design change, and reduced production costs.

The application of microcomputers points up the common ground between software and hardware designers. While software-system designers can use microcomputers most effectively when they are aware of the hardware environment, the hardware designer is well advised to learn the basic techniques of the programmer.

These techniques include how to use assemblers, compilers, and processor simulators, which are effective tools in developing and debugging large and small microcomputer programs. This article introduces these programming tools to the hardware designer and specifically examines the advantages of the PL/M language, which make possible rapid design of systems around the MCS-8 microcomputer, made by Intel Corp.

The MCS-8 is based on the 8008 microprocessor, one of a new class of devices being offered by several manufacturers as a result of recent advances in semiconductor electronics. The PL/M programming aid is a good example of the

service that these manufacturers can offer to simplify the use of their products.

Minimizing software costs

Like other programming tools, the PL/M approach automates the production of programs to counteract the rapidly increasing cost of software production at a time when hardware costs are decreasing. And, in addition to rapid production turnaround, the programs can be fully checked out early in the design process. What's more, the self-documentation of PL/M programs enables one programmer to readily understand the work of another, which dramatically reduces program-maintenance costs and provides transportability of software between programmers and to other Intel processors as they are introduced.

Additional cost reductions will also result from standardization of parts and modules, and alterability of the final program often outweighs benefits of random-logic designs or custom-chip fabrication.

The PL/M compiler, which is another program, translates the PL/M program into machine language. This compiler, which can be run on a medium- or large-scale computer, is available from several nationwide time-sharing services.

Last but not least, PL/M programs can be recompiled as improved optimizing versions of the compiler are released, as Intel has recently done. A recent revision of the PL/M compiler, for example, makes possible reduction of generated code by about 15%.

Although PL/M requires a cross-compiler -- one that runs only on a larger machine -- a resident compiler that uses the microcomputer itself to produce its programs is technically feasible with the advanced state of micro-computer development and today's inexpensive peripherals. Such a compiler would require several passes to reduce a PL/M source program to machine language, using the developmental system itself, and eliminating the need for large-system support.

A program for the Intel 8008 microprocessor is a sequence of instructions from its normal instruction set (see "Hardware for PL/M" box, at the end of this article) that performs a particular task. Given no programming aids, the designer must determine the machine codes that represent each of the instructions in his program and store these codes into program memory. This approach to programming quickly becomes unwieldy in all but the most trivial projects.

Nearly all manufacturers of microprocessors (and mini- and maxicomputers as well) provide symbolic assemblers -- programs that ease the programming task by eliminating the need to translate instructions manually into machine-readable form. The designer can express his program in terms of mnemonics, which are abbreviations that suggest individual instructions. Then the assembler translates each mnemonic instruction into its binary representation.

Symbolic addresses

In addition, the programmer can refer to memory locations by symbolic name, rather than actual numeric address; the assembler translates these, as well as the instructions. The assembler usually runs on a larger computer, although both Intel Corp. and National Semiconductor Corp. have assemblers that run directly on their microcomputer-based development systems, and symbolic programs for Rockwell microcomputers can be assembled on a machine built around that unit by Applied Computer Technology Inc. The assembler requires significantly less development and check-out time than manual translation, and there are fewer coding errors.

Assembly-language programming, however, is necessarily closely related to the machine architecture because instructions in symbolic code have a one-to-one correspondence with those in machine code. As a result, the programmer must spend much more time keeping track of the location of data elements and proper register usage than actually conceptualizing the solution to his problem.

On large-scale computers, high-level languages have been developed to provide important facilities independently of particular machine architectures, while eliminating the trivialities of assembly languages. These facilities include program-control structures, data types, and primitive operations suitable for concise expressions of programs in particular problem environments. For example, a problem environment may be one of numerical computation, in which application-oriented programming languages like BASIC and FORTRAN are appropriate. Or the environment may be the control of a particular class of computer and all its functions, for which system languages, which are necessarily closely related to the machine architecture, are useful.

In a system language, program statements generally correspond directly with machine-level instructions, and conversely, every machine operation is reflected in a high-level language statement. Because of this correspondence, system-language programs usually translate efficiently to the machine-language level, and the programmer finds all the machine's facilities directly available to him. PL/M, an example of such a language, was designed for use with the 8008 microprocessor, and is also usable with Intel's newer 8080 microprocessor ["Electronics", April 18 (1974), p. 95], which has more useful machine-level instructions and a consideraibly faster instruction cycle than its predecessor.

Nevertheless, some hardware designers, particularly those newly introduced to software systems, may prefer to work at a comfortable level, which may mean programming in absolute machine code initially and then moving to assembly language as more capability is required. Similarly, they can easily make the transition to a high-level language when programming in assembly-language becomes tedious.

In any case, the designer soon becomes familiar with various programming levels. One of these levels can then be intelligently selected as most appropriate for a given application. Each level has its own advantages. For example, a program in PL/M that compiles into about 500 bytes of memory space when using the 8008's instruction set might require perhaps as much as 30% less space if it were coded directly in assembly language. But larger programs running 1,000 bytes or more usually turn out to be more compact when written

in PL/M than in assembly language because the compiler can keep track more easily of memory-reference areas, registers, and other resources. The amount of machine code generated in assembly language or PL/M varies, of course, with program complexity and style. Thus, an absolute comparison between the two is not possible.

Simple coding

The PL/M language consists of a number of basic statement types in which complicated arithmetic, logical, and character operations on 8-bit and 16-bit quantities can be expressed in a form resembling usual algebraic notation. Relational tests can be expressed in a natural way to control conditional branching throughout the PL/M program.

For example, to move the larger of two numbers in locations A and B into the location called C, either the PL/M statement,

```
IF A > B THEN C=A; ELSE C=B
```

or the nine-instruction assembly-language program shown in Figure 1 can be used. The statement reads, "If the value of A is greater than the value of B, then set C to equal A; otherwise set C to equal B."

```
+-----------------------------------------------------+
|     LABEL   INSTRUCTION     COMMENT                 |
|     -----  -----------    -------                  |
|     TEST    SHL     B       Load address of B       |
|             LAM             Load B into Accum       |
|             SHL     A       Load address of A       |
|             CPM             Compare B with A        |
|             JFC     L1      Jump to L1 if B <= A    |
|             LAM             Load A into Accum       |
|     L1      SHL     C       Load address of C       |
|             LMA             Store Accum into C      |
|             END             End of program          |
+-----------------------------------------------------+
```

Figure 1. Symbolic. This simple program for choosing the larger of two numbers takes nine lines of code in symbolic or assembly language, but typically only one line in a higher-level language, such as PL/M.

Additional language structures provide iteration control to permit program segments to be "looped", or executed repeatedly a prescribed number of times. Subroutine facilities include mechanisms that are useful for modular programming and construction of subroutine libraries.

The overall structure of the PL/M language is most easily demonstrated by a simple example. Suppose a teleprinter is connected to the least-significant bit of an output port of the Intel 8008. A PL/M program that sends a short message to the teleprinter is shown in Figure 2; it individually times the transmission of the bits through the output port. This program can be translated into machine code loaded into the memory of the MCS-8, and then it

is executed.

```
+-------------------------------------------------------------------+
| LINE  STATEMENT                                                   |
| ----  ---------                                                   |
|   1   DECLARE message DATA ('walla walla wash'),                  |
|   2         (char, i, j, sendbit) BYTE;                           |
|   3                                                               |
|   4   /* Send each character from message vector to teleprinter '*/ |
|   5   DO i = 0 TO LAST (message);                                 |
|   6     char = message (i);                                       |
|   7     sendbit = 0;                                              |
|   8                                                               |
|   9     /* Send each bit from char to teleprinter */              |
|  10      DO j = 1 TO 11;                                          |
|  11         OUTPUT (0) = sendbit                                  |
|  12         CALL TIME (91);       /* Waits 9.1 ms */              |
|  13         sendbit = char AND 1;                                 |
|  14         /* Rotate char for next iteration */                  |
|  15         char = ROR (char OR 1, 1);                            |
|  16      END;                                                     |
|  17   END;                                                        |
+-------------------------------------------------------------------+
```

Figure 2. Serial sender. To print a short message on a Teletype, this routine in PL/M transmits 11 pulses at 9.1-millisecond intervals for each character in the message, stopping after the last one. The pulse train consists of one start pulse, eight data pulses, and two stop pulses.

The program begins with a data declaration that defines a string of ASCII characters -- the words "Walla Walla Wash" as shown in line 1. The 16 individual characters of this string are labeled from 0 to 15 so that they can be addressed by the program (spaces are characters, too). Four variables, or 8-bit memory locations, CHAR, I, J, and SENDBIT, are defined on line 2.

Any names

These designations are wholly arbitrary; the programmer may use any names he wants, so long as he defines them before he uses them. CHAR holds each character of the message in succession for transmission, I identifies the position of the character in the message, and J controls the position of the bit in the character. The right-most bit of location SENDBIT is the next bit to be transmitted.

Since the instructions between lines 5 and 17 are executed repetitively, they are collectively called a loop. Before each repetition, the variable I is incremented until its value indicates the position of the last character in MESSAGE -- in this case, 15.

First, the value of all bits in SENDBIT is set to 0 on line 7 to send a start pulse as the first bit (line 11). Then the individual bits of the selected

character are sent in the inner loop between lines 10 and 16. This loop is executed 11 times, corresponding to the start pulse, 8 data bits, and 2 stop pulses, during each passage through the outer loop, beginning on line 5.

Each successive bit is sent on line 11, followed by a 9.1-millisecond time-out. This time delay is a standard feature in PL/M; the compiler implements it by inserting a wait loop in the program. The wait loop stores an appropriate number in a counter, decrements it once each processor cycle, and allows the program to continue when the counter reaches zero.

On each inner-loop iteration, the right-most bit of CHAR is selected on line 13 by the AND function, and it is stored in SENDBIT. The operation on line 15 places a 1 in the right-most position of CHAR and then rotates the result one step to the right. This step gradually fills CHAR with 1s, working from left to right in each iteration, so that two stop pulses, which are 1s, are sent properly on the 10th and 11th iterations.

The operation of the PL/M compiler and its PLM1 and PLM2 subdivisions is shown graphically in Figure 3. PLM1 accepts a PL/M source program from a card reader, time-sharing console, or other input device. This first pass produces a listing of the source program, along with any error diagnostics, and analyzes the program structure. An intermediate file that contains a linearized version of the original program is written, and the symbols used in it are listed.

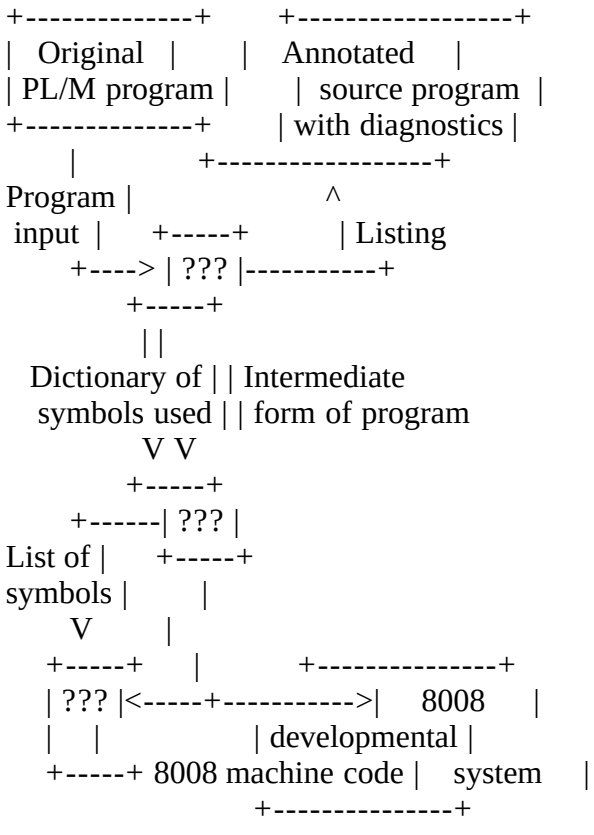


Figure 3. Compiler. Translating PL/M programs into machine language takes two passes with programs called PLM1 and PLM2, run by a larger machine. A third pass, with Interp/8, simulates the microprocessor on the big machine to check out the program.

Although the linearized version does not resemble either an assembly language or PL/M, it has been reduced to a highly simplified form of the original program. PLM2 uses this intermediate file as input and generates machine code for the 8008 microcomputer.

A PL/M program can often be checked out by simulating the 8008 microcomputer's actions on a larger machine. A third program, called Interp/8, is available for this purpose. The three programs PLM1, PLM2, and Interp/8 are written in ANSI standard FORTRAN IV, and will run on most larger computer systems.

A new version of the PL/M compiler is available for use with the extended instruction set of the 8080. Consisting of sections PLM81 and PLM82, it is accompanied by a new simulator called Interp/80. New coding is not required for the 8080. Working with old PL/M programs written for the 8008, the compiler can produce binary code requiring 10% to 20% less storage than the 8008 requires, and having the advantages of new interrupt and decimal-arithmetic capabilities.

Experience with PL/M will enable designers of future Intel microprocessors to incorporate new machine-level instructions that will make more efficient use of the PL/M language. Furthermore, if Intel so chooses, it can alter its processor architecture in future designs, as it did between the 8008 and 8080, without affecting the user of PL/M at all, except possibly to improve the performance of this application.

A number of microcomputer manufacturers are considering the use of high-level languages to augment their assembly-language products, although none have been announced yet. Several minicomputer producers, however, offer high-level applications languages, and at least one minicomputer company, Microdata Corp., provides a systems language. In fact, Microdata's MPL language ["Electronics", Feb. 15, 1973, p. 95] closely resembles PL/M; both of them, in fact, were essentially derived from the same basic system language.

Once the PL/M program is written and checked out, the machine code is punched on paper tape (Figure 3) and loaded into memory of a microcomputer developmental system. Again, the program is verified, and all real-time and environmental considerations are checked out. Final production systems can then be developed from this prototype. The production system, for example, may use read-only memory for the program when the developmental system's memory is read/write.

How to go on the air

Given a PL/M program and an MCS-8 microcomputer, how does a programmer actually go through the compilation and execution process? As mentioned previously, the PL/M compiler is available from several nationwide time-sharing services. These are the General Electric, Tymshare, National CSS, Applied Logic Corp., and United Computing Services facilities. Documentation for general programming is available from Intel Corp., and the time-sharing services provide system-dependent operating instructions.

Once the programmer has a contract with the commercial service, he is assigned

a work area in the host system in which he can store PL/M programs. These programs are created on line by using the time-sharing service's editor, which allows the programmer to enter and alter program files. When a particular program is created, it is saved in a permanent file for subsequent compilation.

In the compilation process, PLM1 is executed first, using the saved PL/M program as input. Any diagnostic messages are printed at the time-sharing console. If no program errors are detected during the PLM1 pass, then the programmer can call for PLM2. This second pass leaves code in MCS-8 machine language, which corresponds to the original program in the user's work area.

With this code, the programmer may execute the Interp/8 program, which reads the machine code and simulates the actions of the MCS-8, as previously discussed. If execution errors appear during simulation, the programmer can alter the original PL/M program and repeat the compilation and simulation process. When the programmer is convinced the program is correct, he can punch the machine language on paper tape or other medium at his local console.

Programming at home

When a large amount of development work is to be done, the user may find it feasible to purchase the PL/M compiler and CPU simulator directly from Intel and run them on an in-house computer system. The user, at his option, can program either in batch or time-sharing mode.

The machine code produced by the compiler can be executed in several different ways. The easiest method is with a developmental system, such as the Intel SIM8-01 or Intellec 8 or equivalent prototyping hardware. These systems include hardware and software for Teletype, as well as facilities for loading and checking out programs.

The machine code is loaded into the SIM8-01 from the Teletype into erasable read-only memories. These chips are then inserted into sockets on the prototype board, and the program is executed. With the Intellec 8 developmental system, the machine code is entered from the Teletype into read/write memory, where the program can be subsequently executed and tested. Both approaches bypass the simulation stage.

After testing the program on a developmental system, a production model making use of MCS-8 and a mixture of read-only and read/write memory can be tailored closely to the final application. Although the hardware is minimized in the production system to reduce costs, the programs remain the same as in the prototype.

Developing systems

Intel Corp. has completed a number of projects using PL/M, including an assembler that runs on the Intellec 8 developmental system. This assembler's characteristics show the effectiveness of the PL/M approach to system

development. For example, it has full macro capabilities, which means that a programmer can define special pseudo-instructions that cause the assembler to insert sequences of instructions in the main program during the assembly process. Macros are like subroutines, except that the main program executes them as it comes to them, instead of branching out of the main stream and then returning, as it does with subroutines.

The assembler is also capable of conditional assembly, which means that it can react to such external signals as the positions of console switches at the time of assembly. Such signals indicate conditions that are not necessarily known to the programmer at the time he writes the code -- such as the availability of particular output equipment to which the assembler's results are to be sent.

Another useful characteristic of the assembler is evaluation of expressions at assembly time, which permits the programmer to specify certain parameters algebraically instead of numerically or symbolically. Then when a program is assembled, the assembler evaluates the algebraic expressions and inserts the correct values in the machine-language program. The process requires the variables to be specified ahead of time, but it permits the programmer to alter these variables by changing their specification only once, rather than every time they are used in the program. It's a great time-saver and bug-killer.

While these characteristics are not uncommon in advanced assembly languages, high-level languages that can handle them are quite rare. Yet by using PL/M, the assembler was coded in approximately 100 man-hours, and it requires 6,000 bytes of program storage -- equivalent to 3,000 words on a minicomputer with a 16-bit word size. Intel estimates that the project would have taken five times as long to code and debug directly in assembly language, with little or no reduction in program-memory space. The resulting assembler is easy to maintain and alter, and, equally important, it can be recompiled for Intel's new 8080 microprocessor without alteration.

A practical example

PL/M permits many programming shortcuts, such as dividing a complex task into individual subtasks, or procedures, that are called upon when needed to simplify the job of writing the program itself. These procedures are conceptually simple and therefore easy to formulate and express in PL/M, as well as easy to check out before being incorporated in a larger program.

For example, consider a simple program for character manipulation -- one that might be part of the work of a more comprehensive word-processing system. The function is relatively simple: the program asks the keyboard for two input-character strings, scans the first string for all occurrences of the second, echoes the first string, and types an asterisk under the starting position in the first string of each occurrence of the second string. A sample interaction with this program is shown in Figure 4.

String comparison program

```
Type source string:__A__B__C__D
Type test string: __
__A__B__C__D
*  *  *  *

Type source string: 666 666 666
Type test string: 6
666 666 666
*** *** ***

Type source string: AAAAAAAABABABA
Type test string: AB
AAAAAAAABABABA
      * * *

Type source string: XXXXXXX
Type test string: XXXX
XXXXXXX
****

Type source string: WALLA WALLA WASH
Type test string: WALLA
WALLA WALLA WASH
*     *
```

Figure 4. Test run. Sample PL/M program produced this printout. Technique is valuable debugging tool.

Stated in this way, this example may seem to have little or no practical value. But it is almost identical to a program needed to fetch the strings from two different data-entry devices and do something more sophisticated than printing an asterisk when it finds a match.

This suggests a practical application -- a teleprinter to check out a routine before it is embedded in a larger program. When all the bugs are out of the routine, the procedures that transfer data to and from the teleprinter can be replaced with other procedures that, for example, check sensors and turn indicators on and off. The new procedures, of course, have to be checked out in a real environment, but that's much easier when the main routine is known to be bug-free.

Box: Hardware for PL/M

The Intel MCS-8 microcomputer consists of the 8008 microprocessor plus a collection of standard read/write and read-only memories and shift registers. The 8008 is a single-chip MOS device with

- 8-bit parallel word size
- Seven 8-bit general-purpose registers
- 16,384-word address capability, in either read-only or read/write memory
- Up to 32 8-bit latched input and output ports

The MCS-8 instruction set includes register-to-register, register-to-memory, and memory-to-register transfers, along with arithmetic, logic, and comparison instructions. Conditional and unconditional transfers and subroutine calls are also provided. Input and output instructions read data from input ports and set data into output-port latches. Each of these instructions is represented in program memory by a sequence of one, two, or three 8-bit words.

(Insert MCS-8 logic diagram here)

EOF

GKLRC.WS4

- "PL/I for Limited Resource Computers"
  Gary KILDALL
  "Microsystems", Jan/Feb 1982, pp.28-29

(Retyped by Emmanuel ROCHE.)

PL/I, Programming Language One, has in form or another been with us for nearly twenty years. Although a pragmatic language, it was considered large, unwieldy, and difficult to implement. Recently, however, the language has been revitalized through the efforts of the American National Standards Institute (ANSI) Technical Committee X3Jl where the General Purpose Subset language was defined. This so-called "Subset-G" language is upward compatible with full PL/I, but is designed expressly for mini-computer implementation. The elements selected for inclusion within Subset-G are the most commonly used facilities used in commercial, scientific, and educational application programming. Redundant language constructs, little-used facilities, and error-prone statement forms were eliminated, resulting in a sub-language which most observers believe is superior to the full language in many ways.

ANSI standard Subset-G is now available for operation on several minicomputer systems, including the Data General Eclipse and MV/8000, Prime computers and the popular Digital Equipment Corporation VAX computer. PL/I-8O, offered by Digital Research, is based upon Subset-G, and brings many mini computer and mainframe facilities to the micro-computer application programmer. The following is a brief history of the PL/I language, a discussion of PL/I facilities, and an overview of the Digital Research implementation.

PL/I was originally conceived in the early 1960's by the Advanced Language Development Committee of the SHARE FORTRAN Project, in the wake of interest created by ALGOL, FORTRAN, and COBOL. Elements of each of these languages were incorporated into the original design: block structure, nested scope of variables, procedure formats, and array referencing were, like Pascal, derived from ALGOL. Scientific facilities came from FORTRAN, including separate compilation, expression formulation, floating-point arithmetic, some I/O formation, and a wide variety of transcendental functions. Commercial processing in PL/I was derived from COBOL, including structures, decimal arithmetic, file processing, and picture formats. A variety of new statement forms were added to allow character string processing and error-exception handling, which where considered essential for high-level application programming. Real-time multi-tasking facilities were also added to allow PL/I to be used for systems programming as well. The language which resulted from this design effort contains more built-in data types, arithmetic operations, and general-purpose programming facilities than any other programming language available today. But herein lies the primary difficulty with full PL/I. The language is too large to implement effectively on any but the largest mainframes. The complexity of the language also inhibited proper use of all language features, while the unwary programmer was often trapped by strange twists and nuances of the language. Nevertheless, PL/I has proved to be a practical, pragmatic language for application programmers over the past

several years, through implementations on a variety of mainframe computers.

The popularity of PL/I led to standardization efforts for the full PL/I language. The document produced by the ANSI committee for full PL/I gives complete syntactic and semantic specifications for the language in a form suitable for compiler and run-time system implementation. That is to say, the language specification describes the manner in which PL/I must be implemented in order to conform to the standard, but does not specifically cover PL/I programming practices. The full PL/I document is considered one of the best language specifications produced to date.

The Subset-G document, in turn, describes the portions of full FL/I which are to be included. Specific features which remain in Subset-G include:

Decimal arithmetic
Character and String Constants
Restricted Arrays and Structure Assignments
Allocate and Free
Record (binary) I/O
Stream (ASCII) I/O
Format Specifications with Pictures
On-Conditions
A wide variety of Built-in Functions
Separate Compilation
Initialized Variables
Based Variables

The Digital Research PL/I-80 programming system project was started in 1978, and completed two years later. PL/l-80 is based upon Subset-G, with nearly all of the Subset-G features, and operates under the Digital Research CP/M, multi-programming MP/M, and CP/NET network operating systems for 8080, 8085, and Z-80 microprocessors. The PL/I-80 programming system itself consists of the compiler, macro assembler, linkage editor, program librarian, and run-time subroutine library.

The PL/I-80 Compiler is a "three-pass" system that reads a FL/I source program prepared using a program editor, and produces a relocatable file as output. The first pass collects declaration information, and produces a symbol table used by subsequent passes. The second pass augments the symbol information and produces intermediate language in tree-structure form for subsequent code generation. Both passes analyze the source program using recursive descent.

