A Block-Structured Language for Microcomputers

You think that there's no viable alternative to BASIC? You haven't heard about XPLO.

Larry Fish
123 E. Arkansas
Denver CO 80210

Ever since BASIC was introduced as a microcomputer language, there has been considerable discussion about an alternative high-level language. A number of languages have been suggested as alternatives; however, none of these languages is really suited to the needs of small systems. XPLO is the first viable alternative language for microcomputers.

Briefly, XPLO is a block-structured high-level language designed specifically for eight-bit microprocessors. It is five to ten times faster than the fastest BASIC. It is a compiled language and yet requires only 16K to 24K of memory, without a disk. But most important of all, it is available now.

Why Block Structured?

Block-structured languages were designed to solve problems that develop in conventional languages. If you have ever written a long BASIC program, you have probably encountered these problems.

```
'BEGIN'
MAKEANUMBER;
'WHILE' GUESS=INCORRECT 'DO'
                'BEGIN'
                        INPUTGUESS;
                        TESTGUESS;
                'END';
'END'
```

Program A.

```
'PROCEDURE'     MAKEANUMBER;
'BEGIN'
        NUMBER:=RANDOM(100);
'END';
```

Program B.

```
'PROCEDURE'     INPUTGUESS;
'BEGIN'
        GUESS:=INPUT(0);
'END';
```

Program C.

1. Complexity. As a program grows in length, its complexity grows geometrically. In languages where any routine can call any other routine, programs tend to become complex webs of subroutine calls. Block structure solves this problem by organizing the program into clean, logical blocks. As a block-structured program grows, it becomes longer but not more complex.

Block structure helps the programmer deal with complexity in another way. The human mind can only grasp a certain amount of information at one time. The easiest way to deal with any program is to break it down into simple, easy-to-understand steps. Even a complex program can be written easily by breaking it down into small modules. Here again, block structure naturally organizes programs into small, easily understood blocks.

2. Collision of Variables. As a program grows in size, more and more variables are used to store data or carry information. With more variables, it becomes easy for the programmer to lose track of what each variable is doing in each subroutine. Eventually, variables collide and you find that your Star Trek program is going out to lunch because the variable that holds the Enterprise's shield power is being eaten by the Klingon navigation routine.

In block-structured languages, the programmer has complete control of each variable. Block-structured languages allow each variable to be defined either locally or globally. This means that variables can be set up to be active only inside certain routines or active for all routines.

To make things clearer, let's write a small program in XPLO. Because of the structure of the language, we can begin by describing the task in plain English. The program we will write is a simple guessing game in which the computer selects a number between 1 and 100 and we try to guess the number. After each guess, the program will tell us whether we are high or low. Here are the steps the program goes through:

1. Think of a number.
2. Get a guess from the keyboard.
3. Test the guess against the computer's number.
4. Do steps 2 and 3 until the guess is correct.

The steps translated into XPLO are shown in Program A. Notice that the program is almost word-for-word the same as the step-by-step description of the task. First we MAKEANUMBER, and WHILE the GUESSES are INCORRECT we INPUT a GUESS and TEST the GUESS. BEGINs and ENDs are used to divide the program up into logical blocks. This part of the program has two logical blocks, one inside the other.

Obviously, there must be more to this program, since XPLO doesn't yet know how to make a number, input a guess or test the guess. Each of these operations is a subroutine to the main program. In XPLO, subroutines are called procedures. We are now going to

```
'PROCEDURE'      TESTGUESS;
'BEGIN'
        'IF' NUMBER=GUESS 'THEN'
        'BEGIN'
                TEXT(0,"CORRECT!!");
                TRY:=1;
        'END'
        'ELSE'
                'IF' NUMBER<GUESS 'THEN'
                TEXT(0,"TOO HIGH")
                'ELSE' TEXT(0,"TOO LOW");
CRLF(0);
'END';
```

Program D.

write each procedure as a complete program block (see Program B).

