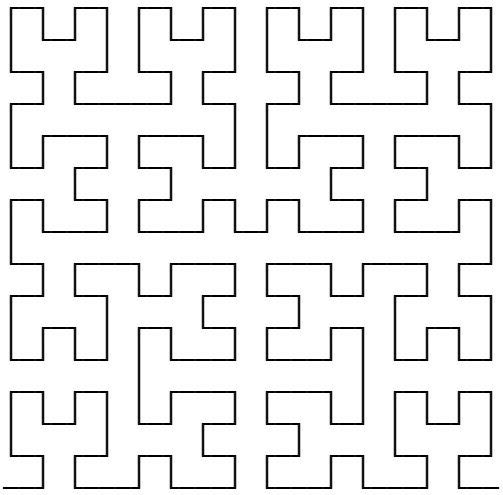
XPL0

PROGRAMMING LANGUAGE MANUAL

VERSION 3.0



All rights to the XPL0 software and its documentation are reserved by the authors. Copyright 2012 software: P. Boyle; manual: L. Fish; revisions: L. Blaney.

This manual is for the small group of individuals who, despite massive support behind other programming languages, continue to use XPL0. It's also for anyone who wonders what all the fuss is about.

Free, open-source versions of the compilers (interpreted, assembly-code compiled, and optimizing) along with many utilities, games and other examples are available from the official website: xpl0.org

C O N T E N T S

* Advanced section

0 : I N T R O D U C T I O N

Welcome to XPL0!

XPL0 is essentially a cross between Pascal and C. It looks somewhat like Pascal but works more like C. It was originally created in 1976 by Peter J. R. Boyle, who designed it to run on a 6502 microcomputer as an alternative to BASIC, which was the dominant language for personal computers at the time. XPL0 is based on PL/0, an example compiler in the book "Algorithms + Data Structures = Programs" by Niklaus Wirth.

Since those early years, XPL0 has been steadily improved and ported to many different computers (6502, PDP-10, IBM-360, homebrews, 8080, 6800, 65802, 680x0, PICs and 80x86). There are versions based on 32-bit integers with megabytes of address space and a version for Windows. This manual describes the 16-bit versions that run on IBM-style PCs under DOS and under versions of Windows that can still run DOS programs.

Programs written in XPL0 include: compilers, operating systems, word processors, video games, and controllers for embedded systems such as medical instruments, astronomical telescopes, and banking machines. These programs might have been written in assembly language, but because they were written in XPL0 they were written quickly, and they are easy to modify.

This manual is both a tutorial and a reference. The information is in a logical order for reference. However, in some cases this makes it more difficult when first learning the language. It's best to skip the sections marked "Advanced" when reading the manual for the first time.

Readers familiar with XPL0 or other programming languages may want to skip to the back. The Addendum lists changes to XPL0 in the last few years. The Syntax Summary and Syntax Diagrams provide a quick way to learn the details of XPL0.

0.0 EXAMPLE PROGRAM: GUESS

A good way to learn a language is to simply jump in and get your feet wet. So let's write a small program in XPL0. We begin by describing the task in plain English.

This program is a guessing game where the computer thinks of a number between 1 and 100, and we try to guess it. After each guess, the program tells us whether we are high or low. The program goes through these steps:

1. Think of a number between 1 and 100.
2. Get a guess from the keyboard.
3. Test the guess against the number.
4. Repeat steps 2 and 3 until the guess is the number.

Here are the same steps translated into XPL0:

```
begin
MakeNumber;
repeat  InputGuess;
        TestGuess
until Guess = Number
end
```

Note that the program is almost word for word the same as the description of the task. First we "make a number" then we repeatedly "input a guess" and "test the guess" until it is the number.

There needs to be more to this program since it doesn't tell how to make a number, input a guess, or test the guess. Each of these operations is a subroutine to the main program. In XPL0 these subroutines are called procedures. We are now going to write each of these procedures.

```
procedure MakeNumber;
begin
Number:= Ran(100) + 1
end
```

This procedure generates a random number between 1 and 100 and puts that number into the variable called "Number".

```
procedure InputGuess;
begin
Text(0, "Input guess: ");
Guess:= IntIn(0)
end
```

This procedure displays the message: "Input guess: " on the monitor (output device 0) and gets a number (INTeger IN) from the keyboard (input device 0). In XPL0 nine different input and output devices can be called from the program. This enables direct access to the monitor, keyboard, printer, disk files, and so forth.

```
        procedure TestGuess;
        begin
        if Guess = Number then Text(0, "Correct!")
        else
                if Guess > Number then Text(0, "Too high")
                else Text(0, "Too low");
        CrLf(0)
        end
```

This procedure is more complicated but still easy to understand. If the computer's number is equal to the guess then we execute one statement; if it's not equal then we execute another statement. 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. CrLf(0) starts a new line on the monitor (Carriage Return and Line Feed).

Here is the complete program:

```
        code Ran=1, CrLf=9, IntIn=10, Text=12;
        integer Guess, Number;

        procedure MakeNumber;
        begin
        Number:= Ran(100) + 1
        end;

        procedure InputGuess;
        begin
        Text(0, "Input guess: ");
        Guess:= IntIn(0)
        end;

        procedure TestGuess;
        begin
        if Guess = Number then Text(0, "Correct!")
        else
                if Guess > Number then Text(0, "Too high")
                else Text(0, "Too low");
        CrLf(0)
        end;

        begin
        MakeNumber;
        repeat  InputGuess;
                TestGuess
        until Guess = Number
        end
```

Two new items are shown here. The command word "code" is used to give names to intrinsics. Intrinsics are built-in subroutines that do common operations. For example, "Ran" is the name of the random-number intrinsic, and "Ran" is used to call this random-number generator as a subroutine. The second item is the command word "integer". This declares

a name and allocates memory space for each variable that follows it.

Note that the main procedure is the last block in the program. An XPL0 program is read starting at the bottom to get the main flow and working upward to get the details in the procedures.

Here is an example of what this program does when it runs:

```
Input guess: 50
Too high
Input guess: 25
Too high
Input guess: 9
Too low
Input guess: 18
Correct!
```

0.1 COMPILING AND RUNNING

After you create a program using a text editor, you compile, assemble, and link it to produce an executable .EXE file. For example, to run the number guessing program, GUESS.XPL, type the following:

```
XN GUESS
GUESS
```

XN is a batch file (XN.BAT) that does these three steps:

1. Run the compiler (XPLNQ) to convert the .XPL source to an .ASM file.
2. Run the assembler (MASM) to convert the .ASM to an .OBJ file.
3. Run the linker (LINK) to combine the .OBJ file with the run-time code (NATIVE.OBJ) and produce an .EXE file.

You can make your program run faster by using the optimizing compiler, XPLX. To do this substitute XX for XN. Also, if your program does many floating-point calculations and is going to run on a computer that has an 80387 math coprocessor (or 486DX or Pentium), you can make it run much faster by linking in NATIVE7 instead of NATIVE.

The above describes how to use the "native" versions of the compiler, but there's another version that compiles "interpreted" code. The native versions produce code in 8086 assembly language. The interpreted version produces code that runs with a program called an "interpreter". Each version has advantages, but the optimizing, native version (XX) is the one that's normally used. Programs are written the same way no matter which version of the compiler is used.

To run the number guessing program using the interpreted version, type:

```
X GUESS
GUESS
```

X is a batch file that does these steps:

1. Run the compiler (XPLIQ) to convert the .XPL source to an .I2L file.

2. Run the interpreter (I2L.COM) to load the .I2L file, combine it with the run-time code, and produce a .COM file.

It's usually preferable to use one of the native versions of the compiler because they produce code that runs about ten times faster than interpreted code. Also, native programs can be much larger than interpreted programs because they are .EXE files instead of .COM files. COM files are limited to 64K bytes in size.

On the other hand, the interpreted version does have some advantages. No assembler or linker is required (which can save a significant amount of space on a floppy diskette). Since the code is not assembled and linked, it can be compiled and run quicker, which may be useful when testing. The interpreted code is also more compact; thus programs require less memory and disk space. In some applications interpreted code is essential because it's being cross-compiled to run on a processor other than the 8086.

0.2 SYNTAX

A program consists of a bunch of characters. The rules that organize these characters into meaningful patterns are called the syntax of a language. Beginning with the most detailed level, the syntax of XPL0 is broken down as follows:

```
Factors
Expressions
Statements
Blocks
Subroutines
```

A factor is the smallest part of a program that can have a numeric value. A factor is usually a constant or a variable. Constants are numbers such as 100, 5280, and 3.14. Variables are places to store numbers. They are given names by the programmer such as "Number", "Percent", and "FEET".

Factors are combined using operators to form expressions. An operator is usually one of the familiar arithmetic operators such as add, subtract, multiply, or divide. An expression calculates to a single value. Here are some examples of expressions:

```
Percent - 10
12.0 * FEET
(Frog + 20.5) / 0.23
```

A statement is a request to do something. A typical statement combines expressions and commands. Here are two statements:

```
Number:= Ran(100) + 1;
if Guess = Number then Text(0, "Correct!")
```

Several statements can be combined into a single statement called a block. A block must start with a "begin" and terminate with an "end". Statements within a block must be separated by a semicolon (;). Here is an example of a block:

```
begin
Number:= 52 + 6;
InputGuess;
if Guess > Number then Text(0, "Too high");
CrLf(0)
end
```

XPL0 is very flexible in the way it allows statements and blocks to be combined. For example, blocks can be placed inside statements:

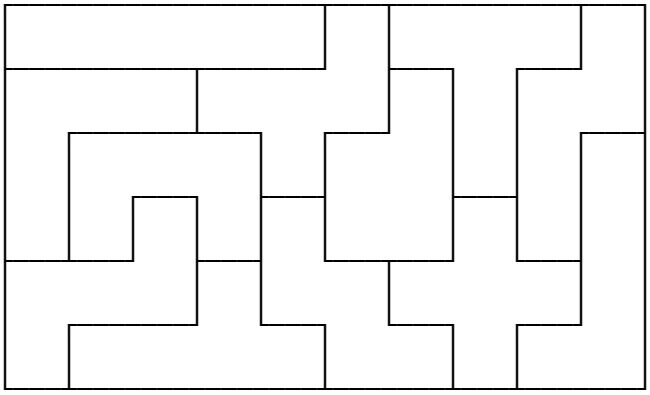
```
if Guess < Number then
        begin
        Text(0, "Too low");
        InputGuess;
        if Guess < Number then Text(0, "Still too low")
        end
```

Here we have an "if" statement containing a block. The block itself consists of three statements separated by semicolons.

Subroutines are the highest level of organization. In XPL0 there are several different types of subroutines; the most common is the procedure. A procedure is a block of statements that does a specific job. A program can contain any number of procedures. Procedures are given names and called as subroutines from other parts of the program. Here is an example of a procedure:

```
procedure InputGuess;
begin
Text(0, "Input guess: ");
Guess:= IntIn(0)
end
```

This gives a you quick idea of what XPL0 is about. In the next sections we will examine each of these levels of syntactic organization in detail.



1 : F A C T O R S

A factor is the smallest part of a program that has a value. Most factors in XPL0 are either constants or variables. A constant is a value that remains unchanged throughout the execution of a program, while a variable is a value that can be changed. Factors are classified as integer or real. An integer is a 16-bit value that represents a whole number. A real number is a floating-point value that's not limited to a whole number and can cover a very large range of values. Thus there are basically four kinds of factors: integer constants, real constants, integer variables, and real variables.

1.0 INTEGER CONSTANTS

In XPL0 an integer constant is a whole number in the range -32768 through 32767. Here are some examples:

```
10              0
-10000          1975
```

1.1 HEX AND BINARY CONSTANTS

Integers can also be written in hexadecimal form. A hex number is an integer in base 16. Hex numbers are indicated by a dollar sign ($). They range from $0000 through $FFFF. Hex is very useful when programming at the machine level. Here are some examples:

```
$123            $1e0
$FFC0           $00ff
```

Note that both upper and lower case letters (A-F and a-f) can be used.

Sometimes it's convenient to think in terms of binary instead of hex. The percent sign is used to represent a binary number. For example, %10011100 is the same value as $9C.

Because binary numbers can blur into meaningless strings of 1's and 0's, underlines can be used to visually break them up, for example, %1001_1100 = $9C. In fact underlines can be inserted into any number, such as $12_34, or -10_000. The underlines are simply ignored by the compilers.

1.2 ASCII CONSTANTS

ASCII characters are often used as constants. A caret (^) converts a character to its ASCII value. For example:

```
^A    =    $41    =     65
^z    =    $7A    =    122
^$    =    $24    =     36
^^    =    $5E    =     94
```

1.3 REAL CONSTANTS

Real constants are distinguished from integer constants by having either a decimal point or an exponent. The exponent is indicated by an "E". For instance, "3E14" means 3 times 10 raised to the 14th power, or 3 followed by 14 zeros. The following are examples of real constants:

```
2.5                    .2
-1000000.              05.e-1
1E6                    -0.00000000000000707
6.63E-34               6.023e+023
```

In XPL0 a real number represents values ranging between ±2.23E-308 and ±1.79E+308 with 16 decimal digits (53 bits) of precision.

Expressions containing reals execute slower than corresponding expressions containing integers. Also, a real number requires four times as much memory as an integer. Thus when an integer is sufficient, it's preferred to a real.

1.4 VARIABLES

Variables are temporary storage places for values. These storage places are given names by the programmer that can be single letters or whole words. Usually names are chosen to describe what the variable contains. For example, if you were calculating interest rates, the interest could be stored in a variable called "Interest". Since XPL0 is a compiled language, long names don't slow execution speed or take up extra memory space at run time (unlike an interpreted language like BASIC).

Variable names contain letters (A-Z, a-z), numbers (0-9), and underlines (_), but the first character must be an uppercase letter or an underline. Here are some examples:

```
X              RATE12              _drawLine
Guess          I_AM_A_NAME         IAmAName
```

Names can be as long as you want, but only the first 16 characters are recognized by the compiler. Upper and lower case letters are equivalent. For example, the following all refer to the same name:

```
Guess           GUESS           GueSS
```

1.5 DECLARATIONS

Before a variable can be used, it must be declared. The integer variable declaration has the general form:

```
integer NAME, NAME, ... NAME;
```

For example:

```
integer Guess, Number, Frog;
```

This declaration tells the compiler that the variables Guess, Number, and Frog are used later in the program.

The word "integer" is a command word. Command words are words that have special meaning to the compiler. They are in lowercase letters. This, for instance, allows you to use the word "Integer" as a variable name.

Since the compiler looks at only the first three characters of a command word, they can be abbreviated. For example, these are equivalent:

```
integer         int
```

Variables that contain real numbers are declared similar to the way integers are declared:

```
real NAME, NAME, ... NAME;
```

In XPL0 all named things, such as variables, procedures, and intrinsics, must be declared before they can be used. The rules for creating variable names, such as starting with a capital letter, apply to all names.

1.6 DECLARED CONSTANTS (Advanced)

Names can also be declared for constants. Constants are different from variables because once they are defined they cannot be changed. Using a

constant is more efficient than using a variable. Giving names to numbers can add clarity to a program. For instance, the name "Highest" might be more meaningful than the number 29028.

Declared constants have the form:

```
define NAME = CONSTANT, ... NAME = CONSTANT;
```

For example:

```
define Summit = 14210, Highest = 29028, Median = 13489.72;
```

In this example Summit and Highest are integer constants, and Median is a real constant.

Any constant can be used in a "define", for example:

```
define A = $41, B = 66, C = -^C, LETTER = B, Number= -3.1E-3;
```

Note that B, once it's defined, can be used to define other constants. Also note that a constant can be signed (- or +).

Sometimes it's useful to have distinct names for things, but the actual value is irrelevant. In fact sometimes we don't want to know the value so that we cannot come to depend on it. For example, we might be working with a set of colors that we just want to distinguish by name. If we come to depend on the particular numerical value of a color, later changes in the program might be difficult. XPL0 has a simple scheme for defining sets of things:

```
define Red, Green, Blue;
```

Here, all you need to know is that these constants have distinct values.

The values actually assigned by the compiler are integers beginning with zero and incrementing up to the last item in the set. In the example, Red equals 0, Green equals 1, and Blue equals 2. This process is called "enumerating".

If an integer value is specified then any following items progress from it. For example:

```
define Jan=1,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec;
```

1.7 EXAMPLE PROGRAM

This program shows some relationships between the various types of integer factors.

```
code ChOut=8, CrLf=9, IntOut=11, Text=12;
integer Counter;
define Tab=$09;

begin
Counter:= $41;
repeat  ChOut(0, Counter);
        ChOut(0, Tab);
        IntOut(0, Counter);
        CrLf(0);
        Counter:= Counter + 1
until Counter = ^G;
Text(0, "That's all folks!");   CrLf(0)
end
```

When run, this program displays:

```
A       65
B       66
C       67
D       68
E       69
F       70
That's all folks!
```

The program begins by declaring the things that are needed to run it. The first line tells which intrinsic subroutines are needed and gives each a name. The second line declares a single variable called "Counter" that will hold integer values. The last declaration tells us that the word "Tab" can be used as a direct replacement for the hex number $09. This replacement is convenient because the ASCII value of the tab character is equal to $09. These three lines of declarations can be in any order. It's conventional that "code" declarations are first.

The rest of the program describes the actions it performs when it runs. Since this executable part of the program is a block, consisting of several statements, it's enclosed within a begin-end pair.

The first statement in the program block puts the value $41 in the variable called Counter. $41 is the value of the ASCII character A.

Next it repeatedly executes a sub-block until Counter contains a value equal to the ASCII character G.

The sub-block begins by calling the intrinsic subroutine ChOut, to which we send 0 and the value in Counter (initially $41). ChOut (CHaracter OUT) sends a value to a specified output device. Here we are specifying device number 0, which is the monitor. When the monitor driver receives a value, it displays the ASCII character that corresponds to the value. So the first time we call ChOut, an "A" is displayed.

The next line calls ChOut again and sends the ASCII value for a tab character. This moves over to the next tab stop on the monitor.

Now it calls IntOut. IntOut (INTeger OUT) is similar to ChOut, but rather than the value being displayed as a character, it's displayed as a decimal integer. The first time IntOut is called, "65" (= $41) is displayed.

The next statement, CrLf(0) (Carriage Return Line Feed), is an intrinsic that moves to the beginning of a new line on the monitor.

Next, a 1 is added to the value in Counter, and the result is stored back into Counter. On the next line we test the value in Counter to see if it's equal to the value of ASCII G. If it's not then the program goes back to the beginning of the repeat block and repeats the statements starting with ChOut. If Counter has incremented up to G then our program falls through to the next line, which is the Text statement.

Text is an intrinsic similar to ChOut, but it sends out a whole string of characters rather than just one.

Notice the overall logic of the program. It started at A and counted up to G. For each count it displayed the character and its decimal value. When it got to G, it broke the repeat loop and displayed the message "That's all folks!"

1.8 FREE FORMAT

In the examples shown so far, a certain formatting has been implied. Statements, for instance, have been written one to a line. XPL0 is a free-format language, which means that the compiler ignores formatting characters such as spaces, tabs, carriage returns, and form feeds. These characters are only used to make the structure of the program more apparent to the reader.

The previous example program could be rewritten as follows without changing the way it compiles or runs:

```
code ChOut=8, CrLf=9,IntOut=11,Text=12; integer
Counter; define Tab = $09; begin Counter := $41;
repeat ChOut ( 0,Counter);ChOut(0,Tab);IntOut( 0

,Counter ) ; CrLf(0);Counter := Counter + 1 until
Counter  =^G;Text(0,"That's all folks!" );CrLf(0 ) end
```

However this hides the structure, making it more difficult to see what the program does.

Formatting characters can be left out, but they cannot be used everywhere. Just as with normal English, words cannot be split apart. For example, this would cause a compile error:

```
Count er:=$41;
```

2 : E X P R E S S I O N S

XPL0, like many computer languages, is a mathematical language. It does arithmetic and other operations on numbers. Expressions consist of factors and operators. Operators perform on anything that has a value, such as constants, variables, and sub-expressions. An expression calculates to a single value. In XPL0 an expression can be used anywhere a value is used and vice versa.

2.0 ARITHMETIC EXPRESSIONS

The common arithmetic operations are done using familiar symbols:

+	Addition
-	Subtraction
*	Multiplication
/	Division

An arithmetic expression is evaluated from left to right, with multiplication and division done first followed by addition and subtraction. The order of evaluation is important because it can affect the result.

Sometimes it's necessary to evaluate an expression in a different order. The part of an expression within parentheses is evaluated first.

Here are some examples of arithmetic expressions:

6 + 4/2	equals 8	(6+4)/2	equals 5
6 - 4*2	equals -2	(6-4)*2	equals 4
6/2*3	equals 9	6/(2*3)	equals 1
6-4+2	equals 4	6-(4+2)	equals 0
3*(6-(4-1))	equals 9		

Integer division gives a quotient and a remainder. The remainder of the most recent division is gotten from the intrinsic "Rem". For example, 19/5 evaluates to 3, and Rem has the remainder 4.

Note that integer division does not work the same way as division using real numbers. For example, these three expressions are not necessarily equal:

```
X/10 * 5
X*5 / 10
X * (5/10)
```

For instance, if X is 15 then the first expression evaluates to 5, the second to 7, and the third to 0.

Integer operations do not give an error if they overflow. Overflowing values wrap around. For example, if you add 32767 + 1, the result is -32768. This is desirable because $7FFF + 1 = $8000, and so forth.

2.1 MIXED MODE

XPL0 does not allow integer and real factors to be used directly together in the same expression. For instance:

```
2 + 3.5
```

This would cause a compile error. It should be changed to:

```
2. + 3.5
```

To do calculations on a mixture of reals and integers, you must convert the factors to a single data type using the Fix and Float intrinsics. Fix rounds a real to its nearest integer, and Float converts an integer to a real. For example, if the variable X is a real and I is an integer then calculations can be done as follows:

```
Fix(X) + I
X + Float(I)
```

2.2 UNARY OPERATORS

Since a constant can be negative, we could have an expression like:

```
2 * -3
```

Do not confuse the minus sign shown here for the minus sign used to do subtraction. This minus sign is called a unary operator because it operates only on the 3 and indicates that the 3 is negative.

Any factor (or sub-expression) can have the unary operators "-" and "+". Because the "+" operator really doesn't do anything, it can always be left out. It's sometimes used to emphasize that a number is positive.

When unary operators are used in expressions with other operators, the unary operations are done first unless parentheses are used to force a different order of evaluation.

Here are some expressions with unary operators:

```
2 * -3   equals -6          +2 +2      equals 4
6+ -4    equals 2           -$40/16    equals -4
-4 - -6  equals 2           -^A + $41  equals 0
-(4+6)   equals -10         -2*-3      equals 6
```

2.3 COMPARISONS

It's often necessary to compare one value to another and make a decision based on the result. The following symbols are used to make comparisons:

```
=   Tests for equal values.
#   Tests for not equal values.
<   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 is greater than or equal
    to the second.
<=  Tests if the first value is less than or equal to
    the second.
```

Here are some expressions containing comparison operators:

```
X = 3
A < 0.91
(X+1) >= Y
```

We have already seen an example of how comparisons are used to make decisions. In the number guessing program, one of two statements were executed depending on a comparison:

```
if Guess > Number then Text(0, "Too high")
else Text(0, "Too low")
```

If the Guess was greater than the Number then it was "Too high"; otherwise it was "Too low".

A comparison evaluates to true or false. These expressions evaluate to true:

```
55 > 23
(3*4) # (3+4)
```

And these expressions evaluate to false:

```
(2+2) = 5
-33.3 > -4.5
```

WARNING: Since XPL0 treats all 16-bit integers as signed,

```
$F000 > $A000   is true, but
$F000 > $7000   is false.
```

Converting the hex to decimal makes the reason apparent:

```
-4096 > -24576   is true, and
-4096 >  28672   is false.
```

2.4 TRUE and FALSE (Advanced)

When a comparison is made, it produces a true or false value, like 2 + 3 produces the value 5. The reserved word "false" is just another way to represent the integer 0, and likewise "true" is equal to -1 (=$FFFF).

Using these concepts and adding the new variable High, the previous example from the GUESS program can be rewritten as:

```
High:= Guess > Number;
if High = true then Text(0, "Too high")
else Text(0, "Too low")
```

Going one step further, since High is assigned either true or false and since:

```
true = true    is true
```

and:

```
false = true    is false,
```

the "if" statement can be simplified to:

```
if High then Text(0, "Too high")
else Text(0, "Too low")
```

2.5 BOOLEAN EXPRESSIONS (Advanced)

A boolean is a value that has two states: true or false. In XPL0 integers are used to represent booleans. Boolean expressions are formed by combining booleans with boolean operators.

XPL0 has four boolean operators: "not", "and", "or", and "exclusive or". The following symbols and words perform these operations:

```
~   not
&   and
!   or
|   xor
```

The "not" operator operates on a single value--it's another unary operator like the minus sign. It simply changes the value to its opposite. For instance, "not true" evaluates to "false". The "and" operator requires two values. If either value is false then the result is false. If both are true then the result is true. The "or" operator also requires two values. If both values are false then the result is false. If either value is true then the result is true. The exclusive or operator "xor" requires two values. If both values are the same then the result is false. If the values are different, the result is true. Here are some examples:

```
if Pig = ~true then Text(0, "Still ok");
if Guess<20 & Number>70 then Text(0, "Way too low");
if Pig ! Bombed then Text(0, "Blew it!")
```

Boolean operators actually use all 16 bits of an integer. Here are some examples, showing 4-bit values for simplicity:

```
~ 1100          1100          1100          1100
= 0011        & 1010        ! 1010        | 1010
              = 1000        = 1110        = 0110
```

Boolean operations set and clear specific bits. One frequent operation is masking, which uses the "and" operator to clear all the bits except the ones of interest. For example, Number & 1 would reveal if Number is even or odd by masking off all but the least significant bit.

The value "true" is not limited to just $FFFF, but is defined as any non-zero value. Thus "anding" an odd number with 1 is 1, which is "true". However, be careful when using values other than $FFFF for "true". There are instances when the "not" of a true value is not false. For example, ~$33 is $FFCC, both of which are non-zero, and thus both are "true".

Expressions can contain boolean operations, comparisons, and mathematical operations. In mixed expressions, arithmetic operations are done first, then comparisons, then boolean "not", then "and", then "or" and "xor". Thus the following expressions are the same:

```
(A = 1) & (B = 2)   is the same as   A=1 & B=2
(X & Y) ! Z         is the same as   X&Y ! Z
```

But these are different:

```
(A & $80) = 0   is different than   A & $80=0
~(X ! Y)        is different than   ~X ! Y
```

A common mistake is to forget to use parenthesis when masking an expression such as this:

```
Number & 7 = 3   is different than   (Number & 7) = 3
```

Boolean operations cannot be done on real numbers. For example, this would give a compile error:

```
Frog & 3.2
```

However, the following example is legal because the comparisons are done first, which produce true or false values for the "and" operator:

```
Frog<3.2 & Toad>=6.3E3
```

Here are some more expressions using boolean operators:

```
true & false            equals false
$A ! 1                  equals $B
false & false ! true    equals true
false & (false ! true)  equals false
~$55AA & $F0F0          equals $A050
~($F0F ! $33)           equals $F0C0
3+1 = 4                 equals true
3=4 & true              equals false
(1 ! $80) = $81         equals true (or $FFFF)
1 ! $80 = $81           equals 1 (or true)
4+1=6-1 & not 10>12     equals true
17/3=5 & Rem(0)=2       equals true
(A&~B ! ~A&B) = (A|B)   equals true
```

2.6 EXAMPLE PROGRAM: SETS (Advanced)

This program shows how boolean operators are used to operate on sets. A single integer can represent a set containing up to 16 elements. The elements are either present or absent, as indicated by set or cleared bits (1 or 0).

