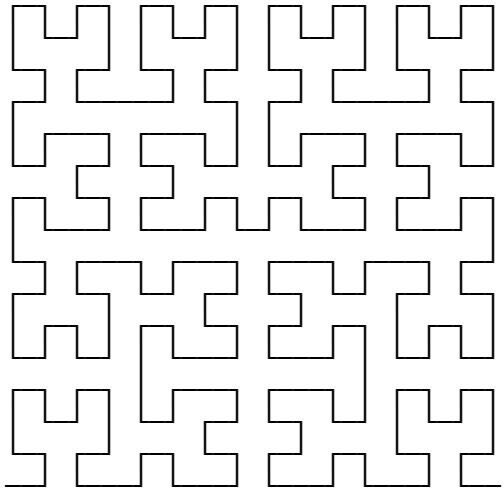
XPL0

PROGRAMMING LANGUAGE MANUAL

FOR THE RASPBERRY PI

VERSION 2.0



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 web site: 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, 80x86 and ARM). There is a Windows-compatible version called EXPL. This manual describes the version that runs on a Raspberry Pi under Raspbian Linux.

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 maintain.

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 Syntax Summary and Syntax Diagrams offer 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 screen (output device 0) and gets a number (INTeger IN) from the keyboard (input device 0). In XPL0 several different input and output devices can be called from the program. This enables direct access to the monitor, keyboard, printer, storage 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 screen (Carriage Return and LineFeed).

Here is the complete program:

```
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
```

The command word "integer" 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 file. For example, to run the number guessing program, guess.xpl, type the following:

```
x guess
guess
```

"x" is a script file (in /bin) that does these steps:

1. Runs the compiler (xplr) to convert the .xpl source to a .s file.

2. Runs gcc to assemble the .s file to produce an executable file.

You can make your programs run faster by using the optimizing compiler, xpl0. To do this substitute xx for x.

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 from 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" (or use brackets []). 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:

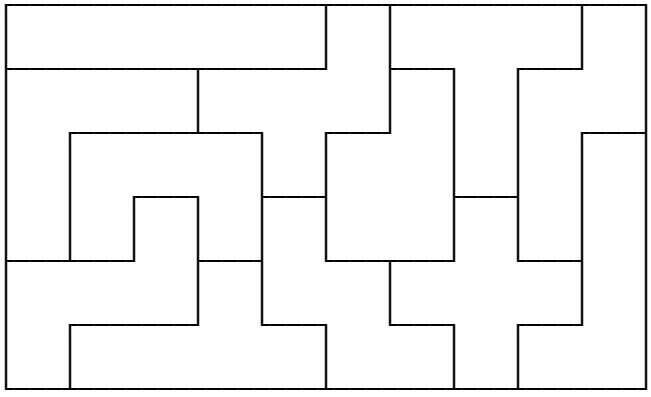
```
if Guess < Number then
        begin
        Text(0, "Too low");
        InputGuess;
        if Guess < Number then Text(0, "Still too low")
        end
```

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 provides a 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, whereas a variable is a value that can be changed. Factors are classified as integer or real. An integer is a 32-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. 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 -2147483648 through 2147483647. 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 $00000000 through $FFFFFFFF. Hex is very useful when programming at the machine level. Here are some examples:

```
        $123            $1e0
        $FFC0           $00fff
```

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

Sometimes it's more convenient to use binary instead of hex. Binary numbers are indicated by a percent sign (%). 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 and -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 twice 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 or Python).

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 can be 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 like 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 its value. For example, this assigns numbers commonly used for months:

```
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.

```
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 declares a variable called "Counter" that will hold integer values. The next 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 two lines of declarations can be in any order.

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 screen. When the screen 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 screen.

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 LineFeed), is an intrinsic that moves to the beginning of a new line on the screen.

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:

```
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, the following three expressions, which are the same mathematically, are not equivalent because of the way integer division works.

```
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 2147483647 + 1, the result is -2147483648. This is desirable because $7FFF_FFFF + 1 = $8000_0000, 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
-7/3     equals -2          -7/-3      equals 2
```

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 was 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 32-bit integers as signed,

```
$F000_0000 > $A000_0000   is true, but
$F000_0000 > $7000_0000   is false.
```

Converting the hex to decimal makes the reason apparent:

```
-268_435_456 > -1_610_612_736   is true, and
-268_435_456 >  1_879_048_192   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 (=$FFFFFFFF).