This procedure generates a random number and puts that number in the variable NUMBER. Program C gets a number from input device number zero and stores it in the variable GUESS. In XPLO, as many as ten different input and output devices can be called directly from the program. This allows XPLO to read and write data directly to disks, printers, CRTs, etc.

Program D is a bit more complicated, but it is still easy to understand. If the computer's NUMBER is equal to our GUESS, then we execute one block of code; if they are not equal, then we execute another block. If the numbers are equal, we tell the user that the guess is correct; if they are not equal, we test if the guess is high or low and tell the user.

The Program

There are two new constructs in the final version of the program (see program listing). CODE allows the programmer to assign names to XPLO functions. For example, the word RANDOM calls the random number function. The programmer need only use those functions necessary to the task and can assign names that add clarity and readability to the program.

INTEGER assigns a name and memory space for each of the variables. Because of the way in which the variables have been set up in this program, each of the variables can be used by any procedure. If we had defined the variables inside a procedure block, the variables would have been active only within the procedure.

Block-structured programs can be thought of as a series of boxes. Each box has only one entrance and only one exit. The program enters at BEGIN and exits through END. Each box can contain sub-boxes, executable statements or calls to procedures (see Fig. 1).

Our program consists of four boxes: three subroutines and a main program box. Each block is a simple, complete operation. Programs are built a brick at a time from these elementary blocks. Even the most complicated programs, such as assemblers and compilers, can be constructed from simple blocks.

Notice that the main procedure is the last block in the program. Reading an XPLO program starts at the bottom to get the main sweep of the program and works up to the details in the subroutines.

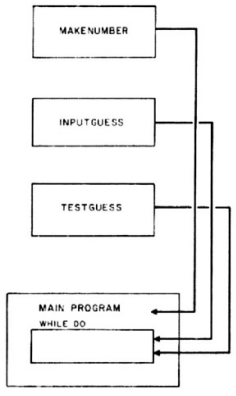

Fig. 1.

Now we can go back and fill in some of the details of the language. One of the most important qualities of a computer language is the way in which it deals with data. XPLO uses three techniques for efficiently dealing with data. These techniques are *scope, dynamic core allocation* and *parameter passing.*

Scope

Scope defines the area in which a variable is active. In many languages, the user has no control over the scope of a variable's activity. For example, in BASIC once a variable is created, it remains active for the entire program. In XPLO the scope of a variable is controlled by where the variable is defined. Variables are active only within their own block or within procedures nested inside that block.

In this way, some variables can be active in only one or two procedures, while others are active for all procedures. It is even possible to have several different variables with the same name and different areas of activity. If more than one variable with the same name is active, the most local variable has precedence.

Dynamic Core Allocation

Dynamic core allocation is a logical extension of the idea of scope. Whenever XPLO completes the execution of a subroutine block, certain variables local to that block are no longer needed by the program. When a variable is no longer active, XPLO returns the unused space to the user's memory pool for use by other routines.

In contrast, variables created in BASIC take up memory space throughout a program's

```
'CODE' CRLF=9,RANDOM=1,INPUT=10,TEXT=12;

'INTEGER'  GUESS,NUMBER,INCORRECT,TRY;


'PROCEDURE' MAKEANUMBER;
'BEGIN'
    NUMBER:=RANDOM(100);
'END';


'PROCEDURE' INPUTGUESS;
'BEGIN'
    GUESS:=INPUT(0);
'END';


'PROCEDURE' TESTGUESS;
'BEGIN'
    'IF' NUMBER=GUESS 'THEN'
    'BEGIN'
            TEXT(0,"CORRECT!!");
            TRY:=1;
    'END'
    'ELSE'
            'IF' NUMBER<GUESS 'THEN'
            TEXT(0,"TOO HIGH")
            'ELSE' TEXT(0,"TOO LOW");
CRLF(0);
'END';


'BEGIN'
INCORRECT:=0;
TRY:=INCORRECT;
MAKEANUMBER;
'WHILE' TRY=INCORRECT 'DO'
                    'BEGIN'
                            TEXT(0,"GUESS: ");
                            INPUTGUESS;
                            TESTGUESS;
                    'END';
'END'
```

Program listing.

execution. The variable space in a BASIC program is always the sum total of all of the variables used in the program. XPLO programs use an absolute minimum of variable space.