The elements that are common to two or more sets are determined by "anding" the sets using the boolean "&" operator. These common elements are called the "intersection" of the sets. Similarly, the "union" of the sets is determined by the "!" operator.

```
\SETS.XPL
code ChOut=8, CrLf=9, Text=12;
int Week, Work, Free;           \Sets of days
int Day;
def \Day\ Mon=1, Tue=2, Wed=4, Thr=8, Fri=$10, Sat=$20, Sun=$40;
\Assign days of the week to the individual bits of an integer

proc Show(SET); \Graphically show the set of days
int Set, Day;
begin
Day:= Mon;
while Day & Week do     \For all of the days of the week do:
        begin
        if Day & SET then ChOut(0, ^X) else ChOut(0, ^-);
        Day:= Day * 2;  \Next day--shift bit left
        end;
CrLf(0);
end;    \Show


begin   \Main
        \Initialize work days and free days to empty sets:
Work:= 0;   Free:= 0;
        \There are 7 days in a week, so set the first 7 bits:
Week:= $7F;
        \Saturday and Sunday are free days:
Day:= Sat;
Free:= Day ! Free ! Sun;   Show(Free);
        \The rest of the week are work days:
Work:= Week & ~Free;   Show(Work);
        \Free is a subset of Week:
if (Free & Week) = Free then ChOut(0, ^O);
        \Week is a superset of Work:
if (Week & Work) = Work then ChOut(0, ^K);
        \Work and Free are mutually exclusive:
if ~(Work & Free) then Text(0, " PETER?");
        \We won't work on Sunday!
if Sun & Work then Text(0, " FORGET IT!");
CrLf(0);
end;    \Main
```

This program produces the following output:

```
-----XX
XXXXX--
OK PETER?
```

2.7 SHIFT OPERATORS (Advanced)

Those familiar with assembly language will recognize the shift operation. The general form of the shift expression is:

```
EXPR << EXPR     or     EXPR >> EXPR     or     EXPR ->> EXPR
```

EXPR is an integer sub-expression--a 16-bit value. "<<" means shift to the left, and ">>" means shift to the right. The value of the first sub-expression is shifted the number of bits specified by the second sub-expression. The value of the second sub-expression should range from 0 through 15. Beware that only the low five bits are used (except on the 8086, which uses 8 bits). This means that attempting to shift 33 places shifts only one place, and attempting to shift -1 places shifts 31 places.

"->>" means shift arithmetic right. The first two shift operators shift in zeros to fill the empty locations. An arithmetic shift fills the empty locations with whatever the most significant bit contains. If the expression on the left side is positive then zeros are shifted in just like the >> operator, but if the expression is negative then ones are shifted in. This preserves the sign bit, and is the same as dividing by powers of two, except it truncates toward minus infinity rather than toward zero.

Here are some examples:

```
1 << 1          = 2
$30 << 2        = $C0
$50 >> 4        = $05
$FF5A >> 4      = $0FF5
$FF5A ->> 4     = $FFF5
```

The shift operator's precedence (priority) is between the unary operators and the multiplication and division operators. The following expressions show this:

```
-1>>8 * 2      =     (-1 >> 8) * 2      = $01FE
2 + 1<<4       =     2 + (1 << 4)       = $0012
```

Multiplying and dividing by powers of two is similar to doing a shift operation. However, note that dividing a negative number gives a negative result, which is not the same as shifting the negative number to the right. Shift operations are faster than multiplying or dividing.

2.8 IF EXPRESSION (Advanced)

Sometimes, rather than calculate a value, we simply want to choose between two values. This can be done using an "if" expression. Do not confuse "if" expressions with the much more common "if" statements that are described later.

The general form of an "if" expression is:

```
if BOOLEAN EXPRESSION then EXPRESSION else EXPRESSION
```

For example:

```
if Guess > Number then 75 else 20+5
```

The "if" expression evaluates to either 75 or 25 depending on the outcome of the comparison. If the comparison is true, that is, if Guess is greater than Number then the entire expression is 75; otherwise it's 25.

Like all expressions, an "if" expression can be used anywhere a value is used. For instance:

```
Text(0, if Guess = Number then "Correct!" else "Incorrect")
```

2.9 CONSTANT EXPRESSIONS (Advanced)

An expression that consists entirely of constants can be used in place of any constant such as in a "define" declaration (or constant array). The compiler calculates the required constant. For example:

```
def     SEC_PER_HR = 60.0 * 60.0;
def     SEC_PER_DAY = SEC_PER_HR * 24.0;
def     HI = ^I<<8 ! ^H;
```

All expression operators can be used. However, function calls, such as Rem(17/5), cannot be used. This means that integers and reals cannot be mixed in an expression since the intrinsics Fix and Float cannot be used.

2.10 CONDITIONAL COMPILE (Advanced)

The command word "condition" is used to conditionally compile sections of code. "condition" must be followed by an expression that evaluates to true or false. If this expression is false then any following code is treated as a comment. This commented-out code must be terminated by a second "condition" that evaluates to true. "condition" works everywhere except inside comments and strings. It can be used to change declarations as well as executable code. For example:

```
def     Debug = true;

condition Debug;
int     X;
condition not Debug;
real    X;
condition true;

begin
cond not Debug;
X:= 3.0;
cond Debug;
X:= 3;
cond true;
. . .
```

"Condition" is intended for commenting out code--not for comments in general. Even though the condition is false, the code that follows is not completely ignored. The compiler is scanning for a lowercase word that starts "con". Also, some minimal syntax checking is done. For instance, a dollar sign ($) must still be followed by a hex digit, otherwise an error is flagged.

2.11 HAZARDS OF REAL NUMBERS (Advanced)

Calculations with real numbers must be done carefully. Unlike integers, there are many instances where a real number is only an approximation of the desired value. For example, just as the value 1/3 cannot be exactly represented by a decimal number (only approximated by 0.333333333333...), it also cannot be exactly represented by an XPL0 real number. The discrepancy is called a rounding error. A real must round the true value to the nearest value it can represent.

Because of rounding errors an expression like:

```
9.0 * (1.0 / 3.0)
```

does not evaluate to exactly 3.0. The intermediate result, 0.33333333333, is not 1/3, and 0.3333333333333333 times 9.0 is 2.9999999999999997. Yet if the order of this calculation is changed, the result is exactly 3.0:

```
(9.0 * 1.0) / 3.0
```

These two expressions are not exactly equal. Thus the first hazard of real numbers is testing for equality. Usually it's only a coincidence if a real expression evaluates to an exact value. This problem is obscured because if we were to output the values of the two preceding expressions using the RlOut intrinsic, we would get 3.0000000000000000 in both cases. The reason is RlOut itself rounds to compensate for slight rounding errors.

The second hazard of rounding errors is that they can accumulate to cause big errors. For example, if an expression such as:

```
3.0 * (1.0 / 3.0)
```

is multiplied by itself 1000 times, the result might be something like 1.000000000000220.

Another hazard to be wary of is loss of accuracy caused by subtracting. For example, the expression

```
1234567890123456. - 1234567890123454. + 1.25
```

equals 3.25, but the same expression evaluated in a different order

```
1234567890123456. - (1234567890123454. + 1.25)
```

equals 1.0.

The discrepancy is caused by not having more than 16 digits of accuracy. When `1234567890123454` is added to 1.25, the result is rounded to 1234567890123455. This discrepancy can be seen two ways. Certainly the difference between 3.25 and 1.0 seems significant, but 2.25 compared to 1234567890123456 is really quite small.

3 : S T A T E M E N T S

Expressions, command words, and sub-statements combine to form XPL0 statements. A statement is a request to do something.

3.0 ASSIGNMENTS

The most fundamental statement is the assignment. It specifies that a value is to be stored into a variable. Assignments have the general form:

```
VARIABLE:= EXPRESSION
```

An assignment uses a colon-equal symbol (:=) to distinguish between comparing two values for equality and storing a value into a variable. The ":=" symbol is pronounced "gets". For instance, the statement X:= 5 + 1 is read: "X gets five plus one."

Here are some assignment statements:

```
Number:= 23;
Time:= Time + 1;
Pig:= Fish = 0
```

In the first statement, 23 is stored into the variable named "Number". The second statement adds 1 to whatever is contained in Time and stores the result back into Time. In the last statement, Pig gets the value "true" or "false" depending on whether Fish is a zero.

3.1 BEGIN - END

"Begin" and "end" are used to designate blocks of code. A block consists of one or more statements that are combined to form a single new statement. This statement has the form:

```
begin STATEMENT; STATEMENT; ... STATEMENT end
```

Note that statements within the block are separated by semicolons.

Each "begin" must have a matching "end". A common programming error is mismatched "begin-end" pairs.

Square brackets ([ ]) can be used instead of "begin" and "end". For example, this is a block:

```
[X:= 12;   Y:= 5]
```

3.2 IF - THEN - ELSE

A characteristic that makes programs seem intelligent is the ability to select alternative courses of action. The "if" statement enables alternatives to be selected based on a condition.

The "if" statement has two forms:

```
if BOOLEAN EXPRESSION then STATEMENT
if BOOLEAN EXPRESSION then STATEMENT else STATEMENT
```

The "if" statement is used to execute statements or blocks of code conditionally. For example:

```
if Number = Guess then Correct:= true else Correct:= false
```

This statement tests to see if Number is equal to Guess. If it's equal, the variable Correct gets the value "true"; if it's not equal then Correct gets "false".

Usually the condition is based on a comparison, but any expression that evaluates to true or false can be used. Here are some examples:

```
if A/B+C-D = (Time+1)/45 then Pig:= true;
if Pig then [X:= 3;  Y:= 4] else [X:= 4;  Y:= 3];
if A=B & C=D then Frog:= 1 else Frog:= 0
```

Two of the examples shown in this section can be simplified:

```
Correct:= Number = Guess;
Frog:= if A=B & C=D then 1 else 0
```

The first simplification is an often overlooked use of boolean expressions. The second simplification uses an "if" expression instead of an "if" statement. Note the difference between the two uses of "if".

3.3 CASE - OF - OTHER (Advanced)

Often a program must decide between more than the two alternatives offered by an "if" statement. Since an "if" statement can contain other statements, "if" statements can be nested. For example:

```
if Guess = Number then Text(0, "Correct!!")
else if Guess < Number then Text(0, "Too low")
else if Guess > 100 then Text(0, "Way too high")
else Text(0, "Too high")
```

However, many levels of nested "if" statements can be inefficient and confusing, so XPL0 has the "case" statement.

The "case" statement has two forms, the first is:

```
case of
        BOOLEAN EXPRESSION: STATEMENT;
        BOOLEAN EXPRESSION: STATEMENT;
        ...
        BOOLEAN EXPRESSION: STATEMENT
other STATEMENT
```

In this form the "case" statement is like the nested "if"s shown above. The first expression that evaluates to true causes the corresponding statement to be executed. If no expression is true then the "other" statement is executed. Note that there is no semicolon before "other". The nested "if" example translates as follows:

```
case of
        Guess = Number: Text(0, "Correct!!");
        Guess < Number: Text(0, "Too low");
        Guess > 100:    Text(0, "Way too high")
other Text(0, "Too high")
```

The "other" cannot be left out, but it can have a null statement:

```
case of
        Number = 1: DoOne;
        Number = 2: DoTwo
other   [];
```

The second form of the "case" statement is used for efficiency. The expressions must all have a common component and must be a comparison for equality, like in the last example. This form is:

```
        case EXPRESSION of
                EXPRESSION: STATEMENT;
                EXPRESSION: STATEMENT;
                ...
                EXPRESSION: STATEMENT
        other STATEMENT
```

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

```
        case Number of
                1: DoOne;
                2: DoTwo
        other   [];
```

Sometimes several different expressions are associated with a single statement. For example:

```
        case Number of
                1: DoOdd;
                2: DoEven;
                3: DoOdd;
                4: DoEven;
                5: DoOdd
        other   [];
```

Here, if Number equals 1, 3, or 5 then the subroutine DoOdd is executed; if Number equals 2 or 4 then DoEven is executed. The "case" allows any number of expressions to select a statement. The form is:

```
        EXPRESSION, EXPRESSION, ... EXPRESSION: STATEMENT
```

So, the example above could be rewritten:

```
        case Number of
                1,3,5:  DoOdd;
                2,4:    DoEven
        other   [];
```

"Case" expressions must evaluate to integers. Reals cannot be used since it's generally a coincidence when two reals are exactly equal. However, a comparison containing reals, such as 2.3 > X, evaluates to true or false, which is an integer expression that can be used by the first "case-of" form.

Note that "case" selectors are not limited to simple constants; they can be any integer expression.

3.4 WHILE - DO

Much of the power of a computer is its ability to do repetitive tasks. In programming it's frequently necessary to make tasks execute over and over. This is called looping. XPL0 has four kinds of looping statements each of which repeatedly execute a block of code if certain conditions are met.

The "while" statement is a conditional looping structure. As long as the condition is met, the following statement or block is repeatedly executed. This statement has the form:

```
while BOOLEAN EXPRESSION do STATEMENT
```

For example:

```
while Guess # Number do
        begin
        InputGuess;
        TestGuess
        end
```

As long as the variables Guess and Number are not equal, the code within the begin-end block is repeated. The program tests the condition at the beginning of the "while" statement. If the condition is false, the block in the loop is ignored. If the condition is true, the block is executed and the code loops back to retest the condition. The condition must eventually become false, otherwise the loop continues forever.

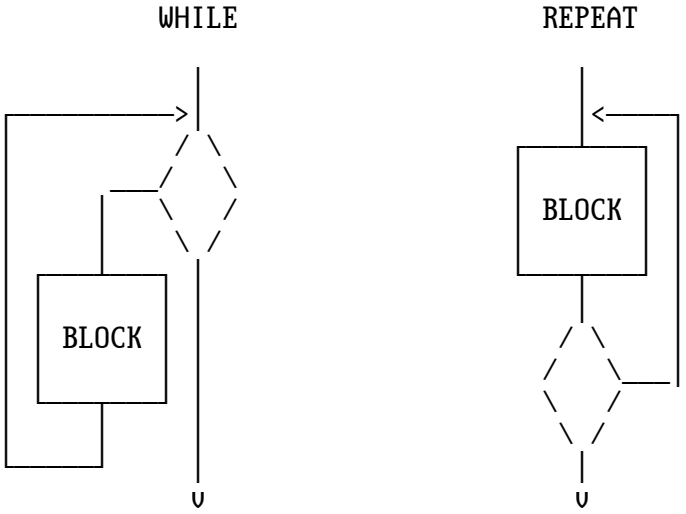
3.5 REPEAT - UNTIL

The "repeat" statement has the form:

```
repeat STATEMENT; ... STATEMENT until BOOLEAN EXPRESSION
```

The "repeat" loop is similar to the "while" loop except that the decision to continue the loop is made after the block.

These flow diagrams show the difference between the "while" and "repeat" statements:



An example of a repeat loop is:

```
repeat  InputGuess;
        TestGuess
until Guess = Number
```

Note that the command words "repeat" and "until" also act as "begin" and "end" for the block in the loop.

3.6 LOOP - QUIT

The "loop" statement has the form:

```
loop STATEMENT
```

A "loop" command repeatedly executes the following statement or block. A "quit" statement is used to exit from any point (or points) within the loop. Usually a "quit" is used in an "if" statement so that the loop exits under certain conditions. For example:

```
loop    begin
        InputGuess;
        if Guess = Number then quit;
        TestGuess
        end
```

3.7 FOR - DO

A "for" loop is a powerful looping statement. It counts one at a time, and for each count it executes a block of code. The starting and ending values of the count are specified, and the count is stored in a variable so that it can be used by the block. This statement has these forms:

```
for VARIABLE:= EXPRESSION, EXPRESSION do STATEMENT
for VARIABLE:= EXPRESSION to EXPRESSION do STATEMENT
for VARIABLE:= EXPRESSION downto EXPRESSION do STATEMENT
```

For example:

```
for Guess:= 1, 100 do TestGuess
```

Guess starts with a value of 1 and steps one at a time up to and including 100. TestGuess is executed 100 times.

The control variable for the loop must be an integer; it cannot be a real nor have a subscript. Negative loop limits are allowed. If the starting and ending limits are expressions, they are evaluated one time before the looping begins. The starting value is assigned to the control variable, and this variable is compared to the ending limit before each pass through the loop.

There are two kinds of "for" loops: incrementing and decrementing. The incrementing version is perhaps the more common, and is shown in the example above. The word "to" can be used instead of the comma if you prefer.

In an incrementing "for" loop if the control variable is greater than the ending limit, the loop is exited; otherwise the block in the loop is executed, and then the control variable is incremented. A decrementing loop uses the "downto" word, and checks if the control variable is less than the ending limit to determine whether the loop is executed or not.

Note that an incrementing "for" loop is not executed if the limits are not in ascending order, as in:

```
X:= -10;
for Guess:= 1, X do Text(0, "Way too low")
```

Also note that 32767 cannot be used as the ending limit because there's not a larger signed number that can be represented with 16 bits. For example, writing "for I:= 32000, 32767 do" causes an infinite loop.

3.8 EXIT

Perhaps the simplest statement is "exit". It terminates the execution of a program at the point it's encountered. This statement is used to halt execution at a point other than the normal end of a program. It's not necessary to put "exit" at the end of a program.

The "exit" statement can also return a code to DOS. The low byte of the value (0-255) of an optional expression following "exit" is returned to DOS interrupt $21 function $4C. This return code can be tested in a batch file with an IF ERRORLEVEL statement. For example, the batch file used to run the compiler (XN.BAT) uses this feature to skip the assembly and link steps if there's a compile error. By convention, a returned value of 0 indicates no errors.

3.9 SUBROUTINE CALLS

Another simple statement is a call to a subroutine. It merely consists of the name of the subroutine, which can be a procedure, an intrinsic, or an external. (This is explained further in 4: SUBROUTINES.)

A call can send some values, known as arguments, to the subroutine. In this case the call has the form:

```
NAME(EXPRESSION, EXPRESSION, ... EXPRESSION)
```

Here are some examples of subroutine calls:

```
MakeNumber;
CrLf(0);
Text(0, "Too low")
```

The first example is a procedure call. The second example calls the new-line intrinsic and passes the argument 0. The last example is an intrinsic call with two arguments.

3.10 COMMENTS

Comments are an important part of a program. Not only do they help others understand what a section of code does, but they often help the programmer understand weeks or years later what was done. A comment can go almost anywhere (except in the middle of a name, inside a string, or in an "include" file name). A comment is enclosed in backslash (\) characters, unless it's the last item on a line, in which case only the leading backslash is needed.

Since backslashes turn comments on and off, a comment cannot ordinarily contain a backslash. However, if two backslashes are used together (\\) then anything on the rest of the line is treated as a comment. This is especially useful when commenting out lines of code that contain comments. Here are some examples:

```
begin           \Move down the page
for X:= -10, 10 \Twenty-one times\ do CrLf(0);
\\for X:= -10, 10 \Twenty-one times\ do CrLf(0);   debug
```

3.11 NULL STATEMENTS

The null statement does nothing. It consists of nothing, and it compiles into nothing. It's useful because in some circumstances we want to do nothing. An example of this was shown with the "other" part of a "case" statement. Here are some more examples:

```
for I:= 1, 1000 do [];          \Kill some time
while not Strobe do;            \Wait for Strobe to be "true"
repeat until KeyStruck          \Another form of wait
```

Each of these statements contains a null sub-statement.

Null statements are frequently used as a coding convenience--a kind of XPL0 slang. For example, these two blocks compile into exactly the same code:

```
begin                   begin
X:= X + 1;              X:= X + 1;
Y:= Y - 1               Y:= Y - 1;
end                     end
```

Note that the block on the right actually contains three statements: the two assignments and a null statement after the second semicolon.

This is convenient because now we can simply insert or delete statements by inserting or deleting lines and not worry about a semicolon on the previous line. Here you might think of semicolons as statement terminators, but they are actually statement separators.

Unless you understand the concept of null statements, you can become confused by semicolons, especially in if-then-else statements. A semicolon is used to separate statements and procedures and to terminate declarations.

3.12 EXAMPLE PROGRAM: THERMO

The following program uses real numbers to convert degrees Fahrenheit to degrees Celsius.

```
\THERMO.XPL     01-AUG-2011
\This program prints a table of Fahrenheit temperatures
\ and their Celsius equivalents.

code    CrLf=9, Text=12;
code real RlOut=48, Format=52;
real    Fahr,   \Fahrenheit temperature
        Cel;    \Celsius temperature

begin
\Print table heading:
Text(0, "FAHRENHEIT     CELSIUS");
CrLf(0);

Format(3, 1);           \Define real-number format

Fahr:= -40.0;
while Fahr <= 100.0 do
        begin
        Cel:= 5.0/9.0 * (Fahr - 32.0);  \Calculate Celsius
        RlOut(0, Fahr);                 \Print out results
        Text(0, "               ");     \(2 tabs)
        RlOut(0, Cel);
        CrLf(0);
        Fahr:= Fahr + 20.0;             \Next step
        end;
end;
```

When THERMO executes, it displays the following:

```
FAHRENHEIT      CELSIUS
-40.0           -40.0
-20.0           -28.9
  0.0           -17.8
 20.0            -6.7
 40.0             4.4
 60.0            15.6
 80.0            26.7
100.0            37.8
```

CrLf and Text are intrinsics we have used before, but RlOut and Format are new. RlOut (ReaL OUT) outputs real numbers in a format specified by Format. Here we are specifying a format of three places (including the minus sign) before the decimal point and one place after it.

4 : S U B R O U T I N E S

One of the most important constructs in programming is the subroutine. XPL0 has four different kinds of subroutines:

Procedures
Functions
Intrinsics
Externals

4.0 PROCEDURES

Scattered throughout most programs are certain operations that must be done over and over. To avoid writing the same code over and over, a programmer puts the common code into a single routine that is called whenever the operation is needed. After the common code is executed, the program resumes at the point following the call. Such a routine in XPL0 is called a procedure.

Any block of code can become a procedure simply by giving it a name. The process of naming a procedure is a declaration. Procedure declarations have the general form:

```
procedure NAME(COMMENT);
DECLARATIONS;
STATEMENT;
```

For example, here's a simple procedure:

```
procedure MakeNumber;
begin
Number:= Ran(100) + 1;
end;
```

Once a procedure is declared, it can be executed simply by calling its name. For instance, here's a block that calls three procedures:

```
begin
MakeNumber;
InputGuess;
TestGuess;
end;
```

A block of code does not necessarily need to be called more than once to justify making it into a procedure. An important use of procedures is to make a program more understandable by breaking it down into smaller, simpler pieces. By making a piece of code into a procedure, you can name it according to its use, test it separately, and keep the main body of code uncluttered.

4.1 LOCAL AND GLOBAL

Names are active only in certain areas of a program. These areas are defined by the rules of scope (see: 4.7 Scope). A name that's declared within a procedure is said to be local to that procedure. A name that's defined for several procedures is global to those procedures.

A procedure is an independent piece of code that can contain its own declarations. For example:

```
code Ran=1;
integer Number;

        procedure MakeNumber;
        integer Times, X;       \Local variables
        begin                   \Randomly pick a random number
        Times:= Ran(10);
        for X:= 0, Times do Number:= Ran(100) + 1;
        end;

begin
MakeNumber;
end;
```

In this example Times and X are local names while Number, Ran, and MakeNumber are global names.

4.2 ARGUMENTS

It's often necessary to send information to a procedure. Values to be sent are separated by commas and placed between parentheses immediately after the procedure call. These values are the arguments of the procedure. When the procedure is called, these arguments are copied into the first local variables of the procedure. Here is an example:

```
integer A, B, C, Result;

        procedure AddTen;       \Subroutine
        integer X, Y, Z;        \Arguments
        begin
        X:= X + 10;
        Y:= Y + 10;
        Z:= Z + 10;
        Result:= X + Y + Z;
        end;

begin                   \Start of the program
A:= 1;
B:= 2;
C:= 3;
AddTen(A, B, C);        \Procedure call with arguments
end;
```

In this example the second block calls the first. In the process it sends the values of the variables A, B, and C, which are 1, 2, and 3 respectively. When AddTen is called, the values in A, B, and C are copied into X, Y, and Z. The procedure adds 10 to each of these values, sums them into Result (= 36), and returns. The original A, B, and C are not changed by the procedure call.

XPL0 allows a special comment to be placed after the name of a procedure and before the semicolon in the declaration. This helps the programmer keep track of which variables are arguments and which are normal locals. Use the comment to list the arguments in the order they are sent when the procedure is called.

Here is an example of an argument list as a comment:

```
procedure Check(Area, Perimeter);
integer Area, Perimeter;        \Arguments
integer Side;                   \Normal local variable
begin
Side:= Perimeter / 4;
if Side*Side = Area then Text(0, "square")
        else Text(0, "rectangle");
end;
```

Writing Area and Perimeter in parenthesis on the first line shows that this procedure has these two values passed to it as arguments, while Side is simply a normal local variable.

Real values can also be passed as arguments. Be sure to declare the local variables in the same order as they are passed. "Real" and "integer" declarations can be mixed in any order to accomplish this.

The ability to pass values to procedures, with the ability to declare in each procedure just those variables it needs, enables each procedure to be a complete and independent piece of code. This enables it to be debugged separately and copied from program to program.

4.3 NESTING

Since a procedure is an independent piece of code, it can itself contain procedures. Procedures can be nested inside each other. For example:

```
        procedure ONE;

                procedure TWO;

                        procedure THREE;
                        begin
                        ...
                        end;

                begin   \TWO
                ...
                end;

        begin   \ONE
        ...
        end;
```

Look at how these procedures are nested. Procedure THREE is nested inside procedure TWO, which in turn is nested inside procedure ONE.

Procedures can be nested up to eight levels deep. Here ONE is at the highest level, and THREE is at the lowest level. Note that the block for the highest level routine is last, but is executed first.

The same order applies to an entire program. The code for the main routine is always the last block in the program, and this highest-level block is always executed first. In fact, a program is just one big procedure.

4.4 RETURN

Occasionally it's desirable to return from a procedure at a point other than its normal end. This is done using a "return" statement. "Return" forces a procedure to immediately return to its caller. At the end of a procedure, a "return" is implied and need not be written.

The TestGuess procedure used in the number guessing program could be rewritten using a "return" statement:

```
procedure TestGuess;
begin
if Guess = Number then [Text(0, "Correct!");   return];
if Guess > Number then Text(0, "Too high")
        else Text(0, "Too low");
CrLf(0);
end;
```

4.5 FUNCTIONS

The "return" statement is also used to return a value from a subroutine to the calling routine. A subroutine that returns a value is called a "function". A function is similar to a procedure except that it returns a value and is used as a value. A procedure call is a statement, but a function call represents a value and is therefore a factor. The general form of a function is:

```
function TYPE NAME(COMMENT);
DECLARATIONS;
STATEMENT;
```

Since all factors must be distinguished as either integers or reals, the function declaration includes a type specifier. This specifier is either "integer", "real", or none. If the type is not specified (none), the function defaults to integer.

The value to be returned by the function is placed immediately following the "return" command. The general form is:

```
return EXPRESSION;
```

Here is an example of how a function is used:

```
integer X, Y;

        function integer Increment(A);
        integer A;
        begin
        return A + 1;
        end;

begin
X:= 3;
Y:= Increment(X);       \Function call
end;
```

This function increments a value. When the function is called, the value in X is sent to it. This value is incremented and passed back to the caller by the "return" statement. The result (4) is then stored into the variable Y.

Here is an example of a function that returns a real value:

```
        real Angle;

                func real Deg(X);
                real X;
                return 57.2957795 * X;

        begin
        Angle:= Deg(3.141592654);
        end;
```

This function converts radians to degrees. Angle gets 180.0.

Here is an example of a function that returns a boolean:

```
        code ChIn=7, ChOut=8, Text=12, OpenI=13;
        integer Ch;

                function Affirmative;
                begin
                OpenI(0);
                return ChIn(0) = ^y;
                end;

        begin
        Text(0, "Do you want to see the ASCII character set? ");
        if Affirmative then for Ch:= $20, $7E do ChOut(0, Ch);
        end;
```

This function returns "true" if the first character typed on the keyboard is a "y" (as in "yes"), otherwise it returns "false". The OpenI (OPEN Input) intrinsic discards any characters that might already be in the keyboard's buffer, thus assuring that the intended character is used.

