XPLO V4D USER MANUAL

MANUAL: L. FISH
SOFTWARE: P.J.R. BOYLE

TABLE OF CONTENTS

THE DISTRIBUTION DISK

The XPLO standard distribution disk is a normal APEX disk, which, when booted, will run a series of simple XPLO demonstration programs. In order to do anything else with this disk you must have a standard APEX system disk.

The disk contains the XPLO compiler as the file "XPL.SAV" and the I2L interpreter/loader as the file "I2L.SAV". These two files should be copied onto your master APEX system disk.

The disk also contains a number of example XPLO programs which illustrate several features of the language discussed in the manual. These programs should be copied onto an APEX task disk.

If you have a dual drive system then you can simply use the APEX "COPY" program to transfer the files.

If you only have one drive you will have to take some care to avoid starting the XPLO distribution disk during the file transfer process. To transfer a file proceed as follows:

1) Boot an APEX system disk.
2) Insert the XPLO distribution disk.
3) Give the APEX command "NEW".
4) Type: EXCH file.ext
5) Follow the prompts, inserting the disk you are transferring the file to.
6) Re-insert the system disk when it is prompted for at the completion of the transfer.

Repeat the process for each file to be moved.

Warning: Do not run the APEX "INIT" or "MAKER" operations on the XPLO distribution disk. If you want to make it a system disk replace the existing ".SYS" files completely.

INTRODUCTION

Welcome to XPLO. XPLO was designed to bridge the gap between machine language and high level languages. It has the speed and ability to manipulate your computer on the machine level and yet has the power and structure of a sophisticated block structured language.

With XPLO you will be able to write programs that are beyond the ability of other high level languages. Programs that have been written in XPLO include: new languages, compilers, operating systems, text processors, medical instrument controllers, graphics programs etc. Normally, these programs would have been written in assembly language, but, because they were written in XPLO they were written quickly and can be readily modified.

This manual is designed to acquaint you with the mysteries of block structured languages. XPLO is as easy as BASIC to learn. If XPLO is your first high level language, or if you are familiar with other block structured languages, then you will find XPLO logical and easy to learn. If your programming experience is with non-structured languages, a structured language may seem clumsy at first. This is because a block structured language requires a bit more setup at the start, but, as the programs grow in length and your skill increases, you will begin to feel the support and power of a block structured language.

The XPLO package actually consists of two programs: the XPLO compiler and the I2L interpreter. In addition, an editor is required to generate the program text.

```
==========      ==========
!        !      !        !
! SOURCE !<<<<<<! EDITOR !
!  file  !      !        !
!        !      !        !
==========      ==========
==========      ==========      ==========
!        !      !        !      !        !
!  I2L   !<<<<<<!COMPILER!<<<<<<! SOURCE !
!  file  !      !        !      !  file  !
!        !      !        !      !        !
==========      ==========      ==========
==========      ==========
!        !      !        !
!  I2L   !<<<<<<!  I2L   !
! INTER  !      !  file  !
! PRETER !      !        !
==========      ==========
```

Writing a program involves three steps:

1. The editor is used to create the program text.

2. The compiler reads the program text and converts it to an executable intermediate language called I2L.

3. The intermediate language (I2L) is executed by the interpreter

The text created by the editor is called XPLO source code. This text is fed to the compiler, which generates the I2L code in a relocatable hex format. The I2L interpreter contains a special loader that loads the I2L code into memory. Once the code is loaded, the interpreter begins execution of program.

Since the XPLO package actually consists of several different programs, the program creation process consists of calling an editor, the compiler and interpreter in sequence.

USING THE COMPILER

Once you have set the APEX default file name to the appropriate file and created a source file using an editor, you run the compiler to translate the source into the I2L code. If your source file is the default file you run the compiler by the APEX command:

```
APX>XPL(space)(return)
```

APEX should respond with something like this:

```
INPUT: FROG.XPL
OUTPUT: FROG.I2L
```

When the compiler starts it will prompt with:

```
XPLO V4D - MAY 1980
BINARY:Y
LISTING:Y
OK?:
```

The compiler is saying that, by default, it will create both a listing to the console and a binary I2L file to the disk. Normally this is what you want to do so you simply respond with a carriage return. If you don't want both options answer with "N" and follow the prompts.

Once the I2L file has been created it must be loaded with the I2L interpreter. The APEX command is:

```
APX>I2L(space)(return)
```

When the interpreter is called, it will begin loading the binary into memory. Some error checking is done during the load. After the load is complete, the interpreter will prompt with:

```
<CR> TO EXECUTE
```

At this point, the loaded program can be executed or saved as a SAV file. To execute the program simply type a return; to save the program, type a CTRL-P at this point, then, when APEX comes back, give the command:

```
     APX>SAVE(space)(return)
```

We will now take an over view of XPLO and of block structured languages in general.

WHY BLOCK STRUCTURED?

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

COMPLEXITY

As a program grows in length, its complexity can grow 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 very complex program can be written easily by breaking it down into small modules. The procedure structure of XPLO naturally organizes programs into small, easily understood blocks.

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 very easy for the programmer to lose track of what each variable is doing in each routine. Eventually, variables collide and you find that your Startrek program is going out to lunch because the variable that holds the enterprise's shield power is getting eaten by the klingon navigation routine.

In block structured languages, the programmer has complete control of each variable. Each variable is defined to be active only within a particular part of your program. This means that variables that are part of some other section of code cannot affect you and you can ignore their existence.

AN EXAMPLE PROGRAM

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 thinks of 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 2 AND 3 UNTIL OUR GUESS IS CORRECT.

Here are the steps translated into XPLO:

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

Notice that the program is almost word for word the same as the step by step description of the task. First we "make a number", and "while" the "guess" is "incorrect" we "input" a "guess" and "test" the "guess". "Begin"s and "end"s 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 write each of these procedures.

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

This procedure generates a random number and puts that number in the variable "number".

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

This procedure gets a number from input device number zero and stores it in the variable guess. In XPLO, as many as 8 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.

```
'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';
```

This procedure is a bit more complicated, but it is still easy to understand. If the computer's "number" is equal to the "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.

Now, let's look at the whole program:

```
'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';
```

There are two new constructs in the final version of the program.

"CODE" allows the programmer to assign names to XPLO intrinsics. Intrinsics are built in procedures which perform commonly required functions. In our example, the word "RANDOM" is assigned to the random number intrinsic and is used to call the random number routine. The programmer need only use those intrinsics necessary to the task and can assign names that add clarity and readablility 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, the variables would have been active only within that procedure.

OVERALL STRUCTURE

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.

```
----------------
I              I
I  MAKENUMBER  I <---
I              I    I
----------------    I
                    I
----------------    I
I              I    I
I  INPUTGUESS  I <------
I              I    I  I
----------------    I  I
                    I  I
----------------    I  I
I              I    I  I
I  TESTGUESS   I <---------
I              I    I  I  I
----------------    I  I  I
                    I  I  I
                    I  I  I
----------------    I  I  I
I              I-----  I  I
I    MAIN      I       I  I
I              I--------  I
I              I          I
I              I-----------
----------------
```

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

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.

ASPECTS OF XPLO

SCOPE

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 memory allocation, and parameter passing.

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 called by 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.

DYNAMIC MEMORY ALLOCATION

Dynamic memory allocation is a logical extension of the idea of scope. Whenever XPLO completes the execution of a procedure, memory space assigned for that procedure is 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 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 some simple languages, the data is sent from one routine to another by placing it in a variable and then 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);
```

This statement calls a procedure named "TEST" and sends the variables X, Y and Z to it. When the call reaches the procedure, the the values of X, Y and Z are placed into the first three variables defined in that procedure.

Thus 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 allows each procedure to be a clean and completely independent operation.

XPLO IN XPLO

One of the most interesting thing 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.

PORTABILITY

The XPLO compiler translates the source program into an intermediate language called I2L. The I2L code is interpreted and executed by an I2L interpreter written in machine language. I2L is very close to machine language. It contains 42 opcodes 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. Exclusive of special intrinsics, 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. This also means that if you should change processors, programs written on your old system can be easily moved to the new machine.

XPLO V.S. PASCAL

Superficially XPLO is very similar to a subset of Pascal. In fact an introductory Pascal text will be most helpful to a begining XPLO programmer. However there are differences, both in syntax and in philosophy.

If Pascal is the Caddilac of computer languages then XLPO is the Volkswagen. XPLO is more energy efficient, less costly and easier to deal with. For many experienced programmers simplicity is the critical element. It is easier to work on a vehicle you understand completely.

XPLO will get you there, everytime. But it wont pull a horse trailer and has no power steering. So, ulitmately, it's a question of taste, efficiency, and your needs.

DATA TYPES

XPLO is a loosely typed language. The standard XPLO sixteen bit quantity can be dealt with as an integer, a boolean true or false, a character, a pointer, etc. In this aspect it is similar to assembly language. Pascal, on the other hand, is an example of a strongly typed language. It keeps these data types logically separate. FORTRAN is an example of a language that cannot decide wether it is strongly typed or not.

The object of strongly typing a language is to allow more extensive error checking as a program is compiled. In Pascal, the compiler will complain if you code syntactically correct nonsense, such as if you store a character into a variable and then try to use the value as an array.

Unfortunately it is impossible to anticipate all possible programming requirements. In a rigid language, as soon as programs become non-trivial you find yourself looking for ways to "trick" the language into doing what you want. The result is that programs become involuted and hard to understand.

Therefore one of the objectives of XPLO is to provide a much needed language which will not do things for you which you then have to find ways to undo. It tries to be direct and obvious.

Freedom and responsibility go hand in hand. In XPLO, making sense is your responsibility.

THE ELEMENTS OF XPLO

Now lets examine the XPLO vocabulary in more detail. Informally XPLO can be looked at as being organized into four levels. From the lowest to highest level, they are:

1) ATOMS
2) STATEMENTS
3) BLOCKS
4) PROCEDURES

Atoms are the smallest complete, understandable piece of program. Here are some examples of atoms: 'FOR', =, +, 9056, etc. Atoms are organized into groups which instruct the compiler to perform a specific task. These groups are called statements. Here is an example of a statement:

```
'FOR' X:=1,10'DO'NUMOUT(0,X);
```

Several statements can be grouped together into a single statement called a block. If several statements are grouped together, the block must start with a 'BEGIN' command and terminate with 'END'. Each statement within a block must be separated by a by a semicolon (;). Here is an example of a block:

```
'BEGIN'
X:=0;
'IF'Y=20'THEN'Z:=25'ELSE'Z:=30;
```

```
          NUMOUT(0,Z);
          'END';
```

Brackets can be used in place of begin-end pairs:

```
          [X:=0;Y:=21]
```

XPLO is very flexible in the way in which it allows statements and blocks to be put together. For example, several blocks can be put together into a single block. Also blocks can placed in middle of statements. For example:

```
          'FOR'X:=0,10'DO'
                   'BEGIN'
                   Y:=X*20;
                   Z:=Y+X;
                   'END'
```

Here we have a statement containing a block. The block itself is made up of two statements separated by semicolons. Notice that there is no semicolon between the 'DO' and the start of the internal block. This is because the block is a part of the statement and not a separate statement.

In XPLO the line by line structure of your program is not important. Statements can be split across several lines or several statements can be put on one line. Extra spaces and tabs are also allowed. By making use of this freedom you can make the structure of your program more clear.

Procedures are the highest level organization in an XPLO program. Procedures are composed of statements and blocks grouped together to perform a specific task. A program may contain any number of procedures. All procedures are named and called as subroutines from other parts of the program.

EXPRESSIONS

XPLO, like most computer languages is a mathematical language. It is capable of performing arithmetic and other operations on numbers. Numbers in XPLO are 16 bit signed integers. This means that XPLO will accommodate any number between -32768 and +32767. Fractional numbers are not normally used because they they take about 100 times longer to process.

Numbers in XPLO are not checked for overflow. Overflowing values simply wrap around. For example, if you add one to 32767, the result will be -32768.

HEX NUMBERS

XPLO also has the ability to deal with numbers in hexadecimal form. Any number preceeded by a "dollar sign" ($) is taken as hex. For example:

```
$FFC0
$A9
```

The same range limits and wrap around apply to decimal and hexadecimal integers.

VARIABLES

Variables are a temporary storage place for numbers. Variable names can be single characters or whole words. Usually, variable names are chosen so they describe what they contain. For example, if you were calculating interest rates, the principle could be stored in a variable called "PRINCIPLE" and the rate stored in "RATE". Since XPLO is a compiled language, large names do not slow program speed or take up extra memory space.

Variable names can contain numbers as well as alphabetic characters, however, the first character must be an alpha (A-Z). For example:

```
X1
B23
RATE12
```

Only the first 6 characters of a variable name have significance to the compiler. Larger names can always be used; the compiler just ignores the extra characters.

Before the compiler will recognize a variable, the name must be assigned memory space. This operation is covered in detail in the "declarations" section below.

Numbers and variables are examples of things which have a "value". Values in XPLO can be formed in many ways, as we shall see in the following pages.

ASSIGNMENTS

Values are usually stored into variables using a statement called an assignment statement. An assignment is distinguished by a combined colon and equals character (:=). Colon-equals is used instead of equals to make a distinction between comparing two numbers for equality and moving a number into a variable. Here are some simple assingnments:

```
FROG:=234
TIME:=50
TEST:=TIME+1
```

In the first example, 234 is stored in the variable named "FROG". The second stores 50 in "TIME", and the last adds one to whatever is contained in "TIME" and stores the result into "TEST".

OPERATORS

XPLO can perform all sorts of operations on numbers. Items which have values can be combined into an "expression" by the use of "operators". An expression calculates out to give a single value. Here is an example expression:

```
TEST+5*FROG
```

Operators can be performed on anything which returns a value, such as numbers, variables, sub-expressions etc. The resulting expression is itself a value. Values are quite general things in XPLO. Any expression can be used wherever a value is required.

The normal arithmetic operations of addition, subtraction, multiplication and division are performed by common symbols:

+	ADDITION
-	SUBTRACTION
*	MULTIPLICATION
/	DIVISION

For example:

```
27+5
TEST*23
P/50+2
```

When an arithmetic expression is evaluated, multiplication and division are performed first, then addition and subtraction. Otherwise, expressions are evaluated from left to right. The order of evaluation is important because it can change the result of a calculation.

In some instances it is necessary to force XPLO to evaluate an expression in a different order. This is done using parenthesis "()". The part of an expression within parenthesis will be evaluated first. Parenthesized expressions can be nested if necessary. Here are some examples:

```
TEST+(FROG/23)
TEST*(TEST+(HIGH/2)/3)
```

COMPARISONS

XPLO allows us to compare values in several ways. They can be tested to see if they are equal(=), not equal (#), etc. For example:

```
X = 3
(X+1) >= Y
A # 0
```

The following characters are used to compare values in XPLO:

```
=       TESTS FOR EQUAL VALUE.
#       TESTS FOR NOT EQUAL VALUE.
<       TESTS IF THE FIRST VALUE IS LESS THAN THE
        SECOND.
>       TESTS IF THE FIRST VALUE IS GREATER THAN
        THE SECOND.
>=      TESTS IF THE FIRST VALUE GREATER OR
        EQUAL TO THE SECOND.
<=      TESTS IF THE FIRST VALUE IS LESS THAN OR
        EQUAL TO THE SECOND.
```

When XPLO evaluates conditional expressions, it decides whether the assertions are true or false. If the specified conditions are met, then the expression is true, if they are not met, the expression is false.

TRUE and FALSE

Truth and falsity in XPLO are values just like other values. In fact, internally, true and false are represented by numerical values. This may seem a bit strange, but the values can be used by the programmer, and they are essential to boolean operations.

The true-false concept is so useful that XPLO has special constants which represent true and false:

```
'TRUE'
'FALSE'
```

These constants have fixed values which represent true or false within XPLO. The constants can be used to set a variable to true or false:

```
FROG := 'TRUE';
```

Or in comparisons, for example:

```
        PIG(1) = 'FALSE'
```

Note that in the latter case the equals sign is not coupled with a colon as in the former. Thus the colon is what distinguishes an assignment statement from an expression involving a comparison operation. An assignment is a complete statement, while an expression is just the value part of some larger statement. As in:

```
        PIG:=FROG=0;
```

BOOLEAN OPERATORS

Booleans are a special form of value in which the 16 bit number is considered in its binary or true-false form. Boolean operators operate on numbers as boolean values.

XPLO uses three different boolean operators: "and", "or" and "not". In XPLO, the following characters are used to to perform these operations:

```
        'NOT'                "NOT" FUNCTION
        &                    "AND" FUNCTION
        !                    "OR" FUNCTION
```

In XPLO, these operators can be used in several different ways. For example they operate on true and false values as you might expect, so that 'NOT''TRUE' evaluates to 'FALSE'

"NOT" operates with a single argument. It simply changes the value to its opposite. For example:

```
        'NOT'DONE
```

"AND" requires two arguments. If either argument is false then result is false. If both are true then the result is true. For example:

```
        SHIPS & SNAILS
```

"OR" also requires two arguments. If both arguments are false then the result is false. If either argument is true the result is true. For example:

```
        CABBAGES ! KINGS
```

In another use of boolean operations they operate on binary bits. Here is a table showing the way in which these operators work on single bits:

BIT	OPERATOR	RESULT
0	'NOT'	1
1	'NOT'	0

BIT	OPER	BIT	RESULT
0	&	0	0
0	&	1	0
1	&	0	0
1	&	1	1
0	!	0	0
0	!	1	1
1	!	0	1
1	!	1	1

"NOT" operates with a single argument. It simply changes the bit to its opposite.

"AND" requires two arguments. If either argument is zero then result is zero. If both are one then the result is one.

"OR" also requires two arguments. If both arguments are zero then the result is zero. If either argument is one the result is one.

In actual XPLO operation, boolean operators work on all 16 bits of a value at once, but the principle is the same. Here are some examples, using 8 bit values for clarity:

NUMBER	OPER	RESULT
00110011	'NOT'	11001100
11110000	'NOT'	00001111

NUMBER	OPER	NUMBER	RESULT
10000001	&	00000001	00000001
11111111	&	00011000	00011000
10000000	!	00000001	10000001
11111110	!	00000001	11111111

Here are some assignments with expressions involving boolean operators:

```
FROG:='NOT'27;
TEST:=FROG & $FE;
TIME:=FROG ! TEST;
```

Boolean operations can be used to link XPLO with machine level of the of the processor. They enables XPLO to set and clear memory bits.

Expressions can contain boolean operations, comparisons, and mathematical operations. In mixed expressions arithmetic operations are performed first, then comparisons and finally, booleans. Therefore the following expressions are the same:

```
        A=1 & B=2  vs.  (A=1) & (B=2)
```

But these are different:

```
        A & $80 = 0  vs.  (A & $80) = 0
```

IF EXPRESSIONS

A common construct in programming looks something like:

```
        'IF' X>Y 'THEN' PIG:=20 'ELSE' PIG:=25;
```

Here the variable "PIG" is set to 20 or 25 depending upon the outcome of the compare. XPLO has a simple mechanism for dealing with this type of assignment:

```
        PIG:='IF' X>Y 'THEN' 20 'ELSE' 25;
```

The entire IF is evaluated and reduced to the value of the THEN or ELSE (here 20 or 25), then the value is passed to PIG.

Like all expressions, IF expressions can be used anywhere in XPLO, not just in assignment statements.

STATEMENTS

XPLO has command words. Each command is set off in single quotes (') to make it more visible in the middle of a program. If you prefer, you can write command words in lower case rather than set them off in quotes. Commands must be written in lower case or enclosed in quotes, or the compiler will not recognize the command.

Command words, expressions and other statements combine to form the XPLO statements. We have already met the first of these statements, the assignment. Now we will cover the rest of the statements.

EXIT

Perhaps the simplest XPLO statement is EXIT. It simply terminates the program normally at that point.

CALL STATEMENTS

Another simple XPLO statement is a call to some other routine. All it consists of is a mention of the name of the routine, for example:

```
          SORT;
```

The routine called in this way can be either a procedure, an intrinsic or an external. They all operate the same way at the call.

A call can send some values to the routine to control the specifics of its function, in which case the call has the form:

```
     name(value,value,...value);
```

For example:

```
          CURSOR(10,13);
```

The details of how the routines being called are defined is covered later in this manual.

BEGIN END

Begin and end are used by the programmer to designate logical programming blocks. Blocks are composed of one or more statements.

This statement has the form:

```
     'BEGIN'statement;statement;...statement'END';
```

Each begin must have a matching end. One of the more common XPLO programming errors is mismatched begin-end pairs.

Where convenient, square brackets ([]) may be used in place of 'begin' 'end'.

IF THEN ELSE

This statement has the form:

```
     'IF'value'THEN'statement'ELSE'statement;
```

It is used to conditionally execute blocks of code:

```
     'IF' (CONDITION) 'THEN' (BLOCK-A) 'ELSE' (BLOCK-B);
```

If the condition is satisfied, BLOCK-A is executed, if the condition is not satisfied BLOCK-B is executed.

Generally, the condition is based upon comparing two or more values. The values can be compared in several ways. They can be tested to see if they are equal(=), not equal (#), etc. For example:

```
     'IF' A=B 'THEN' (BLOCK-A) 'ELSE' (BLOCK-B);
```

This statement tests to see if A is equal to B. If they are equal, BLOCK-A is executed, if they are not equal BLOCK-B is executed.

Conditional statements do not have to be simple compares. The condition can be any XPLO value. Thus XPLO is capable of evaluating complex relationships. For example:

```
          'IF' A/B+C-D=(TIME+1)/45 'THEN'...;
```

Boolean operators can be used inside a compare. For example:

```
          'IF' A=B & C=D 'THEN' (BLOCK);
```

Here A must equal B and C must equal D before the block is executed. When necessary, the ELSE section of the command can be omitted.

CASE

Frequently a program must decide between more than two alternatives. Since an IF statement is defined as containing other statements, which, in turn, can be IF statements, one way to handle this is to nest the IFs, like this:

```
          'IF'A&B=1'THEN'DOONE
```

```
'ELSE''IF'FROGS'THEN'DOTWO
'ELSE'DOOTHER
```

But this can get confusing, so XPLO has the case statement. This statement has two forms, the first is:

```
'CASE''OF'
     value:statement;
     value:statement;
     .
     .
     value:statement
'ELSE'statement;
```

In this form the case is just like the nested IFs above. The first value that evaluates to true causes the corresponding block to be executed. If no value comes out true then the ELSE block will be executed. Translating the above example:

```
'CASE''OF'
A&B=1:  DOONE;
FROGS:  DOTWO
'ELSE'  DOOTHER;
```

The ELSE clause cannot be left out, but it can be blank:

```
'CASE''OF'
A=1:    DOONE;
A=2:    DOTWO
'ELSE';
```

The second form of the case statement is used for efficiency when the values all have a common component and an equality test, as in the example we just used. This form is:

```
'CASE'value'OF'
     value:statement;
     value:statement;
     .
     .
     value:statement
'ELSE'statement;
```

The last example, in this form, looks like this:

```
'CASE'A'OF'
1:      DOONE;
2:      DOTWO
'ELSE';
```

Unlike Pascal, XPLO case values do not have to be simple constants. They can be any expression or value.

WHILE DO

This is a conditional looping structure. As long as the condition is met, the block is repeatedly executed.

This statement has the form:

'WHILE'value'DO'statement;

For example:

```
'WHILE' TEST=FALSE 'DO'
        'BEGIN'
        CLEAR;
        FROG:=25;
        SETUP;
        'END';
```

As long as the variables TEST and FALSE are equal, the code within the begin-end block will be executed. The program tests the condition as soon as the while loop is entered. If the condition is false, the loop will be ignored. Once the loop has been entered, the program tests the condition at the end of each execution of the block. The condition must eventually go false or the loop will continue forever.

REPEAT UNTIL

This statement has the form:

'REPEAT'statement;...statement'UNTIL'value;

The REPEAT loop is similiar to the WHILE loop except that the decision to continue the loop is made at the end of the block. Here is how it looks flow charted:

```
      WHILE                  REPEAT

        I                      I
        I                      I
       / \                 =========
  ----<   >-               I       I
  !    \ /                 I BLOCK I--
  !     I                  I       I !
  !  =========             ========= !
  !  I       I                 I     !
  --I BLOCK I                / \    !
     I       I              <   >----
     =========               \ /
                               !
```

Here is a sample repeat loop:

```
'REPEAT'
        'BEGIN'
        INPUT;
        X:=X*20;
        OUTPUT(SUN);
        'END'
'UNTIL' X>Y;
```

LOOP

This statement has the form:

    'LOOP'statement;

A LOOP command repeatedly executes the statements in the block. Execution will contine indefinitly unless or until a QUIT command is encountered inside the block. A QUIT command can be used to exit from any point within a loop. Frequently, a QUIT will be placed in an IF command so that the loop will exit under certain conditions. For example:

```
        'LOOP'
                'BEGIN'
                DOSOMETHING;
                'IF'A=B'THEN''QUIT';
                'END';
```

FOR DO

A FOR loop is special kind of loop. A for loop counts upward one at a time, and for each count it executes a block. The starting and ending value of the count can be set, and the count is stored in a variable so that the value can be used by the program.

This statement has the form:

    'FOR'variable:=value,value'DO'statement;

Here is an example:

```
        'FOR' X:=1,100'DO' NUMOUT(0,X);
```

Here X starts with the a value of one and steps one at a time to 100. For each step, the block is executed. In XPL0, the steps must always be ascending and the increment is always one. Negative loop limits can be used. Descending loops and other increments can be synthesized.

A for loop might never be executed if the limits are never in ascending order, as in:

```
        X:=-10;
        'FOR'I:=0,X'DO'PRINTIT;
```

COMMENTS

Comments are used by the programmer to make notes to himself or other programmers. They are quite useful as an aid to keeping complex programs straight.

In XPLO, comments can go anywhere in the program, including in the middle of a statement. If a comment is placed in the middle of a statement, it must be enclosed in backslash (\) characters. If the comment is the last item on a line, only the leading backslash is needed. Here are some examples:

```
        'BEGIN'         \GET A CHARACTER
        'FOR' X:=1,10\TEN TIMES\'DO'
```

SEMICOLONS

A semicolon, by itself, is a legal statement. It is the null statement and does nothing. This is not important to the novice XPLO programmer except that it might explain an extra semicolon here or there in an example, such as before an END in a block.

In general the key to understanding the semicolon in XPLO is to realize that they are used to separate statements and declarations where neccessary. Examining the form of each XPLO statement will clarify this.

Don't be misled by the formatting of statements in the examples. As we have said, the end of a line has no relation to the end of a statement, they just happen to correspond in many cases.

In fact the semicolon is not a marker to the end of a statement at all, it separates statements. For example in an IF (or CASE) statement there is no semicolon before the ELSE even though a sub-statement ends there, because a new statement does not start at that point and so there is no need to separate things.

By this reasoning there should not be a semicolon before the END of a block, or before the UNTIL of a repeat loop. In fact, if you look back at the statement forms, you will see that this is indeed the case. However, as we said, extra semicolons do nothing if they do not change the meaning of what is being said, so in this case they are allowed but have no importance.

NULL STATEMENTS

The null, or do nothing statement was just introduced above. It has some relevance to all statements which include other statements because it allows certain parts of statements to be blank. We saw an example of this with reference to the CASE statement above. Here are some other examples:

```
        'FOR'I:=0,1000'DO';      \DELAY A BIT

        'WHILE''NOT'STROBE'DO'; \WAIT FOR SOMETHING

        'REPEAT''UNTIL'KEYSTRUCK;\ANOTHER FORM OF A WAIT
```

Note that the value parts of statements cannot be blank or the statement will be meaningless.

DECLARATIONS

The process of informing the compiler of the existance of a new name is called a declaration. An XPLO program must declare all of the names it will use, and specify what they are names of. There are 7 things one can name:

INTEGER variables
ADDRESS variables
OWN variables
CONSTANTS
INTRINSICS
EXTERNALS
PROCEDURES

LOCAL AND GLOBAL

All names of things are active only within certain clearly defined areas. These areas are governed by the rules of scope which will be described in detail later. A name is said to be "local" to the procedure in which it was declared.

A name which is defined for several procedures is said to be "global" to those procedures.

INTEGERS

The integer variable declaration has the general form:

```
'INTEGER'name,name,...name;
```

More than one variable can be activated in the same declaration:

```
'INTEGER'JACK,JILL;
```

This declaration tells XPLO that the procedure we are in will require the variables JACK and JILL. Memory space is allocated for these variables at this time.

ADDRESSES

The address declaration defines a variable similar to an integer except that it can be used as the address of an area of memory that we want to get at on a byte-by-byte basis. We shall see more about this in the next section. The form of the address declaration is:

```
'ADDRESS'name,name,...name;
```

INTRINSICS

Intrinsics are the way in which XPLO links with necessary machine level operatons. Intrinsics are built in routines which perform a variety of operations, including input and output, math, and special operations.

The current version of the interpreter contains more than forty intrinsics. New intrinsics can be added easily, by placing the machine code for the intrinsic in the interpreter.

Each intrinsic in the interpreter is numbered. When an intrinsic is declared, a name is given to its number. The general form of an intrinsic declaration is:

```
'CODE'name=number,....name=number;
```

Here is an example:

```
'CODE'RANDOM=1,WRITE=11;
```

Intrinsics can be given any name that suits the programmer. Generally, names that describe the intrinsic are used. Only those intrinsics needed by a particular program need be declared for that program.

Information is sent to an intrinsic in the same manner as procedures. The information is placed in parenthesis immediately following the intrinsic name.

Some intrinsics return a value while others do not. In XPLO it is your responsibility to take this into account. Intrinsics that return a value must be used as a value, not as a call statement, or a fatal error will result. Equally, an intrinsic that does not return a value must not be used as a value.

Here are some example uses of intrinsics:

```
CURSOR(25,35);
NUMBER:=RANDOM(100);
```

The first example sends the values 25 and 35 to a cursor position intrinsic. In the second, a random number between 0-99 is placed in the variable "NUMBER".

As an example of the incorrect use of intrinsics, the following statement is illegal and will cause an error when it runs:

```
'FOR'I:=10,100'DO'RANDOM(I);     \A BAD STATEMENT
```

The error arises because the random number intrinsic returns a value which is not used.

PROCEDURES

One of the most important concepts in computer programming is the idea of subroutines. In any program, there are certain operations that must be performed over and over. To avoid having to write the same code over and over, the programmer will put the code into a single module which is executed by the main program whenever the common operation must be performed. After the common code is executed, the program returns to the main line of the program and continues.

In XPLO, subroutines are called procedures. Any block of code can become a procedure simply by giving it a name. The process of naming a procedure is also a "declaration". Procedure declarations take the general form:

```
'PROCEDURE'name;
declarations;
statement;
```

For example:

```
'PROCEDURE' COUNT;
'BEGIN'
'FOR'X:=1,100'DO'
        'BEGIN'
        Z:=22/X;
        ORDER;
        'END';
'END';
```

Once a procedure has been named it can be executed simply by calling it's name. Here is a block that calls four procedures:

```
'BEGIN'
GETNAME;
SORT;
PRINTSORT;
STORENAME;
'END';
```

Since a procedure is a completely independent piece of code, it can itself contain declarations. Declarations for a procedure must be placed between the procedure declaration and the start of active code (usually the first 'begin'). Here is an example of variable declarations within a procedure:

```
'PROCEDURE' RESTORE;
'INTEGER' X,Y,FROG;
'BEGIN'
'FOR' X:=1,20'DO'
        'BEGIN'
        UNDO;
        Y:=Y+1;
```

```
                'END';
        FROG:=X+Y;
        'END';
```

NESTING

Since a procedure can contain declarations, and procedures are a kind of declaration, it follows that procedures may contain other procedures. Therefore, in XPLO we can "nest" procedures inside each other. We say that each such nesting takes us one "level" deeper. For example:

```
'PROCEDURE' ONE;      <---------------
                                     I
'PROCEDURE' TWO;      <---------     I
                               I     I
'PROCEDURE' THREE;    <----    I     I
'BEGIN'                   I    I     I
A:=1;                     I    I     I
'END';                <----    I     I
                               I     I
'BEGIN'\TWO                    I     I
B:=2;                          I     I
'END';               <----------     I
                                     I
'BEGIN'\ONE                          I
ONE;                                 I
TWO;                                 I
C:=3;                                I
'END';               <----------------
```

Look at the way these procedures are nested. Procedure THREE is nested inside procedure TWO which is nested inside procedure ONE. TWO and THREE are sub-procedures to ONE; they are local to it, and do not exist elsewhere in the program.

Procedures may be nested 8 levels deep. Here ONE is at the highest level and THREE is at the lowest level. Notice that the source code for the highest level routine always comes last, and is executed first. In fact, ONE and TWO are not executed unless they are called from a higher level procedure.

The same thing applies to the whole program, the code for the main routine is always the last block in the program. Execution always starts with the highest level blocks.

PARAMETERS

It is frequently necessary to send information from the main program to a procedure. XPLO provides a simple mechanism to send information to a procedure. Information to be sent to the procedure is placed in parenthesis immediately following the procedure call. These values are the "parameters" or "arguments" of the procedure.

If more than one argument is to be sent, they are all placed inside the parenthesis and separated by commas.

When the program arrives at the procedure call, the values sent are placed in the first variables declared. Here is an example:

```
'PROCEDURE' ADDTEN;
'INTEGER' X,Y,Z;
'BEGIN'
X:=X+10;
Y:=Y+10;
Z:=Z+10;
RES:=X+Y+Z;
'END';

'BEGIN'
A:=1;
B:=2;
C:=3;
ADDTEN(A,B,C);
'END';
```

The second block calls the first. In the process, it sends the value of the variables A, B and C, which are one, two and three respectively. When the program executes ADDTEN the values in A, B, and C are passed into X, Y, and Z. The procedure adds ten to these values, sums them into RES and returns. The original A, B, and C are in no way effected by the procedure call.

In the receiving procedure one variable must be declared for each value sent. Otherwise, a fatal error will result.

In order to help the programmer keep track of which variables are parameters, and which are locals, XPLO allows a comment to be placed after the name of a procedure in the declaration. This comment should be used to list the parameters of the procedure in the order they will be used when the procedure is called.

Here is an example of a procedure with the argument list as a comment:

```
'PROCEDURE' CHECK(X,Y);
'INTEGER'X,Y;
'INTEGER'Z,K;
'BEGIN'
Z:=X*Y;
K:=Z/40;
'IF'X<Z'THEN'SAYOK;
'IF'Y<K'THEN'SAYBIG
         'ELSE'SAYSMALL;
'END';
```

The appearance of X and Y in parentheses in the first line tells us that this procedure is intended to have these two values passed to it as parameters while Z and K are simply normal locals.

Occasionally, it is necessary to return from a procedure before the procedure is complete. This can be accomplished using a RETURN command. The RETURN forces a procedure to immediately return to its caller. Unlike BASIC, a RETURN is implied and need not used at the end of a procedure.

RETURN is also used return values from a procedure to the calling routine. The value to be returned is placed following the RETURN command:

```
     'RETURN'value;
```

The argument may be any expression. For example:

```
          'PROCEDURE' INCREMENT(X);
          'INTEGER' X;
          'RETURN' X+1;
          .
          .
          Y:=INCREMENT(X);
          .
          .
```

Here we have a simple procedure to increment a variable. When the procedure is called, the value of X is sent to the routine. This is then incremented and passed back to the caller via the RETURN command. The result is then assigned into the variable Y.

This ability to pass values to and from procedures, with the ability to declare, in each procedure, just those variables it needs, allows each procedure to be a complete and independent module which can be debugged as a separate entity and moved from program to program freely.

One of the important uses of procedures is in making programs "cleaner" and more understandable. By moving a side branch of the main flow of a routine into a separate procedure we can name it according to its function, test it separately and keep the main body of code uncluttered.

In this application it is essential that procedure calls be efficient. Thus XPLO will optimize the procedure calls. In fact, if a routine calls a procedure which is defined within it and has no locals of its own, then such a call is about the fastest complete XLPO statement that there is.

OWN VARIABLES

As described above, variables can be declared so that they are local to a procedure. Any time the program exits from a procedure the memory space occupied by the variables declared in that procedure is returned to the memory pool and the value of the variable is lost.

Most of the time this is exactly what we want, but sometimes it is useful to preserve the value of a variable across procedure calls. One way to do this is to use more global variables, but this makes the variable accessable to other routines and we loose the advantages of the scope rules.

OWN variables can be declared local to a particular procedure, yet the value is not lost when the procedure is exited. It will still have the same value when the procedure is called again. At load time OWN variables are preset to 0 or false.

In general, OWN variables are similiar to normal integer variables. There are two limitations on the use of OWN variables. First, OWN variables cannot be used to receive parameters when they are passed to a procedure. Second, OWN variables cannot be used as the counter in a FOR loop.

OWN variables are declared as follows:

```
'OWN'name,name,...name;
```

For example:

```
'OWN' X,Y,FROG;
```

One application of OWN variables is as program parameters. Many programs require certain setup information, such as the binary and listing switches in the XPLO compiler. We can store this information in OWN variables, run the program long enough to execute the code that sets the OWN variables, and then interrupt it with Control P. Then, when we save the program with the APEX save command, this information will be preserved with the program and be preset when we run the program again.

DECLARED CONSTANTS

One of the things that adds clarity to program is the ability to give names to numbers. The name "LIMIT" is a lot clearer than the number 12419. XPLO allows the user to set up predefined constants. Constants are different from variables in that once they are defined they cannot be changed. Further, the compiler can deal with them as fixed values which makes the code faster and more compact.

The general form of the constant declaration is:

```
'DEFINE'name=number,...name=number;
```

For example:

```
          'DEFINE' MEDIAN=100,LIMIT=27,SUMMIT=14210;
```

Notice that the declaration is made with a simple equal as opposed to colon-equals, because this is a statement of equivalence rather than an assignment to a variable.

In certain instances it is useful to have distinct names for things, but the actual value is irrelevant. In fact in many instances we don't want know the value so that we cannot come to depend on it.

For example, let's say that we are working with a set of colors and we just want to refer to the colors by name. If we come to depend on the numerical value of the color, later changes in the program could be difficult. XPLO has a simple scheme for defining sets of things:

```
          'DEFINE' RED,BLUE,GREEN;
```

Here, all the programmer knows is that these constants have distinct values and that they are in accending order. That is that GREEN > BLUE >RED.

EXTERNALS

Basic has its peeks, pokes and calls. These are weakness in Basic because they result in completely incomprehensible programs.

XPLO has the ability to do the equivalent of peeks and pokes by using address variables. In XPLO the resulting code is readily understandable.

The ability to call machine language routines easily and comprehensibly is provided by the intrinsic structure of XPLO. The external is an alternative form of intrinsic.

(If you are not interested in programming at this level just now, feel free to skip the rest of this section).

To clarify the operation of externals we need to make a few observations about intrinsics. The address of an intrinsic must be known to the I2L interpreter, since it appears in an I2L table. Normally this is as it should be because machine language routines have a way of moving about in memory as we develop things. The table approach makes sure that there is one and only one place where the address of a routine appears. Thus that place is all we have to change if a routine moves. All our XPLO programs which use them do not also have to be changed.

In some cases, however, we want to link to machine code for a purpose which is program specific. This can occur when we want part of the program to be super fast, or when we have a special device to deal with.

When the machine code we want is program specific it is more appropriate for the XPLO program, rather that the I2L itself, to know about the special routines. In that case we use the EXTERNAL declaration. It has the form:

```
    'EXTERNAL'name=address,...name=address;
```

For example:

```
        'EXTERNAL'IRQON=$A000,IRQOFF=$A100;
```

When an external routine is called, XPLO will perform a standard 6502 JSR to the address specified. The external does its job and returns via a standard RTS. Values can be passed to the external routine as normal parameters:

```
        IRQON(TIME,BUFFER);
```

These values are left on the 6502 hardware stack behind the return address and must be removed from the stack by the external. A two byte value may be passed back in the same way, in which case the call becomes a value, as in:

```
        PENDING:=IRQOFF;
```

See the section on memory use for some hints on where you can safely put your own externals.

ARRAYS

It is frequently useful in programming to collect several pieces of data together. Usually, the data values will have something in common. For example, they might be points on a graph or dollars in accounts. In XPLO values can be grouped under a single name. Even though they all have the same name, each item has a separate number. For example:

```
        FROG(11)
```

This refers to the 12th data item in the array named frog. If there are twenty items in an array, they will be numbered 0 to 19. In XPLO there two main types of array: byte arrays and integer arrays. Strings are a special case of byte arrays.

<u>BYTE ARRAYS</u>

Byte arrays are groups of data where each item is a single 8 bit byte. Each item in the array can store either a number from 0 to 255 , a number between -128 and 127, or an ASCII character.

Like procedure and variable names, the name of a byte array must be declared before it can be used. Byte array declarations take the general form:

```
    'ADDRESS'name,...name;
```

To understand arrays more clearly lets take a close look at the way XPLO handles ADDRESS type variables. Array names are really only 16 bit variables that contain the starting address of the data. So the value of the address variable which names the array "points" to a place in memory that is the start of the data. Each numbered item is one byte of memory in sequence from that starting address.