The third compiler pass is largely machine-independent, and consists of a comprehensive code optimization system, along with semantic handlers for 8-bit code generation. The optimizer processes the intermediate tree structures in three stages: first the trees are "normalized" and "flattened", then analyzed by a "frame optimizer", and finally processed by a special- forms recognizer.

The normalization and flattening process reduces alternate of an equivalent expression to the same form, while re-arranging expressions to reduce the number of intermediate temporary variables. The frame optimizer performs common subexpression detection within a limited range of tree-structures in preparation for later processing. This limited window provides optimizing information over a range of approximately ten to twenty statements, thus

avoiding the processing overhead associated with complete program flow analysis. Trees annotated with optimizing information are then passed to the special-forms processor, where approximately three hundred tree-structures of special interest are matched and detected. Special-forms recognition allows concise sequence of code to be produced for many common statements.

As an example, suppose the statements shown below occur in a FL/I-80 source program:

```
K=1+J
I=J+I
A(I)=A(K)+I
```

The normalization process re-arranges the first statement to:

```
K=J+I
```

The frame optimizer then marks I and K as equivalent expressions, so that A(I) and A(K) are known to have the same address. The special-forms recognizer notes that the A(I) array element is simply being incremented, and thus produces an increment memory instruction to affect the operation.

Generally, the PL/I-80 optimizing scheme produces dense machine code for all operations which are reflected in 8-bit and 16-bit architectures, including byte and word fixed-point and bit string operations. More complicated data forms, such as floating-point and decimal arithmetic, are performed out-of-line by calls to subroutines extracted from the run-time library.

The PL/I-80 linkage editor combines relocatable code produced by the compiler and macro assembler into a machine-executable memory image. In addition, subroutines are automatically extracted from the PL/I run-time library when referenced. The linkage editor also allows multi-level overlays, so that a large application, such as a menu-driven inventory control program, can be effectively executed in a small memory region.

The PL/I-80 programming system is currently being transported to 16- bit processors, with initial support for the Intel 8088 and 8086 processors, so that designers may select either 8-bit or 16-bit processors for their applications programs. The transition to the Intel processors is simplified in two ways. First, the compiler itself is written in PL/M, Intel's high-level system language, with portions of the run-time system written in PL/I. Thus, only the semantic handlers need to be altered, along with conversion of the space and time critical run-time subroutines, such as the floating-point library, which are implemented in assembly language.

The PL/I programming system will be transported to all processors and operating systems supported by Digital Research in the future, and serves as the basis for application software written for the microprocessor industry by independent software vendors.

Subset-G is a concise, consistent and practical language for professional programmers who write quality commercial applications programs for their own use or for public distribution. Further, the rapid acceptance of the Subset-G standard in the mini-computer industry opens a wide customer base for

application programs, while ensuring that those programs will not become obsolete.

EOF

GKPLI.WS4

- "PL/I-80"
  Gary Kildall
  "Interface Age", Vol.7, No.6, June 1982, p.71

(Retyped by Emmanuel ROCHE.)

(ROCHE> PL/I-80 fans will note that the program and most paragraphs are taken from Section 12, "Decimal Processing Using PL/I-80", of the "PL/I-80 Applications Guide".)

Since its introduction on mainframe computers some 20 years ago, and more recently on minicomputers, the PL/1 language has been popular, primarily with sophisticated programmers.

PL/1 is useful because it has many of the best elements of early languages such as ALGOL, FORTRAN and COBOL. It has, for example, incorporated such commercial/business processing features as structures, decimal arithmetic, file processing and picture formats from COBOL.

PL/1 has its limitations, though. Redundant language constructs, little-used facilities and error-prone statement forms make PL/1 large, unwieldy and difficult to implement for less experienced programmers.

With the development in 1976 of ANSI Subset-G PL/1, the General Purpose Subset, many of the drawbacks of full PL/1 were eliminated. Almost immediately, Subset-G achieved widespread acceptance in the minicomputer world, with implementations by Data General, DEC, Prime and Wang. With the development of the Subset-G-based PL/I-80 by Digital Research (Pacific Grove, CA), the advanced programming features of PL/1 are available for the first time on microcomputers.

One powerful feature in this micro version of PL/1 is its ability to perform both fixed decimal and floating-point binary operations under programmer control. This makes PL/I-80 particularly useful in business and commercial processing.

In most languages, the programmer has no command over the internal format used for numeric processing. Therefore, the programmer has no control over truncation errors that might arise during internal conversion from binary to decimal operations. These errors are magnified in business and commercial applications because of the need for monetary accuracy.

Differences between the way application programs process data and the way computers perform arithmetic operations make conversion necessary. Internally, computers may perform operations in binary OR decimal numbers, not in both. They generally perform in binary because binary data can be processed directly by most processors.

Commercial programs, on the other hand, usually process decimal values, so

those values must be converted to binary on input, and converted back to decimal on output.

The problem of truncation errors is compounded by differences in internal number formats between languages. For example, among two of the most popular BASIC interpreters for microcomputers, one performs calculations using floating-point binary, while the other uses decimal arithmetic. Pascal language translators generally use implementation-defined precision, while FORTRAN always performs arithmetic using floating- or fixed-point binary.

COBOL, designed specifically for commercial applications in which exact figures must be maintained throughout computations, uses decimal arithmetic.

```
dec_comp:                    bin_comp:
   proc options (main);          proc options (main);
   dcl                       dcl
      i fixed,                   i fixed,
      t decimal (7, 2);          t float (24);
   t = 0;                    t = 0;
   do i = 1 to 10000;             do i = 1 to 10000;
     t = t + 3.10;                t = t + 3.10;
   end;                      end;
   put edit (t) (f (10, 2));      put edit (t) (f (10, 2));
   end dec_comp;                  end bin_comp;
```

Figure 1. Differences between decimal and binary

The two short programs in Figure 1 illustrate the essential difference between the two computational forms: decimal and binary. The programs perform the simple function of summing the value 3.10 a total of 10,000 times. The only difference between these programs is that dec_comp computes the results using a FIXED decimal variable, while bin_comp does it with FLOATING point binary.

Dec_comp produces the correct result, 31000.00, while bin_comp produces only an approximation, 30997.30. The difference is a result of internal truncation that occurs when certain decimal constants, such as 3.10, are converted to binary approximations. The decimal .10 cannot be represented as a finite binary fractional expansion; that is to say, 3.10 is approximated as 3.099999E+00 in floating-point binary. Each addition propagates a small error into the sum that is compounded by the number of additions. In scientific applications, inherent truncation errors are often insignificant and ignored, but such errors are unacceptable in commercial and business applications.

PL/I-80 gives a programmer the choice between decimal and binary representations, so that each program can be tailored to a particular application's exact needs. It converts the internal format of the program in two steps. It first converts values to character format, and then converts to either fixed decimal or floating-point binary, depending on the requirements of the application.

To prevent truncation of digits, which occurs in the least significant position, PL/I-80 considers all digits in a computation equally significant. Since all digits are significant, the programmer must keep track of the range of values that arithmetic operands can take on.

To do this, decimal variables and constants in PL/I-80 have both precision and scale. Precision denotes the number of digits in the variable or constant, while scale denotes the number of digits in the fractional part. Fixed decimal variable and constant precisions must not exceed 15, and the scale must not exceed the precision. The precision and scale of a PL/I-80 variable are defined in the variable's declaration:

```
declare x fixed decimal (10, 3)
```

The precision and scale of a PL/I-80 constant are derived by a compiler counting the number of digits in the constant and the number of digits following the decimal point. For example, the constant -324.76 has precision 5 and scale 2. Internally, fixed decimal variables and constants are stored as binary-coded decimal (BCD) pairs, where each BCD digit occupies either the high or low order four bits of each byte.

Loan schedule as an example

A typical commercial/business application for this particular feature is shown in the accompanying listing for a program that computes a loan payment schedule, while incorporating a number of useful analysis and display formats.

In simplified terms, the algorithm incorporated into this program to compute the loan payment schedule uses three input values: principal (P), the yearly interest rate (i) and the monthly payment (PMT). Each month, the remaining principal is computed as:

(Eq.1) P + i * P

and is then reduced by the payment amount, producing a new principal for the next month:

(Eq.2) Pn = (Po + i * Po) - PMT

As show, beginning on line 116, this program reads several data items:

```
PV  : present value (initial principal)
yi  : yearly interest rate
PMT : monthly payment
ir  : yearly inflation rate
sm  : starting month of payment (1-12)
sy  : starting year of payment (0-99)
fm  : fiscal month (end of fiscal year, 1-12)
dl  : display level (0-2)
```

The initial principal and payment variables are declared as fixed decimal (10, 2), allowing values as large as $99,999,999.99. The yearly interest rate and yearly inflation rate are expressed in percentages as large as 99.99, as defined on lines 24 and 29. The month and year variables, sm, sy and fm, are in fixed binary format, and are assumed to properly represent month and year values. The variable dl defines the amount of information displayed during a

particular iteration of the program, where 0 provides the abbreviated display, 1 provides additional information and 2 gives the full trace.

Using an algorithm similar to the one described in equations 1 and 2, the primary loop in the program occurs between lines 96 and 131, where the principal is increased by the monthly interest, and reduced by the monthly payment until it becomes zero.

```
        S U M M A R Y   O F   P A Y M E N T S

        Output File Name ,

        Principal      3000
        Interest       14
        Payment        144.03
        %Inflation     0
        Starting Month 11
        Starting Year  80
        Fiscal Month   12

        Display Level
        Yr Results : 0
        Yr Interest: 1
        All Values : 2 0

-------------------------------------------------------------------------
|              L O A N   P A Y M E N T   S U M M A R Y                  |
-------------------------------------------------------------------------
|            Interest Rate 14.00%      Inflation Rate 00.00%            |
-------------------------------------------------------------------------
|Date |  Principal   |Plus  Interest|  Payment  |Principal Paid|Interest Paid|
-------------------------------------------------------------------------
|12/80|$    2,890.97|$       33.73|$   144.03|$       219.33|$       68.73|
|12/81|$    1,479.02|$       17.26|$   144.03|$     1,647.75|$      368.67|
|11/82|$        0.25|$        0.00|$     0.25|$     3,000.00|$      456.97|
-------------------------------------------------------------------------
```

Figure 2. Loan payment computation

Figure 2 is a minimal display for a loan of $3,000 at 14% interest with a $144.03 monthly payment. In this case, a 0% inflation rate is assumed with a starting payment in November 1980 and end of the year taxes due in December of each year. The display indicates the principal, interest in December, monthly payment, amount paid toward principal in December, and the amount of interest paid in the last month of the fiscal year.

```
        Principal      ,
        Interest       ,
        Payment        ,
        %Inflation     ,
        Starting Month ,
        Starting Year  ,
        Fiscal Month   ,
```

```
        Display Level
        Yr Results : 0
        Yr Interest: 1
        All Values : 2 1

-------------------------------------------------------------------------
|              L O A N   P A Y M E N T   S U M M A R Y                  |
-------------------------------------------------------------------------
|            Interest Rate 14.00%      Inflation Rate 00.00%            |
-------------------------------------------------------------------------
|Date |  Principal   |Plus  Interest|  Payment  |Principal Paid|Interest Paid|
-------------------------------------------------------------------------
|12/80|$    2,890.97|$       33.73|$   144.03|$       219.33|$       68.73|
-------------------------------------------------------------------------
|            Interest Paid During '80-'80 is         $68.73              |
-------------------------------------------------------------------------
|12/81|$    1,479.02|$       17.26|$   144.03|$     1,647.75|$      368.67|
-------------------------------------------------------------------------
|            Interest Paid During '81-'81 is        $299.94              |
-------------------------------------------------------------------------
|11/82|$        0.25|$        0.00|$     0.25|$     3,000.00|$      456.97|
-------------------------------------------------------------------------
|            Interest Paid During '82-'82 is         $88.30              |
-------------------------------------------------------------------------
```

Figure 3. Execution of the main loop

Figure 3 shows an execution of the main loop using the same values with a display level 1. In this case, the output also contains the yearly interest paid on the loan (which would presumably be deducted from taxable income) for each fiscal year.

```
        Principal      ,
        Interest       ,
        Payment        ,
        %Inflation     ,
        Starting Month ,
        Starting Year  ,
        Fiscal Month   ,

        Display Level
        Yr Results : 0
        Yr Interest: 1
        All Values : 2 2

-------------------------------------------------------------------------
|              L O A N   P A Y M E N T   S U M M A R Y                  |
-------------------------------------------------------------------------
|            Interest Rate 14.00%      Inflation Rate 00.00%            |
-------------------------------------------------------------------------
|Date |  Principal   |Plus  Interest|  Payment  |Principal Paid|Interest Paid|
-------------------------------------------------------------------------
|11/80|$    3,000.00|$       35.00|$   144.03|$       109.03|$       35.00|
```

12/80	$ 2,890.97	$ 33.73	$ 144.03	$ 219.33	$ 68.73
Interest Paid During '80-'80 is $68.73					
01/81	$ 2,780.67	$ 32.44	$ 144.03	$ 330.92	$ 101.17
02/81	$ 2,669.08	$ 31.14	$ 144.03	$ 443.81	$ 132.31
03/81	$ 2,556.19	$ 29.82	$ 144.03	$ 558.02	$ 162.13
04/81	$ 2,441.98	$ 28.49	$ 144.03	$ 673.56	$ 190.62
05/81	$ 2,326.44	$ 27.14	$ 144.03	$ 790.45	$ 217.76
06/81	$ 2,209.55	$ 25.78	$ 144.03	$ 908.70	$ 243.54
07/81	$ 2,091.30	$ 24.40	$ 144.03	$ 1,028.33	$ 267.94
08/81	$ 1,971.67	$ 23.00	$ 144.03	$ 1,149.36	$ 290.94
09/81	$ 1,850.64	$ 21.59	$ 144.03	$ 1,271.80	$ 312.53
10/81	$ 1,728.20	$ 20.16	$ 144.03	$ 1,395.67	$ 332.69
11/81	$ 1,604.33	$ 18.72	$ 144.03	$ 1,520.98	$ 351.41
12/81	$ 1,479.02	$ 17.26	$ 144.03	$ 1,647.75	$ 368.67
Interest Paid During '81-'81 is $299.94					
01/82	$ 1,352.25	$ 15.78	$ 144.03	$ 1,776.00	$ 384.45
02/82	$ 1,224.00	$ 14.28	$ 144.03	$ 1,905.75	$ 398.73
03/82	$ 1,094.25	$ 12.77	$ 144.03	$ 2,037.01	$ 411.50
04/82	$ 962.99	$ 11.23	$ 144.03	$ 2,169.81	$ 422.73
05/82	$ 830.19	$ 9.69	$ 144.03	$ 2,304.15	$ 432.42
06/82	$ 695.85	$ 8.12	$ 144.03	$ 2,440.06	$ 440.54
07/82	$ 559.94	$ 6.53	$ 144.03	$ 2,577.56	$ 447.07
08/82	$ 422.44	$ 4.93	$ 144.03	$ 2,716.66	$ 452.00
09/82	$ 283.34	$ 3.31	$ 144.03	$ 2,857.38	$ 455.31
10/82	$ 142.62	$ 1.66	$ 144.03	$ 2,999.75	$ 456.97
11/82	$ 0.25	$ 0.00	$ 0.25	$ 3,000.00	$ 456.97
Interest Paid During '82-'82 is $88.30					

Figure 4. Full display of data

Figure 4 uses the same initial values, but provides full display of the monthly principal, interest, monthly payment, payment applied to the principal and interest payment.

```
Principal      ,
Interest       ,
Payment        ,
%Inflation     10
Starting Month ,
Starting Year  ,
Fiscal Month   10

Display Level
Yr Results : 0
Yr Interest: 1
All Values : 2 2
```

L O A N P A Y M E N T S U M M A R Y					
Interest Rate 14.00%		Inflation Rate 10.00%			
Date	Principal	Plus Interest	Payment	Principal Paid	Interest Paid
11/80	$ 3,000.00	$ 35.00	$ 144.03	$ 109.03	$ 35.00
12/80	$ 2,864.95	$ 33.42	$ 142.73	$ 217.35	$ 68.11
01/81	$ 2,733.39	$ 31.88	$ 141.58	$ 325.29	$ 99.45
02/81	$ 2,602.35	$ 30.36	$ 140.42	$ 432.71	$ 129.00
03/81	$ 2,471.83	$ 28.83	$ 139.27	$ 539.60	$ 156.77
04/81	$ 2,341.85	$ 27.32	$ 138.12	$ 645.94	$ 182.80
05/81	$ 2,212.44	$ 25.81	$ 136.97	$ 751.71	$ 207.08
06/81	$ 2,083.60	$ 24.31	$ 135.82	$ 856.90	$ 229.65
07/81	$ 1,955.36	$ 22.81	$ 134.66	$ 961.48	$ 250.52
08/81	$ 1,829.70	$ 21.34	$ 133.65	$ 1,066.60	$ 269.99
09/81	$ 1,702.58	$ 19.86	$ 132.50	$ 1,170.05	$ 287.52
10/81	$ 1,576.11	$ 18.38	$ 131.35	$ 1,272.85	$ 303.41
Interest Paid During '80-'81 is $332.69					
11/81	$ 1,451.91	$ 16.94	$ 130.34	$ 1,376.48	$ 318.02
12/81	$ 1,326.68	$ 15.48	$ 129.19	$ 1,478.03	$ 330.69
01/82	$ 1,203.50	$ 14.04	$ 128.18	$ 1,580.64	$ 342.16
02/82	$ 1,079.56	$ 12.59	$ 127.03	$ 1,680.87	$ 351.67
03/82	$ 957.46	$ 11.17	$ 126.02	$ 1,782.38	$ 360.06
04/82	$ 835.87	$ 9.74	$ 125.01	$ 1,883.39	$ 366.92
05/82	$ 714.79	$ 8.34	$ 124.00	$ 1,983.87	$ 372.31
06/82	$ 594.25	$ 6.93	$ 123.00	$ 2,083.81	$ 376.22
07/82	$ 474.26	$ 5.53	$ 121.99	$ 2,183.19	$ 378.66
08/82	$ 354.84	$ 4.14	$ 120.98	$ 2,281.99	$ 379.68
09/82	$ 236.02	$ 2.75	$ 119.97	$ 2,380.19	$ 379.27
10/82	$ 117.80	$ 1.37	$ 118.96	$ 2,477.79	$ 377.45
Interest Paid During '81-'82 is $124.28					
11/82	$ 0.20	$ 0.00	$ 0.20	$ 2,457.00	$ 374.25
Interest Paid During '81-'82 is $0.00					

Figure 5. Loan with inflation adjustment

The same loan and interest rate with an adjustment in dollar value due to inflation is shown in Figure 5. A rather conservative 10% inflation rate is assumed, so that all amounts are scaled to the value of the dollar at the time the loan was issued. For tax reporting purposes, the display showing total interest paid at the end of the year is not scaled and does not match the sum of the interest paid during the year.

If we assume a zero inflation rate, the total loan payment is $3,456.97, taken from the previous output. Assuming an inflation rate of 10%, however, the total cost of the loan in today's dollars is

```
    $2,457.00
+    374.25
    ---------
    $2,831.25
```

resulting in a net gain of $68.75 over a two-year period.

Listing

```
pmt:
    proc options (main);
    %replace
        true    by '1'b,
        false   by '0'b,
        clear   by '^z';
    dcl
        end bit (1),
        m       fixed binary,
        sm      fixed binary,
        y       fixed binary,
        sy      fixed binary,
        fm      fixed binary,
        dl      fixed binary,
        P       fixed decimal (10, 2),
        PV      fixed decimal (10, 2),
        PP      fixed decimal (10, 2),
        PL      fixed decimal (10, 2),
        PMT     fixed decimal (10, 2),
        PMV fixed decimal (10, 2),
        INT     fixed decimal (10, 2),
        YIN     fixed decimal (10, 2),
        IP      fixed decimal (10, 2),
        yi      fixed decimal (4, 2),
        i       fixed decimal (4, 2),
        INF     fixed decimal (4, 3),
        ci      fixed decimal (15, 14),
        fi      fixed decimal (7, 5),
        ir      fixed decimal (4, 2);

    dcl
        name char (14) var static init ('$con'),
        output file;

    put list (clear, '^i^iS U M M A R Y    O F    P A Y M E N T S');

    on undefinedfile (output)
        begin;
        put skip list ('^i^icannot write to', name);
        go to open_output;
        end;

    open_output:
```

```
        put skip (2) list ('^i^iOutput File Name ');
        get list (name);

if name = '$con' then
        open file (output) title ('$con') print pagesize (0);
else
        open file (output) title (name) print;

on error
        begin;
        put skip list ('^i^iBad Input Data, Retry');
        go to retry;
        end;

retry:
        do while (true);
        put skip (2)
                list ('^i^iPrincipal     ');
        get list (PV);
        P = PV;
        put list ('^i^iInterest          ');
        get list (yi);
        i = yi;
        put list ('^i^iPayment           ');
        get list (PMV);
        PMT = PMV;
        put list ('^i^i%Inflation        ');
        get list (ir);
        fi = 1 + ir / 1200;
        ci = 1.00;
        put list ('^i^iStarting Month ');
        get list (sm);
        put list ('^i^iStarting Year  ');
        get list (sy);
        put list ('^i^iFiscal Month   ');
        get list (fm);
        put edit ('^i^iDisplay Level  ',
                '^i^iYr Results : 0 ',
                '^i^iYr Interest: 1 ',
                '^i^iAll Values : 2 ')
                (skip, a);
        get list (dl);
        if dl < 0 | dl > 2 then
                signal error;
        m = sm;
        y = sy;
        IP = 0;
        PP = 0;
        YIN = 0;
        if name ^= '$con' then
                put file (output) page;
        call header ();
                do while (P > 0);
                end = false;
```

```
            INT = round (i * p / 1200, 2);
            IP  = IP + INT;
            PL  = P;
            P   = P  + INT;
            if P < PMT then
                  PMT = P;
            P   = P - PMT;
            PP  = PP + (PL - P);
            INF = ci;
            ci = ci / fi;
            if P = 0 | dl > 1 | m = fm then
                  do;
                  put file (output) skip
                        edit ('|', 100*m+y) (a,p'99/99');
                  call display (PL * INF, INT * INF,
                  PMT * INF, PP * INF, IP * INF);
                  end;
            if m = fm & dl > 0 then
                  call summary ();
            m = m + 1;
            if m > 12 then
                  do;
                  m = 1;
                  y = y + 1;
                  if y > 99 then
                        y = 0;
                  end;
            end;
      if dl = 0 then
            call line ();
      else
      if ^end then
            call summary ();
      end;

display:
      proc (a,b,c,d,e);
      dcl
            (a,b,c,d,e) fixed decimal (10, 2);
      put file (output) edit
            ('|',a,'|',b,'|',c,'|',d,'|',e,'|')
            (a,2(2(p'$zz,zzz,zz9v.99',a),
                  p'$zzz,zz9.v99',a));
      end display;

summary:
      proc;
      end = true;
      call current_year (IP - YIN);
      YIN = IP;
      end summary;

current_year:
      proc (I);
```

```
        dcl
            yp fixed binary,
            I fixed decimal (10, 2);
        yp = y;
        if fm < 12 then
            yp = yp - 1;
        call line ();
        put skip file (output) edit
        ('|','Interest Paid During '',yp,'-'',y' is ',I,'|')
        (a,x(15),2(a,p'99'),a,p'$$$,$$$,$$9V.99',x(16),a);
        call line ();
        end current_year;

    header:
        proc;
        put file (output) list (clear);
        call line ();
        put file (output) skip edit
        ('|','L O A N  P A Y M E N T  S U M M A R Y','|')
            (a,x(19));
        call line ();
        put file (output) skip edit
        ('|','Interest Rate',yi,'%','Inflation Rate',ir,'%','|')
            (a,x(15),2(a,p'b99v.99',a,x(6)),x(9),a);
        call line ();
        put file (output) skip edit
            ('|Date |',
             ' Principal    |',
             'Plus  Interest|',
             '  Payment  |',
             'Principal Paid|',
             'Interest Paid |') (a);
        call line ();
        end header;

    line:
        proc;
        dcl
            i fixed bin;
        put file (output) skip edit
        ('-------','------------',
            '---------------' do i = 1 to 4)) (a);
        end line;
end pmt;


EOF
```

GKPRL.WS4

- "A simple technique for static relocation of absolute machine code"
  Gary Kildall
  DDJ ("Dr. Dobb's Journal"), #22, Vol.3, No.2, February 1978, pp.10-13

(Retyped by Emmanuel ROCHE.)