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 32 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. A frequent operation is masking, which uses the "and" operator to clear all bits except those 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 $FFFFFFFF, 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 $FFFFFFFF for "true". There are instances when the "not" of a true value is not false. For example, ~$33 is $FFFFFFCC, 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 $0000A050
~($F0F ! $33)           equals $FFFFF0C0
3+1 = 4                 equals true
3=4 & true              equals false
(1 ! $80) = $81         equals true (or $FFFFFFFF)
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 SHORT-CIRCUIT BOOLEANS (Advanced)

Some boolean expressions can be executed faster using short-circuit evaluation. It's 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.

Short-circuit evaluation is enabled with the -b command-line switch in the optimizing compiler (xpl0). The xx script enables it by default. The reason a switch is used rather than doing short-circuit evaluation automatically is that it can cause 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 to be 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 old 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 more probable conditions on the left side, for example:

```
if Ch>=^0 & Ch<=^9  !  Ch>=^A & Ch<=^F then DoHex;
```

(Assuming equal probability of all characters, it's faster to test the ten-digit range before the six-digit range.)

It's also more efficient to do comparisons before testing flags. For example, this takes advantage of short-circuit evaluation:

```
if Printer=Epson & Pin9 then ...
```

while this does not:

```
if Pin9 & Printer=Epson then ...
```

Avoid using unnecessary parentheses because expressions enclosed in parentheses are not short-circuit evaluated.

Beware of statements like this:

```
if I>=0 & A(I)=3 then ...
```

If I is negative and short-circuit evaluation is not used (which is always the case with the non-optimizing compiler) then a segmentation fault will probably occur because a location way outside the range of array A gets accessed. The problem can be avoided by rewriting the statement like this (which does the same thing as short-circuit evaluation):

```
if I>=0 then if A(0)=3 then ...
```

2.7 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 32 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
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 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?
```

2.8 SHIFT OPERATORS (Advanced)

If you're familiar with assembly language then you'll 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 32-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 31. Values outside this range can give unexpected results.

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

"->>" means shift arithmetic right. Unlike the first two shift operators listed above that shift in zeros to fill the empty bit locations, an arithmetic shift right 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 it's the same as dividing by powers of two (except that negative values are truncated toward minus infinity rather than toward zero).

Here are some examples:

```
1 << 1              = 2
1 << 0              = 1
$30 << 2            = $000000C0
$50 >> 4            = $00000005
$FFFFFF5A >> 4      = $0FFFFFF5
$FF5A ->> 4         = $0000FFF5
$FFFFFF5A ->> 4     = $FFFFFFF5
```

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

```
-1>>8 * 2      =     (-1 >> 8) * 2      = $000001FE
2 + 1<<4       =     2 + (1 << 4)       = $00000012
```

2.9 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")
```

2.10 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;
def     Pi = 22./7., Dia = fix(25000./Pi);
```

All expression operators can be used. However, function calls, such as Fix and Float cannot be used because they are evaluated at run-time rather than compile-time. In this situation you can instead use the command words "fix" and "float". Note the lowercase. The last example above rounds the diameter of the Earth to an integer 7955 miles.

2.11 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.

2.12 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 above. 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.

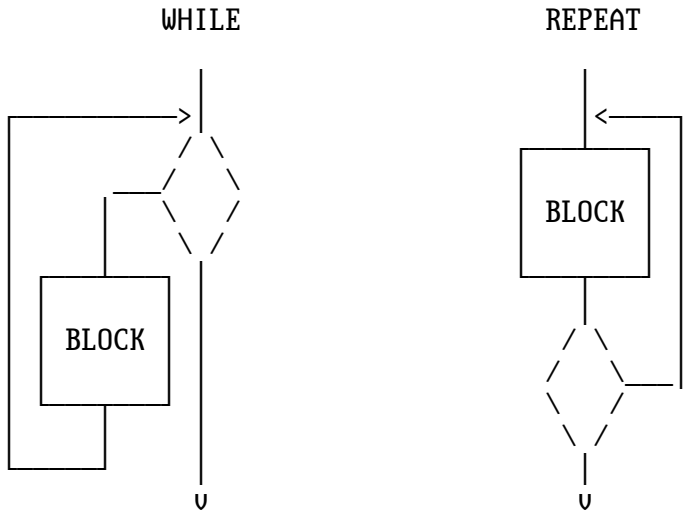
3.5 REPEAT - UNTIL

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 to 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 to X do Text(0, "Way too low")
```