If a "return" is used in the main (highest-level) procedure, it has the same effect as an "exit" statement. If an expression follows such a "return", it also has the same effect as an expression following an "exit" statement. (See: 3.8 Exit.)

4.6 INTRINSICS

Intrinsics are built-in subroutines that do a variety of operations, such as input and output, and math functions. There are 81 intrinsics in the run-time code (NATIVE).

An intrinsic, like any named thing, must be declared before it can be used. When an intrinsic is declared, a name is given to its number. The general form of an intrinsic declaration is:

```
code TYPE NAME(COMMENT) = INTEGER, ... NAME(COMMENT) = INTEGER;
```

Here are some examples:

```
code Ran=1, Text=12;
code real Sin(real)=56, Cos(real)=60;
```

Intrinsics can be given any name, but the established names are usually preferred because they are generally recognizable.

Since some intrinsics are used as functions, and since the compiler must distinguish between integer and real functions, an intrinsic declaration includes an optional type specifier. This specifier works the same way as for function declarations except that it defines the data type of all the names following the declaration. In the example, Sin and Cos are trig functions that return real values.

An intrinsic call is identical to a procedure or function call. Arguments, if any, are placed between parentheses immediately following the intrinsic name.

Here are some examples of intrinsic calls:

```
Cursor(20, 12);
Number:= Ran(100);
Height:= Sin(Angle) * 10.0;
```

The first example sends the values 20 and 12 to the cursor positioning intrinsic. In the second example, a random number between 0 and 99 (inclusive) is assigned to the variable "Number". The last example computes the sine of Angle, multiplies it by 10, and stores the result in Height.

Some intrinsics return a value while others do not. Intrinsics that return a value must be used as functions (factors), not as statements, otherwise a run-time error occurs. Conversely, an intrinsic that does not return a value must not be used as a function.

The following is an example of the incorrect use of an intrinsic. This statement is illegal and will cause a run-time error:

```
for I:= 10, 100 do Ran(I);      \A bad statement
```

The error would occur because the random-number intrinsic returns a value that's not used.

See appendix A.0 for a list of the intrinsics and a description of what they do.

4.7 SCOPE (Advanced)

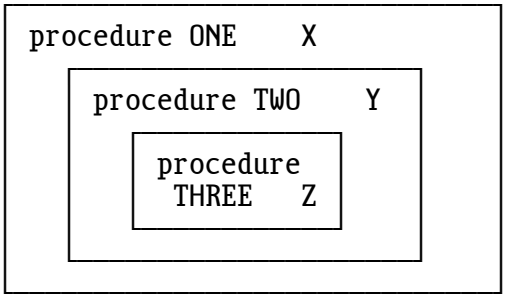
Scope is the feature that makes names active only in certain parts of a program. A name declared in one part does not necessarily conflict with the same name declared in another part. Scope is what makes a program modular.

When a name is active, it's in scope. At any point in the program certain names are in scope and available, while others are out of scope and nonexistent. A name is in scope from the point it's declared to the end of the procedure in which its declaration appears. It is active in any sub-procedures that might be nested in the procedure. Usually we think of scope applying to variable names, but it applies to procedure names, as well as all other names.

Here are some nested procedures with a variable declared in each one:

```
procedure ONE;
integer X;

        procedure TWO;
        integer Y;

                procedure THREE;
                integer Z;
                begin
                . . .
                end;

        begin   \TWO
        . . .
        end;

begin   \ONE
. . .
end;
```

Here is another way of looking at these same nested procedures:

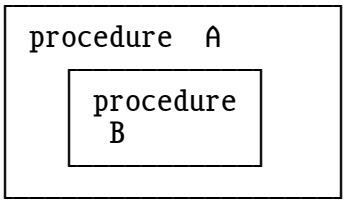


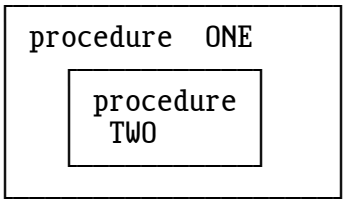
The statements inside procedure ONE can call procedure TWO because both the call and procedure TWO are within procedure ONE. However, the statements inside procedure ONE cannot call procedure THREE because the scope of THREE ends at the end of procedure TWO.

For similar reasons, only the variable X is in scope for the statements inside procedure ONE. Procedure TWO can access variables X and Y, and it can call procedures ONE, TWO, and THREE. Procedure THREE can access all the variables, X, Y, and Z, and can call procedures, ONE, TWO, and THREE.

Note that a procedure is in scope during its own body code, so a procedure can call itself. (See: 4.8 Recursion.)

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





Procedures B and TWO cannot call each other because they are not in scope with each other. The scope of B ends at the end of procedure A. However, statements in procedures ONE and TWO can call procedure A, and conversely, statements in A and B can call procedure ONE. (See: 4.9 Forward Procedures.)

In XPL0 names in scope with each other and at the same level must be unique in their first 16 characters, otherwise a compile error occurs (ERROR 11: NAME ALREADY DECLARED). However, there is no conflict if the identical names are declared in different scopes or in the same scope but in procedures nested at different levels. For example, "integer Frog" can be declared in all four of the procedures: A, B, ONE, and TWO, without conflict. Each declaration creates a separate variable, so there are four unique variables that have the same name.

When the same name is declared at different levels in nested procedures, the most local declaration is used. In the last example suppose the nested procedures A and B both have "integer Frog" declared in them. When a statement in procedure B refers to Frog, it refers to the local Frog declared in B, not the global one in A. Statements in procedure A use the Frog declared in A. It's a good idea to avoid this kind of situation.

4.8 RECURSION (Advanced)

Recursion is a powerful programming technique. It's the ability of a routine to call itself. Recursion provides another approach to solving problems. Some things can be easily defined in a recursive way. For example, an ancestor is a person's father or mother or one of their ancestors. In programming, recursion is used for sorting, searching tree structures, parsing parenthesized expressions, and so on.

XPL0 is designed to facilitate recursive programming. Any procedure (or function) can call itself. A procedure can also call itself indirectly. For instance, a procedure P could call a second procedure Q that calls the original procedure P. Each time a procedure calls itself, the current set of local variables for the procedure is saved and a new set is created.

Here is an example using recursion to compute factorials:

```
        code IntOut=11;

                function Factorial(N);          \Returns N!
                integer N;
                begin
                if N = 0 then return 1          \(0! = 1)
                else return N * Factorial(N-1);
                end;

        begin   \Main
        IntOut(0, Factorial(7));
        end;
```

Seven factorial (7!) is 7*6*5*4*3*2*1, which is equal to 5040.

4.9 FORWARD PROCEDURES (Advanced)

In XPL0 all names must be declared before they can be used. Procedures, in particular, must be declared before they are called. Occasionally a situation arises in recursive programs where a procedure must be called before it's declared. The forward-procedure declaration solves this problem. It has the form:

```
fprocedure NAME(COMMENT), ... NAME(COMMENT);
```

For example:

```
fprocedure MakeNumber, TestGuess, Break, Repair;
```

This declaration tells the compiler that the four names listed are procedures that occur within the present procedure and at the current level. Now that these procedures are declared, they can recursively call each other without regard to the order that they are written.

4.10 FORWARD FUNCTIONS (Advanced)

Forward declarations can also be made for functions. The form is:

```
ffunction TYPE NAME(COMMENT), ... NAME(COMMENT);
```

Forward-function declarations are similar to forward-procedure declarations with the exception that functions must be typed. The type is either "integer", "real", or none. (See: 4.5 Functions.) For example:

```
ffunction real Sinh, Cosh, Tanh;
```

4.11 INCLUDE (Advanced)

Large programs can be broken into smaller, more manageable pieces in several ways. One way is to use the "include" command word to automatically insert another file when you compile your program. For example, it's convenient to "include" the file CODESI.XPL that declares all the intrinsics:

```
include C:\CXPL\CODESI;
```

Note that backslashes specify the path name in the normal DOS manner, and do not indicate a comment in this situation. The default extension is .XPL, so it does not need to be written. Other extensions can be used. Only one file name can follow "include", and it must be terminated by a semicolon.

Any number of files can be included in a program. An included file can itself include other files. Included files can be nested in this fashion up to eight levels.

4.12 EXTERNAL PROCEDURES (Advanced)

When developing a large program, it's inefficient to repeatedly edit, list, and compile the entire program when all the changes are concentrated in one small area. To avoid this, procedures are broken off the main program and put into separate files. These are called "external procedures". They are compiled and assembled separately from the main program, and the resulting .OBJ files are combined using the linker.

Like all named things, external procedures must be declared before they can be called. The form is:

```
eprocedure NAME(COMMENT), ... NAME(COMMENT);
```

For example:

```
eprocedure Baker, Charlie;
```

This means that Baker and Charlie are procedures that are called from the present file, but they exist in another, external file.

The actual procedure in the external file must be prefixed by the command word "public". For example:

```
public procedure Baker;
begin
. . .
end;
```

"Eprocedure" and "public" declarations must be in scope with each other. This means that "eprocedure" declarations must be made at the beginning of the main program, typically right after the "code" declarations, and that "public procedure" declarations must not be nested inside other procedures (except, of course, the main procedure).

Functions also can be external. They are handled like procedures, but since they return a value, they must be identified as "integer", "real", or none. The form of the declaration is:

```
efunction TYPE NAME(COMMENT), ... NAME(COMMENT);
```

External procedures and functions handle local variables and argument passing just as you would expect, but global variables require special consideration. WARNING: Each file must declare global variables in the exact same order. This way the global variables correspond to the same memory locations for each file. A convenient way to make sure that each file has the exact same global variable declarations is to "include" them as shown below.

Here is an example of a program that's divided into three files plus a common global variable file:

```
\GLOBALS.XPL            -- COMMON GLOBALS --
code CrLf=9, Text=12;
code real RlOut=48;

int Flag;
real X;



\PARENT.XPL             -- MAIN PROGRAM --
include GLOBALS;
efunc real Able;        \External procedures & functions
eproc Baker;

begin   \Main
X:= 0.0;
Flag:= false;
Text(0, "EXTERNAL EXAMPLE");   CrLf(0);
RlOut(0, Able(2.0));   CrLf(0);
Text(0, "Global X = ");  RlOut(0, X);   CrLf(0);
X:= X + 1.0;
Baker;
Text(0, "Global X = ");   RlOut(0, X);   CrLf(0);
end;    \Main
```

```
\FILE1.XPL              -- SECONDARY FILE 1 --
include GLOBALS;

public func real Able(X);
real    X;              \Local variable
begin
Text(0, "This is Able");   CrLf(0);
if Flag then X:= X + 1.0;
Flag:= true;
return X * X;
end;    \Able
```

```
\FILE2.XPL              -- SECONDARY FILE 2 --
include GLOBALS;

efunc real Able;        \External function

public proc Baker;
begin
Text(0, "This is Baker");   CrLf(0);
RlOut(0, Able(3.0));   CrLf(0);
Text(0, "Baker's global X = ");   RlOut(0, X);   CrLf(0);
X:= X + 1.0;
end;    \Baker
```

After these files are compiled and assembled, they are linked by the command:

```
LINK /SE:256 PARENT+FILE1+FILE2+NATIVE;
```

The program is run by typing "PARENT", and it displays the following:

```
EXTERNAL EXAMPLE
This is Able
    4.00000
Global X =     0.00000
This is Baker
This is Able
   16.00000
Baker's global X =     1.00000
Global X =     2.00000
```

Several public procedures can be combined into a single file and used as a library. This is like having your own set of intrinsics, and it keeps you from compiling and debugging the same subroutines over and over.

4.13 ASSEMBLY-LANGUAGE EXTERNALS (Advanced)

External subroutines can also be written in assembly language when you want maximum speed or need total control. Assembly-language subroutines are declared using the command word "external". The form is:

```
external NAME(COMMENT), ... NAME(COMMENT);
```

"External" can be declared at any level of procedure nesting, unlike "eprocedure" and "efunction".

In the assembly code, the entry point label must be declared public. For example:

```
CSEG    SEGMENT DWORD PUBLIC 'CODE'
        ASSUME  CS:CSEG
        PUBLIC  DOADD

DOADD:  POP     CX              ;Save return address
        POP     DX              ;Save return segment
        POP     AX              ;Get second argument
        POP     BX              ;Get first argument
        ADD     AX,BX           ;Add arguments
        PUSH    AX              ;Return result
        PUSH    DX              ;Restore return address
        PUSH    CX
        RETF                    ;Far return to caller

CSEG    ENDS
        END
```

This example takes two arguments that are passed on the stack, adds them and returns the result on the stack.

Like intrinsics, it's essential to keep the stack balanced by popping and pushing the correct number of arguments. Also, if you change the stack pointer (SP) or segment registers (CS, DS, SS, ES), they must be restored to their original values before you return. The other registers (AX, BX, CX, DX, SI, DI, BP) need not be preserved. The direction flag bit (D) also does not need to be preserved.

The last example shows one way of getting arguments on and off the stack using PUSH and POP instructions. Another way is to use BP to access all the arguments directly:

```
DOADD:  MOV     BP,SP           ;Get stack pointer
        MOV     AX,[BP+4]       ;Get second argument
        ADD     [BP+6],AX       ;Add to first argument
        RETF    2               ;Drop one argument
```

Local variables can be created by putting them on the stack. This makes a subroutine reentrant, which enables it to be called by an interrupt routine in addition to the XPL0 program (it also enables it to call itself, recursively). For example:

```
        SUB     SP,4            ;Reserve space for two integers
        MOV     BP,SP           ;Get stack pointer
        MOV     AX,[BP]         ;Access one variable
        ADD     AX,[BP+2]       ;Access the other
        ADD     SP,4            ;Drop the local variables
```

You can also create local variables by defining blocks of data using DB, DW, DQ, and so forth, but this makes the subroutine non-reentrant. These variables must be in a segment declared like this:

```
DSEG    SEGMENT WORD PUBLIC 'DATA'
COLOR   DB      0
PIXEL   DW      0
DSEG    ENDS
```

This declaration tells the linker to group your data with the rest of the variables in the program. If you leave the DSEG directive out, your local variables will collide with variables in the XPL0 code. It is also important to link NATIVE last, otherwise similar problems occur.

Accessing global variables is a little more complicated. Generally it's best to pass any variables as arguments. You can even pass the address of a global variable the way arrays are passed. However, global variables can also be accessed using the public label "HEAPLO".

HEAPLO is the bottom of the heap memory space, which is where global variables start. HEAPLO is a public label defined in NATIVE. An assembly-language subroutine can use HEAPLO if HEAPLO is declared external (EXTRN). For example:

```
\MAIN XPL0 PROGRAM
\Start of global declarations
integer Frog, Pig, Cow;
. . .



;EXTERNAL ASSEMBLY-LANGUAGE SUBROUTINE
EXTRN   HEAPLO:WORD             ;Declare HEAPLO as an external
FROG    EQU     HEAPLO+8        ;Define global variables
PIG     EQU     HEAPLO+10
COW     EQU     HEAPLO+12

CSEG    SEGMENT DWORD PUBLIC 'CODE'
        MOV     AX,FROG         ;Accessing global FROG
        MOV     AX,PIG          ;Accessing global PIG
        MOV     COW,AX          ;Accessing global COW
. . .
```

Note that global variables actually start at HEAPLO+8. The first eight bytes are used for a special variable called "global zero". This is used by functions to return values. Eight bytes are used so that reals can be returned as well as integers.

WARNING: Do not give your external assembly-language file the same name as an .XPL file, otherwise when the .XPL file is compiled with one of the native compilers, an .ASM file will be generated that will replace your .ASM file.

4.14 EXTERNAL .I2L PROCEDURES (Advanced)

Up to this point we've discussed externals for the native compilers. In these native versions .XPL code is converted to assembly language, and after being assembled the resulting .OBJ files are combined using the standard linker (LINK). However, the interpreted version does not compile into assembly language, so a different linker is used. XLINK combines the main program with files containing external procedures, and it produces a .C2L file that's loaded and run like a normal .I2L file. For example:

```
XLINK PARENT.I2L+FILE1.I2L+FILE2.I2L
I2L PARENT.C2L
```

The first file after "XLINK" must be the main program, but the other files can be in any order. The linker allows up to 200 external procedures and 1000 calls to those procedures. If two external procedures have the same name, the first one is always called.

The interpreted version can also have external subroutines written in assembly language. An assembly-language subroutine is made into a .COM file, then combined with the main program using XLINK. For example:

```
MASM DOADD;
LINK DOADD;
EXE2BIN DOADD.EXE DOADD.COM
XPLIQ PARENT
XLINK PARENT.I2L+DOADD.COM
I2L PARENT.C2L
```

Note the .COM extension in the XLINK command. This is how the linker distinguishes assembly subroutines from .I2L code.

Assembly-language subroutines used with the interpreted version of XPL0 have several restrictions compared to the native version. Since "public"

names are not used in these .COM files, XLINK uses the name of the file as the name of the subroutine. Because of this, each subroutine must be in a separate file and have its entry point as the first instruction of the file. Also, the name is restricted to eight characters.

Another restriction involves local variables. Subroutines called from the interpreted version must use the stack for any local variables. They cannot be declared using DW or DB because these are not relocated by XLINK.

To get around the limitations imposed by these .COM files, XLINK uses a special type of library file called a "supervisor" file. The supervisor file contains the names of files to be linked. XLINK handles these files just as though their names had been typed on the command line.

The supervisor file does not contain the name of the main program, this must be entered on the command line. All file names in the supervisor file must have extensions, even the .I2L files. Paths can be used. File names are separated by either a plus sign, comma, space, tab, or a carriage return. The supervisor file itself must have the extension ".XLB". Here is an example of a supervisor file and its usage:

```
FILE1.I2L+FILE2.I2L
C:\LIBRARY\DOADD.COM


XLINK PARENT.I2L+SUPER.XLB
```

The simplest use of supervisor files saves the trouble of typing many names on the command line. However, supervisor files can also include other supervisor files to form complex trees of library routines.

5 : A R R A Y S

It is often useful to handle variables as a group when the variables have something in common--like points on a graph or dollars in accounts. In XPL0 variables can be grouped using a single name with each item having a separate number. Such a group is called an array. For example:

```
Account(11)
```

This refers to the 12th item in the array named "Account". If there are 20 items in an array, they are numbered 0 through 19.

In XPL0 there are three types of arrays: integer, real, and character.

Integer arrays are groups of variables where each variable is an integer. Each variable in the array can store a 2-byte value in the range -32768 through 32767 (or $0000 through $FFFF).

The name of an array must be declared before it can be used. Integer array declarations have the general form:

```
integer NAME(DIMENSIONS), ... NAME(DIMENSIONS);
```

For example:

```
integer Account(20);
```

This sets aside memory space for 20 integers and gives this space the name "Account". Now, values can be moved in and out of the elements of this array. For example:

```
begin
Account(19):= 2050;
I:= Account(9) + 100;
. . .
```

Array variables are normally used with an item number in parentheses. This number is called a "subscript", and it can be any integer expression as long as it evaluates to an item number that's in the array.

```
        Account(I+2):= J;
        if Account(0)=$0C then FormFeed;
```

Arrays that contain real numbers are similar to integer arrays. Here is an example:

```
        real  Dollars(70), X;
        int   I;
        begin
        for I:= 0, 70-1 do Dollars(I):= 0.00;
        Dollars(7):= 1.25;
        X:= Dollars(7) - 1.00;
        end;
```

Note that subscripts are always integers, or integer expressions, even for a real array.

Array elements can also be single bytes. Since a byte is often used to store an ASCII character, these arrays are called character arrays. Here are some examples:

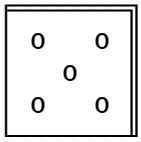
```
        character  Name(20), Address(20), City(10), State(2);
```

Character arrays can have subscripts larger than 32767 (or $7FFF). In this case it's logical to use hex numbers (although negative decimal numbers can be used).

5.0 EXAMPLE PROGRAM: DICE

This little program uses an integer array to represent the six sides of a die. The program simulates throwing the die 10000 times and counts the number of times each side lands up. The sides are numbered 0 through 5 in the array.

```
        \DICE.XPL
        \This program simulates dice throwing
        code  Ran=1, ChOut=8, CrLf=9, IntOut=11;
        integer Side(6), I, N;

        begin
        for I:= 0, 5 do Side(I):= 0;    \Initialize array with zeros
        for I:= 1, 10000 do             \Throw the die 10000 times
                begin
                N:= Ran(6);             \Randomly pick a side
                Side(N):= Side(N) + 1;  \Increment counter for side
                end;
                                        \Show the results
        for I:= 0, 5 do [IntOut(0, Side(I));   ChOut(0, \tab\$09)];
        CrLf(0);
        end;
```



Running this program produced the following output:

```
        1701    1715    1711    1665    1601    1607
```

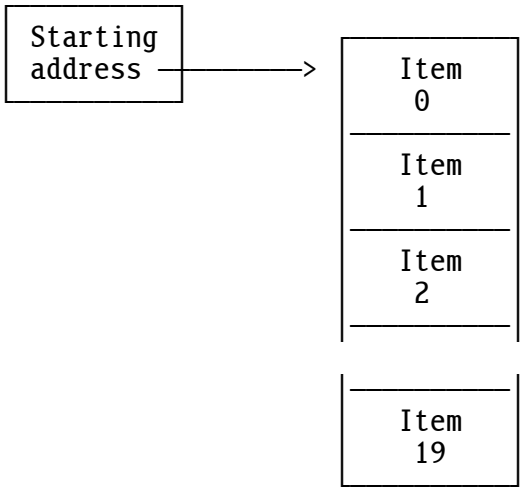
5.1 HOW ARRAYS WORK (Advanced)

When an array name is declared with a dimension in parentheses, memory space is set aside for the items that will be in the array. Memory space is also set aside for the name of the array, just like space is set aside for any variable name. However, the array name is automatically set up with the address in memory where the array items start. The only difference between an array name and an ordinary variable name is that the array name has a value automatically stored into it. This starting address points to the items in the array, and it's called a "pointer".

For example, the declaration

```
        integer Account(20);
```

reserves memory space for 20 integers plus space for one more integer, the variable called Account. The variable called Account is set to point to the start of the space reserved for the 20 integers. Account is normally used with a subscript that refers to one of the items in the array. Account without a subscript refers to the starting address of the array. Here is what this array looks like:



The starting address of an array declared as "real" is handled as a real variable even though it contains a 16-bit address pointing to its data. The address is in the first two bytes, low byte first.

When an array is passed to a procedure, only the starting address is passed, not the actual items in the array. Thus an array passed to a procedure should never have its dimensions declared in the procedure. In other words, the local variable name of the array argument should never have parenthesis showing its size.

Memory used for arrays, as well as variables, comes from an area known as the "heap". The heap has about 60000 bytes and works like a stack but is a little more versatile. When a procedure returns, any arrays and variables that were declared in it are no longer needed. The heap space used by these arrays and variables is released so that it can be used by other arrays and variables in other procedures. This efficient method of using memory is called "dynamic memory allocation". The amount of unused space available in the heap can be determined by calling the Free intrinsic (18). If you have large arrays and need more space, see: 5.9 Segment Arrays.

Declared array dimensions must be constants; they cannot be variables. This is rarely a limitation because any constant expression can be used. For example:

```
def     Size=20;
int     Array(Size);
char    Name(Size*3);
```

If a variable must be used to define the size of an array at run time, it can be done using the method described in: 5.4 Complex Data Structures.

5.2 STRINGS (Advanced)

Another way to set up a character array is to make a text string. For example:

```
"This is a string"
```

This allocates some memory space, fills it with the ASCII for each character, and returns the starting address. If this address is assigned to the character variable S then S is like any other character array except that the contents are already set.

We can read the individual bytes, as in:

```
character  S;
begin
S:= "This is a string";
if S(3)=$73 then Text(0, "It's an s");
. . .
```

Or we can store bytes into this array, as in:

```
S(3):= ^n;   S(5):= ^a;
```

We can output the string to any device using the Text intrinsic. For example:

```
Text(0, S);
```

now displays:

```
Thin as a string
```

on the monitor (device 0).

Note that the quoted string itself allocates the memory space; there is no dimension after the S in the declaration. Writing: "character S(16);" would allocate another 16 bytes that would not be used.

The end of a string is marked by setting the high bit of the last character. This adds $80 (128) to the ASCII value of this character. In the example above, S(15) has the value $E7, which is $80 more than the ASCII for the letter g ($67).

The method for terminating strings can be changed by using the "string" command. If "string 0;" is used then any strings that follow will be terminated with a zero byte instead of having the high bit set on their last character. This has the advantage of making them consistent with the way strings passed to DOS interrupt routines must be terminated. It also enables the extended characters ($80-$FF), such as the line draw characters, to be used in strings. Finally, it provides the possibility for a string that contains no characters, called a "null string".

If you want to change the string termination back to having the high bit set then "string 1;" (or any non-zero integer) will do it. The Text intrinsic (12) works for strings that are terminated by either method.

The caret character (^), besides indicating ASCII values (see: 1.2 ASCII Constants), enables quotes (") and carets to be in strings. For example:

```
Text(0, "^"^^^" is called a ^"caret^"");
```

displays:

```
"^" is called a "caret"
```

A string can contain any printable character. It can also contain control characters like tab, carriage return, bell, and form feed. However, putting a form feed in a string can mess up a program listing, and a control character, such as a bell ($07), won't show in the listing. Thus it's better to use the caret character to put a control character in a string.

Inside a string, ^A means control-A, ^Z means control-Z, and so forth. Do not confuse this use of the caret character with the way it's used to represent an ASCII character outside a string. ^G in a string means control-G ($07, the bell character), but outside a string it means the letter G ($47).

Characters in addition to A-Z can be used with the caret to get the complete range of control characters. The symbols ^@, ^A...^Z, ^[, ^\, ^], and ^_ correspond to the values $00, $01...$1A, $1B, $1C, $1D, and $1F. Note the exception: ^^, which is not $1E but the caret character ($5E) described above. Lowercase letters and characters can also be used. ^`, ^a...^z, ^{, ^|, ^}, and ^~ correspond to the values $00, $01...$1A, $1B, $1C, $1D, and $1E.

5.3 MULTIDIMENSIONAL ARRAYS (Advanced)

Arrays can have more than one dimension. A multidimensional array has multiple subscripts to select an individual element.

A 2-dimensional array can be visualized as a grid of rows and columns that contain data. For example, a 3-by-5 array named "Data" would look like this:

Data(0,0)	Data(0,1)	Data(0,2)	Data(0,3)	Data(0,4)
Data(1,0)	Data(1,1)	Data(1,2)	Data(1,3)	Data(1,4)
Data(2,0)	Data(2,1)	Data(2,2)	Data(2,3)	Data(2,4)

Notice that the order of the subscripts is row followed by column. The rows increase going down, and the columns increase going to the right. (You can reverse this order and think of a 3-by-5 array as having 3 columns and 5 rows, but this is not the order used by matrices and constant arrays.) This kind of data structure is used for many things, such as board games, matrix calculations, and pixel coordinates.

