JANUARY 1979 Volume 4, Number 1 $2.00 in USA / $2.40 in Canada

BYTE

the small systems journal

Languages Forum

IPS, An Unorthodox High Level Language

About the Author and IPS

Karl Meinzer is a research physicist at the University of Marburg, in West Germany. He is a member of a very small, worldwide group of amateur satellite builders, who find an organizational home in the AMSAT project and its ties to the amateur radio community as well as various space agencies. Karl is the principal system designer for the high orbit "OSCAR" amateur radio satellites soon to be launched. His functions include the mechanical design, RF system design, computer system design and software design of the vehicle. IPS was developed in order to provide the AMSAT organization worldwide with machine independent OSCAR ground station software to be run on the typical project member's computer. The spaceborne version of IPS will run on an RCA 1802 processor in orbit, with 4 K (hopefully 16 K if technology becomes available) of on board memory and a radio communications bootstrapping method for loading from the ground control station. After the satellite is launched, IPS may indeed be one of the highest flying high level languages around.

Dr Karl Meinzer
Hoehenweg 38
355 Marburg 1
WEST GERMANY

It seems to me that the general confusion and lack of orientation regarding the best high level language for microcomputers is caused by the combination of two problems:

1. There is a marked lack of stated objectives regarding such a language.
2. It is tacitly assumed that the ease with which a feature can be added to a language is enough justification for adding the feature.

Let us look at the first point. In my experience, the term high level language has nearly the same meaning as "problem oriented" language. But oriented to what problems? It may help to classify computer applications into three large areas:

- *Mathematical problem solving:* Here the amount of data to be processed is usually small, but the data are subjected to relatively complicated numerical operations requiring high accuracy and diversity.
- *Commercial data processing:* Here large amounts of data are subjected to relatively simple procedures. Appropriate file handling techniques are essential.
- *Engineering applications in the wider sense:* By this I mean process control, systems programming, games, AI, robotics and any processes interacting with the real world. Programs in this area are characterized by moderate amounts of data and accuracy requirements. However, these programs usually have rather complicated flow of control.

I think that much of personal computing would fall into the third class. On the other hand, the programming languages advocated so far were really created for the first two groups. To complicate things further, a good programming language not only has to be optimum for the intended problem area, but it also has to appeal to the humans using it. After all, the purpose of a programming language is to be a tool of communication between the human way of understanding a problem and the unambiguous way a computer needs the problem to be explained. Here of course many different opinions are to be expected, particularly because there is a positive feedback mechanism. By using a given language, one adapts to its idiosyncrasies and then, in all truth, can claim the "superiority" of that language. Nevertheless, I feel there are some language properties that would hardly be considered controversial. They can be best expressed by some buzzwords: top down design, structured programming, modularity and good self-documentation. These properties are particularly important in problem areas where the intelligence of the program resides in the flow of control.

The second problem may be explained by having a look at the competitive environment in which languages are created. If a company wants to express the superiority of their product, what better way than to claim to have more language features than the other guy. With this approach to computer languages lots of problems are created. The added features not only require additional implementation effort and an increase in the amount of hardware required (ie: memory), but they also make the language mentally less manageable, both from the

Circle 120 on inquiry card.

user's and the implementer's points of view. In my opinion, PL/I, despite all its virtues, is a good example of where this philosophy leads.

We Need Extensibility, Not Features

It is noteworthy that the need of features in languages arises in the first place because of their restrictive nature. In the traditional data processing environment with many independent users, this is justified by having to prevent inadvertent interactions. But in the personal area, with typically only one user per computer, one may take a much more liberal attitude. If one user bombs, nobody else suffers. So instead of packing a language with features, it makes much more sense to provide only those capabilities which will be required in, say, 70% or more of all programs. *In addition the language should contain means to extend itself.* Thus, it will be possible to add additional capabilities, if required. Another holy cow in high level languages is the use of algebraic notation. Because it is relatively easy to implement, its desirability is hardly ever investigated. Admittedly, expressions in reverse Polish notation are somewhat less readable than the equivalent algebraic forms.