Also note that 2147483647 cannot be used as the ending limit because there's not a larger signed number that can be represented with 32 bits. For example, writing "for I:= 1_000_000_000 to 2_147_483_647 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 Linux. The low byte of the value (0-255) of an optional expression following "exit" is returned. This return code can be tested in a script file with an "if" command. For example, the script files used to run the compilers (x and xx) use 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 or an intrinsic. (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 or inside a string). 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 useful when commenting out lines of code that contain comments. Here are some examples:

```
begin           \Move down the page
for X:= -10 to 10 \Twenty-one times\ do CrLf(0);
\\for X:= -10 to 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 to 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-2016
\This program displays a table of Fahrenheit temperatures
\ and their Celsius equivalents.

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.

3.13 IN-LINE ASSEMBLY CODE (Advanced)

Assembly code is normally not used in an xpl program. It adds complexity, and it prevents a program from running on other kinds of processors. However, there are instances when it's very useful.

The command word "asm" is used to insert assembly code. Characters that follow are sent to the output file (.s). For example:

```
    asm     mov     r0, #102        @ comment
    asm     ldr     r1, Frog        @ Comment
    asm     add     r0, r1
    asm     str     r0, Frog
```

Assembly code must be written in lowercase characters except where an xpl variable or constant name is used. Those are written the usual way with at least the first letter capitalized. This enables the compiler to distinguish them from the rest of the assembly code so that it can substitute them with their assembly code representations. For instance, in the above example, "Frog" might be replaced with something like [r11,#4]. Capital letters may be used in comments preceded with an at-sign because they are ignored by the compiler (as well as the assembler).

A practical application might be to replace xpl code with more efficient assembly code to speed up a critical loop. The following example shows an efficient way to reverse the order of the bytes in an integer. Note that several lines of assembly code can be enclosed in braces:

```
    asm     {ldr    r0, Frog       @ 0x12345678
             rev    r0, r0
             str    r0, Frog       @ 0x78563412
            }
```

Another application is to call Linux system routines. For example, this deletes any existing file in the current directory whose name is pointed to by the variable OutBack (which must be a zero-terminated string). It then renames the file whose name is pointed to by OutFile to the name in OutBack.

```
   asm     {ldr    r0, OutBack     @ delete file if it exists
            bl     remove
            ldr    r0, OutFile     @ rename, for example, outfile.ext
            ldr    r1, OutBack     @  to outfile.bak
            bl     rename
           }
```

Besides variable names, defined constants can also be used. Note that there is no # preceding MinusOne since it's generated automatically. Also note that hex numbers are represented by 0x instead of $.

```
def     MinusOne = -1;
asm     mov     r0, #-1
asm     mov     r1, MinusOne
asm     mov     r2, MinusOne + 0x41
```

Registers r4 to r15 should not be altered. If you must use them, use push and pop to preserve them.

Line labels can be used if they are in lowercase and if they don't conflict with names already used. Labels such as "l" (ell) followed by a number don't conflict with labels generated by the compilers. If there is a conflict, the assembler will display an error message.

Variables should be either local or at global level 0. Intermediate level variables aren't supported.

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:

```
integer Number;

        procedure MakeNumber;
        integer Times, X;       \Local variables
        begin                   \Randomly pick a random number
        Times:= Ran(10);
        for X:= 0 to 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:

```
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 to $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 about a hundred intrinsics in the run-time code (natrpi.s).

An intrinsic, like any named thing, must be declared before it can be used. This is done automatically by including the file codesr.xpl, so you don't need to be concerned about declaring intrinsic names unless you want to use a non-standard name. 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 preferred because they are 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 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 compile 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 compile error:

```
for I:= 10 to 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)

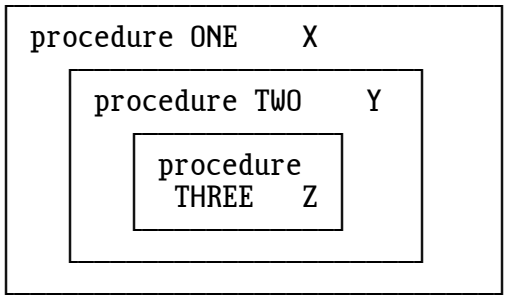
Scope is the feature that makes names active only in certain parts of a program. A name declared in one part does not necessarily conflict with the same name declared in another part. Scope is what makes a program modular.