```
STARTING >  ================
ADDRESS     !              !
            !    ITEM 0    !
            !              !
            ----------------
            !              !
            !    ITEM 1    !
            !              !
            ----------------
            !              !
            !    ITEM 2    !
            !              !
            ----------------
            !              !
```

Once a byte array name has been declared, memory space must be set aside for the actual data it points to. This can be done in several ways. The simplest way is to simply set the address to some absolute memory location. Of course, you must be sure that the memory you address is free for this purpose.

As an example, lets say that we have some free memory at hex 2000. To set up an array in this area, simply store $2000 into the address variable:

```
FROG:=$2000;
```

After the starting address has be assigned, data can be moved in and out of the elements of the byte array:

```
FROG(20):=205;
X:=FROG(9);
FROG(X):=Y;
```

As you see in these examples, the variable without a parenthesized item number refers to the starting address of the array. To refer to individual items in the array, the item number must be included, even if it is simply zero:

```
'IF'FROG(0)=$80'THEN'ERASE;
```

The fact that address variables can be set up to point directly at absolute bytes in machine memory gives XPLO a very simple method of accessing memory and memory mapped I-O devices. As an example, lets say that you have an I-O port that you wish to access directly from XPLO. The address variable could be given the name "PORT" and be set up to point to the memory address of the port. Then a reference to PORT(0) will access the port byte itself.

As useful as direct access to memory can be, most of the time arrays are used simply to store data. When an array is used to store data, the programmer really doesn't want to keep track of what memory is free and available. To solve this, XPLO allows it's own memory to be used and provides a simple method of assigning and keeping track of the memory used.

The operation of reserving some memory for array use is performed using intrinsic number 3, the "RESERVE" intrinsic. RESERVE sets aside a specified number of bytes of memory and returns the starting address. For example:

```
FROG:=RESERVE(25)
```

This sets aside 25 bytes of memory exclusively for the use of the array, and returns the starting address of this array which we pass to the address variable named FROG. The programmer need never know the actual address of the array.

Here is an important note regarding the RESERVE operation. It allocates memory space dynamically, just like other local variable space, so when you return from the procedure in which a reserve was made the reserved space, and its content, is lost. If the procedure is called again the space will be re-reserved once again.

If you have used other langauages you may find you tend to confuse the RESERVE with what some lanuages call "DIMENSION". The vital difference is that RESERVE is an action taken as opposed to a declaration. A RESERVE inside a loop will reserve more space every time the loop is executed!

STRINGS

The third way to set up a byte array is to assign a text string to an address variable. For example:

```
        FROG:="THIS IS A STRING";
```

Here what happens is this: The compiler allocates some space for the string, fills it with the ASCII for each character, and returns the address of the place it put the string as a pointer. We store this address into an address variable.

At this point the variable FROG behaves essentially like any other byte array. There are differences however. In this form the array, and its content, is fixed and preset. Assignments to such arrays are not allowed. We can read the bytes however, as in:

```
        'IF'FROG(3)=$D'THEN...
```

Or we can print the string as in:

```
        TEXT(0,FROG);
```

Since a normal byte array is a convenient form for a string of characters that we will want to change, the two forms together allow powerful string operations to be performed. For example, strings could be compared for matching words by comparing the numerical value of each character.

XPLO employs the convention of marking the end of a string by setting the high bit of the last byte. This adds 128 to the ASCII value of this byte so that, in the example above FROG(15) has the value 199, which is 128 more than the ASCII for the letter G.

To facilitate the handling of characters, XPLO has a special operator that converts an ASCII character into its numerical value. The uparrow character (^) converts the character it preceeds into its numeric value. For example:

```
        X:=^A
```

This operator can be used anywhere a value is called for. For example, it can be used in a compare operation:

```
        'IF' X=^A 'THEN'.....
```

The uparrow character can also be used to indicate characters that would normally be interpreted as controls to XPLO. For example, double quotes cannot normally be included a string because they terminate the strings. An uparrow in a string causes the next character to be taken literally. For example:

```
        FROG:="THIS IS NOT A ^"NEW^" IDEA!!!"
```

As we said before, each item in a byte array is an eight bit value. XPLO normally handles sixteen bit values so we need some way to convert from one to another.

When a 16 bit value is moved into an 8 bit byte, the least significant bits of the 16 bit value is moved into the byte. When an 8 bit value is stored in a 16 bit variable the single byte value is stored in the low byte of the 16 bit variable and the high byte is set to zero.

In addition, there are several intrinsics available that facilitate the handling of 8 and 16 bit values. Intrinsic number 4 ("SWAP") takes a 16 bit variable and exchanges the high and the low byte. For example:

```
        PIG(2):=SWAP(HOUSE);
```

It is common practice in computer programming to treat the highest bit of a number as a sign bit. For example, if the highest bit of a number is a zero, the number is regarded as positive; and if the bit is a one, the number is negative. Any 8 bit number greater than 127 will have its sign bit set and could be thought of as a byte-sized negative number.

Normally, when a number is moved from a byte into a variable the sign bit is lost. As an example, when an 8 bit value is moved into a 16 bit value the sign is no longer in the highest position, so it doesn't act as the sign. Intrinsic number 4 (EXTEND) is used to move the sign bit of an 8 bit value into the sign bit of a 16 bit value. Actually, to be correct, the high byte of the 16 bit value is converted to ones or zeros depending upon the sign of the low order byte, as in:

```
        FROG(0):=255;
        PIG:=EXTEND(FROG(0));
```

Here FROG is an address variable and PIG is a normal integer variable. In the example, if PIG were to be printed or compared, etc., its value would be -1.

INTEGER ARRAYS

Integer arrays are similar to byte arrays except that each item is a 16 bit signed number, which allows us to extend the concept. Each item in an integer array can be a number in the range of -32768 to 32767 or a memory address in the range 0 to $FFFF.

Integer arrays are declared in the same manner as normal integers. The distinction comes in their use. When they are used as arrays the item number, or "index" is included. The presence of the index causes XPLO to use the value of the integer as a pointer to an integer array, rather than as a simple integer.

After the integer is declared memory space must be reserved for the array. Since these arrays are two byte values, the reservation requires twice as many bytes as a byte array. For example, if you wish to make a 20 element array, you must reserve 40 bytes. Here is an example:

```
'INTEGER' FROG;
.
.
FROG:=RESERVE(40);
X:=FROG(0);
Y:=FROG(19);
```

Once the declaration and reservation is made, individual items in the array can be referenced by using the item number as an index after the variable name:

```
Y:=FROG(20)
FROG(14):=3200
'FOR' X:=1,100 'DO' COW(X):=-234;
```

OWN ARRAYS

Normal RESERVED integer arrays are used for most array applications in XPLO. However the fact that they are dynamically allocated can present difficulties. For example if we want to create a sine table array for a sine procedure then we would prefer to not have to refill the array every time we call the sine procedure. We could make the table outside the procedure but this is unclean. The solution is to use an OWN array.

OWN arrays are a declared form of array. The space they use is permanent and is saved with the program. Because they are declared they must be of a fixed size. Here is an example declaration of a 10 element OWN array:

```
'OWN'DATA(10);
```

The elements of this array are DATA(0) through DATA(9).

Note that, although the contents of OWN arrays are 2 byte integers, the dimension we specify is the number of values in the array, not the number of bytes. Also, be aware that since OWN arrays are saved with programs big arrays will make the program big.

MULTI-DIMENSIONED ARRAYS

XPLO has the ability to manipulate INTEGER and OWN arrays with an unlimited number of "dimensions". A multi-dimensional array uses several indices to select the individual items.

In the simplest form, multi-dimensional arrays can be visualized as a grid of row and column squares that contain data. For example a 5 by 10 element array named DATA would have the elements:

```
DATA(0,0)  .......  DATA(4,0)
.........           .........
.........           .........
DATA(0,9)  .......  DATA(4,9)
```

This type of data structure has many uses, such as board games, tables, etc.

The easiest way to setup a regular multi-dimensional array like this is to declare it in an OWN declaration:

```
'OWN'DATA(5,10);
```

When the array should be dynamically allocated so as to limit the program size and memory use we can do the same thing with standard INTEGER arrays by perfoming a multi-dimensional RESERVE.

In particular, to generate a two dimensional array:

1. DECLARE THE INTEGER NAME
2. RESERVE THE FIRST DIMENSION
3. USING A FOR LOOP, RESERVE THE SECOND DIMENSION

For example:

```
'INTEGER' FROG;
.
.
FROG:=RESERVE(50);
'FOR' X:=0,24 'DO' FROG(X):=RESERVE(50);
```

In this example, a two dimensional array is created, with 25 rows and 25 columns. First a 25 element integer array is created, then the address of the start of another 25 element array is stored into each element of the integer array. Once the array has has been reserved, the individual elements are referenced:

```
        FROG(23,1);
        Z:=COW(X,Y);
        PIG(A,B):=COW(X,Y);
```

For more dimensions, another set of reservations can be made:

```
        'FOR' X:=0,24 'DO' 'FOR' Y:=0,24 'DO'
                FROG(X,Y):=RESERVE(50);
        .
        .
        Z:=FROG(A,B,C);
```

COMPLEX DATA STRUCTURES

XPLO handles these arrays in an unusual and very powerful way that allows the user to produce very flexible data structures. With these features, it is very easy to produce linked lists, trees, non-rectangular arrays, arrays of strings, etc.

An XPLO structure can have any number of indices, and hence dimensions. Further, it need not be regular in the sense that the rows need not all be of the same size. In fact the elements of a row need not contain the same thing or be used in the same way.

Since each element in an integer array is a 16 bit number, the value stored in an array element could be an integer, the address of a string or of another array:

```
FROG===========>--------
                FROG(0) ==============>-----------
                -------               !FROG(0,0)!
                FROG(1)=======I       !---------!
                -------       I       !FROG(0,1)!
                              I       -----------
                              I
                              I=======>-----------
                                      !FROG(1,0)!
                                      !---------!
                                      !FROG(1,1)!
                                      -----------
```

This simple technique is used to create all sorts of structures. This manual cannot hope to expand fully on the subject so we will just have to observe that an XPLO structure is, in the end, a "linked list" and refer you to a general programming text on data structures for a comprehensive discussion of the subject.

DATA CONSTANTS

OWN arrays are all very well, but they still have to be filled. If what we want is a fixed table of information a long series of individual stores could be tedious, so XPLO provides the constant data structure.

Here is a simple example:

```
        'INTEGER'DATA;
        .
        .
        DATA:=[2,22,222,2222,22222];
```

This statement is analogous to the text string constant we encountered before. The difference is that the elements are 16 bit integers rather than 8 bit ASCII characters. Thus, in this example, DATA(2) has the value 222.

XPLO stores the information in the saved program space, and when the assignment (:=) is executed, it stores the address of the array into the variable, DATA. The elements of a data constant can be used just like other array elements except that they may not be used to store new data since they are constants.

Data constants can include other data constants and text strings as elements. For example:

```
        INFO:=[1,2,[3,4,[5,6]],"HELLO"];
```

Which is a structure like this:

```
    INFO----> 1
              2
              *---------------------> 3
              *-----> "HELLO"         4
                                      *-----> 5
                                              6
```

Here INFO(0) is 1 while INFO(2,1) is 4 and INFO(2,2,1) is 6. Also, after we store INFO(4) into an ADDRESS variable so that we can use it as a byte array, we have access to the bytes in the string "HELLO". Thus

```
        TEXT(0,INFO(4));
```

will print the string "HELLO" and

```
        'ADDRESS'X;
        .
        .
        X:=INFO(4);
        CHOUT(0,X(1));
```

will print the character "E".

This concept is very flexible and hence its applications potential might be a litle hard to grasp at first sight. It might be wise to experiment with it a bit to become familiar with the whole idea.

ADDR COMMAND

The ADDR command is used to find the actual machine address where a variable is stored.

This operation can be used to pass the address of a variable to a procedure, which provides a mechanism by which we can allow a procedure to store information into a variable which would not normally be in scope, such as in recursive forward references. The purpose of this will be a bit obscure to the newcomer to recursive langauges so don't worry if you don't see it yet, the feature will be there when you need it.

ADDR also gives you an easy way to access individual bytes of a 16 bit integer. To access individual bytes, the address of the integer is passed into the name of a byte array. The high and low bytes of the integer will be in elements 0 and 1 of the byte array.