The 2-dimensional array shown above can be set up and used as follows:

```
integer Data(3,5), I, J;
begin
for I:= 0, 3-1 do
    for J:= 0, 5-1 do
        Data(I,J):= 0;
Data(1,3):= 42;
. . .
```

More dimensions can be easily added. Here is a 3-by-5-by-8 array, this time using a real variable:

```
real    Data(3,5,8);
int     I, J, K;
begin
for I:= 0, 3-1 do
    for J:= 0, 5-1 do
        for K:= 0, 8-1 do
            Data(I,J,K):= 0.0;
Data(1,3,7):= 42.0;
. . .
```

Character arrays can also be multidimensional. For example:

```
character String(100,80);
```

This reserves space for 100 strings that are each 80 bytes long. Note that the number of bytes is specified by the last dimension. Single bytes are accessed using a subscript:

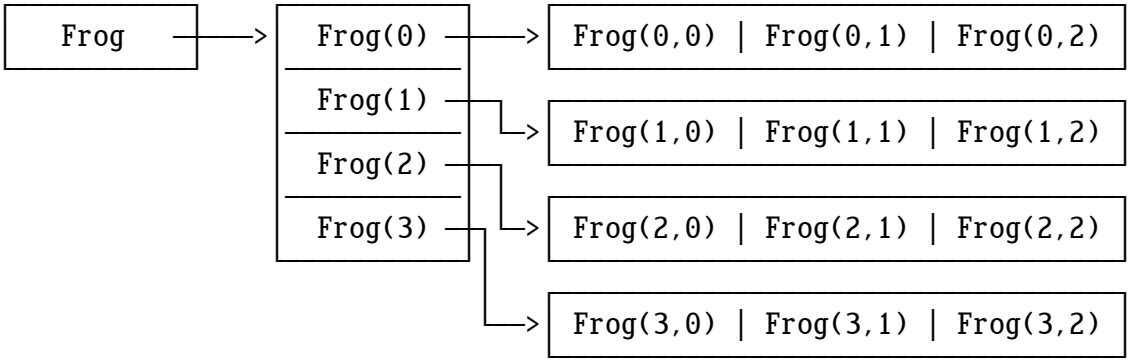
```
String(I,J):= ^A;
ChOut(0, String(99,3));
```

5.4 COMPLEX DATA STRUCTURES (Advanced)

XPL0 implements arrays in a flexible way that lets you build complex data structures that are not limited to the uniform arrays that have been discussed so far.

Each element in an integer array is a 16-bit value. This value can be an integer or the address of another integer array. When a 2-dimensional array is declared, XPL0 reserves the space and sets up pointers to the first and second dimensions. Here is how a 4-by-3 array works:

```
integer Frog(4,3);
```



Like the variable Frog, the elements Frog(0) through Frog(3) contain addresses that point to arrays. These arrays are the second dimension of the original array, Frog.

Normally an element of the array Frog would be accessed like this:

```
I:= Frog(1,2);
```

But note that this is equivalent to these two steps:

```
I:= Frog(1);
I:= I(2);
```

When XPL0 sets up a multidimensional array, it must be uniform. That is, the rows must all be the same length. But you can set up an array yourself and make it any shape you want. The above 2-dimensional array can be set up as follows:

```
integer Frog, I;
begin
Frog:= Reserve(4*2);
for I:= 0, 4-1 do Frog(I):= Reserve(3*2);
. . .
```

The Reserve intrinsic reserves the specified number of bytes and returns the starting address of the reserved memory space. The first statement reserves eight bytes of memory (four integers) and stores the address of this memory space into Frog. Thus the pointer to the first dimension is set. The second statement does the same thing but reserves three integers for each of the four elements in the first dimension of the array.

You could make the first row of the second dimension larger than the others by adding a statement like this:

```
Frog(0):= Reserve(100);
```

Or you could add a third dimension to one of the elements in a row with a statement like this:

```
Frog(1,1):= Reserve(17);
```

Using the Reserve intrinsic, you can make linked lists; you can make trees; you can make any shape data structure you want.

Character arrays and arrays containing real values are set up similar to integer arrays. The only difference for a character array is that the number of bytes is reserved in the last dimension rather than the number of integers (bytes * 2). For example:

```
character Frog(4,3);
```

is equivalent to:

```
character Frog;
int       I;
begin
Frog:= Reserve(4*2);
for I:= 0, 4-1 do Frog(I):= Reserve(3);
```

Setting up real arrays uses the intrinsic RlRes instead of Reserve. The argument for RlRes (an integer) reserves enough memory to hold a real number instead of a byte. A 20-element array would use RlRes(20). For example:

```
real Frog(4,3);
```

is equivalent to:

```
real Frog;
int  I;
begin
Frog:= RlRes(4);
for I:= 0, 4-1 do Frog(I):= RlRes(3);
```

Be careful where you put calls to Reserve and RlRes. Note that the Reserve in the "for" loop reserves more memory each time it's called. Normally reserves are made at the beginning of a procedure to set up a data structure used by the procedure.

Reserved space is allocated dynamically (like any local variable or array space). This means that when a procedure that calls Reserve (or RlRes) returns, the allocated space is released so that other routines can use it. If the procedure is called again, the space is allocated again, but usually the former contents are gone.

A common mistake is to reserve a data structure and use it outside the scope of the procedure that reserves it. A data structure should be reserved in the same procedure that declares the name of the structure. If the name is a global variable then the reserve must be done in the main procedure. Do not call an initialization procedure to reserve this space because the space goes away when the initialization procedure returns.

5.5 CONSTANT ARRAYS (Advanced)

Sometimes what's needed is a fixed table of values. It's possible to assign values to each element of an array, but a better way is to use a constant array. Its general form is:

```
[CONSTANT, CONSTANT, ... CONSTANT]
```

For example:

```
integer Data;
begin
Data:= [2, 22, 222, 2222, 22222];
. . .
```

This array is similar to a text string. The difference is that the elements are 16-bit integer constants instead of 8-bit ASCII characters. In this example, Data(2) contains the value 222. The assignment (:=) stores the address of the array into Data. The elements of a constant array can be used just like other array elements.

Constant arrays can contain real numbers as well as integers and have multiple dimensions. However, reals and integers cannot both be used in a single array. Here is a 2-dimensional, 3-by-5 array:

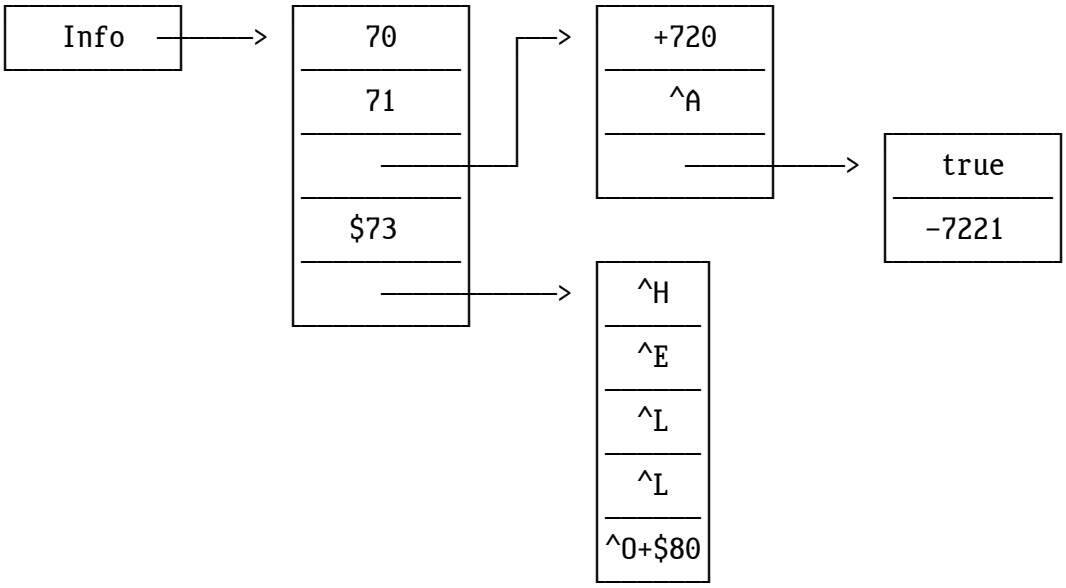
```
real Data;
begin
Data:= [[70.0, 70.1, 70.2, 70.3, 70.4],
        [71.0, 71.1, 71.2, 71.3, 71.4],
        [72.0, 72.1, 72.2, 72.3, 72.4]];
. . .
```

Data(0,0) contains 70.0, and Data(1,4) contains 71.4. Note that the rows are the first dimension.

A constant array can contain other constant arrays and text strings to make complex data structures. For example:

```
Info:= [70, 71, [+720, ^A, [true, -7221] ], $73, "HELLO"];
```

This array has a structure that looks like this:



Here, Info(0) contains 70, Info(2,0) contains 720, and Info(2,2,0) contains "true". Also, after we store Info(4) into a character variable, we can use it as a character array and access the individual bytes in the string "HELLO". For example:

```
character C;
integer   Info;
begin
Info:= [70, 71, [+720, ^A, [true, -7221] ], $73, "HELLO"];
C:= Info(4);
ChOut(0, C(1));
. . .
```

This displays the character "E", and

```
Text(0, Info(4));
```

displays the string "HELLO".

Variables local to a procedure normally don't retain their values from the previous time that the procedure was called. Usually this doesn't matter, but occasionally the value of a variable is needed the next time the procedure is called. A simple way to code this is to make the variable global. However, if the variable is not used by any other procedure, it's best to keep the procedure modular by keeping its variables local. Constant arrays can be used to do this. (Other languages call these "static variables".) Here's an example:

```
proc    MakeNumber;
int     Counter;
begin
Number:= Ran(100) + 1;
Counter:= [0];
Counter(0):= Counter(0) + 1;
if Counter(0) >= 3 then
        begin
        Number:= 50;
        Counter(0):= 0;         \Reset the counter
        end;
end;
```

This procedure sets Number (a global) to 50 every third time it's called. Counter could be declared and initialized in the main procedure, but this way it's kept local to the only procedure that uses it. This makes the overall program more modular and less confusing.

5.6 EXAMPLE PROGRAM: RECORDS (Advanced)

Because of the flexibility of XPL0 arrays, record structures can be made. A record structure is an array that contains elements of different types. In XPL0 integers and reals cannot both appear in a single array. However, integer values can be used to represent such diverse things as numbers, addresses of strings, and elements of a set.

Here is a program that combines the concept of sets with constant arrays and complex data structures.

```
\RECORDS.XPL
code    ChOut=8, CrLf=9, Text=12;

int     File, Person;

def \Person\    Name, SS, Sex, Birth, Dependents, Status;

def \Name\      Last, First;
def \Sex\       Male, Female;
def \Birth\     Month, Day, Year;
def \Status\    Married, Widowed, Divorced, Single;

def \Month\     Jan=1, Feb, Mar, Apr, May, Jun,
                Jul,   Aug, Sep, Oct, Nov, Dec;
```

```
begin   \Main
File:=[ [ ["WIRTH", "NIKLAUS"],
           "701-25-9412",
           Male,
           [Aug, 30, 1944],
           4,
           Married              ],

        [ ["BOREAL", "LENNY"],
           "521-54-1657",
           Male,
           [Oct, 27, 1948],
           1,
           Single               ],

        [ ["MUPPET", "PIGGY"],
           "345-51-7734",
           Female,
           [Feb, 25, 1955],
           1,
           Single               ] ];

for Person:= 0, 2 do
    if File(Person,Sex)=Female & File(Person,Status)=Single then
        begin
        Text(0, "MISS ");
        Text(0, File(Person,Name,First));
        ChOut(0 ,^ );
        Text(0, File(Person,Name,Last));
        CrLf(0);
        end;
end;    \Main
```

This program scans File for nubile females (and old maids) and produces the following output:

```
MISS PIGGY MUPPET
```

The program begins by defining the elements of the set Person. The elements that describe Person are: Name, social security number (SS), Sex, date of Birth, number of Dependents, and marital Status. Some of these elements are in turn defined as consisting of sub-elements. Name, for instance, consists of a Last name and a First name.

All these elements are mapped into the locations of the constant array called "File". The "def" declaration provides names for these locations (subscripts): Name=0, SS=1, Sex=2, etc. File consists of three major elements, or records, of "data type" Person.

5.7 ADDRESS OPERATOR (Advanced)

The "address" operator gives the address where a variable is stored. It has the form:

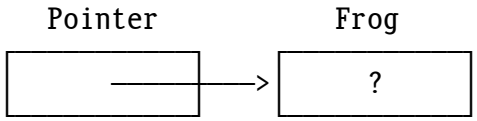
```
address VARIABLE
```

When "address" is written in front of a variable name, the value is no longer the contents of the variable, but the address in memory where the variable contents are stored. Because variable space is dynamically allocated, this address is not determined until a program executes. The variable can be an integer, real, or character, and it can be an array with a subscript. The "address" of a real variable is a 16-bit integer.

The "address" operation on a segment array (see: 5.9 Segment Arrays) is not supported because segment arrays do not have 16-bit addresses.

"Address" is the reverse operation of subscripting an array name with zero. For example:

```
integer Frog, Pointer;
begin
Pointer:= address Frog;
if Pointer(0) = Frog then Text(0, "INVERSE OPERATORS");
. . .
```



"INVERSE OPERATORS" is displayed despite the value contained in Frog because Pointer(0) and Frog both access the same memory location.

The address operator can be used to solve a problem with multidimensional character arrays. Recall that a character array with a subscript always accesses a single byte. However, sometimes we want to access a 16-bit address. Look at this program:

```
char    S;
begin
S:= ["one", "two", "three", "four"];
ChOut(0, S(2,1));
S(1,1):= ^W;
Text(0, addr S(1,0));      \Caution: Text(0, S(1)); will not work
end;
```

When this runs, it displays:

```
htWo
```

Note that "addr S(1,0)" is used in the Text statement rather than "S(1)". This is because S(1) fetches a single byte rather than the entire word that holds the address of the string "tWo". Another solution would be to copy S into a temporary integer variable, for instance I, then I(1) would also fetch the desired address, but this is more awkward.

5.8 RETURNING MULTIPLE VALUES (Advanced)

An "address" operator can be used to return more than one value from a function. Values can always be returned by passing them through global variables, but a better way in some cases is to return them using pointers. For example:

```
code    ChOut=8, CrLf=9, IntOut=11;
int     Frog, Pig(11);
int     Low1, High1, Low2, High2, I;

        proc    MinMax(Array, Size, Min, Max);
        \Returns the minimum and maximum values of the array
        int     Array, Size, Min, Max;
        int     I;
        begin
        Min(0):= Array(0);   Max(0):= Array(0);
        for I:= 1, Size-1 do
                begin
                if Array(I) < Min(0) then Min(0):= Array(I);
                if Array(I) > Max(0) then Max(0):= Array(I);
                end;
        end;    \MinMax

begin   \Main
Frog:= [16, 23, 127, -33, 0];
MinMax(Frog, 5, addr High1, addr Low1);
for I:= 0, 10 do Pig(I):= 2*I*I - 16*I + 20;
MinMax(Pig, 11, addr High2, addr Low2);
IntOut(0, High1);   ChOut(0, $09);   IntOut(0, Low1);   CrLf(0);
IntOut(0, High2);   ChOut(0, $09);   IntOut(0, Low2);   CrLf(0);
end;    \Main
```

This program displays the following:

```
-33     127
-12     60
```

The program displays the minimum and maximum values for two arrays. The calls to MinMax pass the addresses of the High and Low variables, which get values returned to them. The MinMax procedure uses a zero subscript with Min and Max to access the original variables in the calling routine. Compare this to the normal way arguments are passed where only a value is passed to a procedure. This normal way of passing arguments is known as "call by value". What we've done here is what's known as "call by reference" (or "call by address").

Here is another example of using an address operator to pass values to and from a procedure. This program converts rectangular coordinates to polar coordinates, and returns the two polar coordinates back to the calling procedure.

```
        code    CrLf=9;
        code real RlOut=48, Sqrt=53, ATan2=57;

        proc    Rect2Polar(X,Y,A,D);    \Return polar coordinates
        real    X,Y,A,D;
        begin
        A(0):= ATan2(Y,X);
        D(0):= Sqrt(X*X+Y*Y);
        end;    \Rect2Polar

        real    Ang, Dist;
        begin
        Rect2Polar(4.0, 3.0, @Ang, @Dist);
        RlOut(0, Ang);
        RlOut(0, Dist);
        CrLf(0);
        end;
```

Note the use of "@" instead of "addr". The "addr" operator doesn't quite work in this situation because it returns an integer address, and what we need here are pointers to the real variables Ang and Dist. A real pointer is a 16-bit address, but it's packaged in a 64-bit value so it can be handled like a real. It's actually just a 16-bit integer with three more zero integers tacked on. The "@" works exactly the same way as "addr" on integer variables, but it returns a real pointer when used on real variables.

When the above program runs, it displays (angle in radians):

```
            0.64350    5.00000
```

5.9 SEGMENT ARRAYS (Advanced)

Segment arrays solve the problem of the limited 60K heap space. You can have arrays that approach one megabyte in size.

The 8088 microprocessor used in the original IBM PC can address one megabyte of memory. Unfortunately, this memory is divided into 64K-byte blocks called segments. Memory is addressed by a combination of two 16-bit values called a "segment" and an "offset". The value in a segment register is multiplied by 16 and added to the offset to give a 20-bit number that can address one megabyte. This method of accessing memory does not work well for high-level languages because each variable must be addressed using both a segment and an offset. This slows every memory access and complicates pointers.

Other languages deal with this problem by using several different memory models. Each model addresses memory differently. For example, the "tiny memory model" is used by programs that run in less than 64K. In this case addressing is simple: The segment register is set once, and only the offset part of the address is used. If a program needs more than 64K, the "large memory model" might be chosen, which uses both a segment and an offset.

The native versions of the XPL0 compilers (those that produce .EXE instead of .COM files) allow code up to one megabyte, and programs that use segment arrays can address up to one megabyte of data. Of course the actual memory space available is typically less than 640K, the size of conventional memory. If your program needs more memory, you can divide it into modules and use the Chain intrinsic to run a portion at a time. If your data requires more memory, you can use EMS (Expanded Memory Specification) BIOS interrupt $67 or XMS (eXtended Memory Specification) interrupt $15.

Segment arrays are like other arrays except that they can reside in any segment of memory. They can be integer, real, or character arrays. Segment arrays are declared using the form:

```
        segment TYPE NAME(DIMENSION), ... NAME(DIMENSION);
```

For example:

```
        seg int  Length, Angle, Depth;
        seg real Rain, Snow;
        seg char BitMask, OrMask, AndMask;
```

Segment arrays are always 2-dimensional and are normally used with two subscripts. For example:

```
        Depth(X, Y):= 1530;
```

The first dimension contains a list of 16-bit "segment addresses". The second dimension contains a 16-bit offset. When an element of a segment array is accessed, the segment address and offset are combined to form a 20-bit address. Since the microprocessor automatically combines segments and offsets, this operation is relatively fast.

ALLOCATING MEMORY (Advanced)

Segment arrays differ from normal arrays in the way they are reserved. Since segment arrays use memory outside the heap, size declarations (DIMENSIONS) and the Reserve intrinsic are not used for the second dimension. Instead, MAlloc, which stands for "memory allocation", is used. This intrinsic calls DOS and requests some memory. MAlloc allocates memory in 16-byte quantities called "paragraphs". For example, here's how to allocate 64000 bytes for a graphics image:

```
        segment char Pixel(1);
        int     I;
        begin
        Pixel(0):= MAlloc(4000);        \4000 paragraphs = 64000 bytes
        I:= 0;                          \(A "for" loop won't work here)
        repeat  Pixel(0, I):= 0;        \Clear the array
                I:= I + 1;
        until I = 64*1000;
        . . .
```

This provides a segment array that's 1 by 64000. Note that the first dimension is reserved like a normal integer array, and that integers, not bytes, are reserved. The second dimension uses MAlloc, which returns a segment address that points to the start of a 64000-byte block of memory.

If we needed a 320K byte array, a similar process could be used:

```
        segment char Pixel(20);
        int     S, I;
        begin
        for S:= 0, 19 do
                begin
                Pixel(S):= MAlloc(1024);        \1024 * 16 = 16K bytes
                for I:= 0, 16384-1 do Pixel(S, I):= 0;
                end;
        . . .
```

This provides a 20-by-16K array for a total of 320K bytes. Although you could make this a 16K-by-20 array, it's usually better to reserve the large dimension with MAlloc, since this makes better use of the limited (60K) heap space.

Segment arrays also can be used for integer and real arrays. For example, here's how a 10-by-4K array of reals is set up:

```
segment real Data(10);
int     S, I;
begin
for S:= 0, 9 do
        begin
        Data(S):= MAlloc((4096*8)/16);
        for I:= 0, 4096-1 do Data(S, I):= 0.0;
        end;
. . .
```

In the MAlloc statement, 4096 is multiplied by 8 because there are eight bytes in a real number. Note that the total memory allocation is 4096 * 8 * 10 = 320K. This is a large block of memory, and if you have other programs loaded or a limited amount of memory installed, you might not have enough for this array. If DOS is unable to allocate the requested memory, an "OUT OF MEMORY" run-time error occurs.

The largest offset that can be used for each type of segment array is:

```
seg char   $FFFF
seg int    $7FFF
seg real   $1FFF
```

SHORT REALS (Advanced)

To conserve memory, reals can also be stored in segment arrays in a 4-byte short form. Short reals have a range of ±1.2E-38 to ±3.4E+38 with seven decimal digits of precision. Short reals are only used for storage; they are automatically converted to normal 8-byte reals when fetched. As a result, short reals can be used just like normal reals. Segment arrays of short reals are declared as:

```
segment short NAME(DIMENSION);
```

For example:

```
segment short Data(10);
int     S, I;
begin
for S:= 0, 9 do
        begin
        Data(S):= MAlloc((4096*4)/16);
        for I:= 0, 4096-1 do Data(S, I):= 0.0;
        end;
. . .
```

RELEASING MEMORY (Advanced)

A normal array dimensioned or reserved in a procedure only exists as long as the procedure is active. When the procedure returns, the memory used by the array is automatically released. However, a segment array that uses MAlloc does not release memory when the procedure returns. If the procedure is called a second time, more memory is allocated. The Release intrinsic is used to release memory allocated by MAlloc. It requires an argument that is the segment address returned by MAlloc. For example:

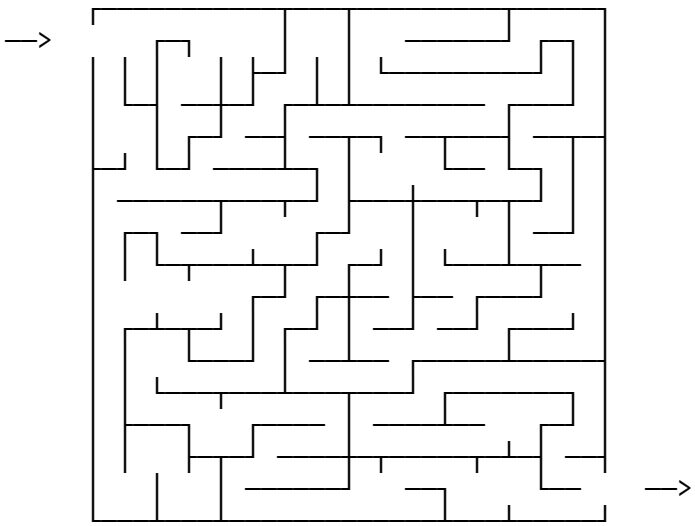
```
proc    Demo;
segment char Pixel(1);
begin
Pixel(0):= MAlloc(4000);
. . .
Release(Pixel(0));
end;
```

It is unnecessary to release memory that's allocated at the global level since the memory is automatically released when the program exits. If you release a block of memory that was not allocated by MAlloc, you get a run-time error. If you write beyond the end of an array, the memory control blocks used by DOS can be corrupted, and you can get a run-time error when you release the memory block. If this occurs, you can find the exact DOS error code by examining the registers in the array returned by GetReg. (See DOS call $21, function $49 for details.)

DIRECTLY ACCESSING MEMORY (Advanced)

A segment array can be used to directly access anything in the first megabyte of RAM. It can be used, for instance, to directly access the video memory, or the program segment prefix (PSP) set up by DOS, or the system interrupt vectors. Any segment address can be used, not just the one provided by MAlloc. Here is an example of a segment array used to clear the video text screen (for modes 0-3):

```
seg int Video(1);
int     I;
begin   \Set up for bright white characters on a blue background
Video(0):= $B800;
for I:= 0, 2000-1 do Video(0, I):= $1F20;  \attribute:space char
. . .
```



6 : I N P U T A N D O U T P U T

Everything XPL0 can do is useless without a way to communicate with the outside world. Input and output (I/O) is done through intrinsics, most of which call I/O device drivers in DOS or BIOS.

The fundamental I/O intrinsics are:

```
variable:= ChIn(device) Input a character, or byte, from device
ChOut(device, byte)     Output a byte to the device
OpenI(device)           Make the device ready for input
OpenO(device)           Make the device ready for output
Close(device)           Close the device (flush output buffer)
```

An input device, such as the keyboard, sends characters (or bytes) that are read in by ChIn. Each time ChIn is called, it returns with the next character. An output device, such as the monitor, receives characters (or bytes) that are sent by ChOut. ChOut sends a single character each time it's called. Some devices must be made ready, or "opened", before they can be used. For instance, a disk file has pointers that indicate where to start filling or emptying its buffer, and these pointers must be set to the beginning of the buffer. Bytes sent to an output file pass through an output buffer, and after the last byte has been sent, this buffer is "closed" so that any bytes remaining in it are written to the disk.

There are other intrinsics that use the fundamental capabilities provided by ChIn and ChOut to input and output integers and reals. For example:

```
variable:= IntIn(device)        Input an integer
IntOut(device,expression)       Output an integer
variable:= RlIn(device)         Input a real
RlOut(device,expression)        Output a real
```

IntIn and RlIn are similar to ChIn, but they input a number consisting of one or more digits instead of just a single character. If a series of numbers are typed on the keyboard and separated by spaces then each time IntIn(0) is called, it returns with the value of the next number. Any non-numeric character (except underline) is used to separate the numbers, such as space, comma, or a carriage return and line feed. If the numbers come from device 3, we have a numeric data file.

Integers and reals are normally represented outside a program as strings of ASCII characters. For example, IntOut(0,35) converts the integer 35 from its 16-bit binary form into an ASCII "3" character followed by an ASCII "5". Conversely, when numbers are input, strings of ASCII characters are converted into binary form.

Unlike some other languages, XPL0 has simple output commands. The advantage is that output can be formatted in a straightforward way. For example, when an integer is output, only the digits of the integer (and possibly a minus sign) are sent out. There are no "helpful" spaces or carriage returns sent that might not be wanted in some cases, and that might be confusing to eliminate. In XPL0 if you want formatting, you do it yourself.

Intrinsics used for I/O specify a device number. Device numbers are assigned to physical devices as follows:

DEVICE NUMBER	OUTPUT DEVICE	INPUT DEVICE
0	Monitor	Buffered Keyboard
1	Monitor	Unbuffered Keyboard
2	Printer	--
3	Disk File	Disk File
4	Serial Port	Serial Port
5	Printer	Printer Status
6	Monitor	Unbuffered keyboard
7	Null	Null
8	Buffer	Buffer

Most intrinsics used for I/O call DOS and BIOS routines. You can learn more about how these intrinsics work by looking up these routines in a book such as "Advanced MS-DOS Programming" by Ray Duncan. This table shows the interrupt and function calls (in hex) used by each device:

Device	OpenI	OpenO	ChIn	ChOut	Close
0	21 0C	--	21 0A	21 02	--
1	16 01	--	21 08	10 0E	--
2	--	--	--	21 05	--
3	21 42	21 42	21 3F	21 40	21 40
4	--	--	14 02	14 01	--
5	17 01	17 01	17 02	17 00	--
6	--	10 02	16 00	10 09*	--
7	--	--	--	--	--
8	*	*	*	*	--

-- Does nothing; simply returns.
* Calls routines that are not in DOS or BIOS (see below).

6.0 DEVICE 0

Output device 0 is the monitor. It displays ASCII characters and handles certain control characters such as tab, form feed (clears screen), bell, carriage return, line feed, and backspace. Text reaching the end of a line automatically wraps to the beginning of the next line. Text written beyond the bottom line scrolls the entire screen up one line. Tab stops are every eighth column. DOS interrupt $21 function $02 is called.