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:

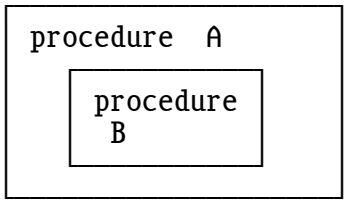


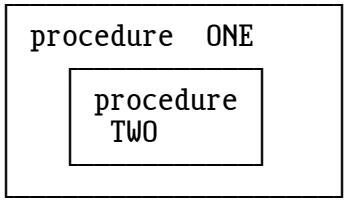
The statements inside procedure ONE can call procedure TWO because both the call and procedure TWO are within procedure ONE. However, the statements inside procedure ONE cannot call procedure THREE because the scope of THREE ends at the end of procedure TWO.

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:

```
                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, the graphics library routines can be inserted like this:

```
include /lib/graphics;
```

Note that forward slashes specify the path name in the normal Linux manner, and do not indicate division 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.

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 4-byte value in the range -2147483648 through 2147483647 (or $0000_0000 through $FFFF_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 to 70-1 do Dollars(I):= 0.00;
        Dollars(7):= 1.25;
        X:= Dollars(7) - 1.00;
        end;
```

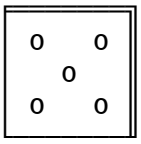
Note that subscripts are always integers, or integer expressions, even for a real array.

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);
```

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
        integer Side(6), I, N;

        begin
        for I:= 0 to 5 do Side(I):= 0;  \Initialize array with zeros
        for I:= 1 to 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 to 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
```

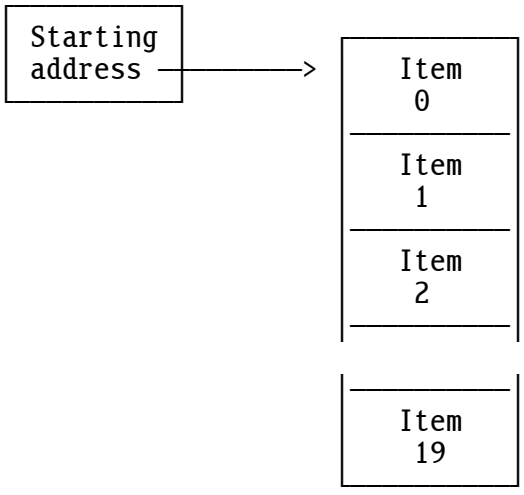
5.1 HOW ARRAYS WORK (Advanced)

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 to 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 is 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 32-bit address pointing to its data. The address is in the first four of the eight 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 is 64 megabytes that 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.

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 screen (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 written then any strings that follow will be terminated with a zero byte instead of the high bit set on their last character. This has the advantage of making them consistent with the way strings passed to Linux system routines must be terminated. It also enables the use of the extended characters ($80-$FF), such as the line draw characters, 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 to 3-1 do
    for J:= 0 to 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 to 3-1 do
    for J:= 0 to 5-1 do
        for K:= 0 to 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:

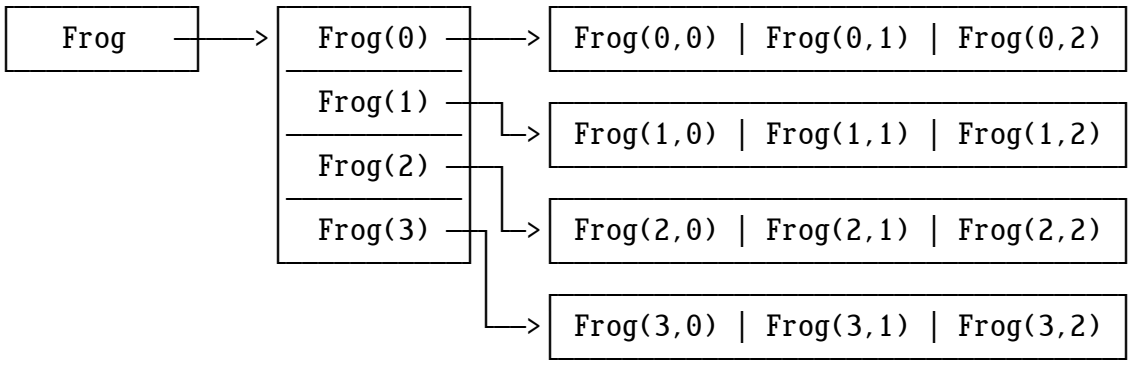
```
String(I,J):= ^A;
ChOut(0, String(99,3));
```