```
'INTEGER' X;
'ADDRESS' FROG;
FROG:='ADDR'X;
LOW:=FROG(0);
HIGH:=FROG(1);
```

Here the low byte is in "low" and the high in "high".

RULES OF SCOPE

Scope is the concept that variables, procedures and all other named things are only active in certain area of a program. Because this obeys a clearly defined rule, the programmer has complete control of the modularity of a program.

When a name is active it is said to be be "in scope". At any particular point in the program, certain items will be in scope and others will be out of scope. If an item is in scope it is available to be used by the program, but if it is out of scope it is completely unavailable. In fact, when a normal variable is out of scope, it is completely non-existent.

The easy way to define the scope of a name is to say that, when reading the source code, the scope of a name is from its declaration down to the end of the procedure in which the declaration appeared, including any other procedures that may occur in this part of the code.

To understand this completely we must add one clarification: The name of a procedure is a part of the declarations of the procedure that encloses it, so a procedure name is in scope until the end of the surrounding procedure.

Here are some nested procedures with a variable declared at each level:

```
'PROC' ONE;       <-------------------
'INT'X;                              I
                                     I
'PROC' TWO;       <-----------       I
'INT' Y;                     I       I
                             I       I
'PROC' THREE;     <-----     I       I
'INT'  Z;              I     I       I
'BEGIN'                I     I       I
'END';            <-----     I       I
                             I       I
'BEGIN'\TWO                  I       I
'END';           <------------       I
                                     I
'BEGIN'\ONE                          I
ONE;                                 I
TWO;                                 I
'END';           <--------------------
```

The code inside procedure ONE can call procedure TWO because both the call and procedure TWO are within the procedure which surrounds procedure ONE (the main program).

However, code inside ONE cannot call procedure THREE because the scope of THREE ends at the end of the body of TWO, since THREE is inside TWO.

For similar reasons, only variable X is in scope for the code inside ONE, while procedure TWO can access variables X and Y, and can call procedures ONE and THREE.

By the rule of scope, it is clear that a procedure is in scope during its own body code, so a procedure can call itself. More about this shortly.

Two procedures at the same level, but nested inside different procedures, cannot call each other.

```
------------------------------
i   PROCEDURE  "A"     i
i   ----------------   i
i   i   PRO B     i    i
i   ----------------   i
------------------------------
------------------------------
i   PROCEDURE "ONE"    i
i   ----------------   i
i   i PRO "TWO" i      i
i   ----------------   i
------------------------------
```

In this illustration, procedure B and TWO cannot call each other, even though they are nested at the same level, because they are nested within different procedures (A and ONE).

FORWARD PROCEDURES

Ideally speaking, the scope of a name should be the entire section of code enclosed by the surrounding procedure. But the XPLO compiler is a one pass compiler so it does not know about names of things it has not seen yet.

Under ideal scope rules, there are items which could come later in the code and yet be within the surrounding procedure, namely other procedures at the same level but occuring later in the code.

XPLO makes a special allowance for this with the forward procedure declaration which has the form:

```
'FPROC'name,...name;
```

For example:

```
'FPROC'DO,UNDO,BREAK,FIX;
```

This declaration tells the compiler that the four names listed will be procedures, and that they will occur within the present scope and at the current level. Now they all know about each other and can call each other without worrying about which is coded before which.

RECURSION

Recursion is a very powerful programming technique. It is the ability of a routine to call and execute itself. In XPLO, any procedure can call itself. Also, a procedure can call itself indirectly through a second or third routine. For example, a procedure can call a second procedure which in turn calls the first.

The structure of XPLO facilitates recursive programming. Each time a procedure calls itself, the old set of variables for that procedure are saved and a new set is created. Each recursion is a complete, independent operation.

Generally, recursion is used to perform those tasks which are complex, but which can be broken down into several lesser tasks of the same general sort.

Certain operations which are normally very complex can be greatly simplfied by using recursion. The classical examples occur in the area of graphics programs, sorting and searching algorithms, and in language processors.

Recusive programming techniques result in whole new ways of thinking about solving programming problems. These techniques are a whole subject in themselves which this manual cannot do justice to.

If you are interested, you can learn more about recursion from any one of several good texts. Ken Bowles' book, "Problem Solving in Pascal", is a pretty good starting point. If you want to research the subject more fully you will find whole books devoted to recursive programming at any good academic book store.

The linked-list data structure mechanism that XPLO uses was designed with recursion in mind and the two fit well together. So if you get involved with any complex data structure problems, that would probably be a good time to dig into recursion more extensively.

The Hilbert curve demonstration program on the XPLO distribution disk is an example of a simple recursive program.

INPUT AND OUTPUT

All device dependent operations like I-O are handled by intrinsics within the I2L interpreter. These intrinsics take a device number argument which corresponds to the APEX devices that you have implemented with handlers on your system.

The basic intrinsics for I-O are:

```
OPENI(device);               = make device ready for input.
OPENO(device);               = make device ready for output.
CHOUT(device,byte);          = output a byte to the device.
variable:=CHIN(device);      = input a byte from the device.
CLOSE(device);               = close the device.
```

These intrinsics call the APEX device handlers at the corresponding standard APEX entry point. Your APEX manual gives more detail on the devices and the handler entry points.

Procedures to do input and output are task specific and so easy to write using the above intrinsics that XPLO has little I-O beyond them. However, to get you started there are a few handy I-O intrinsics:

```
NUMOUT(device,value);            = print the number.
variable:=NUMIN(device);         = read in a number.
TEXT(device,text pointer);       = print the string.
CRLF(device);                    = new line.
```

File input and output in XPLO can be handled either on a byte stream basis (sequential access) through device number 3, or on a block-by-block basis were the details of the data structure are left up to the program. The READ and WRITE intrinsics are used in the latter case. There is more detail on the subject in the APEX manual and in the section on intrinsics. Some simple examples occur in the programs on the XPLO distribution disk.

XPLO also has intrinsics to handle all the special features of your APPLE.

WRITING A PROGRAM

Writing a program in XPLO is a simple and straight foward operation. Because this is a structured language, it is best to proceed in a logical, step by step manner.

Here is a step-by-step description of the process to illustrate what has been called the "top down" process of writing a program. There are other approved techniques such as the "bottom up" process. The whole point is to keep your thinking and your coding incremental, logical, and in step with each other.

1. Write the main procedure.

Think about what the whole program is supposed to do. Break the task down into simple steps. Think of a name for each of the steps and put the names into the main procedure as procedure calls.

2. Set up the procedures.

At this point it is best to declare the procedures. Most of the time at this stage you will not put any code into the procedures. Just add an empty begin-end to each procedure. This way the program will compile without all of the procedures being finished, and you can test each procedure as it is written with a trial compilation.

3. Declare intrinsics and variables.

Think about which intrinsics will be needed by the program, and declare them. If you later find that you need other intrinsics you can always go back and add them. In some cases, you may wish to declare the whole set of intrinsics, to save the inconvenience of adding them one at a time later. If you are aware of any global variables or address your program will need you can declare them now.

4. Fill in the procedures.

Now go back and write the code for each procedure. You may want to do a test compilation after you write each procedure. If the procedure is complex, you should test the procedures individually.

If you find that any of the procedures are too complex to be coded easily, you can break them down into simpler components and nest them as sub-procedures to that procedure.

It is general practice to indent one tab for each level of nesting of blocks (tab is Control-I). This helps you keep track of begin-end pairs and makes the structure of the program readily visible.

DEBUGGING

Debugging a compiled program is a bit different from debugging an interpreted program, mainly because you have to wait until compile time before errors and error messages appear. Actually, once you get used to XPLO, debugging is easier than with most interpreters. Once again, block structure comes to the rescue. Because the program is modular, it is easy to isolate the problem to a single block or procedure. In fact, if you have problems with a particular block, it probably best to break it down into smaller, less complex procedures.

One of the more useful techniques for debugging an XPLO program, is to temporarily insert commands that print the values of certain variables which allow you follow the flow of the program and get an indication of how the variables are changing. It may even be useful to add statements that print the name of a procedure when it is entered. This way you can tell when and if a procedure is being executed while the program runs.

ERROR MESSAGES

The XPLO package provides two different types of error messages. The first type are messages provided by the compiler when it compiles the program. These are usually syntax errors. The second type of error message is provided by the interpreter, and these appear only when the program is being executed. They are called I2L errors.

If the compiler finds an error during the compilation of a program, the compiler will immediately abort. When it aborts it prints an error message. It also prints the line in which it found the error and to make things even clearer, it prints an arrow pointing to the exact place in the line where it was when it realized that there was an error.

One of the disadvantages of block structured languages is that the errors are often not detectable until some time after the actual source of the problem. In this case matters are made worse by the fact that the compiler will by then have lost its place and quite probably give a nonsense error message. This is just one of the things the block structured language user has to get used to.

I2L errors give an error number and question mark For example:

```
3 ?
```

Here is a list of the possible I2L errors:

1 ?
Illegal division by zero.

2 ?
No more memory space. A RESERVE or the loader tried to exceed the allotted memory bounds.

3 ?
Some device handler returned with the carry flag set, which indicates an I-O error. The most common I-O errors are due to a "not ready" disk drive or to exceeding the allotted output file size.

4 ?
Invalid opcode encountered. This means that either the stack or the program has been destroyed. The common causes of this is that an array index was incorrectly computed or that an intrinsic was incorrectly used.

5 ?
Invalid intrinsic number used. This is usually due to an incorrect CODE declaration, but it could be caused in the same way as error 4.

10 ?
Loader failure. The file being loaded is not a legal .I2L file.

COMMON ERRORS

There are several common errors that seem to catch everyone when they first start programming in XPLO. Here is a list of errors in the order of occurence:

1. There are several atoms in XPLO that must be used in pairs. For example, begin-end. The first error that most people make is to omit one of the pairs. The most likely place that you will do this is with begin-end pairs. It is very easy to get an incorrect number of ENDs at the end of a complex procedure. The easiest way to keep track of begin-ends is to use indentation.