However, look what you gain by the explicit presence of a stack:

- Parameter passing between program modules becomes extremely simple.
- The problems regarding the initialization and range of validity of variables in blocks disappear. Local variables are kept on the stack without a name; only global variables are named.
- With variables there is a clear distinction between the "pot" and the "content of the pot." The explicit availability of addresses enables very fancy pointer calculations.
- Testing of program modules is extremely simple. By manually supplying parameters on the stack and observing the results, debugging is a snap.
- The language becomes completely procedural and the statements are executed strictly in the order in which they are written down. There is no guessing or experimental programming to learn what the compiler will do to your program.

I hope that these points show that the decision – reverse Polish notation or algebraic – should not be a closed matter.

Circle 172 on inquiry card.

Another point often overlooked is the fact that the language itself is only part of the total acceptability of an approach. The amount and ease of handling are at least as important. A compiler requiring five passes before an executable code is produced hardly could claim to be very convenient. I feel it is highly desirable that a system should be interactive. This goal includes the requirement that a compiler should only require a single pass; incremental compilation with an interpretive mode should be available to simplify debugging. Usually this translates into the fact that objects need to be defined before they can be referenced, a small penalty for a much more transparent system. (Probably the wide acceptance of BASIC and APL is based on their interactive handling despite their mismatch to engineering problems.) Also, note that a fast single pass translation system eliminates the need for code relocatability. Instead of relocating binary objects, one simply translates the source to a new position in memory; there is no separate loading process.

If one takes the points above as an outline for a language design, it turns out that amazingly few degrees of freedom are still left. The main remaining work is selecting a suitable set of primitive functions, from which the language is built. This cannot be done arbitrarily, as a lot of feedback is necessary from actual problems utilizing the language. Only that way can one assure that the instructions are powerful and self-documenting in a reverse Polish notation environment. Most importantly, the language should have as few underlying rules as possible; and these rules should be applied without exception.

An important question relates to the optimum implementation technique to be used. Essentially three approaches are possible. BASIC and APL are generally realized as interpreters. Classical interpreters are rather slow because the program has to be analyzed at execution time. The other approach, the compilers, produces machine code which executes without syntax and semantics analyzing overhead; although a good optimizing compiler usually is bulky and requires large amounts of memory. Otherwise, the code produced is less than optimal if the compiler is simple.

A third approach tries to combine the best of both techniques. It uses a set of high level instructions defined by code routines. A compiler produces a pseudocode, treating these code routines as the instruction set of a virtual high level machine. Since these

instructions (usually addresses to primitive code) need very little overhead to interpret, such a system is only 2 or 3 times slower than optimum assembler produced code. Assuming that this is acceptable, the technique results in an extremely compact pseudocode requiring significantly less memory than the other approaches. Furthermore, the pseudocode itself is machine independent; only the primitive routines present an interface to the underlying processor. This greatly facilitates program portability, since transcribing such a system to a new processor entails rewriting on the order of 1000 bytes of code (600 in the example which I have implemented). Of course the largest part of such a system is written in the language that is to be implemented.

My interest in computing stems from my involvement with the AMSAT Phase III satellite, a space project for the amateur radio service. For the on board computer as well as for the ground stations I wanted to design a system that met the criteria above and would allow total portability of the programs among the numerous different computers used by radio amateurs worldwide. Naturally the first thing to do, if one tackles a project of this sort, is to study the literature on the subject. A language that meets most of the points just mentioned has been published by CH Moore under the name of FORTH. Particularly, Moore's implementation technique is very ingenious. On the other hand, his user interaction mechanisms were too limited for our problem area. Thus, I rather took FORTH as a platform, from which I designed and implemented my own system which I named IPS (abbreviated from its German language name).

The hardware I had in mind when I designed IPS would typically consist of:

- An 8 bit processor. (The spacecraft computer uses the RCA COSMAC and most ground stations use the 8080 and some the 6502.)
- 16 K bytes of programmable memory. The IPS system itself requires a little less than 6 K bytes, so enough memory is left to the user.
- An ASCII keyboard and a 64 character by 16 line video display memory. The video memory makes possible much more transparent interactions than obtainable with a Teletype or a hard copy oriented system.
- A hardware input of a sort providing 20 ms timing signals. All timekeeping and scheduling is derived from this input.