One principal difficulty with newly evolving computer technology is that software generation tools generally lag corresponding hardware facilities, thus forcing the software engineer to resort to outmoded techniques to produce software systems.

The purpose here is to present one area of difficulty -- that of a static program relocation -- and to offer a simple solution which can be applied to nearly any microcomputer software environment where relocation is not supported by the manufacturer.

The need for static relocation arises most often in a situation where the software systems must be reconfigured in the field. For example, data entry equipment manufacturers often provide a range of optional peripherals which can be attached to user's equipment as processing requirements change. Each peripheral usually requires a software "driver" which is device-specific, and interfaces the device to the operation environment.

A common approach to software reconfiguration is to arrange the individual translated peripheral drivers into distinct machine code modules which can be selectively brought together to form an integral system at the customer site. In order to perform the field reconfiguration, each module is translated so that it originates at location 0 in memory and, when it is brought together with other modules, it is placed at the next available memory location as the system is being constructed. All machine code elements which are location-dependent must, of course, be altered to reflect the actual locations that the driver occupies. Generally, the elements which are affected are the addresses of branch destinations and data addresses. If the locations of the affected addresses in each module are known ahead of the system reconfiguration, the module can be placed anywhere in the final memory image.

Simple static relocation

The process of constructing an executable memory image from a set of relocatable modules, as described above, is called static relocation. Unfortunately, very few microcomputer manufacturers produce the address information with their translator output which is required for the relocation process. The method described below, however, can be applied to the output of most manufacturer's absolute translators to derive the necessary relocation information. In order to be specific, the Intel 8080 microcomputer is used in the discussion, with the understanding that the concepts can be easily extended to differing architectures.

The Intel 8080 microcomputer has a 64K (65,535 bytes) memory space, which can be thought of as 256 "pages" of 256 bytes per page. Data and instructions are intermixed in this memory space, and are addressed with a 16-bit address operand which can be divided into an 8-bit (high-order) page address (0-255), and an 8-bit (low-order) address within a page. Typical 8080 instructions which can use these address operands are shown in Figure 1, where PA denotes the page address, and AWP denotes the address within a page.

```
+--------+-----+
| MVI A, | PA  |       Move immediate to A
+--------+-----+
| MVI C, | AWP |       Move immediate to C
+--------+-----+----+
| LXI D, | AWP | PA |   Load DE with address
+--------+-----+----+
| JMP    | AWP | PA |   Jump to address
+--------+-----+----+
```

Figure 1. Typical 8080 instructions

In general, a machine code memory image consists of instructions, instruction addresses, and data items. The instructions and data items are independent of the actual location at which the module finally resides. Further, only a subset of the instruction addresses are dependent upon the module location. That is to say, a load instruction may reference a buffer address which is fixed outside the relocatable module, in which case it does not change when the module is moved into position. If the address references a branch location or data item within the module, then the value of PA, AWP, or both, must be biased by fixed values, dependent upon the final position of the module in the resulting configuration.

A simpler form of relocation, called "page boundary relocation", is usually sufficient for field reconfiguration. In this case, the module is relocated to a page boundary, so that only the page address (PA) need be changed to perform the relocation, since the address within a page (AWP) remains constant.

Page boundary relocation

In its simplest form, page boundary relocation can be accomplished by constructing two parallel memory images for each module. The first, called the "relative-0", image is created by translating the module for execution at location 0. The second, called the "relative-1", image is produced by translating the module for execution at page 1 (address 256). The relative-0 and relative-1 memory images can then be compared to determine the high-order address elements which must change when the module is moved to its final page boundary location. Figures 2a and 2b show a simple program segment assembled as relative-0 and relative-1 images. The differences in the machine code images are shown in bold characters, and are thus the high-order addresses which must be biased when the module is moved. Figure 2c shows the same program segment assembled at page 5. Note that, if the bolded address fields in the relative-0 image are biased by an amount 5 (corresponding to page

boundary 5), they result in the proper values for the relocated program.

```
0000              ORG    0           ; Relative-0 assembly
0000 3E00    start: MVI    A, d1 SHR 8    ; Page address to A
0002 0E0A          MVI    C, d1 AND 0FFH  ; Address in page
0004 110A00        LXI    D, d1          ; Full address to DE
0007 C30000        JMP    start
          ; Data area
000A        d1     DS     2              ; Two unfilled
000C 00            DB     0              ; One filled element
000D               END

:0A0000003E000E0A110A00C30000C2
:01000C0000F3
:0000000000
```

Figure 2a. Relative-0 assembly

```
0100              ORG    100H           ; Relative-1 assembly
0100 3E01    start: MVI    A, d1 SHR 8    ; Page address to A
0102 0E0A          MVI    C, d1 AND 0FFH  ; Address in page
0104 110A01        LXI    D, d1          ; Full address to DE
0107 C30001        JMP    start
          ; Data area
010A        d1     DS     2              ; Two unfilled
010C 00            DB     0              ; One filled element
010D               END

:0A0100003E010E0A110A01C30001BE
:01010C0000F2
:0000000000
```

Figure 2b. Relative-1 assembly

```
0500              ORG    500H           ; Assembly at page 5
0500 3E05    start: MVI    A, d1 SHR 8    ; Page address to A
0502 0E0A          MVI    C, d1 AND 0FFH  ; Address in page
0504 110A05        LXI    D, d1          ; Full address to DE
0507 C30005        JMP    start
          ; Data area
050A        d1     DS     2              ; Two unfilled
050C 00            DB     0              ; One filled element
050D               END

:0A0500003E050E0A110A05C30005AE
:01050C0000EE
:0000000000
```

Figure 2c. Assembly at page 5

The program which actually performs the relocation process is a simple modification of an absolute loader. The translator output for an 8080 microcomputer is a "hex format" file, containing a sequence of absolute records which give a load address and byte values to be stored starting at the

load address. The exact format of each record, shown in Figure 3, begins with a colon (":") followed immediately by a 2-digit record length (RL) and 4-digit load address (LA). The 2-digit record type (RT) is always zero for absolute records, and is followed by RL pairs of hexadecimal digits to be placed at LA through LA+RL-1 in memory.

```
+---+----+------+----+-------------+----+
| : | nn | aaaa | tt | d1 d2 ... dn | cc |
+---+----+------+----+-------------+----+
```

where:

nn = record length (01-FF)
aaaa = load address (0000-FFFF)
tt = record type (00)
d1 = data byte #1
d2 = data byte #2
...
dn = data byte #n
cc = checksum byte

Figure 3. Hex file format

The record terminates with a pair of checksum digits: if the byte values (hexadecimal digit pairs) are summed, starting immediately after the colon, and continuing through the end of the record (including the checksum byte), then the sum should be zero when computed with an 8-bit counter. The checksum byte is included as an error detection mechanism. The last record of a hex file is denoted by a record length of 00.

An absolute loader reads each record of the hex file, and loads the byte values at the load address specified by LA for the next RL bytes, as shown in the algorithm of Figure 4.

Note: nextchar reads the next ASCII character
nextbyte reads the next pair of digits
nextaddr reads the next pair of bytes
CS is the checksum accumulator (8-bits)
RL is the record length (8-bits)
LA is the load address (16-bits)
M[x] is memory location x (8-bits)

```
A1  [scan for :]      if nextchar <> ":" go to A1
A2  [set checksum]    CS := 0
A3  [get length]      RL := nextbyte
A4  [last record?]    if RL = 0 go to A16
A5  [set address]     LA := nextaddr
A6  [set type]        RT := nextbyte
A7  [load bytes]      if RL = 0 go to A13
A8  [get byte]        b := nextbyte
A9  [store byte]      M[LA] := b
A10 [checksum]        CS := CS + b
A11 [next addr]       LA := LA + 1
A12 [count length]    RL := RL - 1, go to A7
A13 [checksum]        CS := CS + nextbyte
A14 [total  ok?]      if CS = 0 go to A1
```

```
A15 [check error]       halt, "checksum error"
A16 [normal end]        halt, "tape read ok"
```

Figure 4. Absolute loader algorithm

The notation used in this algorithm is that of Knuth [Ref. 2], where each step is labeled with a step name (A1, ..., A16), followed by a [comment] describing the action of the step. The action itself is a series of assignments of expressions to variables, and conditional control transfers. The algorithm begins at step A1, and scans for the beginning colon (":") for each record. When found, the algorithm reads the record length and, if zero, terminates the load operation. If the record length is not zero, the load address is read, followed by the record type (which should be zero). The algorithm then loops between steps A6 and A12, reading successive bytes to memory while computing the checksum. When the entire record has been loaded, the final checksum byte is added, which should result in a zero value. Upon completion of the algorithm of Figure 4, the entire hex file has been read and loaded to an absolute location in memory.

Note: nextchar, nextbyte, nextaddr, CS, RL, LA, and M are identical to Figure 4. PG is the page number where the relocated module will reside.

```
A1  [scan for :]        if nextchar <> ":" go to A1
A2  [set checksum]      CS := 0
A3  [get length]        RL := nextbyte
A4  [last record?]      if RL = 0 go to A16
A5  [set address]       LA := nextaddr
A6  [set type]          RT := nextbyte
A7  [load bytes]        if RL = 0 go to A13
A8  [get byte]          b := nextbyte
A9  [store byte]        M[LA + PG * 256] := b
A10 [checksum]          CS := CS + b
A11 [next addr]         LA := LA + 1
A12 [count length]      RL := RL - 1, go to A7
A13 [checksum]          CS := CS + nextbyte
A14 [total  ok?]        if CS = 0 go to A1
A15 [check error]       halt, "checksum error"
A16 [end rel-0]         go to R1

R1  [scan for :]        if nextchar <> ":" go to R1
R2  [set checksum]      CS := 0
R3  [get length]        RL := nextbyte
R4  [last record?]      if RL = 0 go to R19
R5  [set address]       LA := nextaddr + 256 * (PG - 1)
R6  [set type]          RT := nextbyte
R7  [record done?]      if RL = 0 go to R15
R8  [compare data]      b := nextbyte
R9  [data same?]        if b = M[LA] go to R12
R10 [page diff 1?]      if b <> M[LA]+1 go to R18
R11 [relocate]          M[LA] := M[LA] + PG
R12 [checksum]          CS := CS + b
R13 [next address]      LA := LA + 1
R14 [count length]      RL := RL - 1, go to R7
```

```
R15 [checksum]      CS := CS + nextbyte
R16 [total ok?]     if CS = 0 go to R1
R17 [check error]   halt, "checksum error"
R18 [reloc error]   halt, "relocation error"
R19 [end rel-1]     halt, "relocation done"
```

Figure 5. Relocating loader algorithm

The algorithm of Figure 5 is a simple extension of the previous absolute loader, which reads two successive hex files. The first hex file is the relative-0 machine code, produced by translating the module for execution at location 0. The second hex file is the relative-1 machine code, resulting from the module translation when originated at location 256 (100 in hexadecimal). The first part of the algorithm, given by steps A1 through A16, is similar to that of Figure 4, except that the data is loaded to address LA+PG*256 (which effectively moves the module to the page boundary given by PG) rather than absolute address LA.

Upon reaching step A16, the module has been loaded into memory at page PG, but is translated for execution at location 0 and thus would (most likely) execute improperly, since the high-order branch and data addresses must be biased by an amount PG. Thus, steps R1 through R19 read the relative-1 hex file to determine the addresses which must change. These steps are similar to A1 through A16, except that the input data is compared with memory for differences, rather than actually placed in memory. In step R5, the load address is read as before but, since the relative-1 machine code is biased by one page, the effective address must be reduced by 256 bytes. Step R9 compares the data loaded in the first phase with the data read in the second phase: if the data is the same, then the element is invariant in the relocation process. If the data differs, then it must have been due to the difference in the relative-0 and the relative-1 memory images. Further, this difference must be exactly 1, since differences occur only in the high-order address fields; otherwise, an error occurs and the module cannot be relocated. When a relocatable element is found, the original value loaded and relocated in phase 1 must be biased by an amount PG in step R11. Upon completion of the second phase, the algorithm halts at step R19 with the high-order addresses altered by the proper amount in the relocated module. Note that the algorithm given in Figure 5, when applied to the relative-0 file of Figure 2a, followed by the relative-1 file of Figure 2b, produces the relocated machine code of Figure 2c, when page boundary PG=5 is used.

The algorithm of Figure 5 can be easily translated to an appropriate assembly or high-level language program to perform this relocation process.

The processing of Figure 5 can be used to produce a more compact form of the relocatable module by building a "bit vector" which tabulates the addresses which require relocation, rather than actually performing the relocation process. That is to say, in step R11 the address LA must be biased by an amount PG for proper execution when the module originates at address PG*256. Thus, on the first pass, the data can be read to memory and, upon completion of the pass, a bit vector is constructed, which has one bit position for each address within the module. Before starting step R1, the entire bit vector is zeroed, to indicate that no addresses require relocation. As the second phase processing proceeds, each relocation address determined in step R11 can be

"marked" by setting the corresponding position of the bit vector. Upon completion of the algorithm, the bit vector contains zeroes in the positions corresponding to addresses which are invariant over the relocation, and ones in the positions which require biasing by an amount PG. The entire relocatable module can then be saved for later static relocation.

Given that the relative-0 memory image has been saved along with the relocation bit vector, the page boundary relocation can be simply accomplished by reading the memory image to its relocated page address PG. The bit vector is then read and processed: for each bit position which is set, the value PG must be added to the corresponding element which was previously loaded. Note that this extension to the basic algorithm of Figure 5 is included only for compact representation, and produces exactly the same memory image as the original algorithm.

A case in point

The following situation shows a case where page boundary relocation is useful. The CP/M operating system [Ref. 1] is a simple small computer diskette-based software system, which implements a file management and program loading facility for microcomputer development. The operating system is arranged as a set of modules which are loaded into memory when the computer system is started. User programs are then loaded into memory from the diskette and, because of memory constraints, must overlay non-essential portions of the CP/M system to reclaim storage for program and data areas. In order to allow these areas of memory to be reclaimed, the CP/M system is loaded into the high addresses of the memory space, and the user programs are loaded into the low addresses. Thus, the user programs can overlay the high addresses of memory when necessary and, upon completion, cause the CP/M system to be brought back from the diskette for the next operation.

Given that relocation is not supported by the manufacturer, this memory organization presents a fundamental difficulty: each CP/M operating system must be tied directly to the memory size. If the user of CP/M owns a computer system with 16K bytes of memory, then a 16K version of CP/M must be supplied. If the user adds memory to enhance system capabilities, a different version of CP/M must be supplied to support the larger memory space.

In order to overcome this difficulty, the CP/M system can be reconfigured in the field to accomodate the increased memory, using the page boundary relocation technique described above. In particular, each user receives a 16K version of CP/M (the smallest amount of memory which is useful for CP/M operation), along with a program which implements the reconfiguration. The user may optionally execute the program which rebuilds the CP/M system, according to the existing memory size, and places the relocated memory image back on the diskette, ready for subsequent loading.

The CP/M debugger program, called the Dynamic Debugging Tool (DDT), also resides in the upper portions of memory, so that it can coexist with the programs under test. Again, the area in which DDT is loaded depends upon the current memory configuration, and thus page boundary relocation is performed each time the DDT program is brought into memory. The increased elapsed time

for relocation of DDT is negligible when compared to an absolute load, as long as the bit vector technique of the previous section is used.

Restrictions

It should be noted that the technique described here is by no means a complete linking loader: no address resolution is provided between modules, and no load-time address arithmetic is allowed. Sets of modules which coexist in an integral system must communicate through instruction and data addresses. Using the technique presented here, the communication must be performed through dedicated absolute addresses for data items. Further, instruction addresses must be established through a "root module" which contains a jump vector with vector elements for each possible module which could be configured in a final system.

Address arithmetic is often useful when combining modules. In the simple page boundary relocation described above, all address arithmetic must be performed at assembly or compile time, and must consist only of simple operations which involve a fixed positive or negative offset from a base address, or a shift or logical AND operation which extracts the 8-bit page address of a full 16-bit address. A relocation error will occur, for example, if an 8-bit immediate operand instruction is obtained from a 7-bit right shift, rather than an 8-bit right shift of an address quantity.

In spite of these shortcomings, the technique has particular advantages in being independent of a manufacturer's capabilities, whims, and fancies. All language processors must eventually produce an absolute memory image for execution on the target machine, and thus the relocation process presented here will continue to operate when new software tools are introduced.

Acknowledgements

The author would like to point out that the techniques presented here, although useful, are most likely not original or particularly inventive. In fact, at least one individual, Bruce Van Natta of IMS Associates, San Leandro, California, has independently applied the methods to produce relocatable PL/M programs. There are most certainly many other software designers who have approached the relocation problem in a similar fashion.

References

1. "Microcomputer Software Design: A Checkpoint"
   Gary Kildall
   Proceedings of the Fall Joint Computer Conference
   Spartan Books, New-York, 1975

2. "The Art of Computer Programming. Volume 1: Fundamental Algorithms"
   Don Knuth

Addison-Wesley, Reading, MA, 1975

EOF

GKSL.WS4

- "Systems Languages: Management's Key to Controlled Software Evolution"
  Gary Kildall
  Proceedings of the 1974 western electronics show and convention (WESCON),
  September 1974 ("1974 WESCON Technical Papers", Volume 18, Session 19/2)

(Retyped by Emmanuel ROCHE.)

Abstract

Current industry trends forecast widespread use of microcomputers to simplify the design, development, and manufacture of many digital electronics products. The effects of microcomputer software design upon the production cycle is presented, emphasizing the necessity for well-organized software systems. High-level systems languages are introduced as an aid in software organization, using Intel's PL/M as a specific example.

Introduction

The general availability of the low-cost microcomputer, or "CPU on a chip", is undoubtedly the greatest single breakthrough in digital design technology in this decade. Although relatively inexpensive general-purpose computers have been packaged as end-user oriented minicomputers for several years, it is now economically feasible to design-in a microcomputer set into the heart of a digital system produced in large quantities. Though only recently introduced, microcomputers have been applied to a wide spectrum of digital processing, from simple device controllers through sophisticated word-processing systems. In fact, the ability to treat a microprocessor as simply another relatively inexpensive component has led to simplification of many current product designs, and opened the door to a vast array of digital applications limited only by one's imagination.

Simply stated, the microcomputer allows us to economically substitute programming for wiring. Although there are tremendous savings in software development when compared with hardware breadboarding, there are also inherent difficulties in controlling the evolution of a software-based product. Control over software evolution becomes especially important in the more comprehensive microcomputer applications, such as small business systems. The purpose here is to identify and investigate some of these difficulties from the project manager's viewpoint. The notion of a "systems language" is introduced as an aid to control of software evolution in high-quantity microcomputer-based products where a significant software investment is involved. As a specific example, Intel's high-level systems language, called PL/M, is presented.

The general product evolution cycle is discussed first, in order to characterize the effects of software development upon this cycle.

Product evolution

The wealth of new digital applications, when coupled with rapid technological change, place severe demands upon the project manager. A product must be planned with change in mind, in order to extend its sales window beyond the next unpredictable technological breakthrough. In fact, product evolution can be considered the cyclic process of change involving the product definition in order to adapt to the environment. The product environment, in turn, involves such factors as market trends, customer requirements, competitor reaction, and new technology. It is obvious that the adaptability of a product to a changing environment directly determines its survival in the marketplace. The product evolution cycle is shown graphically in Figure 1, distinguishing the 2 major branches of engineering and manufacturing, and marketing. The advent of the LSI microprocessor is an example of a technological advancement which affects the product evolution cycle. Random logic or custom chip fabrication can now be economically replaced by any of a number of low-cost ROM-driven digital processors which can perform complicated logic, arithmetic, and sequencing operations. As a result, the microprocessor can provide central and peripheral control and processing in many designs, greatly reducing time and cost in product specification, development, and production.

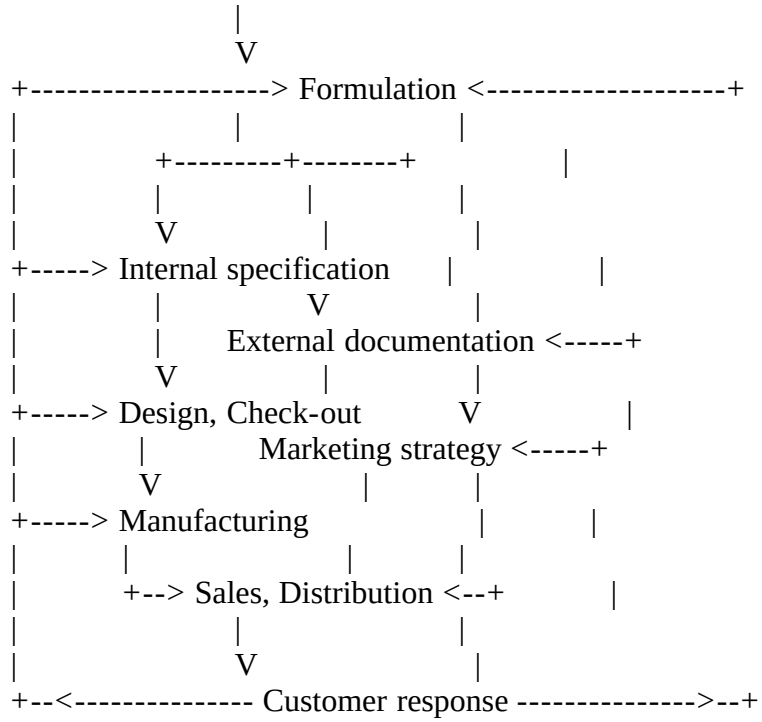


Figure 1. The product evolution cycle

The effects of microcomputer use on the marketing cycle have been investigated elsewhere [Ref. 1], and thus the discussion turns to engineering and manufacturing aspects. Referring to Figure 1, product specification efforts are reduced, since operation of a device can be specified in terms of conceptually simpler computer programs, rather than complicated logic diagrams. Further, the circuitry necessary for interfacing with the electronic environment is generally reduced to specification of simple modular units.

Design, development, and check-out efforts are reduced in a number of ways. First, the flexibility inherent in programming allows principal algorithms to be written, tested, and reprogrammed in a relatively short period, using the software development tools which are available from the major microcomputer manufacturers. This flexibility allows the designer to develop programs in a software "test bed", roughly equivalent to a hardware breadboard for circuit testing. Subroutines communicate with a standard device, such as a teletype, where data is manually entered, representing information which would normally be expected from the corresponding circuitry. This technique allows the principal control functions to be developed and independently checked before system integration. In addition, the forced modularity of the peripheral circuitry implies that each individual module can also be designed, breadboarded, and tested independently.

System integration is thus simplified, since each hardware and software subsystem has been verified. The simulating subroutines and simulated devices are individually replaced by their corresponding actual circuitry and drivers, thus isolating system design errors at each step of the integration.

Finally, manufacturing is simplified, since standard microprocessor modules can either be purchased from OEM suppliers, or developed in-house. These standard modules generally involve fewer parts than corresponding random logic designs, thus reducing both PC (ROCHE> Note that Gary Kildall uses the term "PC" 7 years (!) before the birth of the "IBM Clown"...) board layout efforts and costs for board production and testing. Given that the microprocessor modules are properly checked-out, the transition from software prototype to production software is immediate. Further, production changes often involve software modifications which affect ROM contents, rather than requiring assembly alterations.

Thus, the use of microcomputers and their associated software development tools can significantly reduce the time and costs for the first engineering and manufacturing cycle of an electronic product.

Consider now the cyclic evolution of a microcomputer-based product as it adapts to market pressures. Clearly, the adaptability of the product is directly governed by the adaptability of its software system. That is to say, since most modifications are accomplished through program changes, one can consider the product's evolution in terms of the evolution of its associated software. Changes may arise in a number of ways, including requests from customers for increased facilities, alterations required by design errors detected through field use, or modifications caused by cost advantages in using newly available hardware devices or software techniques. Thus, software evolution must be a major concern of the project manager: with proper control, each cyclic regeneration of software systems improves upon its predecessor. Loss of control over software evolution results in a maze of over-specialized algorithms and data structures which hinder successive product cycles, to the point where entire systems must be re-developed. Factors affecting software evolution are presented in the paragraphs which follow.