Passing Parameters

Passing parameters is the way in which one routine communicates data to another. In many conventional languages, the data is sent from one routine to another by placing it in a variable and calling the routine. The programmer must know in advance which variable names are used by the receiving routine.

In XPLO, information being sent to another routine is simply tacked on to the end of the call. For example,

```
TEST(X,Y,Z);
```

calls a procedure named TEST and sends the variables X, Y and Z. When the call reaches the procedure, the values of X, Y and Z are placed into the first three variable names defined in that procedure. If the first three variables defined in TEST are A, B and C, then the value of X will be passed to A; Y will go into B and Z into C. This technique makes each subroutine a clean and completely independent operation.

Booleans

The Boolean operators AND, OR and NOT are available in XPLO. The Booleans are set up so that they operate upon individual bits. For example, the statement "X: = Y&4" indicates that Y is ANDed with the numerical value 4. Since 4 is the binary number 0100, this operation masks off all bits except bit three.

The ability to operate on individual bits gives XPLO the flexibility to link directly with machine-level operations. This enables the programmer to blast PROMs, read I/O ports and operate joysticks directly from the high-level language.

XPLO in XPLO

One of the most interesting things about XPLO is that the compiler is written in XPLO. This means that the compiler can compile itself and that new features can be added to the language by editing XPLO and compiling the new compiler. Thus each new version of the language is brought to life by the old version. Where did the first version of the compiler come from? The first version of the compiler was written in ALGOL.

Portability

The XPLO compiler translates the source program into an intermediate language called I2L. The I2L code is interpreted and executed in an I2L interpreter written in machine language. I2L is very close to machine language. It contains 28 op codes that are easily implemented in any machine language. Thus, XPLO can be run on any machine by writing the relatively simple interpreter for the particular CPU.

I2L interpreters run about 2K in length. Since all device-specific I/O is contained within the interpreter, the exact same compiler can run on all machines. Once an interpreter is written for a particular CPU, the user need only load the compiler to have the complete XPLO language running on his system.

XPLO is ideally suited to high-speed tasks such as real-time graphics. Arcade-type video games complete with sound effects are easily generated in XPLO. Assemblers, editors and operating systems can also be written in XPLO. XPLO could even be used to father a new language in the same way ALGOL fathered XPLO. A compiler for the new language could be written in XPLO and then each new compiler would be written in the new language.

Availability

XPLO was written specifically for microprocessor systems by Peter Boyle. At the present time, I2L interpreters exist for the 6502, Z-80 and the PDP-10.

A complete program development system is available for Apple II, KIM-1 and TIM systems. The basic package is $45 for KIM and TIM versions and $35 for the Apple II version. A detailed user's manual is also available for $17.25. Packages for other processors will be available in the near future. The packages operate in 16K to 32K of memory depending on the system. They include a compact, versatile editor that allows program creation and compilation to be entirely core resident.

The basic package consists of a memory image cassette or paper tape of the interpreter, compiler and editor. The user's manual is a detailed, step-by-step introduction to the language. It contains over 60 pages of concise description and example programs. It is clearly written so that even a novice can master XPLO.

The Apple II package contains a complete set of functions to drive the high and low resolution graphics, the game paddles and the speaker. A cross-referenced assembly listing for the interpreter and editor is available as a separate package.

KIM and TIM packages are available from The 6502 Program Exchange, 2920 Moana, Reno NV 89509. The Apple II package is available from P. J. R. Boyle, 1337 Adams, Denver CO 80220. ■

✓ Reader Service—see page 179