- Only enough read only memory to provide a simple bootstrap loader. (The COSMAC based systems have no ROM at all due to a hardware load feature on the chip.)
- A cassette interface and one or two cassette recorders. Since all ground to spacecraft and spacecraft to ground communication takes place synchronously at 400 bits per second, the same rate is used for recording. Recordings have a fixed length of 512 bytes (half a TV screen) and may consist of ASCII characters or binary data. Synchronous in this context means that there are no start or stop bits, the data is self-clocking and a valid block is preceeded by a 31 bit long sync vector.

I want to give you now a cursory description of the language. In the context of this discussion, unfortunately, it will be impossible to give you a detailed introduction. Rather, I intend to familiarize you sufficiently for being able to walk with me thru a typical program. I hope that this way you will be able to get some feeling of the nature of IPS. First let me describe how you communicate with IPS. The display screen has 16 lines. The eight lower lines are used for input to the system, either manually or by tape. The upper eight lines are used by IPS for answers. The basic IPS quantity is the 16 bit signed integer. By typing 125, for example, this number is put on the stack and displayed in the first display line. Keying −20 now, you get the display 125 −20. Now keying * will result in −2500 being left on the stack. An arbitrary number of numbers may be held on the stack; operations always refer to the last entries on the stack regardless of the total number of numbers on it.

Besides the normal algebraic operators logical operators are also available to allow bit manipulations. Boolean operations use these operators as well. (Only the effect on the least significant bit is utilized.) Another class of operations allows one to manipulate the order of items on the stack, like DUP duplicating an entry or VERT, which interchanges the two top entries. [Note: *In present versions of IPS, German (the author's natural language) inspired the symbols of primitives. Thus, VERT is from the German word Vertauschen.]*

Numbers may be stored permanently with a name attached to them. They can either be defined as constants (KON), as variables (VAR) or ar arrays (FELD). Calling a constant by mentioning its name pushes its value on the stack. Calling a vari-

Listing 1: The complete text of an IPS program's source. The program used as an example is the software which converts ASCII to Selectric codes, formats a text file and prints that file on a Selectric typewriter. Please note that the IPS primitives used in this listing are based on the German version of the system; equivalent English language names of the primitives are often noted in the text of the article as various features are explained.

```
( IBM SCHREIBMASCHINENSTEUERUNG VOM 2 . 11. 77 )
0 KON CBA
HIER 2 + ? CBA ! HIER 68 + $H !
CBA    VAR CHARP
0      VAR LE
#1000 VAR SBP
#1000 VAR IBP
0      VAR BL
CODE REMOVE 14 LD IM #C PHI 0 LD IM #C PLO 8 LD IM #A PLO
      BEGIN  #D I/O LD MX #20 AND IM ( BELEGT? )
      D=0 Y? #C DEC #C GHI            ( 140 US LOOP )
     VERT D=0 END TH #FF LD IM #A STR 3 I/O #A DEC 4 I/O NEXT

CODE H/TRANSMIT 1 LD IM #D PLO BEGIN 8 LD IM #A PLO PS INC PS LD
A #FF XOR IM  #A STR 100 LD IM #C PHI 0 LD IM #C PLO #A INC
       BEGIN #D I/O LD MX #30 AND IM  ( FREI? )
       D=0 NOT Y? #C DEC  #C GHI
      VERT D=0 END NEXT
              TH #A DEC #D GLO D=0 Y? 3 I/O
                                   N: 4 I/O
                                   TH NEXT
CODE L/TRANSMIT 0 LD IM #D PLO 0 END
CODE MSTATUS PS ->X PS DEC #D I/O #A ->X
            PS DEC 0 LD IM PS STR NEXT
0      VAR LV
0      VAR S
0      VAR EINFL
4    FELD SC        #1020 #400C SC 2 !FK
18   FELD 3ZEICHEN
128  FELD T1

: INCR DUP @ 1 + VERT ! ;
: ADJ DUP @ #3FF UND #1000 ODER VERT ! ;
: LIES LADEFLAGGE @B NICHT DUP LV @ UND
       JA? IBP @ 512 + IBP ! IBP ADJ   S INCR 0 LV !
       DANN
       EINFL @ S @ 2 < UND UND ( LADEFLAGGE NICHT )
       JA? IBP @ DUP #200 + $LOAD 1 LV !
       DANN ;

: NORMALZEICHEN DUP MSTATUS EXO #80 UND =0
                JA? DUP #80 UND >0 ( DREHE KOPF ) JA?    1
                                                  NEIN:  2
                                                  DANN
                    H/TRANSMIT 0 4 JE NUN REMOVE 0 6 JE NUN
                DANN L/TRANSMIT CHARP INCR REMOVE ;


: SONDERBEH #20 - DUP 4 < JA? DUP 2 = JA? 1 LE !
                                    DANN
                               SC + @B H/TRANSMIT CHARP INCR
                               REMOVE
                          NEIN: CHARP @ 2 - CHARP !
                                4 - DUP DUP + + 3ZEICHEN +
                                CHARP @ 3 >>>
                          DANN ;
```