```
'BEGIN'
XXXXXX
XXXXXX
          'BEGIN'
          XXXXXX
          XXXXXX
                    'BEGIN'
                    XXXXXX
                    XXXXXX
                    'END';
          'END';
'END';
```

Each indentation mu have a BEGIN and an END.

ᴾ re are some other pairs to watch for:

```
'--------' SINGLE QUOTES AROUND COMMANDS

"--------" DOUBLE QUOTES AROUND TEXT STRINGS

(---------) PARENTHESIS

[--------] BRACKETS, SAME AS BEGIN-END

\--------\ COMMENTS
```

2. Semicolon can catch you in two ways. One is that there must be one between each statement in the program. The other is that you must not place a semicolon between the 'THEN' and 'ELSE' part of an IF (or CASE) statement.

```
'IF' X=A 'THEN'
          'BEGIN'
          FROG;
          RESTART;
          'END'<=========**** SEMI ILLEGAL HERE
          'ELSE'
          'BEGIN'
          A:=A+1;
          RESTART;
          'END';<==========***** REQUIRED HERE
```

3. Intrinsics require various numbers of arguments. One the of the common errors is to send the wrong number of arguments. This can be fatal and it can cause an I2L error on execution. In fact, there is even the possibilty that I2L will fetch the wrong opcode and destroy part of its own memory image. Under some circumstances, it may be neccessary to reboot APEX.

4. Some intrinsics also return values. If the value is not used as a value, the same kind of fatal error described above can occur. Also, if a variable is stored from an intrinsic that does not return a value, the same kind of error can result.

5. When parameters are passed from one procedure to another, the values passed go into the first variables or addresses declared; and they are always stored in the same order that they are passed. As a program is written, it is quite easy to add new variables or addresses to the declarations which shift the order of declarations and change which parameters are passed to which variables and addresses. Remember that address and variable declarations can be mixed in any way necessary to properly pass values into the correct variables and addresses. It is frequently useful to have separate declarations for values passed and local variables. For example:

```
'PROCEDURE' FROG(A,B,CHAR);
'INTEGER' A,B,CHAR;
'ADDRESS' DEN,SAN;
'INTEGER' X,Y,Z;

'BEGIN'
XXXXXXX
XXXXXXX
```

Here the parameters are passed to the integers in the first declaration and the remaining declarations are locals.

6. XPLO does not do run time bounds checking. Thus a common error in fairly complex programs is to inadvertently store something in an incorrect place in memory.

Almost always, this is due to some error in the calculation of an index to an array. Another way in which you can damage memory content in an Apple is to use graphics and screen intrinsics with incorrect arguments.

The point is simply that the I2L run time routines will not protect you from incorrect operations. This protection is up to the programmer.

If you suspect that this is a problem with some program then an effective solution is to write range checks into the program which you can remove later after the debugging stage is complete.

THE I2L LANGUAGE

It is not necessary to understand the I2L language and interpreter to program in XPLO. All I2L operations are totally transparent to the user.

The I2L language operates in a manner that is very similiar to machine language. It uses single byte opcodes and several different addressing modes.

An I2L program is generally stored immediately following the interpreter. When the program is run, the interpreter begins fetching opcodes from the program store and executes them.

The interpreter uses two different stacks. The first is the regular hardware stack. It is used to store intermediate values while the interpreter evaluates expressions.

The second stack is a special stack built by the interpreter. It is sometimes called the heap. The heap uses the memory immediately following the program store. The heap is used to store the value of dynamically allocated information. The behavior of this stack controls the scope of variables. As the program runs, variables are pushed and pulled and the stack expands and contracts. When a variable comes into scope, it is pushed onto the stack and when it leaves scope, it is pulled from the stack.

Imbedded in the interpreter is a special loader. The loader loads the relocatable hexadecimal version of the I2L code generated by the compiler. Generally, when the I2L interpreter is started, the loader is executed first. When the program is loaded, the interpreter starts executing the program. The program can be saved and re-executed at this point.

INTRINSICS

INTRINSIC 0   value: ABS(value)

This intrinsic returns the absolute value of the argument. If the number is negative, the sign will be removed. Example:

```
        X:=ABS(X);
```

Note that the absolute value of $8000 is $8000

INTRINSIC 1   value: RAN(value)

This intrinsic returns a random number between zero and the argument minus one. Example:

```
        X:=RAN(100);
```

INTRINSIC 2   value: REM(value)

This intrinsic is used with division operations. Since division is an integer operation, a remainder may be left if the numbers are not evenly divisable.

It returns the value of the remainder of the division in the argument expression. If a zero argument is used, the intrinsic will return the remainder of the last division performed.

Care should be taken to keep the intrinsic as close to the target division as possible since some operations perform internal divisions which are not obvious to the user. For example, input and output intrinsics perform division internally. Examples:

```
        X:=REM(X);
        X:=REM(5/2);
        X:=REM(0);
```

INTRINSIC 3   address: RESERVE(value)

This sets aside a certain amount of memory space for use by the program. The intrinsic returns the address of the beginning of the reserved area. The argument specifies the number of bytes to be reserved. Sequential reserves will assign contigious memory space. Example:

```
        DATA:=RESERVE(1000);
```

INTRINSIC 4   value: SWAP(value)

This intrinsic returns the value obtained by swapping high and low bytes of the argument. Example:

```
        X:=SWAP(X);
```

INTRINSIC 5   value: EXTEND(value)

This intrinsic takes the sign bit of the low byte of a 16 bit value and extends this to be the sign of the entire 16 bit number. Example:

```
        X:=EXTEN(X);
```

INTRINSIC 6   RESTART;

This intrinsic immediately terminates execution of the program, sets the RERUN flag to 'TRUE', and restarts the program over from the beginning. All variables except OWN variables are lost.

INTRINSIC 7   value: CHIN(device)

This intrinsic reads in one byte from the specified input device. Example:

```
        X:=CHIN(0);
```

INTRINSIC 8   CHOUT(device,value);

This intrinsic sends one byte to the specified output device. Example:

```
        CHOUT(0,^=);
```

INTRINSIC 9   CRLF(device);

This intrinsic sends a carriage return-line feed to the specified output device.

INTRINSIC 10   value: NUMIN(device)

This intrinsic inputs a decimal integer from the input device specified. Example:

```
        X:=NUMIN(0);
```

INTRINSIC 11   NUMOUT(device,value);

This intrinsic outputs a decimal integer to the specified output device. Example:

```
        NUMOUT(0,X);
```

INTRINSIC 12   TEXT(device,address);

This intrinsic outputs the text string at the address given to the specified output device. The string must be terminated by a byte with the high bit set. Examples:

```
        TEXT(LPT,STRING);
        TEXT(0,"THIS IS A STRING");
```

INTRINSIC 13   OPENI(device);

This intrinsic executes the device handler initialization routine on the specified input device. Example:

```
        OPENI(0);
```

INTRINSIC 14   OPENO(device);

This intrinsic executes the device handler initialization routine on the specified output device. Example:

```
        OPENO(0);
```

INTRINSIC 15   CLOSE(device);

This intrinsic executes the device handler close routine on the specified device. It can be used to close, and thus make permanent, the output file on disk. Example:

```
        CLOSE(3);
```

INTRINSIC 16   ABORT;

This intrinsic forces the program to unconditionally exit through the APEX error exit vector.

INTRINSIC 17   TRAP(boolean);

This intrinsic determines whether or not I-O errors encountered at run time will trapped by I2L. If the trap is set to 'FALSE', then when an error occurs, the error flag (see ERRFLG) will be set to 'TRUE' and control returns to the program for suitable recovery. If, on the other hand, the trap is set to 'TRUE', then an I-O error will cause an I2L error and control will vector through the APEX error exit vector, which usually restarts APEX. The trap is initially set to 'TRUE' at the start of an XPLO program. I-O errors and exit vectors are discussed further in the APEX manual. Example:

```
        TRAP('TRUE');
```

INTRINSIC 18   value: SPACE

This intrinsic returns the number of unused memory bytes available. Be aware that if the free space is greater than 32k, the number returned by SPACE will appear negative. Generally, the maximum reserve is this value minus a few hundred bytes of working space. Example:

```
        BUFFER:=RESERVE(SPACE-1000);
```

INTRINSIC 19   boolean: RERUN

This intrinsic returns a true or false value based on the status of the rerun flag. See SETRUN and RESTART. Example:

```
'IF'RERUN'THEN'....;
```

INTRINSIC 20   value: GETSP

This intrinsic returns the value of the current heap pointer. Be aware that if the heap is beyond $8000 then the number returned will appear negative. Example:

```
X:=GETSP;
```

INTRINSIC 21   SETSP(address);

This intrinsic sets the heap pointer to an absolute memory address. A very risky operation used to force I2L to use abnormal memory locations.

INTRINSIC 22   boolean: ERRFLG

This intrinsic returns true or false depending upon the state of the I2L error flag. This flag starts out 'FALSE', is set to 'TRUE' by I-O errors (if TRAP is false), and is set back to 'FALSE' whenever it is read by ERRFLG. Example:

```
'IF'ERRFLG'THEN'TEXT(0,"TROUBLE!!");
```

INTRINSIC 23   CURSOR(x-value,y-value);

This intrinsic sets the x,y position of the cursor so that the next character printed on the screen will appear in that location. X is horizontal, 0-39 and Y is vertical, 0-23. X and Y must be legal screen positions or this intrinsic will overwrite other parts of memory. Example:

```
CURSOR(3,4);
```

INTRINSIC 24   SCAN(unit,result,address);

This intrinsic looks for a file by name on a unit and returns its starting and ending block number. It operates using the APEX system function "SCAN". For example:

```
'INTEGER'BLOCK;
.
.
BLOCK:=RESERVE(4);
SCAN(4,BLOCK,"WORK     TXT ");
FIRST:=BLOCK(0);
LAST:=BLOCK(1);
```

Here we look for the file WORK.TXT on unit 4. If it is found the block numbers for it are copied into the second argument, BLOCK, which must be a two integer array. If the file is not found a trappable I-O error occurs. The file name must be given as a 12 byte string with the name, padded with spaces, in bytes 0-7 and the extension in bytes 8-10. The period that normally separates the name from the extension is not used.

Scan will use the first three pages of the APEX input buffer when it runs.

Examples of the use of SCAN occur in the sample programs that are distributed with the standard XPLO disk.

INTRINSIC 25 SETRUN(boolean);

This intrinsic is used to force the RERUN flag to true or false.

INTRINSIC 26,27 Unused in this version.

INTRINSIC 28 CHAIN(unit,block);

This intrinsic causes the current program to be terminated and another program to be started. The other program can be any APEX .SAV file. The second argument is the starting block of the file. This can be found using SCAN. Examples of the use of CHAIN occur in the sample programs that are distributed with the standard XPLO disk.

INTRINSIC 29 OPENF(unit,address);

This intrinsic opens a new file for byte input via APEX device number 3. The address argument is the same four byte array that is returned by SCAN, so at this level the file is specified by its starting and ending block numbers. For example, to change device 3 to input from the file "BANANNA.XPL" on unit 0:

```
     INFO:=RESERVE(4);
     SCAN(0,INFO,"BANANNAXPL ");  \FIND IT
     OPENF(0,INFO);                           \OPEN IT
     CHAR:=CHIN(3);                           \READ SOME
     .
     .
```

Note that this is not the normal way of opening an input file for XPLO.

The normal mechanism is simply to specify the input and output files to APEX when the program is being run and then use device 3 as you would any other device:

```
     APX>TEST OUTFIL.DAT<INFIL.DAT
```

Where the program TEST could contain code like this:

```
          'DEFINE'EOF=26;            \END OF FILE MARK
          .
```

```
.
OPENI(3);                     \OPEN INPUT FILE
OPENO(3);                     \OPEN OUTPUT FILE
'REPEAT'
        'BEGIN'
        CHAR:=CHIN(3);  \READ SOME IN
        'IF'CHAR#EOF'THEN'MUNCHONIT;
        CHOUT(3,CHAR);  \WRITE BACK OUT
        'END'
'UNTIL'CHAR=EOF;              \DO WHOLE FILE
.
.
CLOSE(3);                     \MAKE FILE PERM.
```

INTRINSIC 30   WRITE(unit,block,buffer,size);

This intrinsic writes disk blocks to a unit. In APEX blocks are 256 bytes each. The write will occur to blocks on "unit", beginning with block number "block", and continuing for "size" many blocks. The data to be written will come from memory beginning at the address "buffer". The block number to use could have been found for an existing file using SCAN, or possibly read directly from the APEX system page locations which describe the output file, in which case the file may have to be closed as specified in the APEX manual. For example:

```
'ADDRESS'BUFFER;
'INTEGER'SIZE,FIRST;
.
.
SIZE:=10;
BUFFER:=RESERVE(SIZE*256);
FILLBUF;
WRITE(4,FIRST,BUFFER,SIZE);
```

INTRINSIC 31   READ(unit,block,buffer,size);

This intrinsic is the direct companion to WRITE above.

INTRINSIC 32   RESTORE;

This intrinsic calls the APEX system function "RESD", which resets the disk drives.

INTRINSIC 33   SETTXT;

This intrinsic resets the Apple screen to text mode.

INTRINSIC 34   SETHI;

This intrinsic sets the display to high resolution mode on the hires screen buffer from $4000 to $5FFF.

INTRINSIC 35   SETMIX;

This intrinsic sets the display to mixed graphics and text. If the last intrinsic was SETHI the mix will be high resolution graphics and text. If the last intrinsic was SETLO the mix will be low resolution and text.

<u>INTRINSIC 36</u> SETLO;

This intrinsic sets the display to low resolution mode.

<u>INTRINSIC 37</u> boolean: SWITCH(number)

This intrinsic returns the value of a specified paddle switch. It returns a true if pushed and false if not. Example:

```
        'REPEAT''UNTIL'SWITCH(0);
```

<u>INTRINSIC 38</u> value: PADDLE(number)

This intrinsic returns the value of one of four paddles. Values can be in the range 0 to 255 depending upon the position. Paddle numbers can be 0 to 3. The paddles that come with the Apple are 0 and 1. Since the hardware is a bit strange and XPLO is fast you must allow a little time to elapse between paddle reads. Example:

```
        X:=PADDLE(0);
        'FOR'I:=0,9'DO';
        Y:=PADDLE(1);
```

<u>INTRINSIC 39</u> NOISE(volume,period,time);

This intrinsic toggles the speaker at a specified volume, frequency and duration. Volume is either on or off, zero or one. The period of one cycle is about 20 usec times the argument. The duration is about 10 usec times the argument.

<u>INTRINSIC 40</u> CLEAR;

Clears the high resolution buffer so that the whole screen is blank (no points).

<u>INTRINSIC 41</u> DOT(x-value,y-value,mode);

Plots a point in high resolution mode. The point is located at x and y coordinates. "x" being horizonal and "y" being vertical. Since there are 280 possible horizonal points x must be in the range of from 0 to 279. For the same reason, y must be from 0 to 191. A mode argument must also be sent. Off=0, On=$80, Complement (exclusive or) =$40.

<u>INTRINSIC 42</u> LINE(x-value,y-value,mode);

Plots a line in high resolution from the last point plotted to the agument sent.

INTRINSIC 43 MOVE(x-value,y-value);

This begins a line at x and y. It is normally used before LINE.

INTRINSIC 44 value: SCREEN(x-value,y-value);

This intrinsic returns the numerical value (and thus the color) of a specified x,y location on the low resolution screen.

INTRINSIC 45 BLOCK(x-value,y-value,color);

Creates a block in low resolution at the specified x and y coordinates. "x" must be in the range of 0 to 39 and "y" must be in the range of 0 to 47. A color must also be passed. There are 16 possible colors. The values for each color are listed in the Apple reference manual.

INTRINSIC 46-63 Free for user applications.

XPLO REMINDER SUMMARY

STATEMENTS

```
variable:=value;
procname(value,...value);
[statement;statement;...statement];
'BEGIN'statement;statement;....statement'END';
'WHILE'boolean'DO'statement;
'REPEAT'statement;....statement'UNTIL'boolean;
'FOR'variable:=start,end'DO'statement;
'IF'boolean'THEN'statement'ELSE'statement;
'IF'boolean'THEN'statement;
'CASE''OF'value:statement;..value:statement'ELSE'statement;
'CASE'value'OF'value:statement;..value:statement'ELSE'statement;
'LOOP'statement;
'QUIT';
'EXIT';
```

SOURCES OF VALUE

```
VARIABLES:      INTEGERS
                OWNS
                ADDRESSES
                byte array elements
                integer array elements
'ADDR' of a variable
PROCEDURES that return a value
INTRINSICS that return a value
EXTERNALS that return a value
CONSTANTS:      decimal numbers: 123
                hexadecimal numbers: $FE00
                ascii charaters: ^A
                defined constants: 'DEFINE'..
                text strings:   "..."
                data constants: [number,..number]
                'TRUE'
                'FALSE'
```

DECLARATIONS

```
'CODE'name=number,........;        INTRINSICS
'EXTERNAL'name=address,...;        EXTERNALS
'INTEGER'name,name........;        VARIABLES
'ADDRESS'name,name........;        BYTE ARRAYS
'FPROC'name,name..........;        FORWARD PROCEDURES
'OWN'name,name............;        OWN VARIABLES
'OWN'name(dim,dim,..),....;        OWN ARRAYS
'DEFINE'name,name.........;        DEFINED CONSTANTS
'DEFINE'name=number,......;        DEFINED CONSTANTS
'PROCEDURE'name comment;           PROCEDURES
```

OPERATORS

*	MULTIPLY
/	DIVIDE
+	ADDITION
-	SUBTRACTION
>=	GREATER THAN OR EQUAL
<=	LESS THAN OR EQUAL
=	COMPARE FOR EQUALITY
#	NOT EQUAL
>	GREATER THAN
<	LESS THAN
!	BOOLEAN OR
&	BOOLEAN AND
'NOT'	BOOLEAN NOT

OTHER SPECIAL CHARACTERS

space,tab,form-feed,return are separators.

()	EVALUATION PRIORITY
[]	SAME AS BEGIN-END
;	STATEMENT SEPARATOR
\	COMMENT
^	TAKE NEXT CHARACTER LITERALLY
'	SHIFT TO LOWER CASE

MEMORY USE

These are the approximate memory use areas for XPLO V4D. The actual use may be slightly different depending on the exact version you have.

$0056-$00AF	Approximate page zero use.
$0800-$0FFF	Body of the I2L interpreter.
$1000-$117F	Basic set of intrinsics.
$1180-$123F	Intrinsics for disk access.
$1240-$16FF	Intrinsics for Apple special functions.
$1700-$xxxx	User program code.
$xxxx-USRTOP	User program variables and arrays.
$4000-$5FFF	Hires buffer, if used.
$6000-$6FFF	Default I-O buffers, if used.
$7000-$AFFF	APEX executive, if resident.
$B000-$BFFF	APEX required resident code.

The normal XPLO program is, by default, limited to the area from about $1700 to $3FFF, that is, about 10k. If more space is required then the hires buffer can be used. If the program extends beyond $6000 then the I-O buffers will have to be moved and APEX made non-resident by running SET on the I2L.SAV file (see the APEX manual).

The start of the space into which the program will load is mantained in the file "I2L.SAV" at $6D,$6E. By using the APEX GET and SAVE commands the contents of these bytes can be changed to free up extra memory for user functions. If your program does not require the hires and other Apple special functions you can change these bytes to $1240, thus saving a few pages of memory.

IMPORTANT NOTE:

The standard XPLO program space is arranged to use the first text buffer and the second hires buffer. Therefore the MIX intrinsic does strange things. If you need to use the MIX operation, or need more than 10k of program space for a program which uses hires, then you should use I2LBIG intead of the normal I2L. I2LBIG splits the XPLO program across the hires buffer and allows programs to be 24k in size and still use hires. The penalty is that the disk space they use when they are saved is larger since it includes the hires buffer. There is no reason to use I2LBIG if your program does not use hires.