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 32-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*4);
for I:= 0 to 4-1 do Frog(I):= Reserve(3*4);
. . .
```

The Reserve intrinsic reserves the specified number of bytes and returns the starting address of the reserved memory space. The first statement reserves 16 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 something similar. It 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 like 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 * 4). For example:

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

is equivalent to:

```
character Frog;
int       I;
begin
Frog:= Reserve(4*4);
for I:= 0 to 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 to 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 would go 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 32-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:

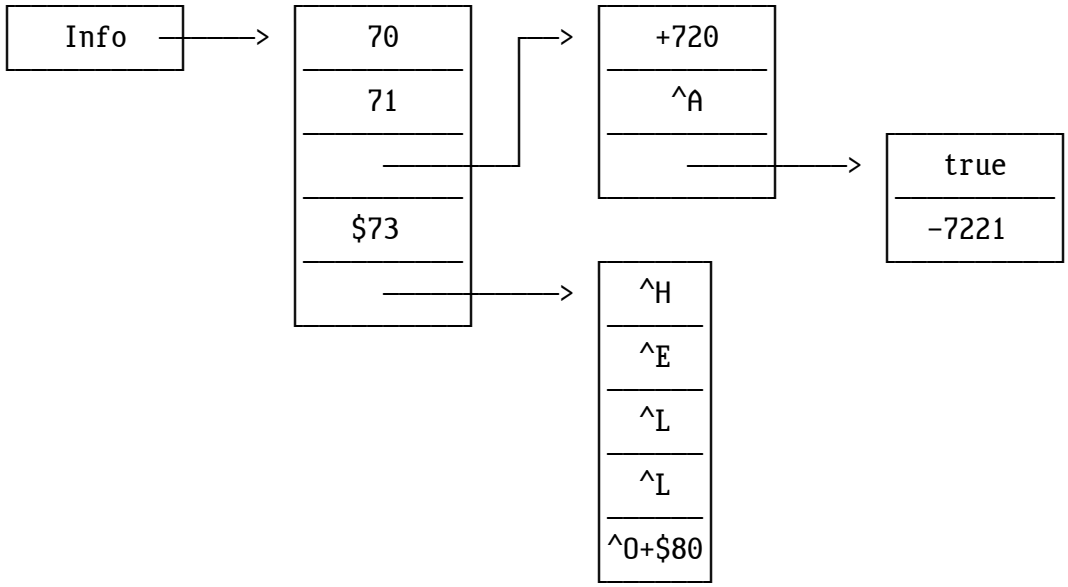
```
real Data;
begin
Data:= [[70.0, 70.1, 70.2, 70.3, 70.4],
        [71.0, 71.1, 71.2, 71.3, 71.4],
        [72.0, 72.1, 72.2, 72.3, 72.4]];
. . .
```

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" (-1). 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 better 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 is 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
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 to 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)

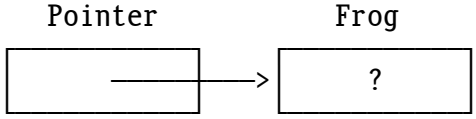
The "address" operator provides the address where a variable is stored. It has the form:

```
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 a subscripted array name. The "address" of a real variable is a 32-bit integer.

"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 32-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:

```
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 to 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 to 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. It uses a different kind of address operator to pass values back from a procedure. This program converts rectangular coordinates to polar coordinates, and it returns the two polar coordinates back to the calling procedure.

```
        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 that "@" is used instead of "addr". The "addr" operator doesn't work in this situation because it returns an integer address and what's needed here are pointers to the real variables Ang and Dist. A real pointer is a 32-bit address that's packaged in a 64-bit value so that it can be handled like a real. It's actually just a 32-bit integer with a second zero integer tacked on. The "@" works exactly the same way as "addr" on integer variables, but it returns a real pointer when used on real variables. For consistency, "@" is normally used instead of the "addr" operator.

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