able or an array pushes its address on the stack; in order to get its value, the operator @ is used. It interchanges an address on the stack with the content of the address. To store a value into an address, the operator ! is used. It expects a value and an address on the stack, and by performing the storage, removes both. There are other similar operators like @B or !B, which perform like operations on single bytes.

Program modules are created by a colon followed by the name of the module. Then the actions are typed in as if the computer were to perform the actions immediately. A module is closed by a semicolon. Later on this module may be executed by typing its name or its execution may be put into another module simply by writing its name.

To control the flow of the program, IPS has some control words consistent with structured programming. The words JA? , NEIN: DANN (In English, respectively: YES? , NO: , THEN) are pretty self-explanatory. They roughly correspond to the IF THEN ELSE construct of other languages, only, to be consistent with reverse Polish notation, the test action precedes the JA? . There are three loop constructs available. The JE . . . NUN (in English, EACH . . . NOW) construct expects two numbers, a loop start index, and a loop limit. The loop is executed for each consecutive value, until the index exceeds the limit. If initially the limit is smaller than the start, the loop is not executed at all. This construct may also employ an increment different from one. The two other two loop constructs are intended for loops with an initially unknown number of repetitions. The ANFANG . . . ENDE? (in English, START . . . END?) structure is functionally equivalent to the DO UNTIL of PASCAL with the test at the end of the loop. The ANFANG . . . JA? . . . DANN/NOCHMAL (in English, START . . . YES? . . . THEN/AGAIN) structure corresponds to DO WHILE with the test at the beginning of the loop. Note that these constructs completely eliminate the need of GOTOs and labels.

Now I want to take you on a little walk thru a program. Its purpose is to read ASCII text recorded on tape cassettes and print them on my IBM Selectric typewriter. (As evidence of the practicality of IPS, this article's manuscript was printed by this program.) As mentioned, a tape block contains 512 characters (eight lines) without any control characters. The program, besides reformating the ASCII characters to the Selectric code, has to insert this control information. The program maintains a cyclic buffer of two blocks; S is a variable