Input device 0 is a buffered keyboard. Characters are echoed on the monitor as they are typed in, but the buffer holds them until the "Enter" (or Carriage Return) key is struck. This enables errors to be corrected, using the "Backspace" key (and other editing keys such as left arrow and F3) before the characters are sent to the program. The buffer holds up to 128 characters including the carriage return ($0D) at the end. Typing an "Esc" deletes all the characters in the buffer (thus Esc cannot be entered as a character). Typing a control-C aborts the program. Typing a control-P turns printer echo on and off (strangely).

Output and input can be redirected using the DOS commands ">" and "<" on the command line when starting your program. The "<" command is useful because it provides a way to make a type of batch file that works inside a program (.BAT files only perform DOS commands).

OpenI(0) initializes the keyboard, which discards any characters that were previously struck and still residing in its buffers. It's a good idea to do an OpenI(0) before getting a reply to a critical question like: "Format Hard Drive?". OpenO(0) and Close(0) do nothing.

6.1 DEVICE 1

Device 1 is similar to device 0 for output, but it calls BIOS instead of DOS. This gives it the following differences: Form feeds and tabs are not handled, they are displayed as characters instead; output cannot be redirected with ">"; and attributes (such as color and flashing) already written to the screen are not changed to white characters on a black background.

For input, keystrokes are not echoed on the monitor, although a flashing cursor is displayed. There is no buffer, so keystrokes are sent to the program as soon as they are struck. Of course, calling ChIn(1) waits until a key is struck. If a non-ASCII key is struck, such as "F1", a zero is returned. ChIn(1) must be called a second time to get the key's scan code (see: A.4: Keyboard Scan Codes). Typing a control-C aborts the program. [A "^C<CR><LF>" is echoed to the display even with TrapC(true)]. Esc can be entered as a character. Input can be redirected from a file by typing "<" on the command line.

OpenI(1) discards any pending keystrokes.

6.2 DEVICE 2

Device 2 is the printer (PRN or LPT1). If the printer is busy, ChOut(2) waits until the printer is ready to accept the character (there is no timeout). Output can be redirected to another printer port or to a serial port using the DOS "MODE" command. For example, to select LPT2 type: "MODE LPT2".

Beware, if the printer is out of paper or powered off, an "Abort, Retry, Ignore?" error can occur. If the user types "A" for Abort, it aborts your program immediately, not giving it a chance to clean up such things as open output files (resulting in lost allocation units) or to restore text mode 3 from a graphics display. You can prevent this by changing DOS's critical-error-handler vector, interrupt $24, to point to your own routine; but an easier way is to use device 5 (see below).

6.3 DEVICE 3

Device 3 is a disk file. Opening, reading, writing, and closing device 3 is more complicated than the other devices. The usual operations are:

```
\Read an input file
Hand:= FOpen("C:\DIR\FILENAME.EXT", 0); \Get handle for in file
FSet(Hand, ^I);                         \Set device 3 to handle
OpenI(3);                               \Initialize input buffer
repeat until ChIn(3) = $1A;             \Read some characters
FClose(Hand);                           \Close out this handle


\Write an output file
Hand:= FOpen("C:\DIR\FILENAME.EXT", 1); \Get handle for out file
FSet(Hand, ^o);                         \Set device 3 to handle
OpenO(3);                               \Initialize output buffer
for Ch:= $20, $7E do ChOut(3, Ch);      \Write some characters
ChOut(3, $1A);                          \Write end-of-file Ctrl-Z
Close(3);                               \Flush output buffer
FClose(Hand);                           \Close out this handle
```

FOpen opens a file and returns a "handle", which is an integer used to refer to the file. FOpen has two arguments: the address of a string giving the name of the file; and the mode, which is either 0 for input or 1 for output. The file name can include the drive and subdirectory path names. If these are omitted, the current drive and subdirectory are used. If you output to device 3 without opening a file, DOS sends this information to the monitor screen, and no error is detected.

FSet assigns the handle to be used by device 3. It also selects a large or small buffer for input or output. The following modes can be selected:

```
^i = Input using small buffer
^I = Input using large buffer
^o = Output using small buffer
^O = Output using large buffer
```

The large buffers are faster than the small ones, but there are only two of them, one for input and one for output. Several files can be open simultaneously if the small buffers are used.

OpenI(3) and OpenO(3) reset the file pointers to the beginning of the file. Close(3) flushes any characters that might be remaining in the large output buffer out to the disk file.

FClose calls DOS interrupt $21 function $3E, which flushes all internal buffers associated with the file handle. If the file was created or changed then the time, date, and size are updated in the DOS directory.

END OF FILE

Character files are usually terminated by a control-Z ($1A). This is merely a programming aid since the file-handling intrinsics and DOS pay no attention to control-Z's. This enables them to handle any kind of data files (such as binary files), not just character files.

Some character files are not terminated by a control-Z, so a control-Z is automatically generated if a program attempts to read beyond the end of the file. If the program attempts this a second time, a run-time I/O error occurs.

When reading binary files, the program must know when to stop. It can get the size of the file from DOS (interrupt $21, function $4E), but an easier way is to use the error trapping intrinsics Trap (17) and GetErr (22) and read until an error is detected. If you use this method, note that an extra control-Z is returned at the end, and it is not part of the file.

OPENING FILES FROM THE COMMAND LINE (Advanced)

The command tail in the program segment prefix (PSP) can be used to specify input and output files. The PSP is 256 bytes of memory that's loaded at the beginning of an .EXE file. It contains useful information such as the command tail, which is the rest of the line typed after the program name when starting the program. For example, the following command line starts the program called LOWCASE and opens FILE1 for input and FILE2 for output:

```
LOWCASE FILE1.TXT, FILE2.TXT
```

```
\LOWCASE.XPL    01-AUG-2011
\This copies a file, shifting all characters to lowercase.

code    Reserve=3,      ChIn=7,         ChOut=8,        OpenI=13,
        OpenO=14,       Close=15,       FSet=24,        FOpen=29,
        FClose=32,      GetReg=35,      Blit=36;

int     CpuReg,          \Register array from GetReg
        HandIn, HandOut, \File handles
        I;               \Scratch
char    CmdTail;         \Copy of command tail

begin
CpuReg:= GetReg;                \Get DOS PSP and data segment
CmdTail:= Reserve($80);         \Get copy of command tail
Blit(CpuReg(11), $81, CpuReg(12), CmdTail, $7F);

HandIn:= FOpen(CmdTail, 0);     \Open first file name for input
FSet(HandIn, ^I);
OpenI(3);

loop for I:= 1, $7F do          \Scan to second file name
        if CmdTail(I) = ^, then quit;

HandOut:= FOpen(CmdTail+I+1, 1); \Open second file name for output
FSet(HandOut, ^O);
OpenO(3);

repeat  I:= ChIn(3);            \Copy and shift to lowercase
        if I>=^A & I<=^Z then I:= I !$20;
        ChOut(3, I);
until I = \EOF\ $1A;

Close(3);
FClose(HandIn);
FClose(HandOut);
end;
```

A much simpler version of this program takes advantage of DOS's ability to redirect I/O devices. FILE1.TXT must be terminated with an EOF. This second version of LOWCASE is run like this:

```
LOWCASE <FILE1.TXT >FILE2.TXT

code    ChIn=7, ChOut=8;
int     C;
repeat  C:= ChIn(1);    \Device 1 doesn't buffer nor echo chars
        if C>=^A & C<=^Z then C:= C+$20;
        ChOut(0, C);    \Device 0 can be redirected to a file
until   C=\EOF\$1A;
```

6.4 DEVICE 4

Device 4 is the serial communications port. The baud rate etc. can be set from DOS using the "MODE" command. For example: "MODE COM1:9600,N,8,1" sets COM1 to 9600 baud, no parity, 8 data bits, and 1 stop bit. The high byte of the device number is used to specify ports other than COM1:

COM1	$0004
COM2	$0104
COM3	$0204
COM4	$0304

The 25-pin RS-232 COM ports are configured as data terminal equipment (DTE). They send data out on pin 2 and receive data on pin 3. When data is sent, input pins 5 (CTS) and 6 (DSR) must be high, and output pins 4 (RTS) and 20 (DTR) are driven high. When data is received, pin 6 (DSR) must be high, and pin 20 (DTR) is set high. To make this all work, it's often convenient to jumper pin 6 to 20 and pin 4 to 5. The program waits until these signals are correct (timeouts are not used).

6.5 DEVICE 5

Device 5 sends characters to the printer like device 2, but it's much faster because it calls BIOS routines instead of DOS routines. The faster speed is noticeable, for instance, when sending graphic images to a laser printer. A consequence of calling BIOS routines is that output cannot be redirected using the DOS "MODE" command. Also, there's no "Abort, Retry, Ignore?" error (which might be desirable). Output can be sent to printers other than LPT1 by using the high byte of the device number:

LPT1	$0005
LPT2	$0105
LPT3	$0205

6.6 DEVICE 6

Output device 6 is similar to devices 0 and 1, but it provides colors and windows. The foreground and background colors used for characters can be defined using the Attrib intrinsic (69), and a window size and location can be defined using the SetWind intrinsic (70). Device 6 is faster than devices 0 and 1 for display modes 0 through 3, 7 and $13 because it writes directly to video memory. Output for the other display modes is done using BIOS interrupt $10, function $09.

Here is a table showing how the different devices handle control characters on the monitor:

DEVICE	BEL (07)	BS (08)	TAB (09)	LF (0A)	FF (0C)	CR (0D)
0	x	x	x	x	x	x
1	x	x	-	x	-	x
6	-	-	-	x	-	x

An "x" means that the control function is done, while "-" means that a character is displayed instead. Control characters not shown are all displayed as characters, including DEL ($7F). All of the extended characters ($80-FF) are displayed. ($00, $20 and $FF are displayed as space characters.)

Input from device 6 is similar to device 1 in that keystrokes are sent to the program as soon as they are struck (there is no line buffer). It differs from device 1 in that keystrokes are echoed to the display. Also, typing a control-C (or control-Break) does not abort the program; it's handled like any other keystroke. If a non-ASCII key is struck, such as "F1", a zero is returned (and echoed--looks like a space character), but the scan code is not available. Use ChIn(1) to handle these special keys.

OpenO(6) resets any window set up by SetWind to the size of the full screen, selects an attribute with white characters on a black background, enables normal scrolling and cursor movement, and moves the cursor to the upper-left corner of the screen.

6.7 DEVICE 7

Device 7 is the null device. It's used to discard unwanted output. For example, the compiler sends its output to a disk file, but if it detects an error, it diverts the output to the null device.

Input from device 7 returns a control-Z (EOF).

6.8 DEVICE 8

Device 8 is a 256-byte circular buffer. It has a variety of uses. For example, the following routine displays the number in X, replacing the decimal point with a comma, which is the format used in some European countries. Note that a control-Z (EOF) is returned when reading beyond the last character written, and it's used to detect the end of the number.

```
OpenO(8);               \Start writing at the beginning of buffer
RlOut(8, X);            \Write the number to the buffer
OpenI(8);               \Start reading at the beginning of buffer
loop    begin
        Ch:= ChIn(8);               \Read character from buffer
        if Ch = ^. then Ch:= ^,;    \Change decimal point
        if Ch = $1A then quit;      \Quit if EOF character
        ChOut(0, Ch);               \Display the character
        end;
```

OpenO(8) and OpenI(8) reset their respective output and input pointers to the start of the buffer.

When a program starts, any characters entered on the command line after the program name are copied into device 8's buffer. This provides a convenient way to pass information to a program, such as file names or numeric values.

APPENDIX

A.0: INTRINSICS

Here is a list of the intrinsics in both numeric and alphabetic order:

```
code    Abs=0,          Ran=1,          Rem=2,          Reserve=3,
        Swap=4,         Extend=5,       Restart=6,      ChIn=7,
        ChOut=8,        CrLf=9,         IntIn=10,       IntOut=11,
        Text=12,        OpenI=13,       OpenO=14,       Close=15,
        Abort=16,       Trap=17,        Free=18,        Rerun=19,
        GetHp=20,       SetHp=21,       GetErr=22,      Cursor=23,
        FSet=24,        SetRun=25,      HexIn=26,       HexOut=27,
        Chain=28,       FOpen=29,       Write=30,       Read=31,
        FClose=32,      ChkKey=33,      SoftInt=34,     GetReg=35,
        Blit=36,        Peek=37,        Poke=38,        Sound=39,
        Clear=40,       Point=41,       Line=42,        Move=43,
        ReadPix=44,     SetVid=45       Fix=50,         POut=64,
        PIn=65,         IntRet=66,      ExtJmp=67,      ExtCal=68,
        Attrib=69,      SetWind=70,     RawText=71,     Hilight=72,
        MAlloc=73,      Release=74,     TrapC=75,       TestC=76,
        Equip=77,       Shrink=78,      RanSeed=79,     Irq=80;
code real
        RlRes=46,       RlIn=47,        RlOut=48,       Float=49,
        RlAbs=51,       Format=52,
        Sqrt=53,        Ln=54,          Exp=55,         Sin=56,
        ATan2=57,       Mod=58,         Log=59,         Cos=60,
        Tan=61,         ASin=62,        ACos=63;
```

```
        Abort=16        Abs=0           ACos=63         ASin=62
        ATan2=57        Attrib=69       Blit=36         Chain=28
        ChIn=7          ChkKey=33       ChOut=8         Clear=40
        Close=15        Cos=60          CrLf=9          Cursor=23
        Equip=77        Exp=55          ExtCal=68       Extend=5
        ExtJmp=67       FClose=32       Fix=50          Float=49
        FOpen=29        Format=52       Free=18         FSet=24
        GetErr=22       GetHp=20        GetReg=35       HexIn=26
        HexOut=27       Hilight=72      IntIn=10        IntOut=11
        IntRet=66       Irq=80          Line=42         Ln=54
        Log=59          MAlloc=73       Mod=58          Move=43
        OpenI=13        OpenO=14        Peek=37         PIn=65
        Point=41        Poke=38         POut=64         Ran=1
        RanSeed=79      RawText=71      Read=31         ReadPix=44
        Release=74      Rem=2           Rerun=19        Reserve=3
        Restart=6       RlAbs=51        RlIn=47         RlOut=48
        RlRes=46        SetHp=21        SetRun=25       SetVid=45
        SetWind=70      Shrink=78       Sin=56          SoftInt=34
        Sound=39        Sqrt=53         Swap=4          Tan=61
        TestC=76        Text=12         Trap=17         TrapC=75
        Write=30
```

This list has evolved over several years. The result is that the intrinsics tend to be grouped, with the fundamental ones first. For instance, intrinsics 40 through 45 all pertain to graphics.

In the descriptions that follow, each heading shows the intrinsic's number and an example call. An assignment such as "variable:=" indicates that the intrinsic is a function that returns a value. All of the values and arguments are integers unless "real" is shown.

0: variable:= Abs(value);

This intrinsic returns the absolute value of the argument. If the value is negative, the sign is removed. For example:

```
X:= Abs(X);
```

WARNING: There's one exception:

```
Abs(-32768) = -32768, or Abs($8000) = $8000
```

A faster way to get the absolute value is to use the "abs" command word. It's available in the native versions but not in the interpreted version. This lowercase "abs" works for both integers and reals.

1: variable:= Ran(value);

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

```
X:= Ran(100);           \Range is 0 through 99
X:= Ran(0);             \Resets seed for a repeatable sequence
X:= Ran(-4);            \Randomizes then returns Ran(4)
```

The random number generator produces a repeatable sequence of random numbers from a particular seed. Each time a program starts, this seed is "randomized" using a counter that's incremented, about 18 times per second, by the system timer.

2: variable:= Rem(expression);

This intrinsic is used with integer division. It returns the value of the remainder of the division in the argument expression. If a zero argument is used, the intrinsic returns the remainder of the last division performed. For example:

```
X:= Rem(7/3);           \X gets 1
Y:= Rem(0);             \Y gets 1
Z:= Rem(-18/-5);        \Z gets -3
```

The remainder gets the sign of the dividend (numerator), which is not necessarily the same sign as the quotient. The command word "rem", which is significantly faster and available in the native compilers, can be used instead of calling the Rem intrinsic.

3: address:= Reserve(value);

This intrinsic sets aside some memory space, which is usually used for an array, and returns the starting address of this space. The argument specifies the number of bytes to be reserved. For example:

```
Data:= Reserve(1000);   \1000 bytes or 500 integers
```

Space reserved in a procedure is released when the procedure returns.

4: variable:= Swap(value);

This intrinsic returns the value obtained by swapping the bytes of the argument. For example:

```
X:= Swap($1234);        \X gets $3412
```

The command word "swap", which is significantly faster and available in the native compilers, can be used instead of calling this intrinsic.

5: variable:= Extend(value);

This intrinsic extends the sign bit of the low byte to a 16-bit integer. It's useful when fetching signed numbers from a character array. For example:

```
X:= Extend($FD);        \X gets $FFFD (= -3)
X:= Extend(3);          \X gets $0003
```

The command word "extend", which is significantly faster and available in the native compilers, can be used instead of calling this intrinsic.

6: Restart;

This intrinsic immediately terminates execution of the program, sets the Rerun flag to "true", and restarts the program from the beginning. This intrinsic is rarely used. Sometimes when procedure calls are nested many

levels down and an error condition is detected that a high-level procedure must handle, it's easier to start the program over than to pass the error indication back through many levels of procedure calls. See intrinsics Rerun (19) and SetRun (25).

7: variable:= ChIn(device);

This intrinsic reads in one byte from the specified input device. The byte is usually an ASCII character (hence: CHaracter IN), but it can be any 8-bit value. After the character is read in, ChIn is ready to read the next character. For example:

```
X:= ChIn(0);            \Get byte from keyboard buffer
```

8: ChOut(device, byte);

This intrinsic sends a byte to the specified output device. For example:

```
ChOut(0, ^=);           \Display "=" on the monitor
ChOut(3, $FF);          \Send $FF to the output file
```

9: CrLf(device);

This intrinsic sends a carriage return ($0D) and line feed ($0A) to the specified output device. It begins a new line.

10: variable:= IntIn(device);

This intrinsic gets a decimal integer from the specified input device. It converts the integer from ASCII digits into a 16-bit binary value. Integers should be in the range: -32768 through 32767. For example:

```
X:= IntIn(0);           \Get an integer from the keyboard buffer
```

After the integer is read in, IntIn is ready to read the next integer. Any leading non-numeric characters, such as spaces and commas, are skipped, and any underlines are ignored. This intrinsic does not return until an integer (or control-Z) is read. The integer must be terminated by a non-numeric character.

11: IntOut(device, value);

This intrinsic sends a decimal integer to the specified output device. It converts the integer from its signed 16-bit binary value into ASCII digits. For example:

```
IntOut(0, X);          \Display the value in X on the monitor
```

12: Text(device, address);

This intrinsic outputs an ASCII text string, beginning at the specified address, to the specified output device. For example:

```
Text(0, "This is a string");
String:= "HELLO";
Text(2, String);       \Print HELLO on the printer
```

13: OpenI(device);

This intrinsic executes the initialization routine for the specified input device. For example:

```
OpenI(0);              \Clear the keyboard buffer
```

14: OpenO(device);

This intrinsic executes the initialization routine for the specified output device. For example:

```
OpenO(3);              \Get ready to write to the disk
```

15: Close(device);

This intrinsic executes the close routine for the specified output device. For example:

```
Close(3);              \Flush output buffer to disk
```

16: Abort;

This intrinsic aborts the program. It does the same thing as the "exit" statement except that it cannot return a value. It is included here for compatibility with other versions of XPL0. New code should use "exit" instead.

17: Trap(integer);

This intrinsic determines which run-time errors stop the program and display error messages. The default is to trap all errors, but they can be individually disabled. The argument is an integer, each set bit of which enables one of these run-time errors:

bit 0: Integer division by 0	bit 7: Real underflow out of range
1: Out of memory space	8: Fix argument out of range
2: I/O error	9: Square root error
3: Invalid opcode	10: Logarithm error
4: Invalid intrinsic	11: Exponential error
5: Real division by 0.0	12: --
6: Real overflow	13: ATan2(0.0, 0.0)

For example, sometimes you don't care if you divide by zero and you certainly don't want your program to stop if you do. Trap($FFFE) will disable this error trap, and the divide will give the best answer it can (32767).

WARNING: These bit assignments are different than those used by the non-PC versions of XPL0, such as on the 6502 and 68000.

18: variable:= Free;

This intrinsic returns the number of bytes of available heap space. Since variables and arrays are dynamically allocated space, the number of bytes returned varies depending on where and when Free is called. The largest possible Reserve is usually this value minus a few hundred bytes of working space. For example:

```
        Buffer:= Reserve(Free-300);     \A big buffer
```

WARNING: If the free space is greater than 32767 ($7FFF), the number returned will appear to be negative.

19: boolean:= Rerun;

This intrinsic returns the value of the Rerun flag, either true or false. The Rerun flag is false when a program starts. It is set "true" by the intrinsic Restart (6), and it can be set "true" or "false" by the intrinsic SetRun (25). These intrinsics are rarely used.

20: address:= GetHp;

This intrinsic returns the current value of the heap pointer. GetHp does the same thing as Reserve(0). For example:

```
X:= GetHp;
```

This intrinsic is rarely used.

21: SetHp(address);

This intrinsic sets the heap pointer to the specified memory address. This intrinsic is very rarely used.

22: integer:= GetErr;

This intrinsic returns the number of the most recently detected untrapped error. If this number is 0 then no error was detected. After returning the error number, GetErr is internally reset to 0, ready for the next call. See the Trap intrinsic (17). For example:

```
if GetErr # 0 then Text(0, "TROUBLE!");
```

When a program terminates, a run-time error message appears if the internal error number is not 0.

23: Cursor(X, Y);

This intrinsic sets the position of the cursor on the monitor screen. The next character output appears at this location. X is horizontal, 0 through 79 (left to right--other screen dimensions are also supported), and Y is vertical, 0 through 24 (top to bottom). For example:

```
Cursor(3, 4);           \Forth column, fifth row
```

WARNING: After calling this intrinsic, tabs can stop in the wrong position.

24: FSet(handle, mode);

This intrinsic assigns the file handle that is to be used by device 3. "Handle" is normally gotten from FOpen (29). "Mode" is one of the following:

```
^i = Input using small buffer
^I = Input using large buffer
^o = Output using small buffer
^O = Output using large buffer
```

There is only one large buffer for input and one large buffer for output, but several small buffers can be open at the same time. The large buffers hold 1024 bytes and are much faster than the small buffers, which hold a single byte each.

25: SetRun(boolean);

This intrinsic sets the Rerun flag directly. This intrinsic is rarely used. See intrinsics Restart (6) and Rerun (19).

26: variable:= HexIn(device);

This intrinsic gets a hex integer from the specified input device. Hex values should be in the range: $0000 through $FFFF. For example:

```
X:= HexIn(0);           \Get hex value from keyboard buffer
```

This intrinsic skips any leading non-hex characters until a hex character is found, thus the dollar sign is optional. Hex numbers are unsigned, and any minus sign is ignored. Any underlines in the hex number are also ignored. Hex digits are read until a non-hex character (or control-Z) is found, thus numbers must be terminated by a non-hex character, and this intrinsic will not return until a hex number is read. If more than four hex digits are read, only the last four are used.

27: HexOut(device, value);

This intrinsic outputs a hex integer to the specified output device. For example:

```
HexOut(0, $a12);        \Displays: "0A12" on the monitor
```

28: Chain("drive:path\filename.ext");

This intrinsic executes another program as a subroutine. The called program is specified by a string containing the file and path name. No wild cards (* or ?) are allowed, and the extension (.EXE or .COM) must be given. The string must be less than 80 characters long, and must be terminated by one of four methods (see 29: FOpen). For example:

```
Chain("C:\WORK\XDEMO.EXE");
```

When an XPL0 program begins, it returns unused memory to DOS. This memory can be used by a subprogram called by Chain. If there's not enough memory or if the execution fails, this intrinsic returns with the carry flag set and error information in the CPU register array (see 35: GetReg). If the memory allocation fails, the DOS function is $4A; and if the execution fails, the DOS function is $4B.

CPU Array	No Error	Memory Error	Chain Error
7 - Carry flag	false	true	true
14 - DOS function	-	$4A	$4B

If there's an error, the DOS return code (GetReg item 15) gives detailed information:

15 - DOS Return Code	
Memory Error ($4A)	$7 = Memory control blocks destroyed
	$8 = Insufficient memory
	$9 = Incorrect segment in ES
Chain Error ($4B)	$1 = Invalid function
	$2 = File not found
	$5 = Access denied
	$8 = Insufficient memory
	$A = Environment invalid
	$B = Format invalid

Large blocks of data can be passed to and from the chained program through the environment block. Every program has an environment block that contains system information. Normally it contains text strings that specify things like the default path and batch file information. For example, if you type "PATH" at a DOS prompt, the information displayed comes from the environment block. If you don't care about this information, or if you save and restore it, you can use the environment block to transfer other information.

The segment address of the environment block is specified at location $2C in the program segment prefix (PSP). The Chain intrinsic automatically

passes the environment block address of the main program to the sub-program. By overwriting $2C with a segment address of your choosing before the Chain call, you can pass a large block of data to the sub-program. Since only the segment address is passed, the data must be aligned to a segment boundary. For example:

```
\MAIN PROGRAM PASSING DATA TO SUBPROGRAM
CpuReg:= GetReg;                \Get info array
PspSeg:= CpuReg(11);            \Get PSP segment

Data:= Reserve(1024+16);        \Reserve data block
Data:= ((Data/16)+1)*16;        \Align block with segment

ThisSeg:= CpuReg(12);           \Get current segment
DSeg:= ThisSeg+(Data/16);       \Calculate data segment
Poke(PspSeg, $2C, DSeg);        \Set environment block
Poke(PspSeg, $2D, Swap(DSeg));

Chain("FROG.EXE");
```

The subprogram can read the segment address of the data block and, using the Blit intrinsic (36), copy the data into an array of its own. Up to 32K of data can be transferred this way.

29: handle:= FOpen("drive:path\filename.ext", mode);

This intrinsic opens a file and returns its handle. The file is specified by a string containing the file name and an optional drive and path name. No wild cards (* or ?) are allowed. Any leading or trailing spaces are ignored. The string must be less than 80 characters long and must be terminated by one of four methods:

- Bit 7 set on the last character
- A zero byte after the last character
- A carriage return after the last character
- A comma after the last character

"Mode" is 0 for read and 1 for write.

FOpen is typically used with other intrinsics as follows:

```
Hand:= FOpen("C:\WORK\FILENAME.EXT", 1);
FSet(Hand, ^O);
OpenO(3);
```

When a file is opened for writing, if it already exists, its contents are discarded; if it does not exist, a new one is created. If you send characters to device 3 without opening a file with FOpen, DOS sends them to the monitor screen and no error is detected. DOS interrupt $21 function $3C is called for writing output files, and function $3D is called for reading input files. (See: 6.3 Device 3).

30: Write(drive, sector, buffer, size);

This intrinsic writes data from memory to disk. "Drive" is 0=A:, 1=B: 2=C:, and so forth. "Sector" is the starting logical sector number. Logical sectors start at 0 (the boot sector). "Buffer" is the address of the data to write. "Size" is the number of sectors to write. There are 512 bytes per logical sector (even on a hard drive in this situation). DOS interrupt $26 is called.

WARNING: Writing to a hard drive (C:) is a very dangerous operation (in fact Windows XP will not let you do it).

31: Read(drive, sector, buffer, size);

This intrinsic is the counterpart to the Write intrinsic described above.

32: FClose(handle);

This intrinsic closes a file handle. All internal buffers associated with the file are flushed, and the handle is released for possible reuse. If the file was modified, the time, date, and size are updated in the directory. DOS interrupt $21 function $3E is called.

When a handle is closed, it ceases to exist. If additional operations need to be made to the file a new handle must be obtained using FOpen (29).

WARNING: If you close handle 0, which DOS uses for the console, your program cannot input from device 0, which is the keyboard.