Software evolution

Similar to product evolution, the evolution of a software system can be considered the cyclic process of change in program design and expression, in order to adapt to the changing product definition. The 3 factors which most affect software adaptability are listed below.

1) Maintainability is a measure of the ease with which a particular program can be corrected when an error is found in the product.

2) Expandability determines the effort required to add new features or subsystems as the product definition changes.

3) Portability among programmers, machine designs, and manufacturers determines the extent to which a software system depends upon a particular software or hardware designer and design philosophy.

Maintainability, expandability, and portability of software directly determine time and cost for program regeneration. Programs are developed only once, but are maintained throughout their lifetime. Thus, the ease of program correction is a major concern in the overall software evolution cycle. Second, as new features or capabilities are added to the product, corresponding extensions are necessary to the programs. Programs written with expansion as a principal design goal adapt easily, while those which are not cause excessive delay during redesign. When time constraints prevent proper redesign of non-expandable programs, the resulting "interim" software is often undependable and cannot be properly maintained, thus adversely affecting subsequent evolution. Finally, unlike random logic designs, software systems easily become dependent upon a particular programmer, or upon a particular machine architecture or manufacturer. In most cases, if the project manager finds advantages in changing any of these variables, the software must be mainly reconstructed, often obviating those advantages. In fact, in this rapidly moving industry, the ease with which programs can be effectively moved between machines of differing design, while being readily understood by a number of different programmers, may be the most important single influence upon the software evolution cycle.

Clearly, there are many aspects of software design which determine maintainability, expandability, and portability, and a complete treatment of all factors is beyond the scope of this paper. The reader is encouraged, however, to refer to notes on structured programming [Ref. 2 and 3], software engineering [Ref. 4], and programming management [Ref. 5] for additional details. One important, but intangible, factor is the training, experience, and problem insight of the project programmers. Even the most experienced programmer, however, depends upon the programming tools that are available to express his or her solution. As a result, these tools have a profound effect upon the adaptability of the product's software.

Programming tools

The discussion of software evolution now focuses upon the degree of adaptability obtained from the various approaches to microcomputer programming. In particular, a programming language is the programmer's principal means of expressing the algorithms and data structures which perform

the specified product function. There are 4 basic methods used in microcomputer software development for expressing programs: machine language, assembly language, macro assembly language, and high-level language programming. These 4 methods are briefly reviewed below for completeness.

1) Machine language programming uses the bit patterns recognized by the microprocessor as a means for expressing programs. All program and data locations are referenced by their absolute addresses in memory.

2) Assembly language programming is one step removed from a machine level expression of a program. It allows the programmer to use symbolic names for each of the processor's operation codes, and automatically translates these codes to the proper bit patterns for microcomputer execution. The programmer references program and data addresses by freely-assigned symbolic names, rather than absolute addresses. In all cases, however, there is a one-to-one correspondence between symbolic instructions written by the programmer and the translated machine level instructions. Thus, an assembly language can be considered a convenient means of expressing machine level instructions. Figure 2 shows a sample assembly language program, and the corresponding machine level code for an Intel 8080 microcomputer [Ref. 6]. Note that the memory locations and machine operation codes are given as hexadecimal values in this figure.

```
 +------------- Location
 |   +-------- Machine language
 |   |       Assembly language and comments
 |   |       |
 V   V       V
             ; Sample assembly language program
             ; for the Intel 8080 microcomputer.
             ;
             ; Compare the values of X and Y,
             ; store the larger value into Z.
             ;
             ;-------------------------------
 0100                ORG    0100H         ; Start program code at 0100H
             ;-------------------------------
             ;
 0100 211301           LXI    H,y          ; Address Y
 0103 7E               MOV    A,M          ; Load
 0104 EB               XCHG                ; Exchange DE,HL
 0105 211201           LXI    H,x          ; Address X
 0108 BE               CMP    M            ; Compare memory
             ;
             ; Carry is set if X > Y
             ;
 0109 DA0D01           JC     setc         ; Jump if no Carry
             ;
             ; X is less or equal
             ;
 010C EB               XCHG                ; Exchange DE,HL
             ;
 010D 7E       setc:   MOV    A,M          ; Load X or Y
 010E 321401           STA    z            ; Store
```

```
          ;
          ;--------------------------------
0111 76           HLT              ; Halt processor
          ;--------------------------------
          ; Variable definitions.
          ;
0112 00     x     DB     0
0113 00     y     DB     0
0114 00     z     DB     0
          ;
          ;--------------------------------
0115              END    0100H
```

Figure 2. Machine language and assembly language programs for the Intel 8080 microcomputer.

3) Macro assembly language is similar to assembly language coding, except that the programmer is allowed to define and use macros. A macro is a predefined group of assembly language statements which is given a macro name. Each use of the macro name causes the predefined instructions or data definitions to be directly substituted for the name. Thus, for example, the programmer can effectively "invent" new machine operations as necessary for concise expression of a particular problem. Additional facilities, such as conditional assembly and assembly-time expression evaluation, are usually present in a macro assembly language, which make it considerably more flexible than a simple assembly language.

4) High-level language programming is one step further removed from assembly language programming. Normally, the notation used in a high-level language more closely resembles common mathematical symbolism, rather than relying upon complicated sequences of machine instructions to perform a specific function. For example, the high-level statement

IF x > y THEN z = x; ELSE z = y

is read as follows: "if the value of X is greater than the value of Y, then store X's value into Z; otherwise, store Y's value into Z". The effect is that Z's new value is the larger of X and Y. Each high-level statement is translated into a sequence of machine level instructions by a compiler for the language. The statement given above is translated into the equivalent machine code shown in Figure 2 by Intel's PL/M compiler for the 8080 microcomputer [Ref. 7]. In general, a high-level language provides primitive operations, data types, and control structures which are appropriate for expressing programs within a particular problem environment. Thus, a high-level language is reasonably independent of a particular machine design and, instead, tends to depend upon the type of problem being solved. These concepts are examined in later sections.

How do these various language levels affect software evolution? First, machine level coding is generally considered inappropriate for even moderate projects, due to the non-symbolic nature of the resulting programs. Assembly language programming, with or without macro capabilities, may be appropriate for moderately-sized programs. However, the adaptability of the final product is highly dependent upon the coding practices of the project programmers, as well

as the coding standards set forth and enforced by the project manager. Portability between programmers is relatively difficult, and depends principally upon the quality of the program's documentation, rather than the programs themselves. Portability from machine to machine is severely restricted, and is usually accomplished only at the option of the manufacturer. Assembly language programs written for Intel's 8008 microcomputer, for example, can be re-assembled for their new 8080 microcomputer with only minor changes in the original program. Although the resulting programs benefit from the increased speed of the new processor, they must be rewritten to take advantage of the 8080's expanded instruction set. High-level languages, however, currently provide the highest degree of maintainability, expandability, and portability of any of these programming tools. In fact, a specific class of high-level languages, called systems languages, are considered the most appropriate tool for controlling software evolution.

Systems languages

As stated previously, a high-level language is relatively independent of a particular machine architecture, and is primarily intended to provide a concise means for expressing programs in a particular problem environment. The BASIC and FORTRAN programming languages, for example, are high-level languages which assume a scientific computation problem environment. The actual machine on which a BASIC or FORTRAN program executes should have little effect on the resulting output.

A high-level systems language is more specialized, since it assumes the problem environment is the control of a particular machine or class of machines. Thus, the goals are somewhat different than those of a pure high-level language: a high-level systems language must be relatively independent of the exact computer, while providing the necessary control structures, data types, and basic operations for clear and concise expression of systems for a particular type of computer. It follows that such a systems language must allow complete control of all machine functions, and, at the same time, eliminate the needless trivialities of assembly language. In addition, a systems language must also be structured in such a way that it can be easily translated to efficient machine language programs for the target machine. Each operation in the systems language has a direct counterpart in the machine, resulting in little or no run-time support software.

The primitive operations and data types for a microcomputer systems language are fairly simple. Bit-level data occur frequently when communicating with peripheral circuitry, such as the status information received from an input or output device. Thus, bit-level data types must be present, along with corresponding shifting, masking, and bit-testing operations. Character handling is also an important function in word-processing and data communication applications. As a result, string data types should be allowed, with facilities for comparing strings, selection of substrings, and conversion between other data types. Fundamental arithmetic processing is also necessary, but the data types may depend heavily upon the microcomputer architecture. Based upon current offerings, the useful arithmetic data types include 4-bit decimal (BCD) operands, and 8-, 16-, and 32-bit fixed-point binary quantities,

along with the usual arithmetic and relational operations. Finally, communication primitives must be provided for environmental monitoring and control.

Given that a particular microcomputer can support these functions, it is certainly the case that the operations and data types can be included in a macro assembly language for the processor. In a systems language, however, these facilities are imbedded in a convenient expressional notation, and enhanced by comprehensive data definition and program control facilities. That is to say, along with the necessary basic functions, the programmer is provided with language statements which allow program expression in a readable or "well-structured" [Ref. 8] form. These structures normally include a natural notation for statement grouping, conditional branching, and iteration or loop control. Furthermore, subroutine definition, parameter specification, and subroutine invocation mechanisms are normally provided. As a result, subroutine linkage standards are enforced, modular programming is encouraged, and construction of comprehensive subroutine libraries becomes practical. The PL/M microcomputer systems language presented in the following section is used to illustrate these capabilities.

A case in point

Intel's PL/M high-level language provides an example of a systems language for programming their 8-bit microcomputers. The statement structure of PL/M resembles IBM's PL/1 programming language and, in fact, was derived from XPL which is a dialect of PL/1. The similarity, however, is only superficial; differences in control structures, operations, and data types make PL/1 useful for general applications programming, while those of XPL make it appropriate for compiler implementation. PL/M was designed with the special needs of microcomputer systems in mind, as given in the previous section, and thus is neither a subset nor superset of these languages. The important point is that PL/M belongs to the general family of "PL languages", and thus a programmer who is familiar with one of these languages finds it relatively easy to learn any of the others. Figure 3 contains a listing of a PL/M program similar to one proposed by Popper [Ref. 9] for comparing high-level and assembly language microcomputer programming. Although Popper gives a complete description of the program in his notes, the overall structure is presented as an example of PL/M program organization.

/* Quicksort procedure

This PL/M procedure sorts an array into ascending order using the QUICKSORT algorithm. Included in this listing is the procedure, QUICKSORT, and a test driver program to demonstrate the calling sequence. Note that the procedure is written with an assumption that the number of elements to be sorted is less than or equal to 256 (low, high, uptr, dptr, lstack, hstack, array$size, a1, and a2 are byte variables) and that the precision of the array elements is 8 bits (list, temp, and ref are byte variables). These restrictions may be lifted by changing the declarations. Note also that the working arrays (lstack and hstack) are dimensioned by stack$size, where

stack$size >= log2 (array$size)

```
*/

quicksort:
  PROCEDURE (array, array$size);
    DECLARE stack$size LITERALLY '10';
    DECLARE array ADDRESS;
    DECLARE array$size BYTE;
    DECLARE list BASED array BYTE;
    DECLARE lstack (stack$size) BYTE, hstack (stack$size) BYTE;
    DECLARE top BYTE;
    DECLARE (low, dptr, uptr, high) BYTE;
    DECLARE (ref, temp) BYTE;

    push:
      PROCEDURE (a1, a2);
        DECLARE (a1, a2) BYTE;

        lstack (top) = a1;
        hstack (top) = a2;
        top = top + 1;
      END push;

    /* Main program */

    top = 0;
    CALL push (0, array$size);
    DO WHILE top <> 0;
      top = top - 1;
      IF (dptr := (low  := lstack (top)))
      <> (uptr := (high := hstack (top))) THEN
      DO ;
        ref = list (low);
    down:
        DO WHILE list (dptr) <= ref AND high > dptr;
          dptr = dptr + 1;
        END;
        DO WHILE list (uptr) >= ref AND low < uptr;
          uptr = uptr - 1;
        END;
        IF dptr < uptr THEN
        DO ;
          temp = list (uptr);
          list (uptr) = list (dptr);
          list (dptr) = temp;
          dptr = dptr + 1;
          uptr = uptr - 1;
          GO TO down;
        END;
        IF uptr > low THEN
        DO ;
          list (low) = list (uptr);
          list (uptr) = ref;
          uptr = uptr - 1;
```

```
        END;
        CALL push (low,  uptr);
        CALL push (dptr, high);
      END;
    END;
  END quicksort;

/* Begin test driver */

DECLARE test$array (16) BYTE INITIAL
  (0,15,1,14,2,13,3,12,4,11,5,10,6,9,7,8);

  CALL quicksort (.test$array, last (test$array));

EOF
```

Figure 3. A QUICKSORT program in PL/M

As described in the program "comment" at the start of Figure 3, the subroutine QUICKSORT is used to sort a list of numbers into ascending order. The subroutine begins with a number of declarations which define variables and macros used within the QUICKSORT subroutine. These declarations are then followed by a nested subroutine, called PUSH, which performs a function local to QUICKSORT. The remainder of the QUICKSORT subroutine then manipulates the locally-defined variables, along with a list of numbers passed to the subroutine, to produce a list in sorted order.

Following the end of the QUICKSORT subroutine, a list of test values is defined, called TEST$ARRAY. This test list is then passed to QUICKSORT in the CALL statement near the end of the program, to verify that QUICKSORT works correctly. The program is terminated by the symbol EOF.

The PL/M compiler is then used to translate this program into machine language. The resulting machine language can be loaded into the memory of an Intel 8-bit microcomputer, and executed. There is no output operation shown in this example, and thus the resulting sorted list is simply left in the machine's memory. In fact, the QUICKSORT subroutine would normally become a distinct software module in a microcomputer application, such as a small inventory control system, and hence the program in Figure 3 is simply a test of the module. The overall organization of the PL/M programming system has been presented in some detail elsewhere [Ref. 10], along with operating and debugging practices.

How does PL/M affect adaptability of software systems? First, the high degree of self-documentation found in high-level language programming greatly enhances maintainability, expandability, and portability among programmers. This fact has been shown many times over in large-scale computer programming using PL languages, and easily carries over to microcomputer programming in PL/M. Portability between machines has been demonstrated by Intel in their 8-bit processor line. Specifically, Intel offers a PL/M compiler for both their 8008 and newer 8080 microcomputer which allow strict upward PL/M compatibility. That is to say, although 8080 PL/M provides additional programming features, any PL/M program written for the 8008 CPU can be recompiled for the 8080 CPU without modification. The only difference is that

the resulting 8080 machine level program requires less storage, and executes faster. Furthermore, Intel's long term commitment to PL/M means that customers can expect their programs to execute on future processors, while taking advantage of each new machine design.

Portability among manufacturers is more difficult in PL/M, but not impossible. First, the 8- and 16-bit class of microcomputer appears to be a popular architecture. Thus, PL/M programs could execute on another manufacturer's machine if a PL/M compiler for that particular machine were available. Construction of a compiler is generally considered a formidable task. PL/M, however, is a small language with simple grammar rules, simple statement execution rules, and no run-time support requirements. Given that a company has a significant investment in software, a specialized compiler for PL/M could be developed in-house for nearly any manufacturer's microcomputer. Using well-known automatic compiler generation techniques [Ref. 11], a specialized PL/M compiler of this sort would require only a few man-months of effort to write, debug, and document. This approach not only allows portability among manufacturers, but also solves the second source problem to some degree.

Thus, given that a high-level language exists for a particular application, one cannot argue its merit as an aid in developing adaptable software.

The efficiency question

High-level languages have traditionally been under attack for their relative inefficiency when compared to tightly-coded assembly language programs. As stated by Weinberg [Ref. 12] "when we ask for efficiency, we are often asking for tight coding that will be difficult to modify. In terms of higher-level languages, we often descend to the (assembly) language level to make a program more efficient. This loses at least one of the benefits of having the program in the higher-level language -- that of transportability between machines. In fact, it has the effect of freezing us to a machine or implementation that, we have admitted by our very act, is unsatisfactory. However, the same managers who scream for efficiency are the ones who tear their hair when told the cost of modifications". Weinberg's comments are especially true in the large-scale computer community. When discussing microcomputer systems, however, one might argue that the relative inefficiencies of high-level language programming are intolerable, due to the added cost of memory in large quantity production.

Concentrating only on the question of efficiency for a moment, one should note that it is impossible to categorically state that all high-level languages are necessarily inefficient. Efficiency depends upon at least 2 factors: the proximity of the language to the target machine, and the quality of the compiler which performs the translation to machine language. In fact, this is a principal point: a high-level systems language is closely related to the architecture of the machines it is to control, which leads to "good" machine level programs. Statements in PL/M, for example, translate into short machine instruction sequences, since PL/M statements reflect the machine architecture. Conversely, FORTRAN language statements are difficult to process on a small machine, and would produce long sequences of machine instructions. As a result, systems languages are potentially the most efficient subclass of the high-level languages.

The quality of the high-level language translator can be measured in terms of the degree of "program optimization" that it performs. That is to say, an optimizing compiler is one which analyzes the program structure to produce machine level code which best uses the target machine's resources. Optimizing compilers are themselves evolutionary, and are generally improved over time. Several versions of Intel's PL/M compiler have been released since its first introduction in June of 1973. Each version has additional program optimizing features which reduce the amount of generated machine language. As an example, consider the QUICKSORT subroutine shown in Figure 3, which was proposed by Popper [Ref. 9] as an indication of relative inefficiency. Popper gives an equivalent QUICKSORT subroutine using Intel 8008 assembly language. The assembly language version is highly 8008 machine-dependent, resulting in a tightly-coded 215 statement program. Table 1 shows a comparison of the PL/M and assembly language versions of QUICKSORT, giving the relative inefficiency of PL/M as the measure

$$\frac{Mp - Ma}{Ma}$$

where Mp and Ma represent the memory requirements of the PL/M and assembly language versions, respectively.

Table 1. QUICKSORT comparison

Translator	Program size (in bytes)	Relative inefficiency
8008 assembler	300	-
8008 PL/M Ver1	426	42%
8008 PL/M Ver3	391	30%
8008 PL/M Ver3 (subscript optim.)	336	12%
8080 PL/M Ver1	330	10%

Note: Program size based upon body of QUICKSORT procedure.

The June 1973 release of PL/M produced 42% more program storage, while the February 1974 release produced 30% more instructions. Because of the relatively small program size, and the large number of subscripted variable references, this particular program can be considered a "worst case" for PL/M. Thus, Table 1 also shows the result of compilation with the 8008 PL/M subscript optimization options enabled. These options allow the PL/M compiler to insert short subroutines for subscript computations, rather than inline code, which both decreases the memory requirements and increases the execution speed. Using these options, the relative inefficiency is reduced to 12%. Note also that the first version of the 8080 PL/M compiler, released in March 1974, produced only 10% more machine instructions.

(ROCHE> PL/M dates:

8008 PL/M Version 1: June 1973
8008 PL/M Version 2: ? 1973

8008 PL/M Version 3: February 1974
8080 PL/M Version 1: March 1974
8080 PL/M Version 2: January 1975)

There are several points to consider in this comparison. First, it is clear that the assembly language version could be completely rewritten for the 8080 CPU, further reducing memory requirements. This, however, is the entire point: given the 2 original programs, one in PL/M, the other in assembly language, the PL/M program improves without alteration as new compilers and machine designs are introduced, while the assembly language version requires reprogramming to adapt. Further, the experience a manufacturer gains in processing the high-level language can be used in designing future processors which more effectively execute these programs.

It must also be noted that the relative inefficiency measure cannot be considered an absolute comparison. The numbers vary widely with program complexity, programming style, and programmer experience. Due to the complexity of large programs, it quite often happens that relative inefficiency becomes negative. That is to say, experience has shown that, for programs larger than 1000-2000 bytes, the PL/M compiler actually produces better machine level programs that hand-coded versions. This effect is principally due to the fact that the compiler more readily accounts for machine resources, where assembly language coding becomes confused and disorganized.

In fact, the entire discussion of relative inefficiency may be a moot point, given the current and projected costs for memory. In quantity, 2K by 8-bit ROM's are currently available for less than $15 apiece, and hence the incremental difference in production cost is hardly appreciable when compared to the adaptability of the product.

Summary

Current microcomputer designs and applications merely predict their promising future. However, due to a microcomputer-based product's heavy dependence upon software systems, the major hurdle in the near future will be in software design methodology. Never before has software design been as important: reliability and correctness of programs directly determines the quality of a product manufactured in the thousands. As customers, we must encourage the industry to offer and support the tools necessary for effective program development and adaptability.

One tool, high-level systems languages, has been shown to be a viable approach to microcomputer systems development. When coupled with proper management and programmer experience, high-level systems languages provide the means to produce quality software systems for supporting a constantly evolving product definition.

References

1. "Processors and profits: how microprocessors boost them"
Davidow, W.
"Electronics", July 11, 1974

2. "Structured Programming"
Dahl, O. et al.
Academic Press, 1972

3. "A brief look at structured programming and top-down program design"
Yourdon, E.
"Modern Data", June, 1974

4. "Software Engineering Techniques"
Buxton, J. (Editor)
Nato Science Committee, OTAN/NATO, 1110 Bruxelles, Belgium, April 1970

5. "Managing a programming project"
Metzger, P.
Prentice-Hall, 1973

6. "8080 Assembly Language Programming Manual"
Intel Corp.

7. "A Guide to PL/M Programming"
Intel Corp.

8. "The Elements of Programming Style"
Kernighan, B. et al.
McGraw-Hill, 1974

9. "Advanced Microcomputer Software Techniques"
Popper, C.
National Electronics Conference,
Professional Growth in Engineering Seminar, 1974

10. "High-level language simplifies microcomputer programming"
Kildall, G.
"Electronics", June 27, 1974

11. "A Compiler Generator"
McKeeman, W. et al.
Prentice-Hall, 1970

12. "???"
Weinberg
???

EOF

GKTMA.WS4

- "They Made America" 2nd Edition
  Harold Evans
  Back Bay Books, 2006

(Retyped by Emmanuel Roche.)

Gary Kildall (1942-1994)

He saw the future and made it work. He was the true founder of the Personal Computer revolution and the father of PC software.

Gary Kildall loved piloting his many aircraft, surfing his speedboats, roaring off on his motorcycles, riding the waves on his Jet Ski, racing his Lamborghini Countach S -- at one time when he had more money than he knew what to do with, he had the pick of 14 sports cars at his lakeside villa. But what Kildall enjoyed most, in his short life, was sitting for hours in a little office writing code for computers. "It is fun to sit at a terminal and let the code flow", he said. "It sounds strange, but it just comes out of my brain; once I am started, I do not have to think about it." He would call colleagues in the middle of the night to tell them that a program had worked. "What a rush!" he would shout. Author Robert Cringely's metaphor is apt: He wrote code as Mozart wrote concertos.

In the early 1970s, he was utterly brilliant at programming -- but that is an understatement of his crucial role in the Personal Computer revolution. He was the first person to realize that Intel's microprocessors could be used to build not just desk calculators, microwave ovens, traffic systems and digital watches, but small Personal Computers with an unimaginable multiplicity of uses. Then, entirely out of his own head, without the backing of a research lab or anyone (ROCHE>??? He was PAID by Intel to develop PL/M... And, to use PL/M, he needed an Operating System...), he wrote the first language for a microcomputer Operating System and the first floppy Disk Operating System before there was even a microcomputer, months before there was an Apple, years before IBM launched a Personal Computer. Kildall did it, moreover, in such a manner that programmers were no longer restricted by compatibility with the computer's hardware. In Kildall's system, anybody's application could run on anybody else's computer. It was the genesis of the whole third-party software industry. This alone would have been an astounding achievement. Yet, Kildall's accomplishment, while revered by experts -- "the world changed dramatically because of him" (Dr. Ken Hoganson of Kennesaw State Universty) -- is relatively unknown to the millions of users of the PC. Professor Sol Libes summed it up: "Every PC owner owes Gary a debt of gratitude. Bill Gates and Microsoft owe him more than anyone else."