Listing 1, continued:

```
: TYPLINE #0D ( CR/LF )  CHARP @ 1 + !B CBA CHARP ! 0 LE !
         ANFANG CHARP @ @B #7F UND T1 + @B
                DUP DUP #1F > VERT #30 < UND
                JA?   SONDERBEH
                NEIN: NORMALZEICHEN
                DANN LE @
         ENDE? ;

: SZEILE SBP @ CBA 64 >>> CBA 1 - CHARP !
         CBA DUP 63 + JE I @B #20 - >0
                        JA? I CHARP !
                        DANN
                     NUN TYPLINE ;


: SBLOCK 0 7 JE SZEILE   SBP @ 64 + SBP ! LIES
                MSTATUS 2 UND >0 ( STOP? )
                JA? 0 EINFL !
                DANN
             NUN SBP ADJ S @ 1 - S ! BL INCR ;


: SCHREIB EINFL @ JA? S @ >0
                     JA? SBLOCK BL @ 4 >
                         JA? 0 BL ! MSTATUS    ( START? )
                                  JA? 0 7 JE CBA 1 - CHARP !
                                          TYPLINE
                                      NUN
                                  NEIN: 0 EINFL !
                                  DANN
                             DANN
                        DANN
                   NEIN: 0 BL ! MSTATUS #80 UND =0
                              JA?  ( TIEFSTELLUNG BEI AUS )
                                   2 H/TRANSMIT REMOVE DANN
                         MSTATUS ( START? ) JA? 1 EINFL !
                                          DANN
                   DANN
       MSTATUS 2 UND >0
       JA? 0 EINFL !
       DANN LIES ;


( ENDE SCHREIBMASCHINE; SCHREIB IST EINGEHAENGT )


( CODETABELLE FUER IBM KOPF CORRESPONDENCE COURIER )
#2020   #2020   #2020   #2020   #2321   #2020   #2220   #2020
#2020   #2020   #2020   #2020   #2020   #2020   #2020   #2020
#B420   #87B7   #CF26   #58D7   #DE96   #4625   #0119   #C634
#7FCC   #3E37   #574F   #5E16   #071F   #FFC9   #BE27   #9928
#9C24   #9A82   #D2DB   #F9B8   #95C3   #93F0   #FCCA   #CCB2
#90D1   #C5DD   #BBF3   #84BD   #C0FA   #54F6   #7531   #8129
#1CD8   #1A02   #525B   #7938   #1543   #1370   #7C4A   #4C32
#1051   #455D   #3B73   #043D   #407A   #D476   #F5B1   #2049

   T1      64 !FK

#084F   #582B   #2B08   #0853   #2D2F   #2708   #082D   #6060
#2708

   3ZEICHEN 9 !FK     ( ENDE TABELLE )
```

indicating the number of blocks available for printing. S equals 0 means an empty buffer, S equals 2 is a full buffer.

Complete details are found in listing 1. Turn first to the last routine SCHREIB (in English, WRITE) which is the main program executed periodically. (Because of the previously mentioned referencing restriction, programs have to be read backwards.) SCHREIB at first checks if EINFLAG (ONFLAG) is set; if so, it starts the printing actions. This action starts by checking if S is larger than 0. Only if this is the case is there a block in the buffer and the routine SBLOCK (WRITE BLOCK) is called. A blockcounter is maintained (BL). If it exceeds four, printing is stopped (PAGE FULL) unless the start switch is on. In this case the 0 7 JE . . NUN does eight linefeeds (endless paper assumed). The other parts of the main program are less important, they mainly make sure that at the end of printing the head is in lower case because if left in upper case the machine cannot be manually switched into lower case.

Now let us look at the routine SBLOCK (WRITE BLOCK). It prints 8 lines (0 7 JE . . NUN) by calling the routine SZEILE (WRITE LINE) and then updating the write buffer pointer SBP by 64. Also it checks for an emergency stop. The routine SZEILE (WRITE LINE) takes 64 characters at SBP and transports them into an auxiliary array CBA (>>> is an array transport in IPS). Then the auxiliary array is scanned in such a way that the pointer CHARP points at the last printing character. The loop index is supplied by I. Then TYPLINE is called. TYPLINE inserts a CR/LF after the last printing character and then uses the characters in the array CBA without the most significant bit (#7F AND) as index to table T1 to find the Selectric code. If it is between #1F and #30 a special treatment is necessary by calling SONDERBEH, otherwise NORMALZEICHEN (NORMAL CHARACTER) is called.

Let's look at SONDERBEH (SPECIAL CHARACTERS). Some actions of the machine are not performed by the printing ball, but by special magnets (blank, for example). In my hardware, these codes have to be output to a different port. Some ASCII characters are not available on the printing ball. These are simulated by two other characters overwritten on each other. The auxiliary array is accordingly manipulated. Both activities are done by SONDERBEH. The NORMALZEICHEN routine checks if the print ball is in the proper case position for the character to be printed. If not, it first executes a head