33: boolean:= ChkKey;

This intrinsic returns a "true" if a key was struck on the keyboard. Interrupt $16 function $01 is called.

34: SoftInt(interrupt);

This intrinsic is used to call a DOS or BIOS "interrupt" routine. Values are passed to and from interrupt routines using the hardware registers of the processor. These values are accessed using the GetReg intrinsic (35). The following example uses SoftInt and GetReg to get the time of day:

```
code    Swap=4, SoftInt=34, GetReg=35;
int     Hour, Minute, Second;
int     CpuReg;
begin                           \Get system time
CpuReg:= GetReg;                \Get copy of CPU registers
CpuReg(0):= $2C00;              \Set register AH to $2C
SoftInt($21);                   \Call DOS function
Hour:= Swap(CpuReg(2)) & $FF;   \Read returned values
Minute:= CpuReg(2) & $FF;
Second:= Swap(CpuReg(3)) & $FF;
end;
```

DOS and BIOS provide many useful routines. These are documented in "Advanced MS-DOS Programming" by Ray Duncan.

35: address:= GetReg;

This intrinsic provides access to the hardware registers in the processor. It returns the address of an integer array that contains a copy of these registers. Before SoftInt calls an interrupt routine, it copies the contents of this array into the hardware registers. After the interrupt routine returns, SoftInt copies the registers back into the array before returning to your program.

The array also contains additional information. The state of the carry flag is returned to aid in error checking. DOS error codes are also returned, which enables more precise error messages than those given by the run-time error traps (22: GetErr).

Some useful values are in the array when an XPL0 program is started by DOS. The array is arranged as follows:

INDEX	CONTENTS	VALUE SET BY DOS
0	AX register	
1	BX register	
2	CX register	
3	DX register	
4	DI register	
5	SI register	
6	BP register	
7	Carry flag ("true" or "false")	
8	CS register	Points to program's code segment
9	DS register	Points to program's PSP
10	SS register	
11	ES register	Points to program's PSP
12	Data segment	Points to program's data segment
13	DOS interrupt	($21, $25, $26)
14	DOS function	
15	DOS return code	
16	Flags register	

Flags Register Bits:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		O	D	I	T	S	Z	0	A	0	P	1	C

NesTed
I/O Privilege Level
Overflow
Direction
Interrupt enable
Trap
Sign
Zero
Auxiliary carry
Parity
Carry

This array is also filled by the Chain intrinsic (28). This enables values to be passed between programs just like values are passed with interrupt routines.

The run-time code copies the array into the processor's registers when starting intrinsics ExtJmp (67) and ExtCal (68), and it copies the registers back into the array when finishing the IntRet (66) intrinsic.

36: Blit(source seg, source off, destination seg, destination off, size);

This intrinsic quickly copies a block of memory from one location to another. The source and destination locations are given by segment and offset addresses. The blocks of memory can overlap. "Size" is the number of bytes to copy, which can be as much as 65535.

37: variable:= Peek(segment, offset);

This intrinsic fetches a byte using segment and offset addressing. It can fetch from any location in the first megabyte of memory.

38: Poke(segment, offset, value);

This is the counterpart to the Peek intrinsic. It can be used to write a byte to any location in the first megabyte of memory. For example, this writes directly to video memory and displays an "A" (assuming the video mode is 2 or 3):

```
Poke($B800, 160, ^A);   \Display "A" at line 1 column 0
```

39: Sound(volume, duration, period);

This intrinsic sounds the speaker. "Volume" is zero for no sound and non-zero for full sound. "Duration" is seconds times 18.2. "Period" is 1190000 divided by the desired frequency. This intrinsic can also be used as a time delay by setting "volume" to zero. For example:

```
Sound(1, 18, 4542);     \One second of Middle C (262 Hz)
```

40: Clear;

This intrinsic clears the screen for either graphics or text modes. The pen position for a graphics line is set to the upper-left corner (0,0). For text modes the cursor is set to the upper-left corner.

41: Point(X, Y, color);

When in a graphics mode, this intrinsic plots a point (pixel) located at the X and Y coordinates. The upper-left corner of the display is coordinate 0,0. X increases to the right, and Y increases downward. The ranges of X, Y, and "color" vary with the video mode (see 45: SetVid). If the mode has 16 colors, they are the foreground colors shown for the Attrib intrinsic (69). If bit seven of "color" is set, the low four bits of "color" are exclusive-ored with the pixel on the screen. If the mode has 256 colors, there is no exclusive-or feature. The colors can be adjusted using several subfunctions of BIOS interrupt $10 function $10.

42: Line(X, Y, color);

This intrinsic draws a straight line from the last point plotted (or moved to with the Move intrinsic) to the specified X and Y coordinates. "Color" is the same as for Point (41), but bits 8 through 15 are used to specify various patterns of dotted and dashed lines. Pixels are not drawn at locations corresponding to set bits. For example, to draw a line with widely space dots, "color" could be $7F01. Horizontal lines are drawn much faster than other lines. This can be exploited when filling areas. For example:

```
Move(0, 0);                     \Set the start of the line
Line(160, 100, 1);              \Draw a solid blue line
Line(319, 199, $AA04);          \Continue with a dotted red line
```

43: Move(X, Y);

This intrinsic is used to set the beginning of a line.

44: color:= ReadPix(X, Y);

This intrinsic returns the color of the pixel (point) at the specified coordinates.

45: SetVid(mode);

This intrinsic sets the video display mode. It also clears the screen (unless bit 7 of "mode" is set) and reinitializes the colors and fonts to their defaults. The text cursor and line pen position are set to the upper-left corner.

Here is an example of a graphics program that plots a sine wave:

```
code ChIn=7, Point=41, Line=42, Move=43, SetVid=45, Fix=50;
code real Float=49, Sin=56;
int X;
begin
SetVid($12);                            \640x480x16 colors (VGA)
Move(320, 0);   Line(320, 479, 1);      \Draw axes in blue
Move(0, 240);   Line(639, 240, 1);
for X:= 0, 639 do                       \Plot in light red
        Point(X, 240 - Fix(180.0 *Sin(Float(X-320) /60.0)), $C);
X:= ChIn(1);                            \Wait for keystroke
SetVid(3);                              \Restore text mode
end;
```

MODE	RESOLUTION	COLORS	TYPE	ADDRESS	MDA	CGA	EGA	MCGA	VGA
$00	40x25 color burst off	16	text	$B8000		x	x	x	x
$01	40x25	16	text	$B8000		x	x	x	x
$02	80x25 color burst off	16	text	$B8000		x	x	x	x
$03	80x25	16	text	$B8000		x	x	x	x
$04	320x200	4	graph	$B8000		x	x	x	x
$05	320x200 color burst off	4	graph	$B8000		x	x	x	x
$06	640x200	2	graph	$B8000		x	x	x	x
$07	80x25	2	text	$B0000	x		x		x
$08	160x200	16	graph	PC JR					
$09	320x200	16	graph	PC JR					
$0A	640x200	4	graph	PC JR					
$0B				EGA BIOS					
$0C				EGA BIOS					
$0D	320x200	16	graph	$A0000			x		x
$0E	640x200	16	graph	$A0000			x		x
$0F	640x350	2	graph	$A0000			64K		x
$10	640x350	16	graph	$A0000			128K		x
$11	640x480	2	graph	$A0000				x	x
$12	640x480	16	graph	$A0000					x
$13	320x200	256	graph	$A0000				x	x
$6A	800x600	16	graph	$A0000	VESA				

Characters can be displayed in graphics as well as text modes.

46: real:= RlRes(integer);

This intrinsic reserves space for real arrays. RlRes(3) reserves enough memory to hold three real numbers. For example:

```
Array:= RlRes(3);      \Reserve elements 0 through 2
```

47: real:= RlIn(device);

This intrinsic gets a real number from the specified input device. It converts the number from ASCII digits into binary form. After a number is read in, RlIn is ready to read the next number. Any leading non-numeric characters, such as spaces and commas, are skipped, and any underlines

in the number are ignored. This intrinsic does not return until a number (or control-Z) is read, thus the number must be terminated by a non-numeric character. For example:

```
Array(2):= RlIn(0);     \Get real number from buffered keyboard
```

48: RlOut(device, real);

This intrinsic outputs a real number to the specified device. It converts the real number from its binary form into ASCII digits. For example:

```
RlOut(2, 3600.0*24.0*365.25);   \Print seconds in a year
```

The number of decimal places shown (etc.) can be specified by the Format intrinsic (52).

49: real:= Float(integer);

This intrinsic converts an integer into its equivalent real value. (See: 2.1 Mixed Mode.) For example:

```
RlOut(0, (Float(35));   \Display "35.00000"
```

50: integer:= Fix(real);

This intrinsic rounds a real to its nearest integer value. (See: 2.1 Mixed Mode.) For example:

```
IntOut(0, Fix(13.002)); \Display "13"
```

Converting a value outside the range -65535.0 through 65535.0 causes a fix overflow run-time error. (It's possibly useful to be able to get an unsigned result.)

51: real:= RlAbs(real);

This intrinsic takes the absolute value of a real number. For example:

```
X:= RlAbs(X);           \Remove the minus sign from X
```

A faster way to get the absolute value is to use the "abs" command word. It's available in the native versions but not in the interpreted version. This lowercase "abs" works for both integers and reals.

52: Format(integer, integer);

This intrinsic specifies the format of reals that RlOut (48) outputs. The first integer is the number of characters before the decimal point, including any minus sign; and the second integer specifies the number of characters after the decimal point. If the first integer is 0 then scientific notation is used instead. If the first integer is -1 (or any value less than zero) then engineering notation is used. For example:

```
Format(5,2)  gives 12345.67
                       3.14
                  -12345.67
Format(0,1)  gives  1.2E+004
Format(-1,4) gives   12.3457E+003
```

One purpose of Format is to align decimal points. However, if the number is too large to fit in the designated space, all of the digits are still sent out, and the decimal point is not aligned. If the format is not specified then RlOut uses the default: Format(5,5). If the number of characters specified after the decimal point is 0 then a decimal point is not sent out.

53: real:= Sqrt(real);

This intrinsic returns the square root of the argument. If the argument is negative, a run-time error occurs. For example:

```
Root2:= Sqrt(2.0);        \1.414213562
```

54: real:= Ln(real);

This intrinsic returns the natural logarithm (base e) of the argument. The argument must be > 0.0, otherwise a run-time error occurs.

55: real:= Exp(real);

This intrinsic computes the exponential function (e^X). This is the inverse operation of Ln.

56: real:= Sin(real);

This intrinsic computes the sine function. All of the trig functions use angles measured in radians. To convert from radians to degrees, multiply by 57.2957795 degrees/radian (= 180 deg/pi). To convert degrees to radians, divide by 57.2957795. For example:

```
X:= Sin(30.0/57.2957795); \Sine of 30 degrees (=0.5)
```

57: real:= ATan2(real Y, real X);

This intrinsic computes the arc-tangent in radians of Y divided by X. If the computed angle is in the range ±pi/2 (±90 degrees) then X can be 1.0. However, if an angle over the entire range of a circle (±pi or ±180°) is to be computed then the signed values of the Y and X coordinates are used. This converts rectangular coordinates to polar coordinates. For example:

```
Angle:= ATan2(0.5, 1.0);        \Angle:= ATan(0.5) (= 26.56505°)
Angle:= ATan2(13.0, -13.0);     \Angle:= 3/4 pi (= 135°)
Angle:= ATan2(-5.0, -5.0);      \Angle:= -3/4 pi (= -135°)
```

58: real:= Mod(real, real);

This intrinsic computes the modulo function. This is the real counterpart to the Rem intrinsic (2). Mod(A, B) is defined as A modulo B, which is defined as A - Int(A/B) * B. Where Int(A/B) extracts the largest integer <= Abs(A/B) and attaches the sign of A/B (i.e. it truncates toward zero). For example:

```
X:= Mod(10.2, 3.0);     \X:= 1.2
X:= Mod(-10.2, 3.0);    \X:= -1.2
X:= Mod(123.456, 1.0);  \Get the fractional part (0.456)
```

59: real:= Log(real);

This intrinsic computes the common logarithm function (base 10). The argument must be > 0.0, otherwise a run-time error occurs.

60: real:= Cos(real);

This intrinsic computes the cosine function. The argument is the angle in radians.

61: real:= Tan(real);

This intrinsic computes the tangent function. The argument is the angle in radians.

62: real:= ASin(real);

This intrinsic computes the arc-sine function in radians. It loses accuracy when arguments are very close to ±1.0 because of an internal subtraction from 1.0.

63: real:= ACos(real);

This intrinsic computes the arc-cosine function in radians. It loses accuracy when arguments are very close to ±1.0 because of an internal subtraction from 1.0.

64: POut(value, port, size);

This intrinsic writes to an output port. "Port" specifies the port address. "Value" is the value written. If "size" is 0 then eight bits are written; if "size" is not zero then 16 bits are written. A faster and more intuitive method to write a byte to a port is to use the command word "port" (see below).

65: variable:= PIn(port, size);

This intrinsic reads an input port. "Port" specifies the port address. If "size" is 0 then eight bits are read; if "size" is not zero then 16 bits are read. A faster and more intuitive method to read a byte from a port is to use the command word "port". For example:

```
POut(PIn($61,0)&$FC, $61, 0);   \Disable speaker
port($61):= port($61) & $FC;    \Disable speaker
```

66: IntRet;

This intrinsic executes an interrupt return (IRET) opcode. This enables an XPL0 program to be used as an interrupt service routine. Just before returning, the processor's registers are restored to the values they had when the XPL0 program was started (unless SoftInt was executed).

67: ExtJmp(segment, offset);

This intrinsic executes a far jump to an external routine. This is useful when XPL0 code is inserted in front of an existing interrupt service routine and control must be passed on to this routine. Just before jumping, the processor's registers are restored to the values they had when the XPL0 program was started (unless SoftInt was executed).

68: ExtCal(segment, offset);

This intrinsic executes a far call to an external routine. The GetReg register array is loaded into the processor's registers just before this call is executed, and the processor's registers are saved into the GetReg array immediately after the call returns. This provides a way to pass arguments to and from the external routine.

69: Attrib(colors);

This intrinsic specifies the colors (attribute) used when sending characters to device 6. The high nibble of the argument sets the background color, and the low nibble sets the foreground color. For example, Attrib($17) displays white characters on a blue background.

BACKGROUND	FOREGROUND
$00: Black	$00: Black
$10: Blue	$01: Blue
$20: Green	$02: Green
$30: Cyan	$03: Cyan
$40: Red	$04: Red
$50: Magenta	$05: Magenta
$60: Brown	$06: Brown
$70: White	$07: White
$80: Flashing Black	$08: Gray
$90: Flashing Blue	$09: Light Blue
$A0: Flashing Green	$0A: Light Green
$B0: Flashing Cyan	$0B: Light Cyan
$C0: Flashing Red	$0C: Light Red
$D0: Flashing Magenta	$0D: Light Magenta
$E0: Flashing Brown	$0E: Yellow
$F0: Flashing White	$0F: Bright White

On a monochrome display (mode 7) the attribute "colors" are as follows:

	VGA	HERCULES
$00:	No display (black)	Same
$01:	Underlined	Same
$07:	Normal	Same
$09:	Underlined intense	Same
$0F:	Intense	Same
$70:	Reverse (black on white)	Same
$77:	No display (white)	Normal
$7F:	Intense on white	Intense on black
$81:	Flashing underlined	Same
$87:	Flashing	Same
$89:	Flashing underlined intense	Same
$8F:	Flashing intense	Same
$F0:	Flashing black on white	Same
$FF:	Flashing intense on white	Flashing intense on black

The underline is continuous on a Hercules display, is dashed on a VGA display, and is nonexistent on a Compaq portable.

Attributes are handled a little differently when the display is in a graphics mode rather than a text mode. The graphics hardware cannot flash characters so bit 7 from the Attrib intrinsic is used to specify intense (bright) colors for the background instead. This gives 16 background colors identical to the 16 foreground colors. Characters with a black background (Attrib($0x)) are written almost twice as fast as characters with colored backgrounds. If the foreground color is the same as the background color and the color is not black (0) then the character is written by XORing it with the screen. This makes the background transparent. In other words, characters can be drawn on top of a pattern without a surrounding background box. If the character is XORed a second time, it's erased, and the original background pattern is restored.

Graphics mode $13 is different because it has 256 colors. The high byte of the argument (Attrib($xx00)) sets the background color, and the low byte sets the foreground color. There is no XOR feature.

OpenO(6) sets the attribute to white on black ($07) (and also sets the cursor to the upper-left corner of the display).

70: SetWind(X0, Y0, X1, Y1, mode, fill);

This intrinsic specifies the window used when sending characters out to device 6. X0, Y0 sets the upper-left corner of the window; and X1, Y1 sets the lower-right corner.

"Mode" specifies how the window behaves. The high byte controls the visible cursor:

= 0: Visible cursor moves normally.

0: Visible cursor does not move (but the invisible character insertion point moves normally).

The low byte of "mode" controls the text:

0 = Scroll: Text scrolls up when the cursor reaches the bottom of the window, and an automatic CR and LF is done at the right edge. (Writing to the bottom-rightmost character cell scrolls.)

1 = Wrap: Text does an automatic CR and LF at the right edge, but it wraps to the top of the window when it exceeds the bottom edge.

2 = Clip: Text is clipped beyond the right edge and beyond the bottom of the window.

The visible cursor moves when sending a character out to device 0, even if visible cursor movement is turned off for device 6. Visible cursor movement can only be turned off for video modes 0, 1, 2, 3, and 7. When the visible cursor is off, text can be output about twice as fast.

If the "fill" flag is "true", the window is erased and filled with the background color specified by Attrib (69). When in graphics video modes 4, 5 and 6, the background is not filled with a color as you would expect but is instead filled with vertical stripes. If the "fill" flag is "false", the window is set up without changing any characters on the screen.

Opening device 6 for output resets the window to the full screen size and enables normal scrolling and cursor movement.

WARNING: The Cursor intrinsic (23) is not affected by the position of a window; it always uses the upper-left corner of the entire screen as position 0,0.

71: RawText(device, address);

This intrinsic is the same as the Text intrinsic except that strings are terminated by a space character with its most significant bit set ($A0). This enables the extended ASCII codes to be used. The terminating space is not sent out. For example:

```
RawText(0, "╔══════╗ ");
```

The normal Text intrinsic (12) can display strings containing extended ASCII characters if the command word "string" is used to specify zero-terminated strings. For example:

```
string 0;
Text(0, "╔══════╗");
RawText(0, "╔══════╗ ");       \will not work with string 0;
```

72: Hilight(X0, Y0, X1, Y1, attribute);

This intrinsic changes the colors in a specified area on the text screen without changing the characters. The area is defined by the corners of a rectangle. X0, Y0 is the upper-left corner, and X1, Y1 is the lower-right corner. "Attribute" defines the background and foreground colors (see 69: Attrib).

Hilight is typically used to highlight selected menu items, but it can also be used to make such things as drop shadows for windows. If the foreground color is the same as the background color, characters are invisible (there is no XOR feature).

73: segment address:= MAlloc(paragraphs);

This intrinsic returns the starting segment address of a block of memory. (A segment address is a physical address divided by 16.) Memory is allocated in 16-byte quantities called "paragraphs". For example:

```
Seg:= MAlloc(4000);     \Allocate 64000 bytes
```

Unlike the Reserve intrinsic (3), MAlloc does not automatically release memory when a procedure returns. If MAlloc is used in a procedure and the procedure is repeatedly executed, more memory can be allocated each time (resulting in the infamous "memory leak" problem). If insufficient memory is available then run-time error 2: OUT OF MEMORY is trapped. Allocated memory is automatically released when a program terminates. DOS interrupt $21 function $48 is called.

74: Release(segment address);

This intrinsic deallocates a block of memory that was allocated by MAlloc. The segment address of the block is passed to indicate which block to deallocate. For example, this would deallocate the 64000 bytes allocated above:

```
Release(Seg);
```

75: TrapC(boolean);

This intrinsic turns control-C trapping on and off. "True" is passed to turn on control-C trapping, which prevents the control-C and control-Break keys from aborting a program. Control-C trapping is normally off. Any change to the way control-C and control-Break are handled is restored when a program terminates.

76: boolean:= TestC;

When control-C trapping is on, TestC is used to determine if the control-C or control-Break keys were struck. If either one was then TestC returns "true".

Each time the control-C or control-Break key is struck, a status flag is set. When TestC is called, it returns the state of this status flag and then resets it to "false".

TestC can give unexpected results. A control-C is not checked for until some I/O is done through DOS or BIOS. Also, the keyboard hardware is buffered, and a control-C is not detected until it's actually read in. Control-Break, on the other hand, is detected immediately.

77: address:= Equip;

This intrinsic returns the address of an array that describes the equipment that the program is running on. The array contains the following information:

0: Processor type
1: Math coprocessor type
2: Video configuration
3: Processor speed
4: Run-time code version
5: Run-time code type

0: PROCESSOR TYPE. The type of processor is indicated by one of the following integers: 86, 286, 386, 486, or 586.

1: MATH COPROCESSOR TYPE. The type of math coprocessor is indicated by one of the following integers: 0, 87, or 387. The 80287 is indistinguishable from the 8087, so 87 refers to both the 8087 and 80287. Zero means there is no math coprocessor. A DX386 or Pentium (586) effectively has a 387 math coprocessor built into it.

2: VIDEO CONFIGURATION. This gives the type of video adapter and monitor. The possible configurations are:

VIDEO ADAPTER (low byte)

$01: MDA Monochrome Display Adapter
$02: CGA Color Graphics Adapter
$04: EGA Enhanced Graphics Adapter
$08: VGA Video Graphics Array
$10: MCGA Multi-Color Graphics Array
$20: HGC Hercules Graphics Card

MONITOR (high byte)

$01: Monochrome
$02: Color (or enhanced monochrome emulating color)
$04: Enhanced color
$08: Analog monochrome only
$10: Analog color only
$18: Analog monochrome and color

3: PROCESSOR SPEED. This gives an indication of the processor's speed. The original 4.77 MHz IBM PC-XT returns a value of 5. Interrupts are left enabled so the value reflects the actual running environment. Interrupt-intensive environments, such as Microsoft Windows, give lower speeds. The speed test takes 55 to 110 milliseconds to run no matter how fast the processor.

4: RUN-TIME CODE VERSION. This gives the run-time code (NATIVE or I2L) version number. The value is the version number times 10. For example, if the current version is 3.0, the value returned is 30.

5: RUN-TIME CODE TYPE. This gives the type of run-time code used.

```
^N: NATIVE
^7: NATIVE7
^S: NAT
^I: I2L
```

For example, this displays the processor type:

```
int     EqList;
begin
EqList:= Equip;
IntOut(0, EqList(0));
. . .
```

78: Shrink(value);

This intrinsic returns unused heap space to DOS. The argument is the number of bytes to keep (above the current heap pointer). A minimum of a couple hundred bytes are usually kept for local variables and small, temporary arrays.

When an XPL0 program starts, it's given about 60K of memory space for its heap, which is where variables and arrays are stored. Usually this is more than what is needed. In situations where memory is scarce, any excess can be given back to DOS and used by other programs that are also loaded in memory at the same time. This can occur, for instance, when using the Chain intrinsic (28).

Shrink is normally called from the main procedure after reserving global arrays. This sets a new limit for the heap space. You'll get a run-time error (OUT OF MEMORY) if you attempt to reserve past this limit.

79: RanSeed(integer);

This intrinsic sets the seed for the random number generator (1: Ran). This allows up to 65536 different, repeatable random number sequences.

80: Irq(boolean);

This intrinsic turns interrupts on if boolean is "true" and off if boolean is "false". This is useful, for instance, to prevent a mouse interrupt from interfering while manipulating the VGA registers. The NMI interrupt is not affected. Interrupts, of course, should not be left turned off for very long. Be aware that most BIOS and DOS calls will temporarily re-enable interrupts.

A.1: COMPILE ERRORS

XPL0 has two different types of error messages. The first type, called "compile errors", occur when a program is being compiled; and the second type, called "run-time errors", occur when the program runs.

If the compiler detects an error, it stops and asks if it should attempt to continue. A "Y" (or just hitting the Enter key) continues; an "N" aborts the compile. The ASM output file is discarded if any error is detected.

For example, if we try to compile:

```
Frog:= 2 + 3.5 + Frog;
```

the compiler stops and displays:

```
Frog:= 2 + 3.5 + F
***** ERROR NO. 46 *****
MIXED MODE
ATTEMPT TO CONTINUE (Y/N)?
```

Compile error messages can sometimes be misleading because the actual error might have occurred prior to the point that the compiler flags as the error. The reason these two points don't always coincide is because the compiler finds the code at the actual error to be syntactically correct, but it interprets it in a way other than what was intended. This alternate interpretation can go for several lines before an error is finally flagged. An extreme example of this is failing to terminate a string with a close quote. In this case the compiler simply interprets the following code as being part of the string, and an error is not detected until either a quote mark or the end-of-file is encountered. Particularly misleading error messages can result from unpaired begin-ends.

All of the compile error messages are listed below along with some helpful comments and suggestions.

1: TOO MANY VARIABLES. Procedures with too many variables should be broken into smaller procedures. Perhaps the variables are more global than necessary. Perhaps several variables could be combined into an array.

2: TOO MANY REAL CONSTANT NAMES. There are too many constants "define"d as real values in scope at one time. The maximum number is 160. Perhaps they are more global than necessary.

3: TOO MANY NAMES. There are too many names (variables, procedures, intrinsics, constants, etc.) in scope at one time causing the symbol table to overflow. The maximum number is 1600. Perhaps some intrinsics are declared but not used. Perhaps some names are more global than necessary. Perhaps several variables could be combined into an array.

4: TOO MANY 'QUITS'. There cannot be more than 160 total "quit" statements inside a "loop". This total includes "quit"s for other "loop"s that are nested inside the outer "loop".

5: TOO MANY STATIC LEVELS. Procedures can be nested to a maximum depth of eight levels.

6: NUMBER OUT OF RANGE. Integers are limited to the range of -32768 through +32767.

7: NUMBER OUT OF RANGE. Intrinsic "code" declarations are limited to 0 through 127.

10: UNDECLARED NAME. The name is undefined here. It might be out of scope or be forward referenced. A procedure declaration that is missing a semicolon causes the rest of the line to not be seen (it's taken as a comment).

11: NAME ALREADY DECLARED. This name conflicts with a previous declaration at this level. Only the first 16 characters are significant to the compiler.

20: ILLEGAL START OF A STATEMENT. Missing an "end"? Unpaired "begin-end"s? If "procedure" is flagged then there's a missing "end" in the previous procedure.

21: ":=" EXPECTED BUT NOT FOUND. Illegal variable in a "for" or an assignment statement? The control variable in a "for" loop cannot have a subscript.

22: 'THEN' EXPECTED BUT NOT FOUND. Illegal expression in an "if" statement?

23: 'DO' EXPECTED BUT NOT FOUND. Illegal expression in a "for" or "while" statement?

24: 'TO' OR 'DOWNTO' EXPECTED BUT NOT FOUND. Illegal expression in a "for" statement? A comma does the same thing as 'to'.

26: ILLEGAL FACTOR. Incomplete expression or an illegal operator? Semicolon or "of" before an "other" in a "case" statement? Perhaps parentheses are needed around an "if" expression. Perhaps a word should start with a capital letter.

27: STATEMENT STARTING WITH A CONSTANT. The name is declared as a constant, which cannot be assigned a value.

28: 'UNTIL' EXPECTED BUT NOT FOUND. Perhaps the previous statement is missing a semicolon. Unpaired "begin" "end"s within a "repeat" block?

29: 'OTHER' EXPECTED BUT NOT FOUND. A "case" statement must be terminated with an "other" statement. Perhaps the previous statement is missing a semicolon.

30: 'ELSE' EXPECTED BUT NOT FOUND. An "if" expression must have the "else" clause. Illegal expression after the "then"? Do not confuse an "if" expression for the more common "if" statement.