Kildall stayed ten years ahead of his time and never stopped pushing the boundaries of technology up to his untimely death just as the Internet was beginning to take hold. He pushed for pre-emptive multi-tasking, window capabilities and menu-driven user interfaces. He laid down the basis for PC

networking. He created the first computer interface for videodiscs to allow non-linear playback and search capability, presaging today's interactive multimedia. He built the first consumer CD-ROM filing system and data structures for a PC. With all this inventiveness, the "Edison of computers" was also a dedicated teacher; as his son, Scott, noted, it was his devotion to creating tools to help the world, rather than money-making, that led him to devote a great deal of time to a product called Dr. Logo, an intuitive, non-abstract computer language program geared toward teaching kids to program, to use computers as learning tools, not merely game-playing machines. By the end of his life, he was working on wireless hardware connections. In all he did, he epitomized the openness of the early days of Silicon Valley, the zest for the next frontier, the conviction that the best technology would succeed in the marketplace on its own merits. He had the faith of the academic scientist that mankind advances less by the protection of knowledge than by its diffusion. Jacqui Morby, a venture capitalist, has an affectionate remembrance of his idealism from their first lunch appointment. "He said to look for a red-bearded man in cow-boy boots at San Jose airport, then he rolled up in a light plane and yelled from the cockpit for me to jump in." She had no idea that she would be whisked off to the Nut Tree restaurant, in Gold Country 80 miles North of San Jose, which just happened to have a little landing strip. On the lunch napkin, her host drew a visionary plan of an industry in which the owner of the Operating System would forswear going in for applications like word processing. "He said that would create a dangerous monopoly, and stiffle innovation."

Kildall was hardly a humorless missionary; he was unassuming, droll and generous. The bitterness that darkened the last decade of his life was similar to that inflicted on radio's Edwin Armstrong. Both men discovered that the sublime could come off a bad second to the mediocre, that misrepresentation and manipulation could prevail over truth and justice.

"The day Gary went flying" has entered legend as the explanation of how a deal with IBM came to make Bill Gate's fortune. Kildall, so it is said, preferred a joyride to a meeting with IBM and was too prickly to sign IBM's standard confidentiality agreement. The story has been swallowed whole by computer historians without the benefit of Kildall's own testimony. It is misleading. Bill Gates certainly saw and seized an opportunity, but IBM was not straight with Kildall -- and Kildall was not a natural fighter.

The loss was not Kildall's alone. Had IBM embraced Kildall's genius (and Kildall returned the embrace), the majority of computer users would have had multi-tasking and windows a decade earlier. Not long after the release of the IBM PC, Kildall's company was able to demonstrate Concurrent CP/M, a single-user system (ROCHE> False! CP/M was single-user single-tasking. MP/M was multi-user multi-tasking. Concurrent CP/M was designed because of the technical limitations of the hardware of the IBM Clown, to use "virtual" screens. The standard version of Concurrent CP/M was running several separate tasks on 3 screens: the PC screen and two Wyse 60 terminals connected to the RS-232C ports. The last version of Concurrent CP/M was running 64 terminals...) to run multiple jobs at the same time, a feature that did not occur in Microsoft products until some ten years later. By adopting Microsoft Disk Operating System (MS-DOS), IBM and Microsoft forced users to endure years of crashes, with incalculable economic cost in lost data and lost opportunities.

At the end of his life, Kildall wrote an autobiography, "Computer Connections", which has never been published. It is incisive, unaffected, moving and funny, suffused by Kildall's romance with technology. It informs part of the narrative that follows, and is the source of the kildall quotations, but nothing may ever be enough to drive a stake through the heart of the appealing myth of how Kildall missed becoming the richest man in the world. In his manuscript, Kildall writes: "I think I will make a cassette tape of the 'IBM Flying Story'. I will carry a few copies in my jacket, to give out on occasion. There is only one problem. I tell this story and, after I am done, the same person says: "Yeah, but did you go flying and blow IBM off?"

Gary Kildall's precise seafaring father, Joseph, long dreamed of building a simple machine to take the tedium out of finding just where a ship was on the face of the Earth. Having taken a sextant reading and checked a chronometer, a navigator still had tedious calculations to do, based on tables from the "Nautical Almanac" to correlate the exact time and date. Joe, who taught navigation at the family nautical college, envisaged just punching the data into his machine of cams and gears, and turning a crank for the answer. "It was not until the microcomputer was invented", writes Kildall, "that the 'crank' was truly feasible", but his father's idea stayed in his mind.

Gary was a poor performer at Seattle's Queen Anne High School. He applied his gifts to rebuilding old cars and boats, and carrying out pranks. He rewired neighborhood phone lines, so as to eavesdrop on his sister's conversations with her boyfriend. But his English grades at Queen Anne were so bad he had to stay back a year. It turned out to be a stroke of luck: When he squeezed his lanky frame into the desk for his repeat year, he found himself sitting next to a beguiling and witty young woman, Dorothy McEwen. His focus on irregular verbs suffered -- they talked so much they had to be moved to different corners of the classroom -- but she became his bride a few years later. Dorothy remembers: "He was inventive. He was like a little kid in a candy shop."

After high school, Gary followed his father, who had followed *his* father, Harold, in becoming teacher at the Kildall Nautical School. Teenage Gary taught navigation and trigonometry for several years, alongside his father and grand-father. The family tradition was strong, so Gary's father did his best to sabotage Gary's plans when, at the age of 21, he announced he was abandoning ship to go to college. His ambition ran afoul of not only his father's protests, but the fact that his grades at high school had not been good enough to qualify for the University of Washington. He petitioned the university regents to take into account his teaching at the Kildall Nautical School, and "by entirely too close a margin", he was admitted in 1963, the year of his marriage to Dorothy. She supported him while he studied -- and study hard he now did. "The Kildall Nautical School", he writes, "taught me processes that high school had not. Such as the ability to do mathematics of a sort and, most important, the mental tools to dissect and solve complicated problems, and to work from the beginning to the end in an organized fashion." He got nothing but top grades.

Kildall found himself in a pivotal moment in the transition between mechanical and digital computing. He studied both; of the mechanicals, he dryly remarked

that, after a lot of complicated button pushing, "Sometimes, the resulting number was correct." His deepest passion was for an important piece of the computer software called a compiler. Compilers are translators. They take programs written in computer languages understandable by people and turn them into the famous binary digits -- ones and zeros -- called "bits" for short (BInary digiTS), that the computer understands. "They are sort of like natural language translators", writes Kildall, "who sit in a business conference and make English into Japanese. Compilers, when perfected, can be elegant to the point where you want to paste a printout on your wall, like artwork. OK, you have to be into writing compilers to get my meaning, but when your compiler works, you are very proud and want to show it off."

In 1966, the University of Washington bought a new Burroughs B5500, a computer powerful enough to run ALGOL, or ALGOrithmic Language -- a series of procedures done by numbers. The computer follows algorithms to do mathematics much faster than people ever could. ALGOL was a precursor for today's Pascal programming language. Kildall got himself a part-time job maintaining the Burroughs. He writes: "That old B5500 became my learning machine. I saw a ton of sunrises over that Computer Center." He became so gleeful having the computer to himself that, at midnight, he would put up a sign saying "B5500 Down for Maintenance". At 6 a.m., he would take down the sign after having played with the machine all night.

His nocturnal exercises paid off. In 1967, he was accepted as one of 20 students in UW's first Master's degree program in Computer Science. What the left hand of Providence bestowed, the right threatened to take away: He received a draft notice consigning him to the Army and the Vietnam War. "Damn! All of a sudden, visions of rice paddies flew through my head. I know you are not supposed to use connections but, quite frankly, I did not want to get shot at. Dad connected me with one of his [Navy] buddies, and I got a reprieve to finish my Master's degree while I worked toward my commission as an officer." He spent two Summers at the Navy's Officer Candidate School in Newport, Rhode Island, in 1967 and 1968, became an Ensign and, while waiting for assignment, taught data processing to sailors in Seattle. "It was a bummer. I was destined to become an officer on a destroyer tossing shells into the forests of Vietnam." Unbeknownst to Kildall, the President of the University of Washington, Dr. Charles Odegaard, had been impressed by Kildall's computer work in 1969, and arranged for him to have a decisive interview shortly before he was due to be posted. The Navy Captain he met with stared Kildall in the eye. "Mr. Kildall", he said, "you have a choice to make." He could become either an officer on a destroyer, or an instructor in Mathematics at the Naval Postgraduate School (NPS) in Monterey, California. Kildall recalls: "This particular question made me understand the length of a microsecond. "Well, Sir", he said, "I would like dearly to serve my country in battle, but I think I shall take the second option, if you please." The Captain warned him that, if he taught at the Naval Postgraduate School, he would probably not reach the level of Admiral. "I took a pensive stance for a moment, and then told him that I would accept that risk."

The Navy did, however, benefit greatly from Kildall's teaching Mathematics at NPS. While he and Dorothy settled down to family life in Seaside, on the beautiful Monterey peninsula, he became lifelong friends with Dan Davis, who was assigned by the Navy to NPS at the same time, having completed his Mathematics Ph.D. at Caltech and, together with others in the Math department,

they later founded the NPS Computer Science department. They even learned to fly together in the NPS Flying Club. When his three-year tour of naval duty was up in 1972, Kildall kept a link with the School as an assistant professor, but returned to the University of Washington to continue work on his Ph.D. His thesis topic was to optimize the translation of programming languages into computer-readable form, by analyzing the flow of execution of the program. He called the project "Global Flow Optimization". After several months, Kildall found a method that seemed to work, but he could not prove its correctness mathematically. He slept little, struggling vainly for an answer. "I just sat and sat and sat in my UW grad student office, resting my head in both hands until my eyes shut by themselves late [one] evening. Nothing. Then, in an instant, the proof came to me. I was not even paying attention to it. I awoke in an instant and wrote the entire proof of my central theorem, not finishing until sunrise. I guess that is why they put lightbulbs over cartoon characters. The discovery of this proof was one of the grandest experiences of my life, except, of course, for the time I visited Niagara Falls."

It was also a manifestation of Kildall's genius. His colleague Dan Davis says: "This was the first time anyone had created a general and mathematically rigorous approach to code generation for compilers. An unusual fact about Gary was that he was equally creative in both the practical nuts and bolts of building systems and the theoretical knowledge underlying the practical. His practical genius is known, but he also made a major contribution to the advance of theoretical knowledge of code optimization in compiler theory through his Ph.D. work. Not many people could match the breadth and depth of his practical and theoretical knowledge of computing, certainly not Bill Gates."

In 1972, a colleague showed Kildall an ad in "Electronic Engineering Times" saying: "Intel Corporation offers a computer for $25." Actually, it was offering the four-bit computer chip (ROCHE> The Intel 4004.), containing 2,300 transistors but measuring only approximately 0.8 by 0.3 inches. It was designed by Intel's young Ted Hoff for a Japanese desktop calculator, but released for general sale at Hoff's urging. The cost was $25 only if you bought 10,000 of them; the price jumped to between $45 and $60 if you bought just one. But customers using the Intel 4004 chip would first need to design a custom board-level or box-level system with memory, power supplies, keyboard, display and cables. Kildall was intrigued. He had never heard of this "little chip company", but he sent for specifications for the first development system for the Intel 4004. It was a little foot-square blue box called the SIM 4-01, with read-only memory (ROM), but the price was $1,000 plus $700 for a Teletype. He did not have enough money for both on his $20,000-a-year salary.

He got around this by faking the operation of the little Intel 4004 on the big IBM 370 (ROCHE> of the NPS). As he programmed the simulator, the limitations of the chip drove him crazy, but he saw the potential of escaping from large immobile computers. "This [Intel 4004] was a very primitive computer by anyone's standard, but it foretold the possibility of one's own Personal Computer that need not be shared by anyone else. It may be hard to believe, but this little processor started the whole damn industry... There, in 1972, my dad's navigation 'crank' had arrived in the Intel 4004, but there appeared to be some major programming work to get the crank to actually work."

The Intel 4004 had no trigonometric functions, so Kildall spent months

programming the chip to find Sines and Cosines. After debugging the program on his simulator, Kildall called a friend at Intel and offered to swap the Intel 4004 simulator for a development system built around a real chip, a $1,000 SIM 4-01. The Intel engineer was less interested in the simulator than in the trigonometric functions Kildall had written. They made the trade, and Kildall had his own 4004.

There was a tedious year's journey to make anything of a machine that could fed data only four bits at a time, and had no monitor. Kildall describes the process: shining a Ultra-Violet light through a quartz window for 30 minutes to erase 256 bytes of space on the EPROM (erasable programmable read-only memory), so there was room for his own little program; feeding paper tape into a Teletype and then, line by line, typing a program written in hexadecimal code (ROCHE>??? This must be a misunderstanding. Kildall was not PHYSICALLY typing the hex code: the hex code was read from the paper tape (output by the compiler) and written in the EPROM.), known as machine language; fixing the typing errors by going back to the beginning; running the corrected code to load each EPROM. "We, pioneers, had to do all this stuff two decades ago, so you can enjoy your sweet little laptop while cruising placidly over Colorado at 37,000 feet... For reference, an average JFK to SFO flight takes about six hours. That's the time it takes to program twelve EPROMs of 256 bytes each, or a total of 3,072 bytes of memory."

A laptop today does all this work in a fraction of a second.

Nonetheless, Kildall built a briefcase computer -- "It may have been the first Personal Computer". He proudly showed it to Dan Davis, then took it around for demos, lugging with him the 60-pound Teletype. He inspired hundreds, one of them a young engineering graduate at the University of Washington, Tom Rolander, who later became important in his life. Intel, too, was impressed by Kildall's bubbling imagination, and engaged him as a part-time consultant, initially to build a simulator for the new microprocessor the company was working on (ROCHE> The Intel 8008.), which was to be more sophisticated than the Intel 4004, and ten times faster. Software applications were a low priority then at Intel; the "software group" Kildall joined part-time was only two people tucked away in a space the size of a small kitchen. Kildall devised a video game for his briefcase computer based on a 1972 idea -- something like the future "Star Wars" -- by Intel engineer Stan Mazor, a co-developer of the microprocessor. The pair of them showed it off to one of the founders of Intel, Bob Noyce, a gentle, smilling presence who occasionally walked encouragingly through the little software corner in his white lab coat. Kildall writes: "Noyce peered at the LEDs blinking away on my 4004. He looked at Stan and me, and said, bluntly, that the future is in digital watches, not in computer games." Intel had just bought Microma, one of the first digital watch companies which was, not long afterward, beaten into the ground by a flood of Japanese digital watches. Intel thus passed up an opportunity to lead the video game industry. Kildall, in a judgment that would reverberate for him, too, writes of Noyce: "He, like all of us, made some decisions that are right, and some that could have made the future unfold in a different manner." What mattered to Kildall was that, in building an industry in microprocessors, "Bob treated his people with dignity".

Intel was abuzz in 1973 with the triumph of the Intel 8008 chip, which doubled the power of its first microprocessor, and Kildall was drawn to spend more and

more time there. After his "eyeballs gave way", he would spend the night sleeping in his Volkswagen van in the parking lot. He became a trader in an electronic bazaar, swapping his software skills for Intel's development hardware. One morning, he knocked on the door of Hank Smith, the manager of the little software group, and told him he could make a compiler for the Intel 8008 microprocessor, so that his customers would not need to go through the drag of low-level assembly language. Smith did not know what Kildall meant. Kildall showed how a compiler would enable an 8008 user to write the simple equation x = y + z instead of several lines of low-level assembly language. The manager called a customer he was courting, put the phone down and, with a big smile, uttered three words of great significance for the development of the Personal Computer: "Go for it!"

The new program (ROCHE> The simulator was called "Interp".), which Kildall called PL/M, or Programming Language for Microcomputers, was immensely fruitful. Intel adopted it, and Kildall used it to write his own microprocessor applications, such as Operating Systems and utility programs. (ROCHE> It was a "System Language" (like C), but with Kildall's optimizer...) It was the instrument for developing the PL/I-80 compiler that he worked on with Dan Davis for three years. "Gary was very visual", Davis told me. "He would design things more or less graphically, and then transfer his design into code. He even had an aesthetic about his drawings. He was very thorough, patient and persistent in ensuring his solutions were not only correct, but elegant." Kildall's reward was Intel's small new computer system, the Intellec-8. (ROCHE> There is an obvious error in dates, here. He got his Intellec-8 for PL/M, not PL/I-80.) It must have been the first commercial Personal Computer, Kildall notes, though no one thought of it as that. He borrowed $1,700 to buy a printer and a video display. What irritated him was that he could not operate the Intellec-8 independently of the expensive DEC PDP-10 minicomputer now installed in the Navy's classrom at Montery (ROCHE> Error: There was an IBM 370 Mainframe at NPS, and a DECsystem-10 at Intel! The S/370 was running under CP/VMS, hence the name "CP/M"...) -- unless he could contrive a way for the Intellec-8 to store data.

Experiments with cassette tape did not work; then Memorex (ROCHE>??? Kildall says "Shugart Associates" in his DDJ article?), just down the street from Intel, came up with an eight-inch floppy disk for mainframes (ROCHE>??? Wasn't it IBM who invented the 3740?). It held 250,000 characters (ROCHE> Well, more exactly (since it is a power of 2), 256KB.), moved data at 10,000 characters a second (compared with ten characters a second with the Teletype paper reader) and, in theory, gave nearly instant access to any portion of the stored data without rewind or fast-forward. Wonderful -- but the communication between Kildall's small computer and the disk drive needed a controller board to handle the complex electronics, and there was no such thing. "I sat and stared at that damned diskette drive for hours on end, and played by turning the wheels by hand, trying to figure a way to make it fly. The absence of a controller for that floppy drive was the only thing between me and a self-hosted computer. It drove me nuts." The equipment sat in his office for a year, the software genius defeated by hardware. "I would just look at it every once in a while. That did not seem to work any better."

He went reluctantly back to his DEC minicomputer, and built an Operating System he called CP/M, or Control Program for Microcomputers, mimicking the name PL/M. He knew the program was sound, but he still could not get it to

communicate with the disk. Desperate, he called his friend from the University of Washington, John Torode, who had a Ph.D. in Electrical Engineering. Torode worked on it for a few months, and came up with a neat little microcontroller. Kildall held his breath: "We loaded my CP/M program from paper tape to the diskette, and 'booted' CP/M from the diskette, and up came the prompt:

*

"This may have been one of the most exciting days of my life, except, of course, when I visited Niagara Falls, one day."

Kildall opened a file, stored it on the floppy, and it appeared in the directory -- common place stuff now, but a dramatic achievement then, the world's first Disk Operating System for a microcomputer. As Al Fasoldt writes, without a Disk Operating System, a computer is just too dumb to do anything useful. Walking back to Kildall's home for a celebratory bottle of wine, the programmer and the engineer told each other: "This is going to be a big thing". But where was the market? Ben Cooper, an entrepreneur from San Francisco, paid Kildall to write a program for an arcade astrology machine he was making: Put in a quarter, dial your birth date and out comes your future. Kildall built the software system in a converted tool shed behind his home. When Ben mistakenly entered the command "del *.*" instead of "dir *.*" to get his files, he deleted all of the files on the diskette. And that is the origin of the prompt: "Are you sure? (Y/N)".

Cooper finally got his machine installed on Fisherman Wharf in San Francisco, and the entrepreneur and programmer sat on a bench one Summer evening, to see what would happen. A hand-in-hand couple put in a quarter, did not bother with the dial, and walked off happily enough with someone else's horoscope. "Because of it", writes Kildall, "they are probably married with seven children to this day". But nobody wanted to buy the 200 machines Cooper had built. (ROCHE> This must be quite a collector, today!)

The first big break was a sale of a word-processing program (ROCHE> Which one?) in 1975 to Omron, which made newspaper display terminals. It was the first company to build hardware using CP/M. Kildall and Torode split the $25,000.

Earlier in the year, in Albuquerque, New Mexico, Ed Roberts had come out with a mail-order kit for hobbyists for the first commercially successful Personal Computer, the Altair, which sold for $500. It had an Intel 8080 microprocessor inside -- a larger, faster and more capable successor to the Intel 8008 -- with indicator lights and toggle switches on the front panel for entering programs (ROCHE> Like all previous computers, since the front panel was the simplest way to test the components... In addition, if you had nothing more, you could still play with the LEDs!). It was notoriously difficult to use, with only 256 bytes of memory, and no screen or keyboard.

A company called IMSAI (ROCHE> The first "clone" maker, using the same "S-100 Bus" chosen at random by MITS.), in San Rafael, across the Golden Gate Bridge from Silicon Valley, had promised delivery of a Diskette Operating System for the general public, but had not even begun to figure it out when Glenn Ewing, a former naval student of Kildall's, engaged as a consultant, told IMSAI about CP/M. "Glenn came to my toolshed computer room in 1975", writes Kildall, "so

we could 'adapt' CP/M to the IMSAI hardware. What this means is that I would rewrite the parts of CP/M that manage things like diskette controllers and CRTs (screens). Well, come on, I had already done this so many times (ROCHE> ???) that the tips of my fingers were wearing thin, so I designed a general interface, which I called the BIOS (basic input/output system) that a good programmer could change on the spot for their hardware. This little BIOS (ROCHE> Less than 1KB...) arrangement was the secret to the success of CP/M."

Kildall had, in essence, created a digital pancake. The underside could be adapted to fit different hardware configurations. But the top part was truly revolutionary; it did not have to be rewritten. (ROCHE> Well, that's the purpose of any Disk Operating System... But it was the first to run on microcomputers, whose users were experiencing, until then, programs tied to a particular hardware. Like Microsoft BASIC, which was "bundled" to the MITS 8800 hardware until Version 4.51... It is only after CP/M became bigger than MITS, that Bill Gates ported it to CP/M.) Kildall developed a general-purpose easily expandable mechanism through which any application program could interface with his Operating System, by executing a simple "CALL 5" instruction (ROCHE> Well, more precisely, calling the BDOS with an appropriate system function number in a register.) This was a phenomenal advance. (ROCHE> For microcomputers. Disk Operating Systems existed for years before, on minicomputers and Mainframes. Indeed, CP/M commands were patterned after the "Monitor" of the DECsystem-10, which had an history going up to the first DEC PDP-1.) It liberated software from hardware. Any application could, thereafter, run on any computer. Another way of visualizing the revolution comes from former DRI programmer Joe Wein: "Kildall's Application Program Interface created a virtual program 'socket'."

According to Kildall, he and Ewing completed the system on a lovely afternoon, sitting in the toolshed behind Gary's house on Bayview Avenue in Pacific Grove with its hummingbird feeders, a pastoral scene for a computer revolution, for that is what it portended. (It was still there in 2005, and ought to be an historic site.) Here, they created the "universal software bus" to run programs on any home-brew computer based on the Intel 8080; a lower-cost Intel device dubbed the 8085; or on a more sophisticated microprocessor, the Zilog Z-80, prodded by an Intel spin-off. Kildall's friend and future partner, Tom Rolander, puts into context what we take for granted today: "Think how horrible it was for the software vendors before that time. They would have to have different copies of their program configured to different pieces of hardware -- and there were scores of specialized pieces of hardware. Imagine a world where each model of car required a different kind of gasoline -- that's what it was like for computer operators, before Kildall's innovation." (ROCHE> Obviously, Tom Rolander has not used CP/M for quite a while, since he no longer remembers that CP/M provided a standard for system calls dealing with a floppy disk and a virtual Teletype, THAT'S ALL... There was not one single system call dealing with screens, printers, graphics, or networks. (Remember: Gary used a Teletype (like everybody else, since the Teletype was the standard I/O device used by computers during 30 years...) when he created CP/M.) As a result, each CP/M program had its own INSTALL program for using screens or printers. Tom Rolander also forget to mention that, as a result, CP/M was PORTABLE: it was not tied to a particular hardware. Indeed, to this day, some people are still using CP/M and its programs, thanks to this portability. Have you ever tried to port Windows on a Z-80? Most people are shocked when they learn that the IBM PC cannot PHYSICALLY underline a line on screen, since

Internet links used to be displayed in Blue underlined. They simply ignore (and Microsoft does not want them to know) that Windows use a "virtual machine" to separate it from the limitations of the hardware of the IBM Clown. That's why Windows is so slow: it is doing in software what should have been done in hardware. But, then, it would not run on IBM Clowns...) Kildall created the bedrock and subsoil out of which the PC software industry would grow. He licensed his system to IMSAI for $25,000 and felt rich.