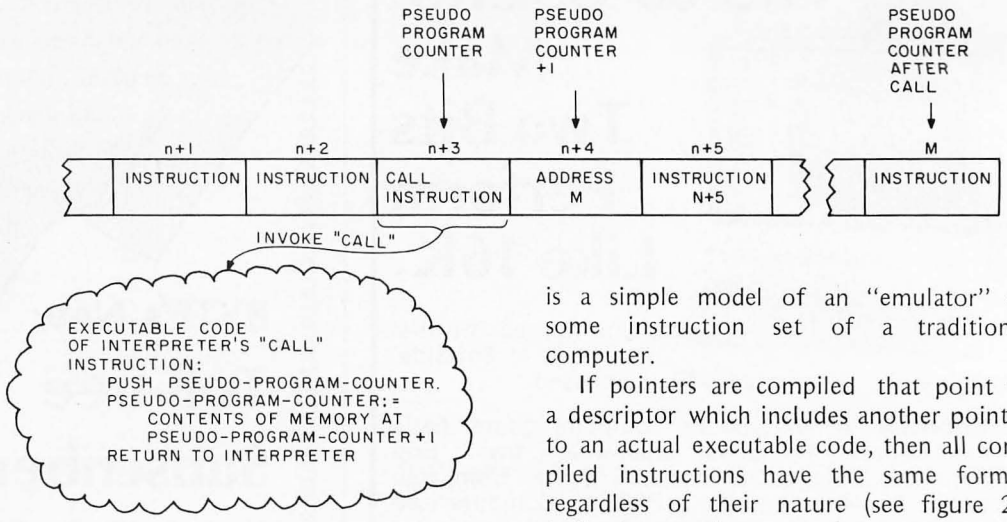

Figure 1: The temptation on the part of an interpreter writer is to implement direct execution as if the interpreter were simulating some form of traditional machine. Thus in this example, the operation code for a CALL to a subroutine is followed in the interpretive text by the address of the subroutine being called. This requires, for example, two entries in the string of interpretive text.

rotate action before printing. Each machine action is started by issuing the magnet code via H/TRANSMIT or L/TRANSMIT. These routines check the machine for "not busy" and then output the code to the magnets. The routine REMOVE does the opposite; if the machine acknowledges a code by "busy," the magnets are deenergized. Thus the necessary handshaking is established.

These interface routines are particularly noteworthy, because they are coded in assembler. IPS has an integral assembler; thus procedures dealing with special hardware or extremely time critical jobs can run at the maximum speed of the processor. This assembler also employs structured programming. The branch codes have been replaced by the Y? N: TH and BEGIN END mnemonics. The assembler, like the compiler, uses the stack to keep track of the addresses to be inserted into the branches. The routine LIES finally serves to read in the cassette blocks.

I hope this walk was not too tedious for you and was justified by giving you some insight into IPS programs and IPS-like languages. Let us have now a look at some selected topics regarding the implementation of IPS. These items maybe useful to you, if you want to tackle a similar project.

As mentioned earlier, the stack keeps 16 bit numbers. In addition to the normal operand stack, the design is simplified if there is a second stack, which is primarily used by the system to keep return addresses. This stack is available on a limited scale to the programmer as well.

A very useful trick in such a system is the use of "indirect execution." When implementing a simulated high level computer one is tempted to compile addresses of the appropriate code routines performing the instructions as shown in figure 1. This is a simple model of an "emulator" of some instruction set of a traditional computer.

If pointers are compiled that point to a descriptor which includes another pointer to an actual executable code, then all compiled instructions have the same format regardless of their nature (see figure 2). Indirect execution not only saves memory, it also simplifies the compiler because it needs no more checking the semantics of the instruction to be compiled. Thus in figure 2, a subroutine reference reduces to a pointer to the descriptor which begins the interpretive code of the subroutine. Thus, it boils down to having a descriptor at the head of each routine, defining its nature. This also makes the language naturally extensible since new descriptors may be defined anytime, and it eliminates the distinction between application routines and the primitive routines constituting the language.

In IPS the names along with the pointers to the pseudocode are kept in a hash table with a size for 512 entries. Experience shows that using this size in a 16 K system, memory and table saturate at about the same time. I preferred the hash table over a linked list because the compilation process is much faster and enables transcompilations for machines having no interaction facilities like the computer running the typewriter on which I printed this manuscript. (Names are separated from the program.)