31: DIGIT EXPECTED BUT NOT FOUND. Either the exponent of a real number or a hex digit is missing.

33: INTEGER VARIABLE EXPECTED BUT NOT FOUND. The control variable in a "for" statement must be an integer or character variable.

38: ">" EXPECTED BUT NOT FOUND. Arithmetic shift right "->>" incomplete?

39: "(" EXPECTED BUT NOT FOUND. Parentheses must enclose arguments.

40: "=" EXPECTED BUT NOT FOUND. In a "code" declaration every name must be set equal to an integer.

41: ";" EXPECTED BUT NOT FOUND. A semicolon must be at the end of a declaration, must separate procedures, and must separate statements within a "begin-end" (or a "repeat-until") block. The first letter of a variable name must be uppercase.

42: CONSTANT EXPECTED BUT NOT FOUND. In a "define" or a constant array the values must be previously declared constants or be integer or real constants; they cannot be variables.

43: VARIABLE EXPECTED BUT NOT FOUND. The "address" and "@" operators can only return the address of a variable.

44: ")" EXPECTED BUT NOT FOUND. Parentheses must be balanced. Even though balanced, extra sets of parentheses around arguments and subscripts are illegal.

45: NAME EXPECTED BUT NOT FOUND. There must be a name in a declaration. At least the first letter of a variable name must be uppercase.

46: MIXED MODE. Reals and integers cannot be mixed within an expression without explicitly doing the type conversions using the intrinsics Fix and Float. This message can be triggered if a variable is undefined. Also, a forward-function declaration and its function must be the same data type.

47: INTEGER EXPECTED BUT NOT FOUND. The indicated value or expression is not of type integer. Subscripts, the control variable in a "for" loop, and "case" expressions cannot be reals.

48: 'OF' EXPECTED BUT NOT FOUND. Illegal expression in a "case" statement?

49: ":" EXPECTED BUT NOT FOUND. Illegal expression in a "case" statement?

50: "]" EXPECTED BUT NOT FOUND. Constant-array brackets must be balanced. Perhaps a comma is missing.

51: NO ARGUMENTS DECLARED. The called procedure has no local variables declared and therefore cannot have arguments passed to it.

52: STATEMENT STARTING WITH 'ELSE'. An "else" is never preceded by a semicolon.

53: STATEMENT STARTING WITH 'OTHER'. An "other" is never preceded by a semicolon.

60: 'QUIT' NOT IN A 'LOOP'. The "quit" statement is legal only inside a "loop" block.

61: EOF EXPECTED BUT NOT FOUND. More code after the apparent end of the program. Unpaired "begin" "end"s? Too many "end"s or missing a "begin"?

62: EOF INSIDE A BLOCK. End-of file (Control-Z, $1A) is inside a block statement. Too many "begin"s or not enough "end"s? Incomplete or missing statement?

63: EOF INSIDE A STRING. Unpaired double quote (")? A caret (^) can cause a quote to not be seen.

65: 'FPROC' & ITS 'PROC' NOT AT SAME LEVEL. A forward procedure declaration and its corresponding procedure declaration must be at the same static level and must be in scope with each other.

66: 'FPROC' REFERENCE NOT FOUND. Unresolved forward procedure or function reference. Perhaps it's out of scope. "fproc" and its corresponding "proc" must be at the same static level. Maybe a "begin" is missing.

67: 'PROC' OR 'FUNC' EXPECTED BUT NOT FOUND. "public" must be followed by "procedure" or "function".

68: 'EPROC'S AND 'PUBLIC'S MUST BE GLOBAL. "eproc"s, "efunc"s, and "public"s must be at level zero; they cannot be inside a procedure (except the main routine).

69: 'INCLUDE'S NESTED TOO DEEP. A file that is included can itself include other files. These files also can include files, but the chain of includes is limited to eight levels. Perhaps a file is including itself, or is including a file that includes the original file.

70: BAD FILE SPEC. The specification should be: C:\path\filename.ext; Everything but the file name is optional. The semicolon is required. Backslashes do not designate comments in a file name.

71: FILE NOT FOUND. Perhaps the file is not in the current directory.

72: 'INT', 'REAL', 'CHAR', or 'ADDR' EXPECTED BUT NOT FOUND.

73: DIVIDE BY ZERO. A constant expression is attempting to divide by 0.

74: MATH ERROR IN A CONSTANT EXPRESSION. A floating point overflow or underflow occurred.

75: EXPRESSION MUST BE ENCLOSED IN PARENTHESES. Exclusive-or operations (|) and "if" expressions must be enclosed in parentheses when the short-circuit boolean command-line switch (/b) is used.

L2002: FIXUP OVERFLOW. This error is generated by the linker (LINK). There is more than 64K of code in external routines that are declared after a dimensioned array declaration. Rearrange your declarations, putting "external" "eproc" and "efunc" ahead of any dimensioned arrays. This moves the EXTRN declarations in the assembly code ahead of the code segment.

A.2: RUN-TIME ERRORS

If an error is detected while a program is running, it aborts and a run-time error message is displayed.

Aborting points out errors in the code, but sometimes it's more of a nuisance than a help. The Trap intrinsic (17) can be used to disable the abort for selected run-time errors.

This is a list of all of the run-time error messages:

1: DIV BY 0. Illegal division by zero for an integer. If this is untrapped, 32767 is returned.

2: OUT OF MEMORY. No more memory space. An array declaration, a Reserve, or the I2L loader tried to exceed the allotted memory bounds.

3: I/O ERROR. Some device driver returned with an error. The most common I/O errors are caused by forgetting to specify an input or output file on the command line, or mistyping the name of an input file. Perhaps a device number in an intrinsic call is missing.

4: BAD OPCODE. Invalid opcode encountered. When this occurs, the stack is out of balance and the program has probably blown-up. It's not a bad idea to reboot your computer to be absolutely safe, although this is unnecessary if running under a protected operating system such as Windows XP. The common causes of this error are an array subscript was incorrectly computed or an intrinsic was incorrectly used.

5: BAD INTRINSIC. 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. Some versions of the run-time code (NAT) do not support floating-point calculations and their related intrinsics (such as Fix).

6: DIV BY 0.0. Floating-point divide by zero. If untrapped, the largest possible real value is returned.

7: OVERFLOW. Floating-point overflow. Some calculation exceeded the upper limit of ±1.79E+308. If untrapped, the largest possible real value is returned.

8: UNDERFLOW. Floating-point underflow. A calculation exceeded the lower limit of ±2.23E-308. If untrapped, 0.0 is returned.

9: FIX OVERFLOW. Fixed-point overflow. Attempted to Fix too large or too small a number (greater than 65535.0 or less than -65535.0). If untrapped 32767 is returned with the appropriate sign. This error can also be caused by the Mod intrinsic if an internal calculation exceeds 15 bits of precision.

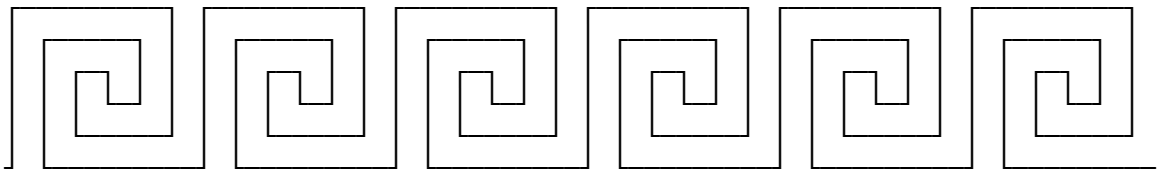
10: SQRT < 0. Square-root error. Attempted to take the square root of a negative number. This error is also trapped by the ASin and ACos intrinsics if the argument is outside of its legal range (-1.0 <= arg <= 1.0). If untrapped, the square root of the absolute value is returned.

11: LOG <= 0. Logarithm error. Attempted to take the logarithm of a number that's less than or equal to zero. If untrapped and the argument is 0.0, the smallest possible negative value is returned (minus infinity). If the argument is less than 0.0, the logarithm of the absolute value is returned.

12: EXP OVERFLOW. Exponential error. Exp intrinsic caused an overflow. If untrapped, the largest possible value is returned.

13: Unused.

14: ATAN2(0.0, 0.0). ATan2(0.0, 0.0) is undefined. Returns 0.0 if untrapped.



A.3: COMMON ERRORS

There are some errors that seem to catch everyone when they first start programming in XPL0. Here is a list of these errors beginning with the most common.

1. There are several commands or symbols that must be used in pairs. Many newtimers omit one of the pairs. The most likely place that you might do this is with "begin-end"s. It's easy to get the wrong number of "end"s at the end of a complex procedure. The easiest way to keep track of these is to use indentation:

```
        begin
        . . .
                begin
                . . .
                        begin
                        . . .
                        end;
                end;
        end;
```

Each indentation must have a "begin" and a corresponding "end".

Here are some other pairs to watch out for:

```
        " . . . "       Double quotes around text strings
        ( . . . )       Parentheses
        [ . . . ]       Brackets (same as  "begin-end"s)
        \ . . . \       Comments (when not the last item on the line)
```

2. A semicolon can catch you in two ways. One is that there must be a semicolon between all statements in the program. The other is that you must not place a semicolon before the "else" of an "if" statement or the "other" of a "case" statement. For example:

```
if N = Guess then
        begin
        Restart;
        MakeNumber;
        end  <————————————— semicolon is illegal here
else    begin
        N:= N + 1; <———————— semicolon is required here
        Restart; <—————————— semicolon is optional here
        end;
```

3. Intrinsics require various numbers of arguments. A common error is to pass the wrong number of arguments or the wrong type of arguments (integer versus real). This causes a stack imbalance and your program blows up.

```
     WRONG                  CORRECT
     ———————                ———————
Text("message");        Text(0, "message");
ChIn(0);                I:= ChIn(0);
I:= ChOut(0,^A);        ChOut(0, ^A);
X:= Sqrt(100);          X:= Sqrt(100.);
```

4. When arguments are passed to a procedure, the values passed are stored into the first variables declared and in the same order that they are passed. As a program is written, it's easy to add new variables to the declarations, which shift their order and change which arguments are passed into which variables. "Integer", "real", and "character" declarations can be mixed in any way necessary to properly pass values into the correct variables. It's often useful to have completely separate declarations for arguments and local variables. For example:

```
procedure Oink(I, X, Ch);
integer I;              \Arguments
real X;
integer Ch;
integer A, B, C;        \Local variables
begin
. . .
```

5. XPL0 does not do run-time array bounds checking. Thus it's possible to store something into an incorrect location in memory. Almost always, this is due to an error in the calculation of a subscript for an array.

6. Avoid using the same name both locally and globally. You can easily get confused as to which is which, and this can be a difficult error to find. If you use a local variable with the same name as a global variable, the compiler does not give a NAME ALREADY DECLARED error; the local variable is used instead of the global variable. As a consequence you should make global names longer and more formal than local names. For example, avoid using a name like "I" for a global. At the very least call it "II".

A.4: KEYBOARD SCAN CODES

3B	F1	68	Alt-F1	1E	Alt-A
3C	F2	69	Alt-F2	1F	Alt-S
3D	F3	6A	Alt-F3	20	Alt-D
3E	F4	6B	Alt-F4	21	Alt-F
3F	F5	6C	Alt-F5	22	Alt-G
40	F6	6D	Alt-F6	23	Alt-H
41	F7	6E	Alt-F7	24	Alt-J
42	F8	6F	Alt-F8	25	Alt-K
43	F9	70	Alt-F9	26	Alt-L
44	F10	71	Alt-F10		
				2C	Alt-Z
54	Shift-F1	78	Alt-1	2D	Alt-X
55	Shift-F2	79	Alt-2	2E	Alt-C
56	Shift-F3	7A	Alt-3	2F	Alt-V
57	Shift-F4	7B	Alt-4	30	Alt-B
58	Shift-F5	7C	Alt-5	31	Alt-N
59	Shift-F6	7D	Alt-6	32	Alt-M
5A	Shift-F7	7E	Alt-7		
5B	Shift-F8	7F	Alt-8	03	Ctrl-2
5C	Shift-F9	80	Alt-9	0F	Shift-Tab
5D	Shift-F10	81	Alt-0	47	Home
		82	Alt-Hyphen	48	Up arrow
5E	Ctrl-F1	83	Alt-=	49	PgUp
5F	Ctrl-F2			4B	Left arrow
60	Ctrl-F3	10	Alt-Q	4D	Right arrow
61	Ctrl-F4	11	Alt-W	4F	End
62	Ctrl-F5	12	Alt-E	50	Down arrow
63	Ctrl-F6	13	Alt-R	51	PgDn
64	Ctrl-F7	14	Alt-T	52	Insert
65	Ctrl-F8	15	Alt-Y	53	Delete
66	Ctrl-F9	16	Alt-U	73	Ctrl-Left arrow
67	Ctrl-F10	17	Alt-I	74	Ctrl-Right arrow
		18	Alt-O	75	Ctrl-End
		19	Alt-P	76	Ctrl-PgDn
				77	Ctrl-Home
				84	Ctrl-PgUp

A . 5 : S Y N T A X S U M M A R Y

FACTORS		SECTION
CONSTANTS:	Decimal integers: 123, -19375	1.0
	Hex and binary integers: $FE00, %11_0110	1.1
	ASCII characters: ^A, ^z	1.2
	Real numbers: 6.63e-34	1.3
	Declared constants: define Pi=3.14;	1.5, 2.9
	True and false	2.4
VARIABLES:	Integers: Guess	1.4
	Reals	1.4
	Array elements: Side(N)	5
FUNCTIONS		4.5
INTRINSICS that return a value		4.6
EXTERNALS that return a value		4.13
TEXT STRINGS: "..."		5.2
CONSTANT ARRAYS: [CONSTANT, ... CONSTANT]		5.5
ADDRESS of a variable or array: addr Frog, @Array(3)		5.7

OPERATORS

The operator precedence (priority) is shown in parentheses; 1 is highest.

Unary minus (or plus):	- (+) (1)		2.2
Shifts:	<< >> ->> (2)		2.7
Multiplication:	* (3)		2.0
Division:	/ (3)		2.0
Addition:	+ (4)		2.0
Subtraction:	- (4)		2.0
Equal:	= (5)		2.3
Not equal:	# (5)		2.3
Less than:	< (5)		2.3
Less than or equal:	<= (5)		2.3
Greater than:	> (5)		2.3
Greater than or equal:	>= (5)		2.3
Boolean "not":	~ (6)		2.5
Boolean "and":	& (7)		2.5
Boolean "or":	! (8)		2.5
Boolean "xor":	| (8)		2.5
If expression:	if (9)		2.8

SPECIAL CHARACTERS

Space, tab, carriage return, and form feed are formatters		1.8
()	Expression evaluation priority, arguments, and subscripts.	2.0, 3.9, 4.2, 5.0
;	Statement and procedure separator and declaration terminator	3.1, 3.11
\\ \	Comment (except in strings and "include" path names)	3.10
^	ASCII constants, and ", ^ and ctrl chars in strings	1.2, 5.2
_	Underline in a variable or procedure name or in a number	1.4
{}	Assembly code	Addendum

STATEMENTS

```
VARIABLE:= EXPRESSION;  . . . . . 3.0
begin STATEMENT; STATEMENT; ... STATEMENT end;  . . . . . 3.1
[STATEMENT; STATEMENT; ... STATEMENT];  . . . . . 3.1
if BOOLEAN EXPRESSION then STATEMENT;  . . . . . 3.2
if BOOLEAN EXPRESSION then STATEMENT else STATEMENT;  . . . 3.2
case of  . . . . . 3.3
        BOOLEAN EXPRESSION, ... BOOLEAN EXPRESSION: STATEMENT;
        ...
        BOOLEAN EXPRESSION, ... BOOLEAN EXPRESSION: STATEMENT
        other STATEMENT;
case INTEGER EXPRESSION of  . . . . . 3.3
        INTEGER EXPRESSION, ... INTEGER EXPRESSION: STATEMENT;
        ...
        INTEGER EXPRESSION, ... INTEGER EXPRESSION: STATEMENT
        other STATEMENT;
while BOOLEAN EXPRESSION do STATEMENT;  . . . . . 3.4
repeat STATEMENT; ... STATEMENT until BOOLEAN EXPRESSION;  . . 3.5
loop STATEMENT;  . . . . . 3.6
quit;  . . . . . 3.6
for VARIABLE:= INTEGER EXPRESSION, INTEGER EXPRESSION  . . . 3.7
        do STATEMENT;
for VARIABLE:= INTEGER EXPRESSION downto INTEGER EXPRESSION  . 3.7
        do STATEMENT;
exit;  . . . . . 3.8
exit BYTE EXPRESSION;  . . . . . 3.8
SUBROUTINE NAME(EXPRESSION, ... EXPRESSION);  . . . . . 3.9, 4.0
return;  . . . . . 4.4
return EXPRESSION;  . . . . . 4.5
;   (null statement)  . . . . . 3.11
```

DECLARATIONS

```
integer NAME, NAME, ... NAME;  . . . . . 1.5
real NAME, NAME, ... NAME;  . . . . . 1.5
define NAME=CONSTANT, ... NAME=CONSTANT;  . . . . . 1.6
define NAME, NAME, ... NAME;  . . . . . 1.6
procedure NAME(COMMENT);  . . . . . 4.0
function TYPE NAME(COMMENT);  . . . . . 4.5
code TYPE NAME(COMMENT)=INTEGER, ... NAME(COMMENT)=INTEGER;  . 4.6
fprocedure NAME(COMMENT), NAME(COMMENT), ... NAME(COMMENT);  . 4.9
ffunction TYPE NAME(COMMENT), NAME(COMMENT), ... NAME(COMMENT); 4.10
eprocedure NAME(COMMENT), NAME(COMMENT), ... NAME(COMMENT);  . 4.12
efunction TYPE NAME(COMMENT), NAME(COMMENT), ... NAME(COMMENT); 4.12
public procedure NAME(COMMENT);  . . . . . 4.12
public function TYPE NAME(COMMENT);  . . . . . 4.12
external TYPE NAME(COMMENT), NAME(COMMENT), ... NAME(COMMENT);  4.13
integer NAME(DIMENSIONS), ... NAME(DIMENSIONS);  . . . . . 5
real NAME(DIMENSIONS), ... NAME(DIMENSIONS);  . . . . . 5
character NAME(DIMENSIONS), ... NAME(DIMENSIONS);  . . . . 5
segment TYPE NAME(DIMENSION), ... NAME(DIMENSION);  . . . . 5.9
```

I N D E X

A D D E N D U M

VERSION 3.0

The double backslash "\\" comments out everything on the rest of line regardless of any backslashes it might contain. This is handy for commenting out sections of code.

When a program starts, any characters entered on the command line after the program name are copied into device 8's buffer. This provides a convenient way to pass information to a program, such as file names or numeric values.

The at-sign (@) is an alternative "addr" operator that works better when returning reals from a procedure that uses the call-by-reference technique. It works exactly the same as "addr" when used on an integer variable, but it returns a real pointer when used on a real variable. Note that "addr" and @ also work on subscripted variables.

The starting (integer) value for an enumeration can be specified instead of always starting at zero. For example:

```
define Jan=1,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec;
```

The "string 0" option for zero-terminated strings is now available in all versions. The Text intrinsic works no matter which termination is used.

The random number generator has been improved over several versions to where it now passes all the Diehard tests.

VERSION 2.9

The command-line switch /? displays a compiler's usage and lists its switch options.

VERSION 2.8

The command words "and", "or" and "xor" can be used in place of "&", "!" and "|".

VERSION 2.7

"For" loops can decrement using the "downto" command. What was previously kludged using negative values can now be written in a straightforward way, for example:

```
for I:= -10, -1 do    [IntOut(0,-I); CrLf(0)];  \old method
for I:= 10 downto 1 do [IntOut(0,I); CrLf(0)];  \new method
```

For symmetry, a normal incrementing "for" loop can use the command word "to" in place of the comma, for example:

```
for I:= 1, 10 do   [IntOut(0,I); CrLf(0)];      \existing method
for I:= 1 to 10 do [IntOut(0,I); CrLf(0)];      \makes same code
```

An arithmetic shift right (that preserves the sign bit) can be done using the "->>" symbol. This provides a fast way to divide integers by powers of 2. This is obvious for positive integers, but it does not give the exact same value as a divide for negative integers. The difference is that if there is a remainder, an integer divide truncates the quotient toward zero, whereas the arithmetic shift right truncates the quotient toward negative infinity. Here are some examples:

```
 27 / 4 = 6
 27 >> 2 = 6
 27 ->> 2 = 6
-27 / 4 = -6
-27 >> 2 = 16377
-27 ->> 2 = -7
-24 ->> 2 = -6
```

The interpreted version, XPLI, supports the "port" command, like all the other compilers, but it does not support the "abs", "rem", "swap" or "extend" commands. These commands must be replaced with their equivalent intrinsic calls.

I2LS.COM is a version of the interpreter that makes programs about 7K bytes smaller. It has no floating point capability or high-speed line draw.

VERSION 2.6

Variables can be declared after procedures. This makes it easier to break programs up into separate files, which can make them more modular and easier to understand and manipulate.

For example, oftentimes there is a group of procedures that share common variables that are not used by the rest of the program. You can put these procedures and their variables into a single file and "include" the file in the main body of the program. Previously, these global variables had to be declared at the beginning of the program.

Another advantage of this feature is that you can declare a variable immediately above the Main procedure if that's the only place it's used. For example, if you use "I" as a scratch index in Main, it's nice if "I" is not global to the entire program where it might mistakenly be used by a nested procedure.

The percent sign can be used to represent binary numbers. For example, %10011100 is the same value as $9C. Because binary numbers can blur into meaningless strings of 1's and 0's, underlines can be used to make them more recognizable, for example: %1001_1100 = $9C.

For consistency underlines can be inserted into any number. For example, $12_34, or 123_456.78. The underlines are simply ignored by the compiler. Underlines may also appear in any number read in by the intrinsics IntIn, HexIn and R1In.

VERSION 2.5

Assembly code can be inserted directly into an .XPL source file using the new command word "asm". This is described in detail in a section ahead.

SWITCHES

Command-line switches are used to modify the behavior of the compilers. Many of them are not normally used. However, for completeness of documentation they're all listed here.

The command-line switches for XPLNQ are:

```
/?  Display list of switches
/A  divert Assembly code to monitor instead of the output file
/C  insert I2L Comments into the output file
/L  List source code to the monitor screen
```

XPLX also recognizes these switches:

```
/B  do short-circuit evaluation of Boolean expressions
/D  Debug: include XPL0 source code in .ASM output file
/J  generate short conditional Jumps (to be fixed by MASM 6)
/S  generate near procedure calls for code Smaller than 64K
/2  align loops to word boundaries to speed them up
/3  align loops to double-word boundaries to speed them up
```

XPLIQ only recognizes these switches:

```
/?  Display list of switches
/D  list Debug information to the monitor screen
/L  List source code to the monitor screen
/W  display Warning messages
```

Procedure calls can be optimized if a program is less than 64K by using the /S (Small) switch. This replaces the normal far call instruction with a near call. Calls to external XPL0 procedures (eprocs) are handled the same as normal procedures. Calls to intrinsics and external assembly language routines are always far regardless of the /S switch. (These calls can be optimized a little using the /F switch in LINK.) If several modules are linked together, they must all be compiled the same way--either with /S or without it.

Conditional jumps can be optimized using the /J switch and MASM 6. The /J switch makes XPLX output short conditional jumps, such as JNE. It relies on MASM 6 to automatically replace them with jumps over long jumps if necessary. For example, MASM 6 automatically replaces JNE with JE $+5 and JMP if the target location is more than 128 bytes away. It also replaces a normal 3-byte JMP with a short JMP wherever possible.

MASM 5.10 nor TASM 3.1 support the /J switch, that is, they do not fix short jumps that are out of range. TASM does not support the environment variable INCLUDE, which tells MASM where to look for include files (such as RUNTIME.ASM) if they are not in the current directory.

SHORT-CIRCUIT BOOLEANS

Some boolean expressions can be executed more quickly by using the /B command-line switch to enable short-circuit evaluation. Short-circuit evaluation is used in conditional statements to bypass the rest of a boolean expression when the result is already known. For example:

```
if A=3 ! B=5 ! C=7 ! D=11 then Prime:= true;
```

In this expression if B is equal to 5 then there's no reason to compare C to 7 and D to 11 because Prime will be assigned the value "true" despite these additional comparisons.

The reason this feature uses a command-line switch rather than being done automatically is that it can cause some errors, although they're very unlikely.

An error can occur if a term in the boolean expression contains a function call that does more than simply return a value. Such a function is said to have a "side effect", and it's generally considered a bad programming practice. Here is an example:

```
if P<10 & P/3=N then DoSomething;
R:= Rem(0);
```

Rem(0) is not defined when P is >= 10 and short-circuit evaluation is enabled. The divide operation not only returns the quotient, but also sets the remainder as a side effect.

Another reason for not automatically using short-circuit evaluation is that some older programs might give a compile error unless a small modification is made. For instance, the statement: "while A | B do..." gives the new compile error 75: EXPRESSION MUST BE ENCLOSED IN PARENTHESES. Adding parentheses solves the problem: "while (A | B) do..." This problem only occurs with the exclusive-or operator and the "if" expression, and these are rarely used in the boolean expression of a conditional statement. For example: while (if A=1 then F1 else F2) do....

Since expressions are evaluated from left to right, it's faster to test for frequent conditions on the left side, for example:

```
if Ch>=^0 & Ch<=^9  !  Ch>=^A & Ch<=^F then DoHex;
```

(Hint: There are ten digits in the range 0..9 and only six in the range A..F.) Also, it's more efficient to do comparisons before testing flags. For example, this takes advantage of short-circuit evaluation:

```
if Printer=Epson & Pin9 then ...
```

This does not:

```
if Pin9 & Printer=Epson then ...
```

Avoid using unnecessary parentheses because expressions enclosed in parentheses are not short-circuit evaluated.

IN-LINE ASSEMBLY CODE

The native compilers have the ability to handle assembly code that's inserted directly into a program. The reserved word "asm" designates that the following characters on the line are assembly code, and they are to be copied to the output (.ASM) file. For example:

```
asm     cli
asm     mov     ax, 102         ;comment
asm     mov     bx, Frog        ;Comment
```

Assembly code must be written in lowercase characters except when an XPL variable or constant name is used. These are written in the usual way with at least the first letter capitalized. This enables the compiler to distinguish them from the rest of the assembly code and to substitute them with their corresponding code. For instance, in the above example, "Frog" might be replaced with something like "[SI+4]". Capital letters may be used in comments because comments are ignored by the compiler.

If several lines of assembly code are needed, they may be written this way:

```
asm     {
        cli
        mov     ax, 102         ;comment
        mov     bx, Frog        ;Comment
        }
```

The ability to insert assembly code into a high-level language program is a two-edged sword. In general it should be avoided, but there are instances when it's very useful.

The most obvious application is to replace compiled code with more efficient assembly code. For instance "Irq(false)" can be replaced with "asm cli", which is at least ten times faster (except under Windows XP, which simulates the cli). Similarly, "POut(Time, $40, 1)" could be replaced with:

```
asm     mov ax, Time
asm     out 40h, ax
```

Assembly language provides low-level control that a high-level language can't. Consider this expression: Frog * 777 / 1000. If Frog is above 42, the calculation will overflow in 16-bit XPL. However, the following will not overflow even when Frog is 32767:

```
asm     {
        mov     ax, 777
        imul    Frog            ;ax:dx := ax * Frog
        mov     cx, 1000
        idiv    cx              ;ax := ax:dx / cx
        }
```

Here is an example of a double-precision add:

```
TimeLo:= TimeLo + 143;
asm     {jnc    tm10
         inc    TimeHi
        tm10:};
```

RULES AND RESTRICTIONS

With the power of assembly language, it's easy to shoot yourself in the foot. When using XPLX, the SI and DI registers must not be altered, and of course altering DS, SS or CS is fatal.