Clearly, there was a business here, but Kildall found the transition from inventor (ROCHE>??? What did Kildall invent? Certainly not CP/M, since it is a port of the Disk Operating System used by the University of Washington, that he describes at length in his Master Thesis, dated 1968, 5 years before CP/M...) to innovator wrenching.

His happy marriage (with "two great kids, Scott and Kristin") hit a reef in 1974, but it was retrieved by Dorothy's willingness to help make a business out of the CP/M program. She had not had a formal college education, but she had worked in a phone company's customer service department and, as Kildall writes, often outsmarted the grads who came to him. Gary continued to teach at Monterey while Dorothy handled the early business, sending diskettes to customers responding to a $25 advertisement she and Gary had bought in the famous insider (ROCHE>???) magazine "Dr. Dobb's Journal of Computer Calisthenics and Orthodontia" at the suggestion of its founding editor, Jim Warren. Demand for the diskettes was slow at first; the market was made up of early computer enthusiasts. "We started in a corner of the bedroom", Dorothy told us. "There was no long-term plan. We put no money into the operation. We didn't have much savings. We lived off Visa and MasterCard."

Gary had fun with his classes at Monterey, where the graduates revved up on his enthusiasm and readiness to give everyone a chance. He led them through the steps to design a wristwatch computer that monitored a Navy diver's nitrogen pressures at varying depths, to avoid the "bends". His classrom, in the words of Michael Swaine, editor at large for "Dr. Dobb's Journal", was probably the world's first academic microcomputer lab. But it was a time to move on.

"He just loved teaching", said Dorothy. "It was a hard decision for him to quit school full-time." But Dorothy encouraged the choice they made in 1976 to start a full-time mail-order business they called "Intergalactic Digital Research" -- "Intergalactic" only because someone else held claim to "Digital Research" for a couple of years. They moved to an office on Lighthouse Avenue in Pacific Grove, where Gary worked from a cupola on top of the building. He initially proposed selling his system for $29.95 a disk, i.e., giving it away. At Dorothy's insistence, he asked $70 -- which was still absurdly cheap. (ROCHE> Compared to prices of software for IBM Mainframes and minicomputers, whose users were banks and companies.) She remembers going down the block to the Pacific Grove Post Office in 1976, hoping to find checks that would keep the company alive a little while longer but, by 1978, it was a roaring success, leaving other proprietary systems in its wake. CP/M made the Intel Operating System look like a scam (ROCHE>??? What the Hell is he saying? Intel had *NO* Operating System! The only one which was developed "in-house" was CP/M, but they decided to concentrate on hardware only, kept PL/M, and gave CP/M to Gary, free to do whatever he wanted with it... Just a few months before the boom of the microcomputer revolution!), in addition to being cheap,

Kildall's system was small, it was fast, and it would run on all Intel computers and competing Zilog Z-80s. "No other software product had been priced our way before", Kildall writes. "OK, CP/M's price came up to $100 per copy with Version 1.4, but no one seemed to care." That denomination was, in itself, another Kildall invention: The first digit was a "major" revision, and the (ROCHE> digit following) the decimal point indicated a minor revision for update. "You charge the manufacturers and customers a 'minor' fee to get the minor revision, and then issue a 'major' revision, like CP/M 2.0, and charge a major fee. That became the way microcomputer softare was labeled, and for that purpose only."

In 1978, when sales were $100,000 a month with a 57 percent profit margin, Gary and Dorothy moved the business into a more spacious old Victorian house at 801 on Lighthouse Avenue, overlooking the waves of Monterey Bay. Gary worked on the top floor and Dorothy ran the business office on the ground floor, Dorothy abandoning the name Kildall for her maiden name, McEwen, to avoid the aroma of a mom-and-pop operation. "It was a very exciting time, and we were just very naive about everything, about starting a business, about the industry", Dorothy recalls. "We were young. The grownups had not come yet." They gradually recruited a young staff, students, professors and friends, and installed the programmers out of sight on the second level of the house. The hiring was casual. Dan Davis was laid up at home, recovering from a motorcycle accident, when Kildall walked in with a computer and asked if he would like to work full-time on a language compiler. "This was a wonderful period in DRI's history", recalls Dan. "Beer and pizza every Friday. People like John Pierce, Kathy Strutynski, Dave Brown, Bob Silberstein and others working like crazy and having a lot of fun." The atmosphere was certainly zany; as Kildall put it, a lot of marriages, a bunch of babies. People came to work barefoot, in shorts and in hippie dresses; anyone in a suit was a visitor. One candidate for interview with Kildall found herself talking technology with a red-bearded Roman imperor in a toga. Tom Rolander, visiting Kildall after three years working as an engineer at Intel, remarked that, as a pilot, he recognized the model airplane on Kildall's desk. Within minutes, Kildall bundled him into a sports car for a fast drive to the airport and a flight in the real plane, a Cherokee 180. Two days later, Rolander was at work in Pacific Grove writing the multitasking version of CP/M. But everyone worked hard. Strutynski was known as "the mother of CP/M 2.2", the largest money-spinner for DRI, for the hundreds and hundreds of hours she put in with Dave Brown and others, perfecting Kildall's original design.

Rolander was with Kildall through all the later triumphs and crises. Kildall writes: "Tom and I had a knack about how we worked together. I would create new stuff, write programs, and he would clean things and make them products. Sometimes the products were good, and sometimes they weren't. But that's how this world works. You don't get a home run every time." Rolander, the son of a preacher, was described by one associate as "Tom the Cannon". "What he meant is that you aim Tom in a particular direction, and light the fuse. Tom really doesn't care what direction it is; he only wants to work 80 hours a week on an interesting software problem." He is, today, still a lean, focused man exuding fitness. Visitors to his office inclined to pick up his bicycle in a corridor find it impossible to move. Rolander loads it with bricks, to make sure he gets a proper workout. He might equally be called T-for-Thoroughness Tom. "Tom learned and practiced calligraphy", writes Kildall. "During our friendship, he wrote [out] "The Prophet" in calligraphy for me. I know it took him many, many

hours to do this." The two men flew together, jogged the Asilomar Beach and confided to each other. Writes Kildall: "At the time, he was my co-pilot in flying and in life." On one scary night flight, Rolander saved both of them. Kildall, untypically distrusting his instruments, mistook a string of lights crossing Lake Ponchartrain outside New Orleans for the horizon. They were half a second from crashing when Rolander, looking out the right window, yelled the alarm. "With the airplane now in a bank", writes Kildall, "I went back on the gauges. Righting that Aerostar to 'instrument level' may have been the hardest thing I have done in my lifetime."

Kildall was not a daredevil pilot. He was fully instrument rated. But, on the ground, he relished risky fun. For his 39th birthday in 1981, he was given a pair of roller skates, "the kind that look like tennis shoes mounted on a Formula One car". When the party ran out of Champagne, he sped downhill on them to get some more, stumbling over small acorns to everyone's merriment. He liked the skates so much he rolled around on them in the corridors of the office. Alan Cooper, who made an accounting system using CP/M on an IMSAI computer, says Kildall got frustrated only when the company did not function like a college. "Employees would come to him, expecting him to solve business problems, marketing problems, personnel problems. He did not know the answers; did not really want to think about the problems. What he wanted to do was write code."

There was nothing wild about that. Flying more than 1,000 hours on business trips with Kildall, Rolander -- like Dan Davis and Andy Johnson-Laird -- came to appreciate Kildall's very methodical approach, whether for brief acrobatics in his Pitts biplane or for a journey across the country in a twin-engine Aerostar. "While my own personality would have prompted more spontaneous departures", says Rolander, "Gary's would always be done after detailed weather briefings, fuel loading, and weight and balance calculations."

"Gary's programming was just as methodical. It always began with complete and detailed sketches of data structures on large sheets of paper. The coding never started until he had visualized and comprehended the overall design. From the preflight to landing, Gary was a consummate professional in his flying, paying attention to every detail and never getting flustered. He was always calm, confident and equally demanding of detail from his copilot. He would have me rehearse my air traffic control transmissions over and over, so that I would sound like a professional. After all, we were flying up at 25,000 feet, close to the big commercial jet traffic. Gary paid just as much attention to detail in his programming. Unlike other designers, who are often content to paint the broad picture and then let the more junior programmers fill in the details, Gary designed, implemented and debugged his products."

By 1980, Kildall had sold hundreds of thousands of copies of CP/M, and had redesigned his system for the new hard drives. His was the standard Operating System for most PCs. (ROCHE> It all depends what he means by "PC". In one article, Gary uses the expression "personal computer" several years BEFORE the IBM Clown. But, at the time, everybody was talking about microcomputers, or home computers. It is only after the IBM Clown that the expression "personal computer" and its abbreviation "PC" came into widespread use, along with the "shareware" concept for office users, not home users. Before, it was not uncommon for societies to sell THE SOURCE CODE of their programs... Example: IMSAI 4K and 8K BASIC, Tarbell 24K BASIC, which had a full-screen editor,

while MBASIC used a Teletype, even on the first IBM PCs. Just try MBASIC v5.28 in a "DOS box" to experience how old-fashioned MBASIC was, on screens.) The most popular software programs, like WordStar and dBase II, ran on CP/M -- and nothing else. (ROCHE> The only significant software ever developed for the Apple II was VisiCalc. Every other professional software (1: Word-processing: Word-Master, then WordStar, 2: Programming: CBASIC, then CBASIC Compiler, 3: Spread-Sheet: Multiplan and SuperCalc, 4: Database: dBase II, 5: Communications: XMODEM (still used 30 years later!), 6: Graphics: GSX, Dr Graph, DR Draw) was developed under CP/M, then, sometimes, ported to others niche systems.) For the young couple, it was a heady time. Gerry Davis (no relation to Dan), who was then the Kildall attorney, remembers the bank calling to ask if DRI's profits were real. Davis said they were. "But they are making 85 percent profits. That's not possible." Davis assured the banker it was true. The Kildalls had a virtual monopoly. The natural question, then, is how Bill Gates got into the act.

Bill Gates was a 13-year-old hacker when Gary Kildall had already written his first compiler and was pursuing his Doctorate. Gates and Paul Allen first came into the fringes of Kildall's consciousness in 1968. Several professors at the University of Washington formed a company called "Computer Center Corporation" ("C-Cubed"), to rent out time on the new DEC PDP-10, the first real minicomputer. It allowed remote access through early modems. The Seattle area high school attended by Gates and Allen, Lakeside, had an account, and the teenagers used their school connection to break into C-Cubed's memory, where other people's passwords were kept. When a customer started complaining about charges that were not his, Allen and Gates were found out. Kildall writes that one of the professors "found the culprits and cleverly allowed them access, so that he could recode and test the Operating System to prevent illegal access".

Later, Gates and Allen famously simulated one of Ed Robert's computers (ROCHE> The MITS 8800 had nothing particular. In reality, Allen wrote an Intel 8080 simulator, then wrote a BASIC interpreter. After that, all that was needed was to write I/O subroutines for the MITS 8800, that Bill Gates wrote, since Paul Allen was busy writing Version 2. Both were totally ignorant of how computers performed computations, so paid someone else who had studied Floating-Point software at university: Monte Davidoff, to write the Floating-Point math package, a good third of the program, and the only technically difficult part of the program.) on the Harvard mainframe (ROCHE> Note that only Bill Gates was a student at Harvard, while Paul Allen was working as a programmer... It would be funny to check if Harvard students were allowed to make professional work on an academic computer...) and installed on it a simple program invented at Dartmouth (ROCHE> College) by John Kemeny and Thomas Kurtz called "BASIC", meaning "Beginner's All-Purpose Symbolic Instruction Code". It was primitive (ROCHE>??? It was LEADING EDGE TECHNOLOGY, at the time! Today, 30 years later, according to Microsoft, Visual BASIC is the #1 Programming Language in the world. No other Programming Language has been sold in such a number, in the history of computing.), but it enabled hobbyists to write their own simple programs. In 1975, Gates and Allen formed a company called Microsoft (ROCHE> Historically, "Micro-Soft". It is only 2 years later, that the name was changed to "Microsoft".) to sell this BASIC interpreter out of Albuquerque, not far from Robert's factory (ROCHE> Indeed! Since the mail address for Altair BASIC was *INSIDE* MITS...) but, two years into it, Gates wondered if that was the right location for his little business (ROCHE> After he started having bad relations with Ed Roberts...).

Gates came to consult with Kildall (ROCHE>??? Why? Remember that, at the time, Altair BASIC was NOT running under CP/M...), who drove him along the central California coastline and, while commiserating about the speeding tickets they both routinely collected, they talked of merging their two companies. "We invited him to stay that night at our home. Dorothy fixed a nice roast chicken dinner", writes Kildall in his memoir. But he adds: "For some reason, I have always felt uneasy around Bill. I always kept my hand on my wallet, and the other on my program listings. I found his manner too abrasive and deterministic, although he mostly carried a smile through a discussion of any sort. Gates is more an opportunist than a technical type." David Kaplan, the author of the engaging "Silicon Boys", says there seemed to be a gentleman's agreement that neither would get involved in the other's business. "DRI would stay away from languages, and Microsoft would leave Operating Systems alone."

Around this time, Kildall says he was encouraged by an engineer at Data General Corporation, located outside Boston, to write a whole new compiler for a newly-defined subset of IBM's Mainframe PL/I. The original PL/I was a very powerful language originally developed by IBM in the early 1960s -- "a dinosaur every bit as well done as Disney could have produced", Kildall writes. But he was impressed by the PL/I Subset-G design, and became excited about building a compiler for microcomputers, partly to show that Personal Computers could be used as serious machines, not just for running BASIC programs and games. Dan Davis, who worked on it with him, recalls: "Gary would write code for this compiler ten hours a day, day in and day out for months and months. It was a monumental undertaking." In the end, it took not the nine months Kildall thought it needed, but nearly three years. Using Kildall's algorithm from his Ph.D. work, it was, by far, the most sophisticated compiler ever built for the Intel chip set, enabling a host of new applications. But it interfered with the completion of CP/M-86, a 16-bit CP/M version to run on Intel's 8086 chip -- a delay that gave Bill Gates the opportunity of a lifetime, one he was to seize with alacrity.

Gates settled his enterprise near Seattle, Washington, of course. His breakthrough, in 1978, was Allen's design of a "Microsoft Softcard". This was an add-in board to the Wozniak-Jobs Apple IIe, so that it would run CP/M and Microsoft BASIC. The addition of CP/M gave Apple II users access to a large software base from the CP/M application suppliers. "I wanted a royalty", writes Kildall, "but Bill wanted a buyout and was stuck on that point. I sold him 10,000 copies for $2.50 each." Kildall adds with emphasis: "He signed agreements to protect the CP/M design under this license."

It was a necessary precaution. As Kildall writes: "CP/M was and always has been a copyrighted product, with external and internal markings to that effect", but many people where mimicking Kildall's design. By the late 1970s, hundreds of "clones" had been made. Some were sold in minor quantities, but they gained little market share and, in Kildall's view, hardly merited a lawsuit. Gerry Davis would issue warning letters, but Kildall liked to drop in on a more noticeable copyright infringer and try a little shame. Roger Mellon bought an Operating System from the Palo Alto Computer Store and was assured it was original. He was astounded when Kildall used the machine's built-in debugger to view Mellon's memory storage and, embedded there, was the message: "Copyright 1978, Digital Research". Mellon promptly signed up for a license. Kildall writes: "I put the copyright message in the object code for exactly

that purpose, and you had to be a very sophisticated programmer to remove that message. Not only that, if it was removed, CP/M would not run because the Operating System checked to see if the message was there before starting, using an encryption scheme that worked quite well." (Kildall had learned the encryption techniques at the Naval Postgraduate School.) In the fall of 1979, Roger Billings was doing very well selling a computer system out of his company in Provo, Utah. Kildall and Rolander flew seven hours in a single-engine Piper Archer, only to have Billings make them cool their heels in the waiting room. With nothing to do, Kildall played with a sample Billings computer in the waiting room. Using his debugger program, he quickly entered the innards of the computer Operating System. There, again, was his copyright message. Kildall writes: "Roger became quite friendly, all of a sudden."

Another participant in these little morality plays was Rod Brock, an enterprising neighbor of Bill Gates's in Redmont. Brock, who owned a mom-and-pop company called Seattle Computer Products (SCP), was impatient in 1979 for the CP/M-86 Kildall was developing to maximize the potential of the more powerful (ROCHE>??? In which sense? Complexity? Slowness? Lack of hardware? (The IBM PC used the 8-bit Intel 8088... That Intel was obliged to introduce, since all the hardware existing at the time used 8-bits I/O busses.) Lack of software?) 8086 16-bit Intel chip. Brock has said he was told it would be ready in December, but his revenues were running down so, in the hope of filling the gap, he hired 24-year-old Tim Paterson, a University of Washington graduate with a Bachelor's degree in Computer Science. In August 1980, Brock shipped an early version of the system Paterson came up with. It was officially known as 86-DOS, but Paterson called it QDOS, for Quick'n'Dirty Operating System, and Brock shipped a finished version in December 1980. Paterson borrowed Kildall's basic applications architecture, to make QDOS compatible for users of the dominant CP/M-80 and upcoming CP/M-86, which was shipped on January 23, 1981. Andy Johnson-Laird was a savvy computer specialist in Oregon, who had a business customizing the CP/M Operating System for hardware vendors. He heard that Seattle Computer Products had produced the first S-100 (ROCHE> Bus) card with a 16-bit processor. Out of technical curiosity, he dropped in on SCP. "It was a small one man, a boy, and a dog kind of operation", he recalls. "Messrs. Brock and Paterson were the only people I saw working out of one of those generic business park offices. I bought one of their boards. Later, when I looked at the documentation for 86-DOS that was supplied by SCP, it made it clear that, in the API calls, there were substantial similarities to CP/M." (APIs, meaning "Applications Programming Interfaces", are ground rules that tell the application developer what the Operating System will do in response to a defined set of requests or "calls".) Johnson-Laird just happened to be a licensee of CP/M since about 1977 who had become friends with Kildall and others, so he phoned Kildall to ask if he knew about this.

In his memoir, Kildall writes: "Paterson's Seattle DOS was yet another one of the rip-offs of the CP/M design. The CP/M machine code was taken apart, using CP/M's own DDT [its debugger], to determine the internal workings of CP/M in order to make a clone of CP/M's operation." Paterson, who vehemently denies using Kildall's source code, explained in a subsequent PBS television documentary what he did: "I took a CP/M manual that I had gotten from the Retail Computer Store for five dollars or something, and used that as a basis for what would be the application program interface, the API of my Operating System." According to James Wallace and Jim Erikson in "Hard Drive", Kildall

lost his traditional cool. They report an interview they had with Paterson, in which he said Kildall phoned him and accused him of "ripping off" CP/M: "At the time", said Paterson, "I told him I did not copy anything. I just took his printed documentation and did something that did the same thing. That's not, by any stretch, violating any kind of intellectual property laws. Making the recipe in the book does not violate the copyright on the recipe. I would be happy to debate this in front of anybody, any judge."

The analogy is questionable. Another way of looking at it is that a cookbook may give a chef an implicit right to prepare its recipes, but it does not give a chef the right to borrow extensively from the text for his own book or, say, translate the most important passages into foreign languages and sell it at a profit. Paterson was not writing a computer application according to DRI specs, i.e., cooking from a recipe. He was creating a derived work, based on the cookbook copyrighted by someone else. When Paterson wrote QDOS with Kildall's manuals "at his side" (in the words of Gary Rivlin in "The Plot to Get Bill Gates"), he was using materials marked on every page: "All Information Contained Herein is Proprietary to Digital Research". The title page just about covered every contingency: "Copyright (C) 1979. Digital Research. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language, or computer language, in any form by any means, magnetic, optical, chemical, manual, electronic, mechanical, or otherwise, without the prior written permission of Digital Research, Inc."

Computer commentators have variously referred to QDOS as a clone -- "an obvious CP/M knock-off" (Gary Rivlin); "almost a direct copy of CP/M" (Ward Christiansen); "mostly a copy of CP/M" (Joe Wein); and "kind of like CP/M" (Microsoft co-founder Paul Allen). In 1994, following the death of Kildall, Paterson and John Wharton, a former Intel engineer and computer specialist who was a friend of Kildall's, got into a vigorous debate in "Microprocessor Report". Wharton referred to QDOS as "an unauthorized 'quick and dirty' knock-off of CP/M". He added: "I can empathize somewhat with the bind in which SCP found itself: unable to sell its 8086 hardware for lack of software, and unable to buy the software it wanted. But, for Mr. Paterson to cite the unavailability of CP/M-86 as justification for appropriating the 'look and feel' of a competing Operating System and its utilities seems, to me, analogous to telling a judge: 'I needed a car, Your Honor, and the plaintiff wouldn't sell me his, so I was forced to take it.'"

Paterson resented the analogy. In a long response, he admitted drawing on the 1976 CP/M Interface Guide "so that the interface used by applications to request Operating System services would be exactly the same as CP/M's after applying the translation rules". But he also vigorously defended his work as original: "Because of the completely different file-storage format, none of the internal workings has any corresponding relation to anything like CP/M. I never used CP/M source or disassembly at any time while I was developing 86-DOS."

In 1980, the argument would have been a bagatelle, soon disposed of by lawsuit or a license from Kildall, but for the curious behavior of IBM. Everybody in the computer world knew that Kildall had created and owned CP/M -- everybody, it seems, except the biggest beast in the Mainframe jungle, at which Personal Computer had, hitherto, been almost invisible. In July 1980, IBM's top

managers in Armonk, New York, set up a taks force in Boca Raton, Florida, to report on the feasibility of urgently mass-producing a desktop computer. Philip "Don" Estridge was given just one year to get the secret project, code-named "Project Chess", into the marketplace. This left no time to do everything within IBM, so they chose an Intel processor and looked for an Operating System with open architecture to facilitate add-ons -- exactly as Kildall had designed -- then called not on Kildall and DRI in California, but on Bill Gates in Seattle.

Jack Sams, the IBM Operating Systems expert charged with buying the software, had visited Microsoft in July, and had been stunned to find that the "office boy" who came to greet him was "the most brilliant mind I have ever dealt with". (ROCHE> Yes, the one who wrote (twice!) publicly that the DAA instruction was useless... MITS "Computer Notes", September 1975, page 7. Instruction that still exists on IBM Clowns, 35 years later...) Sams wanted a license for the source code for CP/M because he had been impressed with its success in the Microsoft Softcard for Apple II. (ROCHE> Eureka! The IBM PC was designed to fight against the Apple II, not the hundreds of thousands microcomputers running under CP/M! But why the Apple II? Because, at the beginning, VisiCalc was running ONLY on the Apple II... and many customers of IBM were so stricken by the first Spread-Sheet of history that they were buying them "en masse", since they were so incredibly cheap, compared to their incredibly expensive Mainframes. IBM commercial engineers noticed those non-IBM "things" appearing in their customers's offices, and IBM decided to kill it. Now, you know the REAL origin of the IBM Clown. If you study the history of the IBM PC, you will find that the first version had no floppy disk drive (!), but a magnetic cassette interface, 64KB of RAM, and a ROM-version of BASIC. That is to say: the specifications of the Apple II...) "I just assumed Microsoft had a license to market the CP/M source code", Sams told me. (ROCHE>??? Had the customers of IBM a license to market the source code of IBM's Operating System? This answer is, simply, incredible.) "But I also very much wanted Microsoft's languages." (ROCHE> This sounds more logical.) When Sams flew cross country again to Seattle, on a Wednesday in August, he brought with him a whole IBM team. Having ensured that Gates and his partner, Steve Ballmer, signed a tight confidentiality agreement and a consulting agreement, they opened negotations to buy a license for CP/M from Microsoft. Hello? Gates had to say it was not his system to license. "IBM had not done their homework", said Gates later. Gates, there and then, phoned Kildall, only to say that a "big client" was going to contact DRI, and that Kildall should "treat them well". Sams took the phone to schedule a meeting with DRI two days later.