Use of IPS in program development situations also requires the possibility of eliminating entries from memory. Generally a hash table would be unsatisfactory for this purpose. Nevertheless, since the program grows linearly in memory, the pointers to the routines in effect represent a secondary index by which the order of the entry and creation is available. Thus, pruning is no problem.

Reverse Polish notation languages can be essentially syntax free and by indirect execution the compiler can compile virtually blind, that is, without knowledge of what is being compiled. This is not exactly true, though. In fact, there are four different sorts of entries regarding the action the compiler is expected to take. This 2 bit

information is kept in the name table.

Normal entries execute in the interpretative mode and compile in the entry mode. See table 1 for the other modes. INT is used for variable definitions or for the colon to prevent definitions within definitions. PRIOR, on the other hand, is used for the JA? NEIN: DANN and the semicolon. These entries are executed at compilation time to handle jump addresses on the stack or to reset the compile flag. At the beginning of compilation (:), a special number is placed on the stack. At the end of compilation (;), this number is checked. If it is wrong there has been a structuring error (such as DANN following a JA?) forgotten and the entry is rejected.

At this point I would like to point out that much of the readability of structured programs seems to come from a geometrical representation of the source text as exemplified by the indented writing. Hardly any language software analyzes this geometry, though, and it is in effect redundant to the semantics of statements.

There is another useful technique which I call "pseudointerrupts." On a stack computer programs are naturally reentrant; and recursive programming is possible. Thus, the inner interpreter can be designed to accept "interrupts" between actions specified by interpretive code. These interrupts interrupt a stack oriented computer simulated by the interpreter, so no state saving operations are necessary (except for the pseudoprogram counter, of course.) Machine interrupts may also be implemented independently when required. The 20 ms clock input assumed by IPS forces such a pseudointerrupt to keep time in a normal clock and to service four "stopwatches."

Engineering problems usually require a few programs running independently of each other. I solved this problem by providing