The compilers generate the correct code for named constants such as: "def Frog=123; asm mov ax, Frog". XPLX also generates the correct code for variables except as follows. If the variable is at an intermediate level (neither local or global), the inserted assembly code must make sure the correct BASEn is loaded into the BP register. With XPLN the correct BASEn must be loaded in the SI register for all variables other than globals.

Here is an example that fills an integer array with a pattern. It runs about seven times faster than the equivalent "for" loop in XPLX, and 24 times faster than XPLN (as measured on a Duron 850).

```
\XPLN needs the following line, but it must not be in XPLX
\asm    mov     si, base1
asm     {
        push    ds              ;es:= ds
        pop     es
        push    di              ;save di register
        mov     di, Array
        mov     ax, Pattern
        mov     cx, Size
        cld                     ;set direction flag to increment
        rep stosw               ;es:[di++] := ax; cx--
        pop     di              ;restore di register
        }
```

Segment variables are not supported by in-line assembly code.

Line labels are allowed if they are in lowercase and don't conflict with names generated by the compiler. Certain names are reserved and cannot be used such as: l208, l13, cseg, dseg, base2, intr10. The assembler will flag an error if there's a conflict.

Expressions are evaluated by the assembler, not by the compiler. Thus, for instance, hex numbers are represented with a trailing "h" instead of a leading "$".

The bracket "}" ends an assembly-code section, even if it occurs inside a "; comment", but not if occurs inside a "\ comment".

WARNINGS

A separate version of the run-time code is required for XPLX (XX.BAT). Code compiled with XPLX must be linked to NATIVEX, NATIVE7X, or NATX. Code compiled with XPLN must be linked to NATIVE, NATIVE7, or NAT. This is handled automatically when using the batch files (X, XX, XN, XJSB, and XS).

XJSB.BAT makes programs about 7K smaller than with XX.BAT. It accomplishes this mostly by linking in NATX.OBJ instead of NATIVEX.OBJ. This small version of the run-time code does not support floating-point calculations nor does it support high-speed line draw (intrinsic 42: Line). XJSB also enables the /J /S /B switches. This means that MASM 6 must be used to fix any short jumps, that the program code cannot be larger than 64K, and that short-circuit boolean evaluation is done. Floating point calculations will not give a compile error, but they will give the run-time errors 4 and 5 (unless they are turned off with the Trap intrinsic). Line draw still works, but it's many times slower than with the full version of the run-time code. XS.BAT does the same thing but with the interpreted version instead.

Beware of STDLIB.OBJ. It must be compiled with the same compiler used to make other .OBJ files. If your main program module uses XPLX then STDLIB.XPL must be compiled using XPLX, not XPLN. One critical difference between these two compilers is that XPLX returns integer values from function calls in the AX register while XPLN returns them in global 0. Another difference is that XPLX uses the DI register as the heap pointer. Also, be sure that the /S switch is used consistently on all .XPL modules that are linked together.

XPLX puts the control variable of a "for" loop in a register. Although extremely unlikely, a possible problem is using a pointer to change this variable. For example, the following is not an infinite loop if compiled by XPLX, but it is if it's compiled by the other compilers:

```
A:= addr I;
for I:= 1, 10 do
        A(0):= 3;
```

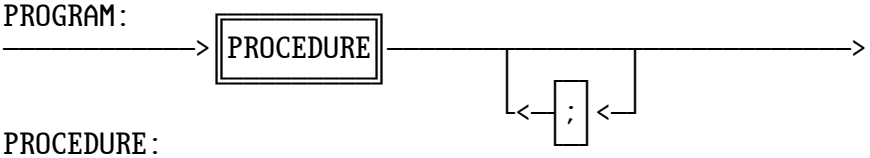
XPLI's /W switch displays a warning message for the above situation.

Be aware that using the Reserve intrinsic to reserve an odd number of bytes in a character array can misalign the heap, which makes any variable declared from that point on (including in called procedures) misaligned, which can significantly slow a program. The Reserve intrinsic does not automatically align to a word boundary because some early programs took advantage of consecutive Reserves allocating contiguous space.

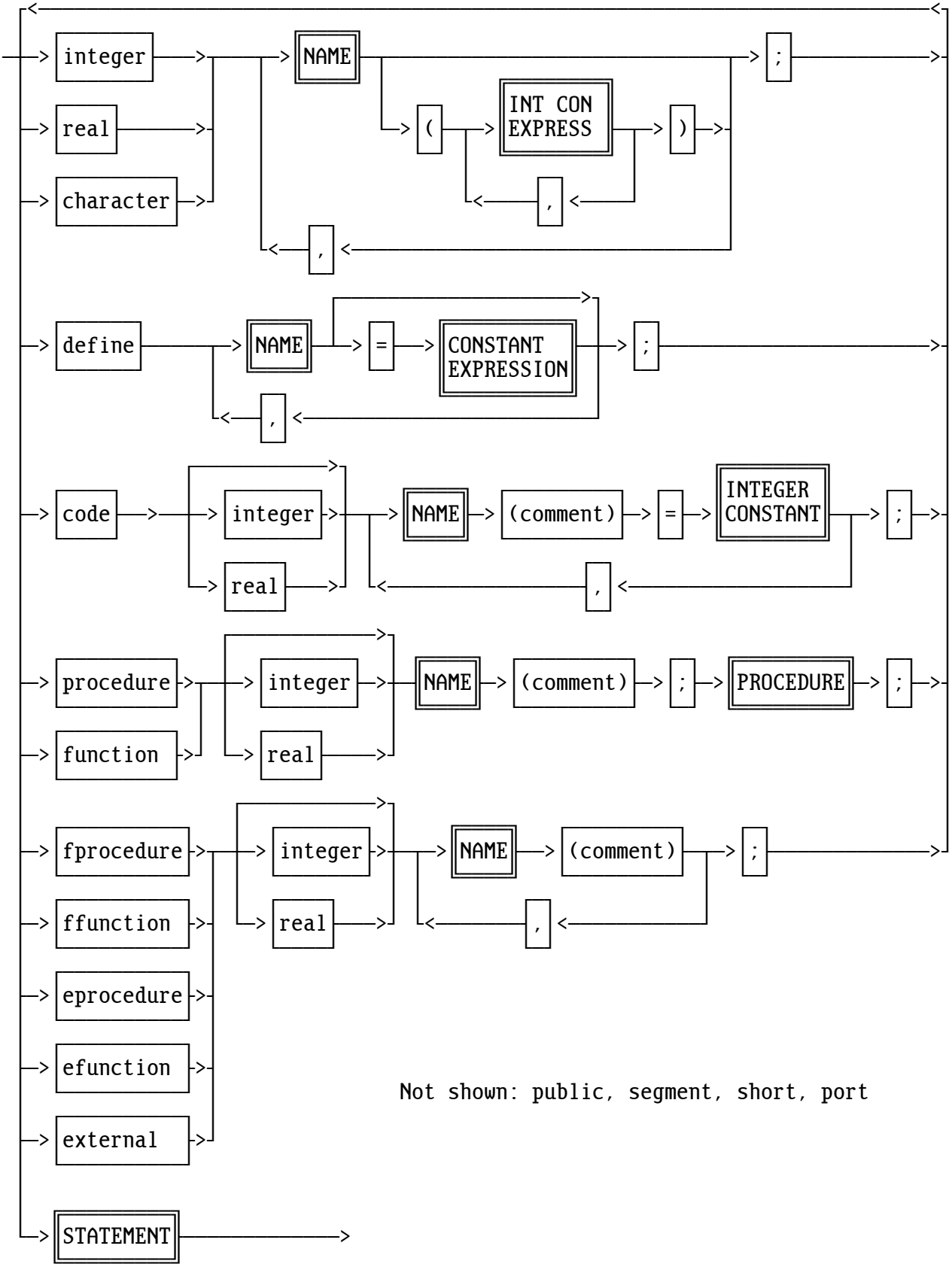
Consecutive dimensioned arrays cannot allocate contiguous space because the pointer to the array precedes the reserved space. (Declared arrays are automatically aligned by the native compilers to a word boundary regardless of whether an odd number of bytes are used in a character array.)

It is more efficient to declare dimensioned arrays last, after all other variables have been declared. This allows single-byte offsets to address these variables; whereas if an array is declared with 128 bytes or more, then double-byte offsets are required to access the variables.

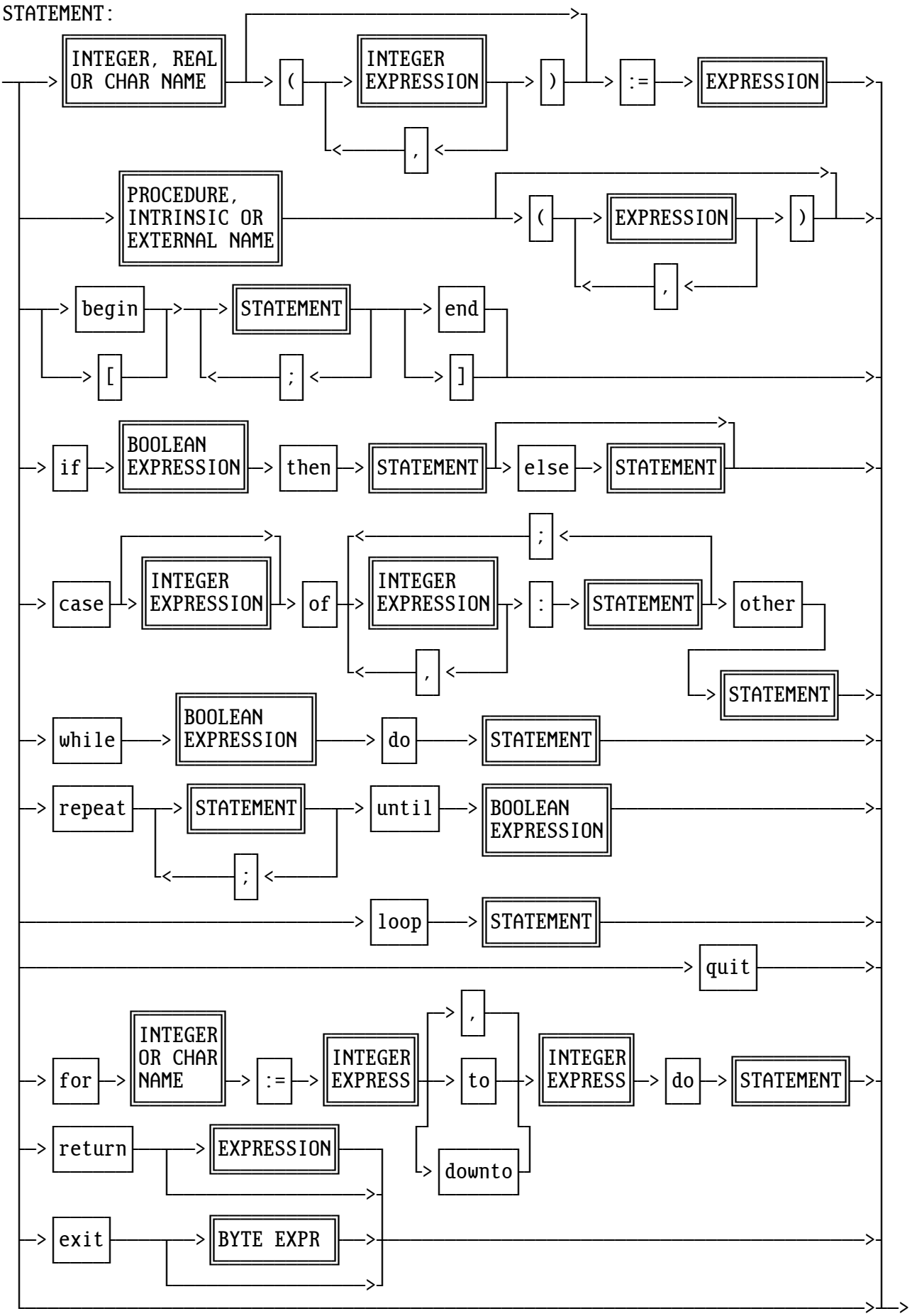
PROGRAM:



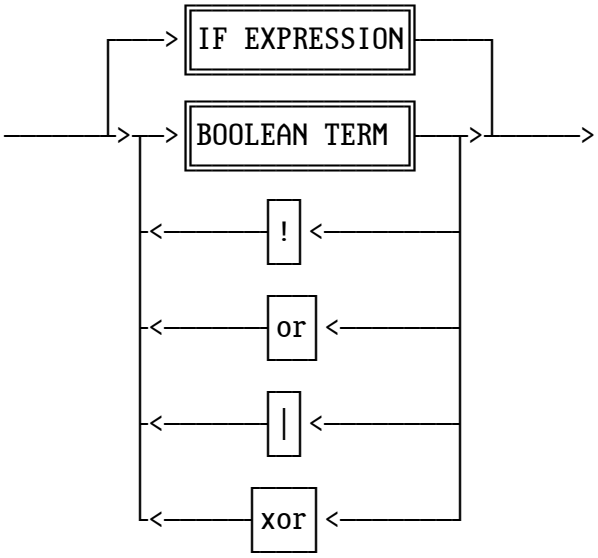
PROCEDURE:



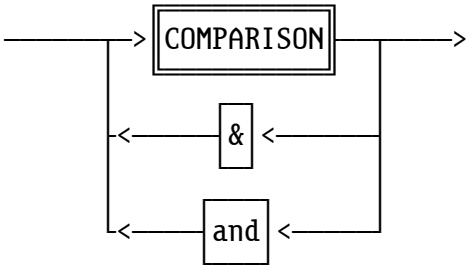
Not shown: public, segment, short, port


STATEMENT:
INTEGER, REAL OR CHAR NAME
(
INTEGER EXPRESSION
,
)
:=
EXPRESSION
PROCEDURE, INTRINSIC OR EXTERNAL NAME
(
EXPRESSION
,
)
begin
[
STATEMENT
;
end
]
if
BOOLEAN EXPRESSION
then
STATEMENT
else
STATEMENT
case
INTEGER EXPRESSION
of
;
INTEGER EXPRESSION
,
:
STATEMENT
other
STATEMENT
while
BOOLEAN EXPRESSION
do
STATEMENT
repeat
STATEMENT
;
until
BOOLEAN EXPRESSION
loop
STATEMENT
quit
for
INTEGER OR CHAR NAME
:=
INTEGER EXPRESS
,
to
downto
INTEGER EXPRESS
do
STATEMENT
return
EXPRESSION
exit
BYTE EXPR

EXPRESSION:

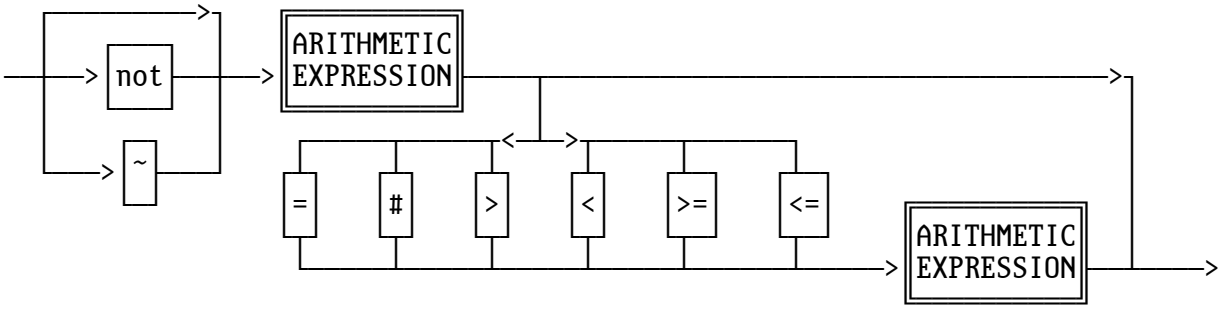


BOOLEAN TERM:

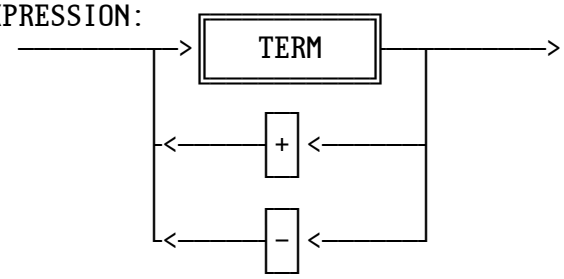


IF EXPRESSION:

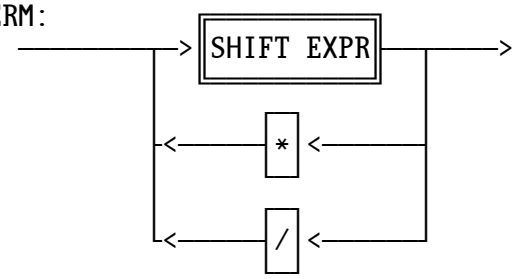
COMPARISON:



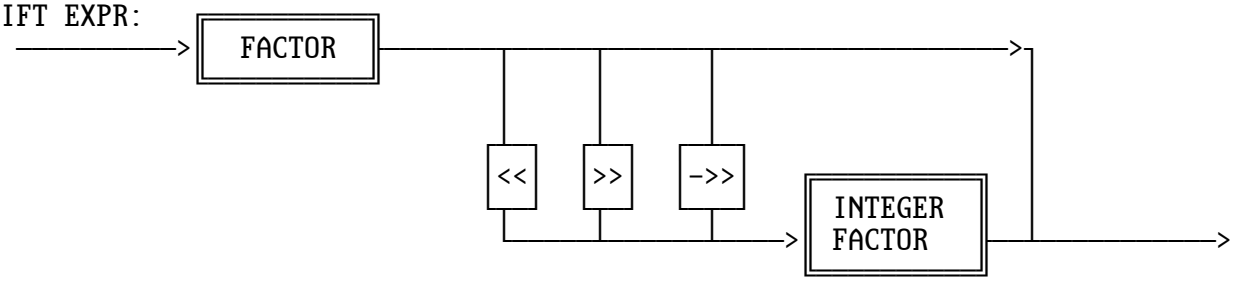
ARITHMETIC EXPRESSION:



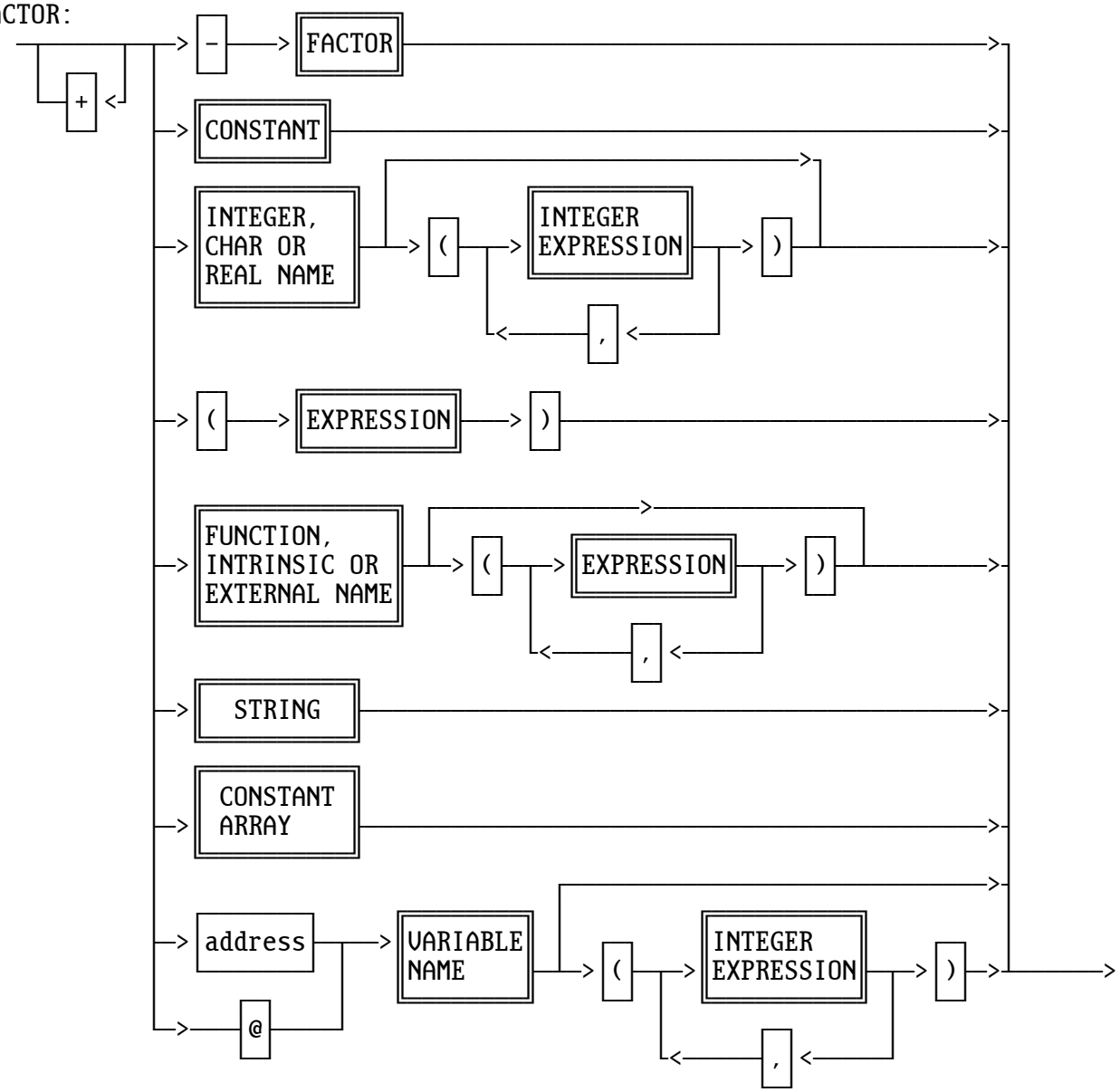
TERM:



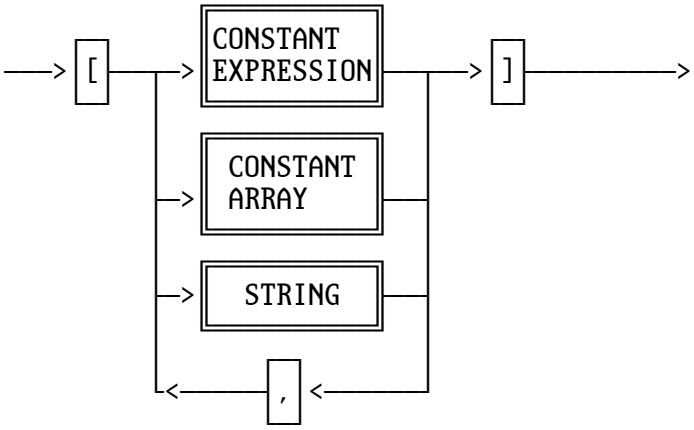
SHIFT EXPR:



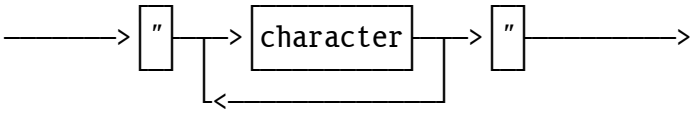
FACTOR:



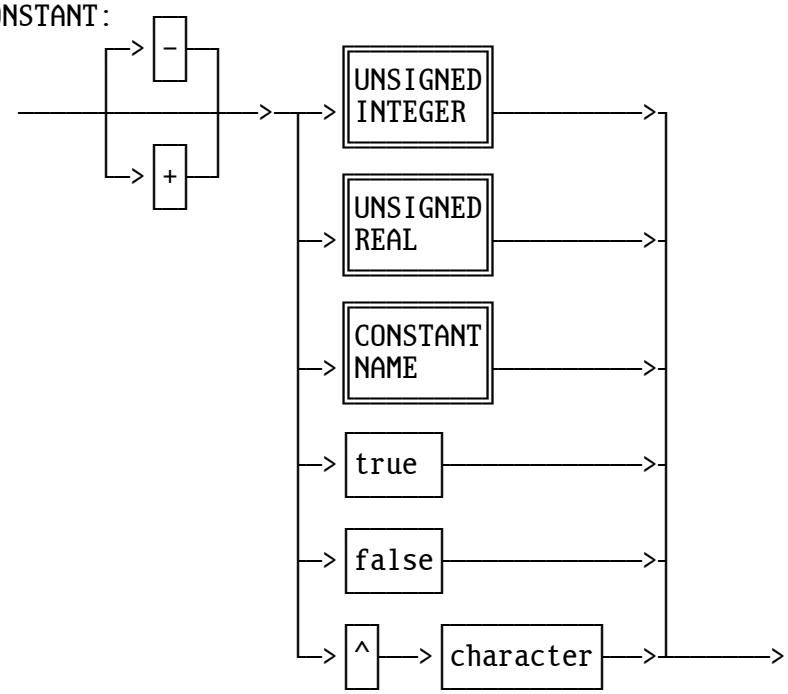
CONSTANT ARRAY:



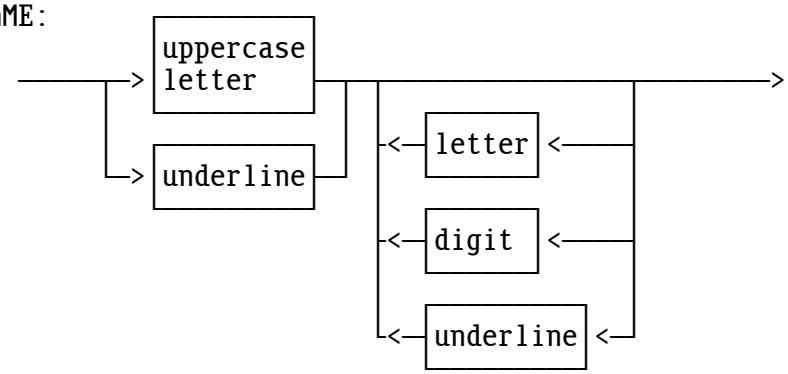
STRING:



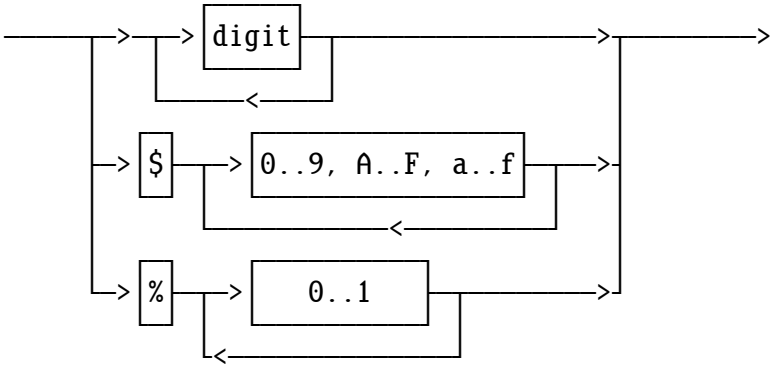
CONSTANT:



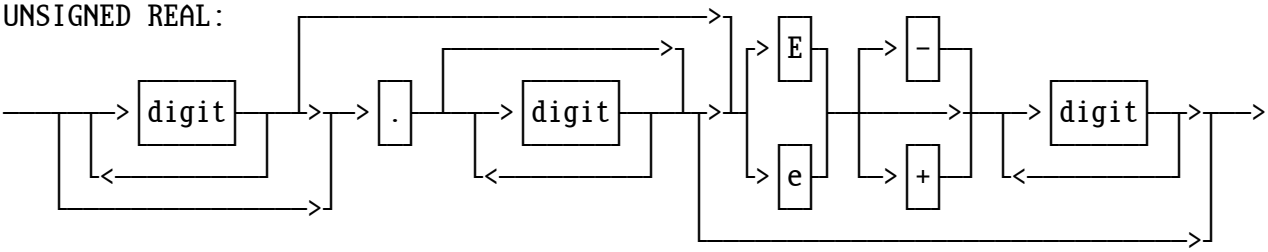
NAME:



UNSIGNED INTEGER:



UNSIGNED REAL:



≤