This is where the myth begins. In his memoir, Kildall is quite specific (and Rolander confirms) that he arranged to meet the "Project Chess" team on a Friday afternoon. Knowing and explaining that he had a previously-scheduled business trip on Friday morning (taking urgent documentation to CP/M distributor Bill Godbout at his factory in Oakland), he arranged an initial meeting between the visitors and Dorothy, who negociated contracts.

The IBM team showed up as scheduled at 10 a.m., and the lawyer, well-known for his aggresive style, presented Dorothy with a ludricrously far-reaching non-disclosure agreement. According to Kildall, it stated: "All Ideas, Inventions, or other Information become the property of IBM." Anything IBM said would be confidential, whereas anything DRI said was not. For its part, IBM was unhappy

that every page of the DRI manuals was stamped: "Confidential". Dorothy gave the IBM team DRI's standard licensing agreement, which Kildall says more than 1,000 manufacturers had already signed. There was a stalemate for a few hours. Dorothy would not sign IBM's broad agreement without knowing what IBM wanted. IBM would not reveal what it wanted until DRI signed. Dorothy sought the advice of Gerry Davis. He agreed with her that the undertaking asked for was too broad, but thought it might be modified. He says, "Bill [Gates] signed that agreement because he had nothing to lose, because he did not have an Operating System."

Dorothy decided not to negotiate further until Kildall came back for the afternoon session. In the meantime, it appears, the IBM team fumed. There is an exponential arc to the revisionism that was to so amaze and dismay Kildall. In an interview with the "Times" of London in 1982, Gates is reported as saying: "Gary was out flying when IBM came to meet, and that is why they did not get the contract". Robert Cringely's "Accidental Empires" (1992), one of the seminal works on Silicon Valley, states that Kildall never bothered to show up at all. The Long Island newspaper "Newsday" wrote: "In a story often told, the starched-shirt IBM guys, after CP/M long-hairs canceled an appointment, turned to an unknown company called Microsoft, headed by an unknown computer geek named Bill Gates." (On a smaller point of accuracy, Tom Rolander was quite bald by that time.) The source for the absent-Kildall story is Sams. In 1992, he told Wallace and Erikson he was sure Kildall did not turn up for the meeting, "unless he was there pretending to be someone else". (ROCHE> The behavior of Sams is really more and more curious.) When I asked Sams about this in the light of Kildall's memoir to the contrary, he conceded: "I believed we had not met, but he may have been one of the three or four people round the table." Alfred Chandler Jr., who does not doubt Kildall's presence, writes in his 2001 book "Inventing the Electronic Century": "Kildall was unwilling to sign the standard non-disclosure agreements on which IBM insisted... If Kildall had been willing to accept the non-disclosure clause, and if Motorola's chip had been the first choice over Intel's commercially unpopular one, the underlying history of the Personal Computer during the critical decade of the 1980s might have remained much the same. But the industry's two most powerful players in the 1990s might not have been Intel and Microsoft." David Kaplan explains the prevalence of these stories: "That's the Microsoft, and popular, version -- and since winners tend to write history, it is the prevailing one."

This is what Kildall has to say about his role as the invisible man: "Tom Rolander and I flew back in the early afternoon, to join the IBM discussion. The group consisted of me, the IBM Chess Team, Dorothy, and attorney Gerry Davis. Tom was around for a time. The non-disclosure agreement was signed, and the IBM team revealed their plans for their new Personal Computer."

There is a conflict on timing, here. Sams maintains that, at this first meeting, he did not reveal IBM's plans for a PC (ROCHE> While he had done so with Microsoft, which had no Operating System, 2 days earlier... The behavior of Sams, the "Operating System expert", is really more and more curious!), but one of the key DRI staffers (Kathy Strutynski, see below) has a distinct memory of Kildall talking about the PC only a week or so later. John Katsaros was not with the company at the time of the 1980 meeting but, in July 1981, six months after he took over marketing, Dorothy let him see an agreement Kildall had signed at the first meeting with IBM. "What I saw was a short one-

page document saying that Digital Research would not disclose to others that IBM came by for a discussion. I was told that one of the hang-ups was the fact that Digital Research manuals contained the word "Confidential" on every page, and that IBM would not accept these manuals without their non-disclosure agreement being signed. Dorothy told me that, when IBM left the meeting, they said she should call them whenever DRI was ready to proceed."

At this meeting, that the conventional histories say never took place, Kildall writes that Rolander presented Frank Holsworth's new MP/M-86 for DRI, the Multi-Tasking Operating System that they envisaged working with Intel's 16-bit computers. (ROCHE> Can you imagine an "Operating System expert" not realizing the power of MP/M-86, and forgetting it?) Kildall says he also briefed the IBM team on DRI's transitional Operating System, CP/M-86, to help CP/M customers move to new 16-bit Intel chips.

Ultimately, though, Kildall wanted MP/M-86 to become the new standard. He writes: "The new MP/M-86 was the Operating System for the future, because it had built-in Multi-Tasking that supported the existing software base. It had built-in networking. Only today [1994] are we even considering these prospects. Clearly, the PC industry would be much more advanced if DRI had been allowed to introduce these products a decade ago." According to Rolander, Kildall felt uncomfortable around the stiff, overdressed (by California standards) IBM men. They probably saw him as a hippie. Despite the awkward beginning to the IBM meeting, and the arguments over the wording of the non-disclosure agreement, Kildall still believed they could strike a deal. As far as he knew, no one else had an Operating System that IBM could use on its hardware. ("We looked at Unix and other possibilities", Sams told me, "but all were too big or too slow for our machine.") (ROCHE> Can you imagine? An "Operating System expert" having been presented with several people using several programs AT THE SAME TIME on a 4-MHz chip, and forgetting it?) Kildall writes: "We broke from discussions but, nevertheless, handshaking in general agreement on making a deal."

Kathy Strutynski, the DRI CP/M-86 project leader, told me that, around 8 p.m. that evening, she had a call at her home from Bill Gates. "He asked me to contact Gary and urge him to restart negotiations with IBM." But she could not find Kildall, for the good reason that he and his family had, that evening, set out for a vacation in the Carribean. On the initial flight to Florida, the Kildalls ran into the IBM team returning to Boca Raton. Kildall says he spent much of the flight discussing how he would adapt CP/M-86 to IBM's needs. Dorothy described the team as "friendly". "One of them kissed me on the cheek", she told me. I asked Sams about this manifestation of the Kildalls. He said he missed it because he parted from his team headed for Boca Raton, and went to IBM in White Plains, New York. (ROCHE> Why?)

When Kathy Strutynski did finally pass on the message from Bill Gates "three or four days later", she says Kildall gave her three reasons why he had not already done a deal with IBM. "First, no large computer company had been successful in the PC market. Second, we owed loyalty to many manufacturers [OEMs] who had purchased our products and we could not conceal information about IBM. This was a major concern. Third, he did not want IBM employees stationed in our two office buildings." Strutynski comments: "I think these concerns of Gary's were valid, especially the second. Bill Gates did not have the business relationships with all these players that DRI had, and that gave

him free rein to do what he did. One could argue that loyalty is a bad business practice."

In his memoir, Kildall says that, when they returned to Monterey after a week away, "I repeatedly attempted to contact the Chess Team, headed by Jack Sams. Sams never returned any of my calls." (ROCHE> Yet one more curious behavior from Sams...) Sams told me: "He may have tried to call, and I may not have been where he was calling. I did make an appointment to see him around Thanksgiving." We now know that, by this time, Sams had gone back to Microsoft. Gerry Davis said DRI caught wind, later, that IBM might be talking to Gates again, but Kildall told him: "Bill is a friend of mine. He would not cut my throat." (ROCHE> How true!)

When IBM revisited Gates with news of the encounter with Kildall, Gates jumped in with the observation that Kildall had not yet finished designing CP/M for a 16-bit machine (ROCHE>??? You need a microscope to find the differences between CP/M-80 and CP/M-86. So, what "design" was difficult to finish?) and that Microsoft could, itself, meet IBM's requirements. He accepted IBM's deadline of October for delivery of a proposal. As soon as the IBM visitors had left, Paul Allen called Rod Brock at Seattle Computer, to say an OEM customer he could not name might be interested in QDOS and could Microsoft as licensing agent. Brock agreed. He ended up receiving $25,000 from the licensing deal.

Gates was taking three gambles. The first was whether he could meet IBM's tight schedule. The second was that Paterson's adaptation of Kildall's architecture risked a damaging legal suit: Gates never told IBM how close QDOS was to CP/M. "We had no idea it was similar", Sams told me. The third was that IBM might pull out. They had done that before; back in 1974, IBM had made a $10,000 PC, the IBM 5100, which failed to sell. (ROCHE> In big number: it was too expensive, as usual for IBM products. But I saw one which worked 17 years. Anyway, IBM could not "pull out". In 1974, there was no competition. In 1980, the Apple II, thanks to the VisiCalc spreed-sheet, was invading IBM's customers! So, it was anything but a gamble: it was a counter-attack.) "They seriously talked about cancelling the project, up until the last minute", says Gates (ROCHE> Rubbish!), "and we had put so many of the company resources into this thing." (ROCHE> That is to say: Gates had GAMBLED on the outcome. Would you invest in a man who has such a dangerous behavior?) Free of Kildall's limiting relationships, and with a keener appetite for business, Gates was willing to bet everything. His dominating vision, even then, was of building a company that would span all three sectors of the nascent industry -- Operating System, development tools, and office software -- establishing the kind of monopoly Kildall had feared in his lunchtime conversation with Jacqui Morby.

At the end of September, Gates and Ballmer flew to Boca Raton, to present their proposal for using Paterson's version of Kildall's interface program. On the drive from the airport to the meeting, Gates panicked when he realized he had forgotten a tie. (They stopped at a department store on the way in.) Gates understood how to behave around IBM. His culture meshed with theirs far better than Kildall's did. He had other advantages. Estridge, who was to die in a famous Dallas "wind sheer" crash in 1985, told Gates over lunch that, when IBM's new chief executive, John Opel, heard Microsoft might be involved with the PC, he enthused: "Oh, is that Mary Gates's boy's company?" Opel and Gates's mother served together on the board of the United Way. (ROCHE> What a

coincidence! So, IBM, the biggest computer maker in the world, was to bet its future in microcomputers with a "Harvard dropout" who had a version of a public domain programming language, while ignoring a Ph.D. in Computer Science whose name is still found, 35 years later, in books about compilers... Is it serious?)

Gates is reported to have done a bargain basement deal, licensing DOS (ROCHE> What's that? DOS is an abbreviation for Disk Operating System. Several companies had already used "DOS" in their product's names.) system for only $15,000, with almost nothing in royalties, but the agreement with IBM in November 1980 left him free to license the Operating System to other equipment manufacturers. IBM's acceptance of this has long been regarded as one of the greatest blunders in business history. Microsoft went back to Brock, and bought the system outright for $50,000 -- thereafter known as Microsoft-DOS (MS-DOS), or PC-DOS on the IBM machine. Sams's explanation is that IBM let Gates keep control because it had too many problems "being sued by people claiming we had sold their stuff. We had lost a series of suits on this, and so we did not want to have a product which was clearly someone else's product worked on by IBM people." (ROCHE> And, in such a context, he did not care how Gates could have invented, overnight, an Operating System? Who would believe this?) Of course, when MS-DOS eventually became the industry standard, Microsoft left IBM in its wake. Nobody had seen the size of the market quite as Gates had done. "We had no idea", Paterson has said, "that IBM was going to sell many of these computers." (ROCHE> I never bought an IBM Clown. During 15 years, I continued to use, totally alone, an Epson QX-10 running CP/M Plus. Now, I am given IBM Clowns, that I use under CP/M-86 Plus. I have never understood why people where buying technically (both in hardware and software) inferior computers for more money.) Neither Brock nor Paterson profited all that much for their enterprise. They had retained licenses on DOS (ROCHE> Which one?) after its sale to Gates but, when Brock later tried to sell the license, he was blocked by Microsoft. He sued for $60 million, and settled for a million. Paterson, too, was eventually to receive a million for giving up his DOS license; he took a job with Microsoft. Steve Ballmer, looking back, acknowledged the ironies: "Tim Paterson's Operating System, which saved the deal with IBM, was, well, adapted from Gary Kildall's CP/M." (ROCHE> Where is the irony? This is a FACT.)

None of this was known, at the time, at Pacific Grove. Kildall was relaxed about not hearing from Sams. He shared Silicon Valley's view of IBM as a dinosaur. "A lot of us, in the microcomputer world, in the early days," says Rolander, "saw IBM as all fluff and marketing, big, lumbering, slow, uninteresting, not clean, exciting, fast." (ROCHE> There is a famous saying among computer programmers: "An elephant is a mouse with an IBM Operating System.") In 1981, Kildall's CP/M ran on 90 percent of the roughly 500,000 or so Intel chip-based Personal Computers in existence. (Apple and Commodore were the exceptions, using their own proprietary system.) Where else could IBM go?

But, about six months after IBM's visit to Pacific Grove, Andy Johnson-Laird again alerted Kildall to dissemination of his system. Now, it was not a small computer company, but the great IBM! So that software developers would know how to write programs for its still-secret project, IBM had to let selected programmers have a list of API function calls. Kildall was angry to find how much of CP/M's proprietary list appeared, there. He had no idea IBM had a deal with Gates. He was just upset that IBM itself seemed to have copied his

interfaces. In his unpublished memoir, he says he furiously got through to IBM. They immediately dispatched a manager and an attorney to Pacific Grove. "I showed the IBM attorney definitive evidence that PC-DOS was a clone of CP/M, and immediately threatened a lawsuit for copyright infringement. The IBM attorney compared the API interface, and I can clearly that he fairly blanched at the comparison and stated that he was not aware of the similarity. I told him that he should take note and become aware at the earliest opportunity, or else he should face a major lawsuit."

The upshot was that Don Estridge -- The "Don of Boca" in Johnson-Laird phrase -- immediately invited Kildall to Boca Raton. The decisive meeting was on July 21, 1981, when he had with him Gerry Davis, John Katsaros and Johnson-Laird. Katsaros was strongly in favor of suing. IBM knew it had to appease Kildall in some way, since a lawsuit for "injunctive relief" could, at the very least, delay its entire secret venture, due to be launched only the next month. Estridge at once offered to market CP/M-86 alongside PC-DOS. Kildall writes: "IBM offered to buy out CP/M for its new PC for $250,000. You might be saying: 'Hey, Gary, sell the whole damn thing to IBM, then just wrap MP/M on top of that, say hey?' That strategy may have worked [for IBM]. So, I countered with a $10-per-copy royalty for CP/M -- as was paid by all other manufacturers." Gerry Davis points out that DRI had contracts with "most favored nation" clauses, meaning that, to sell CP/M to IBM for a flat flee, might have caused DRI to be sued by its other customers. When they turned to discuss the retail price to be charged for the PC with CP/M-86, IBM insisted they could not agree to set a price. "They told us", said Davis, "they feared it would be a violation of antitrust laws." In the end, Kildall thought he was getting exactly what he wanted. "CP/M-86" would not be changed to "PC-DOS", and IBM accepted that it would market CP/M and pay DRI. "Once the heavy negotiations were over", Johnson-Laird says, "Estridge, who had presided like a benevolent dictator, gave us a demonstration of the new PC, with a color display and two 160-Kilobyte floppy drives. It looked like a toy. I had no idea that it would sell in the millions." (ROCHE> Notice that, again, all the people who were used to something better, like single-user single-tasking CP/M or multi-user multi-tasking MP/M, judged the IBM Clown to be a toy... And why would IBM, the Giant of computer Mainframes, want to introduce a toy? Because its specifications matched the one of the Apple II... which was the real target.)

Kildall left the details of the contract to Gerry Davis, who worked into the small hours with Estridge, both dictating to a secretary. "Unhappily", Davis remembers, "she was typing on a magnetic card, it got knocked, so that all the data was lost, and we had to do it all over again." A final sticking point, Davis recalls, was that IBM wanted a guarantee it would not be sued for infringement of CP/M copyrights. Kildall agreed.

Immediately after this agreement, IBM sent their prototype machine to Kildall, so that CP/M-86 could be installed. IBM had not yet announced their project to the general public, and was in paranoid mode. Kildall was told the prototype had to be chained to a desk, strong locks put on the doors of the bedrooms where the DRI developers would work, and shutters installed, so that neighbors could not see in and take pictures. No phone was allowed nearby. Any document printed out had to be shredded and burned. Several times, IBM technicians appeared on nearby roofs, armed with special meters to detect if anyone was able to eavesdrop on the electromagnetic signals emitted by the new computer's keyboard. Johnson-Laird, who had been invited by Kildall to install CP/M on

the prototype, recalls: "I sat in this secure room equipped with a three-drawer safe with a giant steel bar and padlock running down the front. I do not know what it was for but, in the spirit of the enterprise, I locked my sandwiches in it. I know they would have been impressed."

His very first hour on the prototype startled him. "I put in the IBM floppy, and was stunned by what I saw. There, in the 'boot' sector, the very first sector of data on the diskette, was the name of Bob O'Rear. I knew him to be a Microsoft programmer. I called Gary up to look. He went ashen." The shock to Kildall was, suddenly, to find that Microsoft was in bed with IBM, and must have been so secretly before Kildall's angry visit to Boca Raton. Bill Gates, of course, had signed the celebrated IBM confidential document -- "Bill kept our deal very secret", Sams affirmed. Kildall was not only alarmed by that collusion, but worried that Microsoft might license the cloned software to other hardware vendors, meaning DRI would be facing competition in a market Gates had previously left to DRI, while DRI stayed out of languages. The non-litigious Kildall could have sued Microsoft, but took comfort in the fact that both PC-DOS and CP/M would be sold with the new PC. The marketplace would decide the victor, and he had no doubt of DRI's technical superiority. What he did not know was that Gates had licensed MS-DOS with almost no royalty, nor how deeply committed IBM was to Microsoft. Sams told me they preferred to deal with a single supplier. For his part, Gates was reported to be furious his old friend had been allowed back in the game, insisting that IBM had been "blackmailed into it".

In August 1981, IBM's PC finally came out. Rolander remembers driving with Kildall to the nearest store, both of them brimming with excitement. They knew a knife had been plunged in their backs the moment they saw the labels on the software boxes: Microsoft's price advantage was a multiple of six. IBM asked $240 for CP/M-86, and only $40 for Microsoft's PC-DOS. (ROCHE> PC-DOS was the name of the Operating System sold by IBM, and MS-DOS the one sold by Microsoft.) Rolander says seeing the price difference was probably the biggest shock of his life. "It was just as if I were to reach across the table, right now, and give a slap across the face, something completely off the wall. Looking at the price and knowing you had been completely screwed, that there was no intention whatsoever on their part to sell CP/M-86. No intention at all. There was such a trusting nature, especially in the academic world that was collegial. This was so big-business, agressive, killer." He and Kildall felt so naive. They called IBM to demand the company reduce the price of CP/M-86, but no one called back. Gerry Davis says: "IBM clearly betrayed the impression they gave Gary and me." (ROCHE> Why IBM did not want CP/M-86 and, even more MP/M-86? Because THEY WERE TOO MUCH POWERFUL. IBM wanted an Apple II-killer, not something even more appealing! Else, how would you explain that one "Operating System expert" would not be attracted by one multi-user, multi-tasking Operating System already running?)

Kildall writes: "The pricing difference set by IBM killed CP/M-86. I believe, to this day, the entire scenario was contrived by IBM, to garner the existing standard at almost no cost. Fundamental to this conspiracy was the plan to obtain the waiver for their own PC-DOS produced by Microsoft." As psychiatrists like to say, even paranoids are persecuted. Kildall clearly was.

Yet another explanation of what happened came from Bill Gates in an interview with "PC Magazine" in 1997. (ROCHE> Who would believe an explanation from the

man who benefited from the crime?) He said: "The IBM guys flew down there, and they could not get the non-disclosure signed. Because IBM non-disclosures are pretty unreasonable. It is very one-sided. And we just went ahead and signed the thing. But they did not. Subsequently, Digital Research woke up to the fact that this was a pretty important project, and convinced IBM to also offer their product. But they priced it very high." There are, of course, two problems with these two sentences. First, by the time the PC was launched, Kildall had clearly signed a non-disclosure agreement. Second, who is "they" -- IBM or Kildall? The implication that DRI itself set the price of the retail product is misleading. Though Kildall had asked for a ten-dollar royalty, IBM could have priced both products equally, or with a ten-dollar difference. The obvious question is why Kildall did not sue Microsoft. That hectic August, Kildall flew to Seattle with Katsaros, to confront Gates and Allen. He writes: "Allen was worried about a lawsuit, and asked if DRI had ever sued anyone over copying CP/M. I said I had not. I was telling the truth. Paul is a gentle person, but he saw my chink and said that we were now engaged in OS-Wars."

By the time he wrote his memoir, Kildall saw the decision not to sue as a fateful error. He grew increasingly irate about the similarity of PC-DOS and CP/M. (ROCHE> PC-DOS was IBM's version, written by Microsoft...) He writes: "The first twenty-six function calls of the API in Gates's PC-DOS are identical to and taken directly from the CP/M proprietary documents [CP/M manuals]." Then, he poses a challenge for his old friend: "If you think Bill Gates invented those function calls, ask him why "Print String" (function 9) ends with a dollar sign. He will not know." (Bill Gates's office said he felt unable to give me a personal interview for the first edition of this book but, after publication, a letter from Microsoft said: "Your book implies that Gates' and Kildall's relationship was competitive and contentious in nature. Mr. Gates has stated on the record, on a number of occasions, that he had a good relationship with Gary Kildall, and valued his contributions to the industry.") (ROCHE> I like the hypocrisy of this letter. What "contributions to the industry" Bill Gates made? He did not invent BASIC (Dartmouth College), he did not invent MS-DOS (QDOS), he did not invent Xenix (Unix), he did not invent Windows (Apple Macintosh), he did not invent the CD-ROM (KnowledgeSet), he did not invent the Internet (Netscape Communicator)... Yet, he is the richest man in the world! Is it just?)

What Paterson, essentially, did was rewrite the bottom part of the software -- improving the way files were stored (ROCHE>??? Is the author stupid, or has he NEVER heard of FAT crashes? CP/M is bullet-proof. I have used CP/M Plus during 15 years, and never lost a file. It is impossible for me to say the same for MS-DOS and Windows. It is against the truth. But who needs the truth when there is the marketing of Microsoft?) and adapting the program to a 16-bit machine (ROCHE> Everybody knows how to do it. There are even programs (XLT-86) which do this automatically. Paterson had a Z-80-to-8086 translator. Have a look to the registers, and you will see that it can be done automatically.) -- while copying most of the top part (ROCHE> That is to say: the most important part.) of Kildall's Operating System interfacing mechanisms. Even if QDOS and CP/M were 80 percent different (ROCHE> Which is impossible.), as Peterson has insisted, he took almost unaltered Kildall's interrupt mechanism -- the key innovation. (ROCHE> I am afraid that the author does not understand, on a technical level, how an Operating System, and CP/M is particular, works. The "interrupt mechanism" is part of the hardware, decided by the manufacturer (Intel, in this case). What we are talking are the ways to

have an Operating System obey its user. In the case of CP/M, you have two parts: the BDOS and the BIOS. (In addition, on the IBM Clown, there is a ROM BIOS, providing lots of functions directly from the hardware.) So, the BIOS is quite small on the IBM Clown, thanks to the ROM BIOS. What remains is the BDOS, the Heart of the Operating System, without which no computer can work. And Paterson's QDOS copied the first 26 BDOS calls... when he could have invented any other way. So, Paterson simply copied a Copyrighted program. That's all. And, of course, Bill Gates knew it.) One curious feature of the systems is that both CP/M and DOS began each new line with A>. (ROCHE>??? What is curious? QDOS was a copy of CP/M, and CP/M was typical of minicomputers's Operating Systems of its time, down to its "system prompt". See the DECsystem-10 "Monitor".) Paterson's original 86-DOS, or QDOS, began with the slightly different A:. After Microsoft acquired 86-DOS rights, the prompt was changed back to being identical with CP/M's A>, thereby eliminating this slight cosmetic difference. To demonstrate how far Paterson mimicked CP/M's interface, the first 36 Interrupt 21 functions, Kildall's memoir devotes an appendix to comparing the sequence and language of CP/M, and those of QDOS and MS-DOS. A few words were changed. Kildall's "Read Sequential" function became "Sequential Read"; "Write Sequential" became "Sequential Write"; "Read Random" was called "Random Read". And so on.