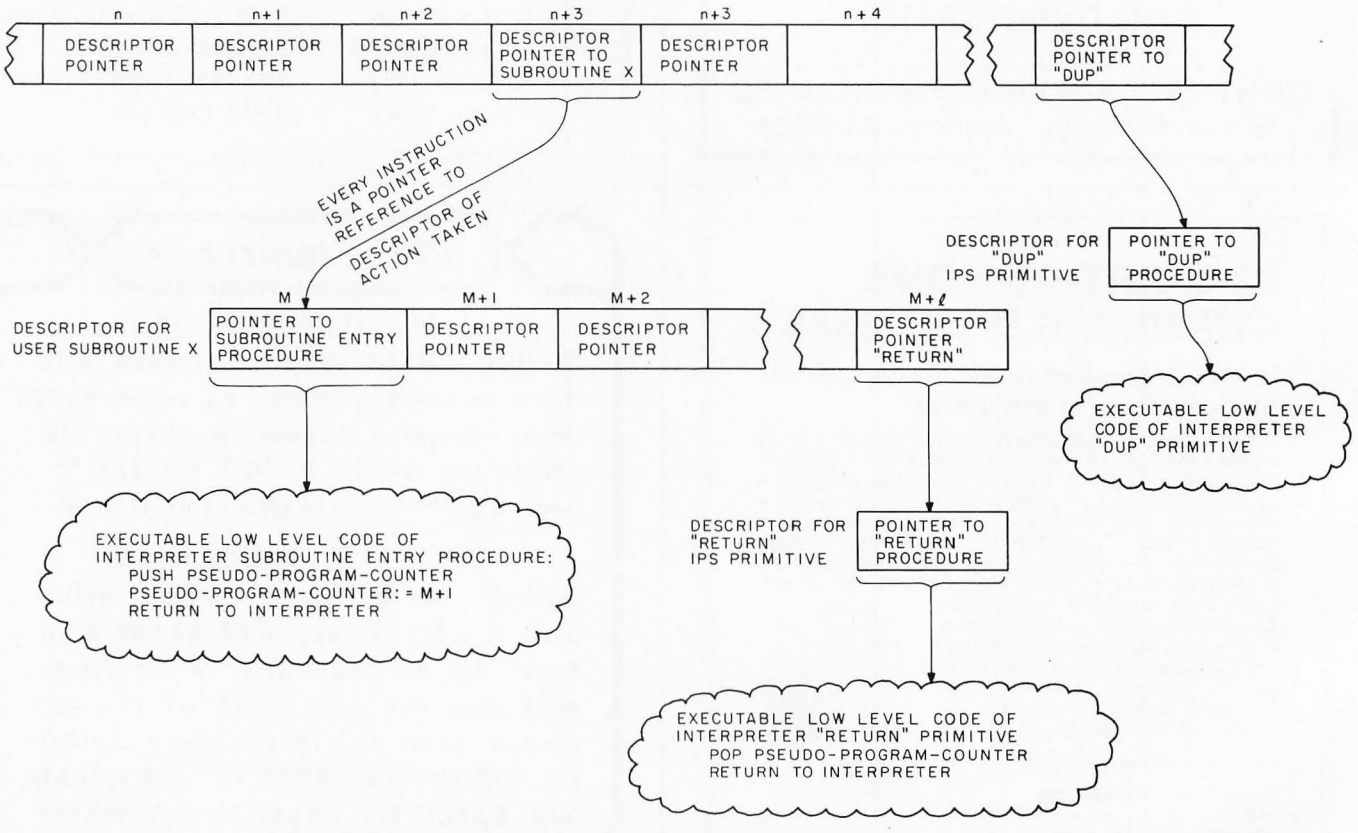

Figure 2: If a 2 level indirect form of interpretive execution is defined, then simplicity is achieved by making every token in the text of a user program a pointer reference to a descriptor of the operation. The descriptor in turn contains information such as the pointer to a low level routine to be executed upon reference to the descriptor during interpretation. Thus, a subroutine call which required two interpretive text entries in figure 1 is replaced by a simple pointer reference to the subroutine's descriptor. This descriptor in turn contains a reference to the low level interpreter routine which saves the program counter on the subroutine stack, then points to the first entry of the subroutine's interpretive text before resumption of interpretation. For simple built-in primitives of the language, the descriptor simply points to the low level executable routine of that primitive's operation, as shown for example in the case of RETURN and DUP. Note that the interpreter increments the pseudo program counter after each interpretation cycle.

a chain, that is, an array of programs executed in a cyclic fashion. Initially, this chain is only occupied by the compiler, while the other positions contain null entries. Programs may put other programs into this chain, or programs may remove themselves or other programs from this chain. It turns out that by this method, very general multiprogramming is possible without needing separate stacks for each chain member. The act of scheduling or descheduling a real time process is implemented by this chaining technique. Apart from the "chain in" and "chain out" operators two other operators are necessary. A suspend and a resume operator are provided to be able to wait for internal or external events before a program's execution is continued. Amazingly, the four chain operators eliminate the necessity of a centralized operating system without imposing undue restrictions.

Presently at AMSAT we have IPS versions for the RCA COSMAC and the 8080, and a 6502 version is in preparation. Also, there exists a special version for the COSMAC aboard the AMSAT satellite, performing all interactions by radio link. Except for the last they occupy a little less than 6 K bytes of memory, having everything resident including the text editor. The radio version is slightly smaller.

entry-type	compiler-mode: interpreting	compiling
Normal (:)	execute	compile
INT	execute	error action
PRIOR	error action	execute
HPRI	execute	execute

Note: error action marks input-word with question mark and stops further processing of input.

Table 1: The modes of IPS operation (interpreting or compiling) and the action the compiler takes depending on the type of entry found. Normal refers to standard modules (:), variables and constants.

In mid 1976 the first IPS version was up and running. Since then I have done some fine tuning as a result of feedback from some ten scientific projects using IPS within the University of Marburg. If you feel that IPS could be useful to you or if you have some comments or suggestions, please do write me.

(The original paper which discusses such a machine interpreter is: "FORTH, a New Way to Program a Minicomputer," by C H Moore, Astron Astrophys Suppl 15, 497-511, 1974. In the years since the first FORTH paper, FORTH has become a registered trademark of Mr Moore's consulting firm FORTH Inc.)■