```
            0.64350    5.00000
```

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, which call system routines in Linux.

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 screen, 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 storage 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 storage device (SD card).

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, carriage return, or linefeed. If the numbers come from device 3, it's 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 32-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 linefeeds 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 Screen	Buffered Keyboard
1	Monitor Screen	Unbuffered Keyboard
2	Printer *	--
3	File	File
4	Serial Port *	Serial Port *
5	Printer *	Printer Status
6	Monitor Screen	Unbuffered keyboard
7	Null	Null
8	Buffer	Buffer

* Not implemented.

6.0 DEVICE 0

Output device 0 is the monitor screen. It displays ASCII characters and handles certain control characters such as tab, form feed (clears the screen), bell, carriage return, linefeed, 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.

Input device 0 is a buffered keyboard. Characters are echoed on the monitor screen as they are typed in, but the buffer holds them until the "Enter" (Carriage Return) key is struck. This enables errors to be corrected using the "Backspace" key before the characters are sent to the program. The buffer holds up to 128 characters including the carriage return ($0D) at the end. Typing a Ctrl+C aborts the program.

Output and input can be redirected using the Linux commands ">" and "<" on the command line when starting a program. The "<" command is useful because it provides a simple way to make an input data file.

OpenI(0) initializes the keyboard, which discards any characters that were previously struck and still residing in any of its buffers. For example, it's a good idea to do an OpenI(0) before getting a reply to a critical question like: "Delete all files?". OpenO(0) and Close(0) do nothing.

6.1 DEVICE 1

Device 1 is identical to device 0 for output.

For input, keystrokes are not echoed on the monitor screen, although a flashing cursor is displayed (when in text modes but not graphic modes). 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. Typing a Ctrl+C aborts the program (unless TrapC(true) has been called).

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). F11 and F12 and Ctrl+Function keys are not available. Only the Alt values for Alt+A through Alt+Z are available. (The Alt+Function keys are used by Linux to switch terminals.) If the Pause key is struck, another key must be struck before ChIn(1) will return (with a zero).

ChIn(1) can get characters from an input file by typing "<" on the command line. This is not suitable for reading binary files because escape codes are converted to scan codes and a control-C will abort the program.

OpenI(1) discards any pending keystrokes.

6.2 DEVICE 2

Device 2 is the printer. (Not implemented.)

6.3 DEVICE 3

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

```
\Read an input file
FD:= FOpen("/path/filename.ext", 0);    \Get input file descriptor
FSet(FD, ^I);                           \Set device 3 to descriptor
OpenI(3);                               \Initialize input buffer
repeat until ChIn(3) = $1A;             \Read some characters
FClose(FD);                             \Release descriptor


\Write an output file
FD:= FOpen("/path/filename.ext", 1);    \Get output file descriptor
FSet(FD, ^o);                           \Set device 3 to descriptor
OpenO(3);                               \Initialize output buffer
for Ch:= $20 to $7E do ChOut(3, Ch);    \Write some characters
Close(3);                               \Flush output buffer
FClose(FD);                             \Release descriptor
```

FOpen opens a specified file and returns a "file descriptor", 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 a path name. If the path name is omitted, the current directory is used. If you output to device 3 without first opening a file, Linux sends the bytes to the monitor screen, and no error is detected.

FSet assigns the descriptor 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 their file pointers to the beginning of their files. Close(3) flushes any characters that might be remaining in the large output buffer and writes them to the storage device (SD card).

FClose flushes all internal buffers associated with the file descriptor. If the file was created or changed then its time, date, and size are updated in the Linux directory. When a program terminates, any open file descriptors are automatically closed.

END OF FILE

Character files may be terminated with a control-Z ($1A). This is merely a programming aid since the file-handling intrinsics pay no attention to control-Z's, which enables them to handle any kind of data files, such as binary 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. An easy way to do this is to use the 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)

When a program starts, any characters entered on the command line after the program name are copied into device 8's buffer. Any input or output file names are typed in after the program's name. 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    17-Dec-2016
\This copies a file, shifting all characters to lowercase.

int     FDIn, FDOut, Ch, I;
char    CmdLine($80);
begin
I:= 0;                          \Get copy of command line
repeat     Ch:= ChIn(8);
           CmdLine(I):= Ch;
           I:= I+1;
until Ch=\EOF\$1A;

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

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

FDOut:= FOpen(CmdLine+I, 1);    \Open second file name for output
FSet(FDOut, ^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(FDIn);
FClose(FDOut);
end;
```