In addition, The PC-DOS EDLIN editor program was almost the same as CP/M's ED program. (ROCHE> Something with can ONLY be the result of a disassembly... So, why would Paterson have disassembled a program two times bigger than the BDOS, and not the BDOS itself? Nonsense.) Paterson has continued to justify his work with varying degrees of emphasis on his claims to originality. "This was a real product", he told Wallace and Erikson. "Everyone always thinks IBM was the first to have it. That's crap. We shipped it a year before they did." In a 1997 "Forbes" magazine article under Paterson's byline, he said: "I was 24 when I wrote DOS. It is an accomplishment that probably cannot be repeated by anyone else. (ROCHE> *BULLSHIT!*) More copies of DOS have been sold than any other program in history." (ROCHE> But not QDOS... QDOS only dealt with floppy disk drives, and everybody knows that MS-DOS Version 2 became a bastard of CP/M and Unix, to manage hard disks.)

A year later, Paterson protested his apparent boastfulness. "That makes me sound egomaniacal", he told Doug Conner, who interviewed Paterson for "MicroNews" in 1998, when Paterson was in his eight year as an employee of Microsoft. Conner remarked that it was surprising Paterson was, then, not as well-known as his computer system -- surprising in light of the fact "that he sometimes bears the heavy mantle: 'The Father of DOS'." Conner writes: "It is a quieter celebrity the amiable software design engineer carries around, and it is a celebrity he is comfortable with -- when the stories are accurate. He squirms, for instance, at the implication that he is fixated on his authorship of DOS." To that title, "Father of DOS", he reports Paterson responding as follows: "I prefer 'original author'", he explains, "I don't like the word 'inventor' because it implies a certain level of creativity that was not really the case. (ROCHE> This is an understatement...) Besides," he laughs, "there are enough people who think it is nothing to be proud of. If I say: 'I invented DOS', they say: 'Well, good for you, Sucker.'" Rolander observes: "If Tim did not consider himself the inventor of MS-DOS, and felt his own creative contribution was minimal, it is hard to see how he can complain if his dead predecessor is given credit for showing the necessary inventiveness and creativity Paterson has declined."

In the same year of 1981, when IBM launched its PC, venture capitalists invested in DRI -- Jacqui Morby of TA Associates in Boston and the venture capital companies Hambrecht and Quist and Venrock Associates -- and they helped the company move into the big time with a new president, John Rowley, relieving Kildall of management. But the Board also dithered about suing, as time ran out under the statute of limitations. Gerry Davis had to advise them of the uncertainties -- the computer illiteracy of courts, at the time, and the deep pockets of IBM, which would have to back Microsoft. The Copyright Law of 1976 was not amended until 1981, specifically to cover the "look and fell" of software. Gerry Davis himself won one of the first cases, putting a Bay Area infringer of CP/M out of business. So, was it a mistake to hold back? "Yeah", says Davis now, "what we should have done, in retrospect, was gone in and sued Microsoft very early on, even with the uncertainty of the law, because it would have stopped the development of a competitor. And, if we had stopped them to begin with, they would never have gotten the foothold they have." Jacqui Morby agrees. But aggression ran contrary to Kildall's character. Davis remembers him saying: "It is not nice to sue people, and we are going to succeed, anyway." Everybody in the company was in denial for a couple of years, says Davis. "There was a lot of naivety on the part of a lot of us, the board, me, and then the venture capital people." Katsaros believes DRI should have gone back to IBM and asked for a repricing agreement, "but, by then, John Rowley was on board, and did not prioritize that." The complacency at DRI was understandable. In 1981, CP/M was used worldwide in close to 200,000 installations, with more than 3,000 different hardware configurations. (ROCHE> You will note that, now, there is only one hardware configuration: the IBM Clown, who froze the hardware. Remember the 80%-compatible IBM Clowns? This never happened during CP/M days, as the Microsoft Softcard (running on an 6502!) proves.) There were nearly 500 software products in the shops. (ROCHE> Today, nothing new exists, thanks to the IBM Clown (MS-DOS and Windows). Programs are still classified in 6 categories: 1) word-processor, 2) spread-sheet, 3) data base, 4) business graphics, 5) communications, and 6) programming. At the time, people where thinking that the 640KB of MS-DOS would lead to "integrated software" (one single program doing everything) but, 30 years later, Microsoft is still selling its programs separately...)

The company doubled its space, moving from the Victorian house to offices on Central Avenue. By the end of 1982, DRI employed more than 500 people, and had operations in Europe and Asia. Revenues skyrocketed from $6 million in 1981 to $44.6 million in 1983. Everyone was confident -- they knew that DRI's technology was superior, so it must surely prevail in the marketplace. Engineers at DRI had, under Gary's leadership, moved beyond CP/M and MS-DOS, which was based on it, and they had a poor view of the IBM machine itself. "That machine was a piece of crap", says Rolander, "compared to other machines. I would defy you to find anyone else who was around the industry 20 years ago who would have thought the IBM would be successful." Soon after the IBM machine came out, DRI engineers already had Concurrent CP/M as a single-user multi-tasker up and running. (ROCHE> Previously, there was MP/M-86, a multi-user multi-tasking Operating System for S-100 Bus computers. Concurrent CP/M was designed because of the limitations of the hardware of the IBM Clown which prevented several users to use the machine at the same time. The biggest limitation of the IBM Clown was its "memory mapped screen" as it limited the number of people able to use the machine. That's why DRI was obliged to design a card to connect other monitors. At the beginning, "only" 3 terminals could

use Concurrent CP/M running on a single IBM Clown. At the end, up to 64 persons could use one single PC! Compare this with the slowness of Windows...) It did things computer users take for granted now, such as printing a file while editing a spread-sheet, or cutting and pasting between spread-sheet and text. The IBM-Microsoft Operating System, being single-tasking, did nothing like this. Sometime after the release of the IBM PC, Dan Davis and a team of engineers had visited the IBM PC team in Boca Raton, and demonstrated their PL/I and some other products on an IBM PC, including DRI Operating Systems. "We dit it to convince them that we had better system software than Bill Gates", recalls Davis. "The IBM engineers were simply flabbergasted that we had been able to create a multi-tasking Operating System on such a small (by IBM standards) machine. We demoed several programs running while, at the same time, we were printing to a printer, and the engineers said: 'How do you do this, without printer jobs interfering with each others?' We explained we had mutual-exclusion queues in the Operating System. We realized, talking to them, that not only were they very unfamiliar with microcomputer system software, they were not even that familiar with their own Operating System from Microsoft."

Still, IBM stuck to Microsoft, perhaps out of incomprehension, perhaps because it knew what a relentlessly-driven super-salesman Gates was, or the fact that it was making so much money with its virtually free PC-DOS and booming sales of its PC. IBM let it be known it would only provide further technical support for DOS. John Wharton concludes that IBM "consciously chose to kill CP/M-86 because it was machine-independent. There was clearly a strategic advantage in IBM promoting an Operating System that locked customer software into its, then, proprietary hardware." (ROCHE> Hum... I don't agree with everything, so I will only add that CP/M (a family of Operating Systems, 8- and 16-bits) was not only portable, but probably too much powerful for IBM, who probably did not want to threaten its own machines, just kill the Apple II. MS-DOS Version 1 was so pitiful that it was not a threat. It was not before Version 2 (with hierarchical directories, etc, from Unix) that MS-DOS became usable. Now, why did so many buy IBM Clowns, when the S-100 Bus computers were at their heyday? My Epson QX-10 was making circles around the IBM PC. I simply don't understand why people where buying such a bad computer.)

Kildall simmered, the tensions reflecting in his personal life. He and Dorothy separated and then divorced after 18 good years together; she opened a lovely guest ranch in Carmel Valley. How utterly maddening it must have been: With Microsoft and IBM controlling the market, Kildall could not push MP/M-86, the multi-tasking 16-bit version of CP/M. (ROCHE>Well, well... The author is mixing a few facts, here. CP/M was a single-user single-tasking OS. Its multi-user multi-tasking version was called MP/M. Both were running on S-100 Bus computers. Due to the limitations of the hardware of the IBM Clown, Gary Kildall had the idea of Concurrent CP/M, a version of MP/M-86 specially designed for the IBM Clown. The first version used BDOS Version 3, then added MS-DOS Version 1 compatibility, then MS-DOS Version 2 compatibility, etc.) "I was competing with an Operating System clone, MS-DOS, of my original design, and both Operating Systems were, by this time, completely out of date." (ROCHE> Because MS-DOS (and CP/M-86) were both using BDOS 2, not BDOS 3.) In Europe, at least, Kildall could push forward. Digital Research had four European offices, two in England, one in Paris and one in Munich. (ROCHE>??? Where is Digital Research Japan?) IBM and Microsoft had much less market clout abroad, and DRI's European operations kept the company afloat during the mid-

1980s. (ROCHE> In 1986, (English) Amstrad made the best selling CPC-6128 and PCW-8256. Amstrad was paying DRI only 1 Sterling Pound per machine... But it sold more than 1 million of both machine!) Paul Bailey, DRI's head of UK operations, beat out Microsoft for big accounts like Siemens (ROCHE> Which used CP/M-86 Plus under the name "Personal CP/M", a denomination also used for CP/M Version 2.8, a version of 8-bits CP/M in ROM that was designed to counter-attack the Japanese MSX machines.) and Nixdorf. DRI software was used to automate industry in Europe; (ROCHE> Both German companies produced components using ARCnet (a network only used by industry) via DR-Net and Personal CP/M.) Microsoft still could do only single-tasking, while DRI's software allowed manufacturers to track multiple pieces of data.

Wharton writes that the impression he got of Microsoft programmers, at the time, was that they were "untrained, undisciplined... They did not seem to appreciate the importance of defining Operating Systems and user interfaces with an eye to the future. In the end, it was the latter vision, I feel, that set Gary Kildall so far apart from his peers." (ROCHE>??? Who else had an Operating System, in 1981? QDOS = MS-DOS = clone of CP/M, while he had MP/M running in 1978... Gary Kildall had no peers.) What Kildall saw, and what Paterson, Gates and IBM did not, is that CP/M-86, itself, would soon be antiquated. (ROCHE> That's why Gary pushed and pushed MP/M-86!) The real problem for computer users was not that QDOS was similar to CP/M, but that it did not have the stable multi-tasking capabilities that Kildall was developing. (ROCHE> Totally false! MP/M was running THREE YEARS before the IBM Clown! Re-read the microcomputer magazines of the time!) Dan Davis says he has always believed that what depressed his friend and colleague in later years "was not so much that Bill Gates got undeserved credit for his creations, but that the vision Gary had for an industry he helped to create would never be realized".

While the salesmen fought the battle over Operating Systems, Kildall could not stop inventing and innovating. Videodiscs were still new -- they were the beginning of "multimedia" -- and he and Rolander pushed the boundaries to fashion interactive hardware and software for the Commodore 64 computer. They labeled the system "VidLink". Kildall astonished Grolier Publishing by storing Grolier's entire nine-million-word encyclopedia on a single videodisc. Grolier gave the go-ahead for Kildall to develop a commercial version of their "Academic American Encyclopedia" on videodisc. Ironically, the new management of DRI did not take the job, so Kildall and Rolander independently made the first encyclopedia videodisc in Kildall's garage. (ROCHE> And what was doing Microsoft, meanwhile? Nothing. It is only AFTER Gary Kildall demonstrated that CD-ROM were working that Bill Gates decided to use them on IBM Clowns...) In 1984, Kildall set up a new company with Rolander called "Activenture"; the name later changed to "KnowledgeSet". It was small, just like the early Digital Research, with Kildall and Rolander doing the engineering, and Kildall's new wife, Karen, doing the bookkeeping.

Kildall, ever prescient, set out in 1985 to build a CD-ROM version of the encyclopedia, called the "Grolier Electronic Encyclopedia". Rolander remarks: "This was in June of 1985. Here we are, 17 years later. At that point in time, we said, absolutely every new computer will have a CD-ROM drive. You will not be able to buy a new computer without a CD-ROM drive. And it took at least 10 years to get to the point where they were commonplace, and 12 or 13 before they were a standard device." KnowledgeSet made CD-ROMs for the Boeing 767

manuals, with vector drawings -- and Rolander's daughter, Kari, got an A+ on a paper, astounding the teachers with her knowledge of Costa Rica from the CD-ROM searches.

Bill Gates, not realizing who KnowledgeSet really was, wrote the company a letter, saying Microsoft might be interested in acquiring a CD-ROM firm. (ROCHE> Typical of Bill Gates: He never develop a new thing, just buy the company who created it.) When he discovered that Kildall was the man behind it, he wrote him what Kildall describes as "a fine letter". It is not clear whether Kildall is paraphrasing, but his memoir says it went like this: "Dear Gary, It has been a long time since we have been together. Next time you are in Seattle, maybe we can get together and go water-skiing, and talk about CD-ROMs." In the spring of 1985, Kildall visited Seattle to see his family and met Gates in a suite at the Olympic Four Seasons Hotel. The ever-generous Kildall writes that the meeting was pleasurable "and, for some reason, I opened up to Bill. I told him about the CD-ROM work that I was doing. We talked of standards. We talked for hours." Kildall mentioned his intention to hold a CD-ROM seminar at the Asilomar Conference Center, in Monterey, for publishers, and was somewhat taken aback, shortly afterward, when Gates invited him to be the (unpaid) keynote speaker at a $1,000-a-head Microsoft CD-ROM conference. Only when he had given his speech did he hear, from a Microsoft friend in the audience, that Gates had come straight back to his office from the Four Seasons meeting to order a conference, to preempt Kildall's. Kildall writes: "It was clever. It was divisive. It was manipulative. It is Bill Gates's nature. I must give him credit for being a very opportunistic person."

By 1984, DRI was enabling PC users to link their computers through a program primiarily designed by Joe Wein (ROCHE>???) called Concurrent DOS, a clone of MS-DOS with StarLink software. (ROCHE> "StarLink" was the name of the board used to connect additional terminals to an IBM Clown running under Concurrent CP/M.) You could buy one single IBM-compatible PC to serve as a hub to other PCs, linked by cable and with shared access to a common database. Again, Kildall was a decade ahead with PC networking. (ROCHE>??? The author seems to mix "multi-tasking" with "networking". Networking is via telecommunications software, like DR-Net, that was developed (among others) by Joe Wein... And DR-Net is a version of CP/NET-86, itself a 16-bit version of CP/NET for Good Old CP/M...)

By the middle of the decade, for all these innovations (ROCHE>??? The author forgets GSX, the PORTABLE graphics system for CP/M, and GEM, the Graphics-User Interface ("GUI") with pull-down menus and mouse... GEM is GSX Version 2.), DRI was losing its principal business against the muscle of IBM in alliance with Microsoft. The board fired John Rowley, and authorized Kildall to sell the company. Recognizing his responsibility to shareholders, he gritted his teeth and called Gates. Kildall flew his airplane to the San Francisco airport, and met Gates in the United Red Carpet Room. "This is a very sticky situation", he writes. "Bill, although once a good friend, had taken advantage of me at least twice. Bill appeared nearly on time at 2:00 in the afternoon. I learned what 'eating crow' means." No doubt fearing he might be taken advantage of again, Kildall gave Gates only public information, and suggested $26 million would be a fair price. Gates replied that DRI was probably worth only $10 million. "We parted friends for some reason I do not understand today. However, this rejection by Bill was one of his big business mistakes."

Kildall made one deal with Atari's Jack Tramiel for its graphic-display technology (ROCHE> GEM for the Atari 520ST.) and another with Kay Nishi, a Japanese programmer and entrepreneur who had fallen out with Gates. (ROCHE> Personal CP/M?) Many people, like Nishi, wanted DRI to create an MS-DOS direct competitor. DRI was the only company that could legally parallel DOS, Kildall believed, because DOS was simply "a derived work of CP/M". Microsoft seemed vulnerable, because it had not improved its Operating System (ROCHE> Still based on BDOS 2.), had done nothing to support the new larger disk drives until Compaq move in to do that and had failed to improve memory management for the larger applications programs (such as desktop publishing). (ROCHE> Compare the concepts of "memory model" of CP/M-86's CMD and MS-DOS's COM and EXE files, and find which one is technically primitive...)

When DRI's first version of DR-DOS was released, Kildall must have loved the irony that the company he founded was now selling a clone of MS-DOS. The new single-tasking (ROCHE> Maybe Version 1, but the doc available online now explains how to use it for multi-tasking...) Operating System was MS-DOS-compatible, and gave Microsoft a run for its money. On AUgust 6, 1989, Bill Gates wrote in an e-mail to Steve Ballmer: "DOS being cloned has had a dramatic impact on our pricing for DOS. I wonder if we would have it around 30-40% higher if it was not cloned. I bet we would!" This was a loss of millions of dollars. Users started calling DRI's new Operating System "Doctor DOS", not "Dee Are DOS", since it cured so many of the bugs found in MS-DOS. The August 1990 "Byte" magazine commented: "The latest incarnation of DR-DOS, Digital Research's MS-DOS clone, is an innovative and intriguing Operating System that is thoughtfully designed. Version 5.0 is also packed with the extra features that Microsoft's own Operating System should have (and might, eventually, have if the long-rumored MS-DOS 5.0 becomes a reality)." (ROCHE> Another particularity of Bill Gates is "vaporware": announcing programs 2 or 3 years before they arrive, that is to say: announcing a program that does not exist, but is started to been made AFTER announcing its arrival...)

Microsoft responded by announcing in May 1990 that, within a few months, it would issue a new release of MS-DOS that would catch up on the DRI system. Industry experience indicates that it would have been near impossible for Microsoft to so soon develop and release a commercial version. Nonetheless, Microsoft repeated this vaporware announcement throughout the Summer and into the Fall of 1990. In fact, MS-DOS 5.0 was not released until June 1991 and, when finally released, it did not offer the features Microsoft had promised.

On July 17, 1991, Ray Noorda, the founder of Novell, announced that his company was acquiring DRI -- not for the $26 million Kildall had asked or the $10 million Gates had offered, but for $120 million. Using DR-DOS and its networking software, Novell became one of Microsoft's biggest rivals. Now, Gates was up against a tougher opponent than Kildall. Noorda devoted himself to fighting Microsoft by acquiring a small start-up called "Caldera", which employed the Linux system, and he used Caldera as a battering ram to sue Microsoft for monopolistic practices. His petition noted that "QDOS borrowed heavily from an Operating System developed by Digital Research", but it concentrated on the "predatory" way Microsoft had cut DR-DOS sales by 91 percent. "This action", said Caldera's claim, "challenges illegal conduct by Microsoft calculated and intended to prevent and destroy competition in the computer software industry." Caldera alleged Microsoft would falsely announce

new software that did not exist, engage in exclusionary licensing, create false warning messages (ROCHE> True!), criticize DR-DOS, use product tying, and threaten customers who used DR-DOS with retaliation. According to Judge Dee Benson, who oversaw the lawsuit, "On September 23, 1991, IBM officially endorsed DR-DOS 6.0, which was scheduled to be released to the public in September or October of the same year. Plaintiff alleges that, in response to IBM's endorsement and in anticipation of an IBM/Novell alliance, Bill Gates publicly threatened retaliation against IBM, should it choose DR-DOS. Caldera claims that, as a result of the threatened retaliation and intense FUD [Fear, Uncertainty, Doubt campaign] concerning DR-DOS incompatibility with Windows, IBM withdrew its consideration of DR-DOS."

The lawsuit stretched three and a half years. On January 10, 2000, just weeks before the lawsuit was to go to a jury, Caldera and Microsoft settled. The deal was secret, but Microsoft announced a one-time charge against earnings of three cents per share. Observers of the case quickly noted that, since there were over five billion shares of Microsoft stock, that came to a charge of over $150 million. The "Wall Street Journal" estimated the cost of settlement at $275 million, but some estimates go up to half a billion.

Kildall and his second wife, Karen, had moved to Austin, Texas, in 1991 after the sale to Novell. Again, Kildall was ahead of his time, provoked by technical conundrums encountered even by an undaunted computer wizard. His son, Scott, created a desktop publishing system using the Apple Macintosh, impressing Kildall enough to want to give it a try himself. He found setting up his own Macintosh "one of the worst [experiences] of my life, except for the day I visited Philadelphia." Then, he wrestled with a Murata F-50 fax machine and found it "a switch-o-manic's nightmare", with 17 switches and such confusing instructions he ended up finding that his fax machine rang his personal phone day and night.

"OK", he writes, "so I am complaining about switches. How about proposing a solution to this stuff. I mean, plugging in a stereo these days seems to require a degree in electrical engineering. But there seems to be something on the horizon that may help. It is called digital wireless." Kildall set up a company called "Prometheus Light and Sound", working closely with Japanese company DDI, to exploit the fact that the one-dollar chips for cordless phones, communicating at 32KB in a frequency range around 1.9-GHz, could also be used for stereos, VCRs, security systems, heating "and you-name it, because, for the local area, you need no wires... Buy a stereo at macy's. Plug a unit into the wall and turn it on. No speaker connections. No CD player connections. No tuner connections. It just works... It just works."

He predicted: "Switches, cables, wiring. We cannot live with it in the future, because of the complexity of the interconnections. Wireless will solve part of this. Some 'switch standards' will solve the rest." He might have made another fortune. But making money was never what drove him. He had a beautiful lakeside ranch in the West Lake Hills suburb of Austin, a mansion with a splendid sea view in Pebble Beach, California, and all his fast toys, but his second marriage was heading toward divorce. He got some satisfaction from charitable work for pediatric AIDS, but the continual anointment of Bill Gates as the founder of the PC revolution finally got to him. Jim Warren says: "In his personal one-on-one candid comments to me, Gary was intensely upset and depressed about Bill Gates and what Microsoft had done. And it continued and

increased, unabated, until his death. Gary was a super good guy." Rolander remembers: "The more the fortune and influence of Bill Gates grew, the more he became obsessed. Day and night, the film of that day played in his head. It was not a question of money. What really hurt him was the myth. Gary felt no one accorded any importance to what he had accomplished."

Everywhere Kildall went, people would ask why he had "gone flying" the day IBM came. Cruelly, the University of Washington triggered an emotional decline. It invited Kildall -- surely its most lustrous graduate -- to the 25th anniversary celebration of UW's Computer Science program, but just as an ordinary member of the audience; and he was mortified to hear that they had asked Bill Gates -- "a generous donor" -- to be the speaker that evening. When Kildall rang to question that, the chairman of the Computer Science department hung up on him. Kildall writes: "The UW Computer Science Department educated me so that I could produce compilers like PL/M. Then I made CP/M a success through millions of copies sold throughout the world, again using my knowledge gained through education at the UW. Gates takes my work and makes it his own through divisive measures, at best. He made his 'cash cow', MS-DOS, from CP/M. So Gates, representing wealth and being proud of the fact that he is a Harvard dropout, without requirement for an education, delivers a lecture at the twenty-fifth reunion of the Computer Science class. Well, it seems to me that he did have an education to get there. It happened to be mine, not his."

So Kildall ends his manuscript.

His health deteriorated. When he was afflicted with arrhythmia of the heart, his doctor banned him from flying. Kildall gave Rolander his pilot's helmet. It was a bittersweet moment. He had so loved flying. Now, one of his last refuges was taken away from him.

During the Summer of 1994, he returned to Monterey for a visit. Shortly before midnight on Friday 8, 1994, he stumbled and hit his head inside the "Francklin Street Bar and Grill" in downtown Monterey. The place was packed, and he was found on the floor, next to a video game. He went to the hospital twice over the week-end, but was released. Doctors saw nothing wrong. (ROCHE> This is America. In France, a radio of the head would have been mandatory, before leaving.) Three days later, on July 11, he died of a cerebral hemorrhage. A blood clot had formed between his brain and skull.

He was 52. More than 300 people came to his memorial service at the Naval Postgraduate School. Bill Gates was not among them. Microsoft issued a statement that Kildall's passing was "a loss to the industry". Kildall's ashes were buried next to his father and grand-father, the sources of his love for teaching, not far from the lakefront where Gates was building his $60 million home.

Etched on Kildall's tombstone is a simple image: a floppy disk.

EOF