A simpler version of this program takes advantage of Linux's ability to redirect I/O devices. This second version of lowcase is run like this:

```
lowcase <file1.txt >file2.txt

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. (Not implemented.)

6.5 DEVICE 5

Device 5 is the printer. (Not implemented.)

6.6 DEVICE 6

Output device 6 is similar to devices 0 and 1, but characters can be displayed in color and be confined to a window. Output cannot be redirected to a file using ">" on the command line.

The full IBM/OEM character set of the original IBM-PC is supported in three font sizes. A character's foreground and background color can be specified using the Attrib intrinsic (69). A window size and location can be defined using the SetWind intrinsic (70). Character positions aren't restricted to character-cell boundaries, such as set by the Cursor intrinsic, but instead can be set to any pixel using the Move intrinsic.

Device 6 normally uses an 8x16-pixel serif font, device $106 uses an 8x8 sans-serif font, and device $206 uses an 8x14 sans-serif font. If the video mode is set to a low resolution then device 6 uses a shorter font so that 25 lines fill the screen. For instance, if video mode $13 is set, which is 320x200, then device 6 uses an 8x8 font (200/8 = 25). This is consistent with the way the IBM-PC works.

Device 6 does not position characters at the location of the flashing cursor set by Linux before a program starts and by devices 0 and 1. Instead, it positions characters at the graphic pen position set by the Move intrinsic (43). Thus it's generally desirable to turn off the flashing cursor by calling ShowCursor(false) when device 6 is used. It's automatically turned off if SetVid (45) sets a graphic mode (consistent with the IBM-PC).

A particularly noticeable problem occurs if the flashing cursor is left on and the Cursor intrinsic is used. Old characters (often spaces) will be displayed at any location that Cursor moves to.

This table shows how the different devices handle control characters on the monitor screen:

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	x	x
6	-	-	-	x	-	x

An "x" means that the control function is done, while "-" means that a character is displayed instead. Device 6 displays all byte codes as characters (including $00-$1F and $7F-$FF) except $0A and $0D. ($00, $20 and $FF are displayed as space characters.)

A linefeed (LF $0A) moves to the beginning of a new line, consistent with the Linux convention. It's equivalent to the DOS/Windows convention of a carriage return (CR) plus a linefeed (LF).

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 Ctrl+C (or Ctrl+Break) does not abort the program; it's handled like any other keystroke. If a non-ASCII key is struck, such as an arrow key or F1, its Linux escape sequence is returned (and echoed); and its scan code is not available. Use ChIn(1) to handle these special keys.

OpenO(6) moves the graphic pen position to the upper-left corner of the screen and sets the attribute to white characters on a black background. It also resets any window set up by SetWind to the size of the full screen and enables normal scrolling and cursor movement.

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 file, but if it detects an error, it diverts the output to the null device.

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

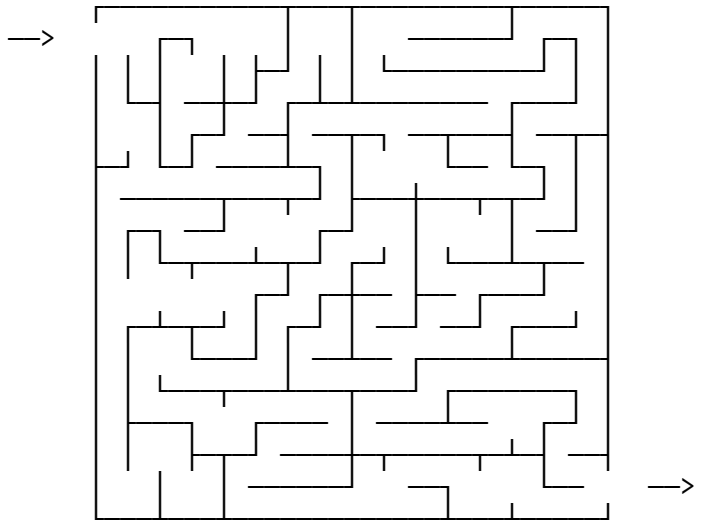
6.8 DEVICE 8

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 alphabetic order:

```
Abort = 16                              var:= Log(real) = 59
var:= Abs(int) = 0                      adr:= MAlloc(bytes) = 73
var:= ACos(real) = 63                   var:= Mod(real,real) = 58
var:= ASin(real) = 62                   MoveMouse = 78
var:= ATan2(realY,realX) = 57           Move(X,Y) = 43
Attrib(bg:fg) = 69                      OpenI(dev) = 13
BackUp = 83                             var:= OpenMouse = 85
var:= ChIn(dev) = 7                     OpenO(dev) = 14
var:= ChkKey = 33                       Paint(X,Y,W,H,image,W2) = 81
ChOut(dev,byte) = 8                     PlaySoundFile(pathname) = 103
var:= Cos(real) = 60                    Point(X,Y,color) = 41
Clear = 40                              var:= Ran(range) = 1
Close(dev) = 15                         RanSeed(int) = 79
CopyMem(dst,src,bytes) = 100            RawText(dev,str) = 71
CrLf(dev) = 9                           var:= ReadPix(X,Y) = 44
Cursor(X,Y) = 23                        Release(adr) = 74
DelayUS(int) = 94                       var:= Rem(expr) = 2
var:= Exp(real) = 55                    var:= Rerun = 19
var:= Extend(byte) = 5                  adr:= Reserve(bytes) = 3
FClose(hand) = 32                       Restart = 6
FillMem(adr,byte,bytes) = 101           var:= RlAbs(real) = 51
var:= Fix(real) = 50                    var:= RlIn(dev) = 47
var:= Float(int) = 49                   RlOut(dev,real) = 48
var:= FOpen(pathname,0=r/1=w) = 29      adr:= RlRes(int) = 46
Format(int,int) = 52                    SetFB(W,H,D) = 84
var:= Free = 18                         SetFont(height,adr) = 92
FSet(hand,^I/^O) = 24                   SetHP(adr) = 21
adr:= GetDateTime = 95                  SetPalette(reg,R,G,B) = 90
var:= GetErr = 22                       SetRun(bool) = 25
adr:= GetFB = 97                        SetVid(mode) = 45
adr:= GetFont(set) = 91                 SetWind(X0,Y0,X1,Y1,mode,fill) = 70
adr:= GetHP = 20                        ShowCursor(bool) = 88
var:= GetKey = 89                       ShowMouse(bool) = 77
adr:= GetMouse = 86                     ShowPage(0/1) = 99
adr:= GetMouseMove = 87                 var:= Sin(real) = 56
var:= GetShiftKeys = 93                 Sound(vol,dur,period) = 39
var:= GetTime = 82                      var:= Sqrt(real) = 53
var:= HexIn(dev) = 26                   var:= Swap(int) = 4
HexOut(dev,int) = 27                    var:= Tan(real) = 61
Hilight(X0,Y0,X1,Y1,bg:fg) = 72         var:= TestC = 76
InsertKey(byte) = 96                    Text(dev,str) = 12
var:= IntIn(dev) = 10                   Trap(bits) = 17
IntOut(dev,int) = 11                    TrapC(bool) = 75
Line(X,Y,color) = 42                    WaitForVSync = 98
var:= Ln(real) = 54
```

Intrinsics have been added over the years as they were needed. The result is that they 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 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 is one exception:

```
Abs(-2147483648) = -2147483648, or Abs($8000_0000) = $8000_0000
```

A faster way to get the absolute value is to use the "abs" command word. 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 the system time in microseconds.

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 faster, can be used instead of calling this 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 250 integers
```

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

Array space is normally reserved in the declaration of the array name. For example, assuming that Data is defined as "char", this does the same thing as above:

```
char Data(1000);
```

4: variable:= Swap(value);

This intrinsic returns the value obtained by swapping the low two bytes with each other and the high two bytes with each other. For example:

```
X:= Swap($12345678);    \X gets $34127856
```

The command word "swap", which is faster, can be used instead of calling this intrinsic.

5: variable:= Extend(value);

This intrinsic extends the sign bit of the low byte to a 32-bit integer. It's useful when fetching signed numbers from a character array. For example:

```
X:= Extend($FD);        \X gets $FFFF_FFFD (= -3)
X:= Extend(3);          \X gets $0000_0003 (= +3)
```

The command word "extend", which is faster, 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 deep and an error condition is detected that a high-level procedure must handle, it's simpler to restart the program than to pass the error indication back through the 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 screen
ChOut(3, $FF);          \Send $FF to the output file
```

9: CrLf(device);

This intrinsic sends a carriage return ($0D) and linefeed ($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 32-bit binary value. Integers are in the range: -2147483648 through 2147483647. 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, which returns 0) is read in. 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 32-bit binary value into ASCII digits. For example:

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

12: Text(device, address);

This intrinsic sends an ASCII text string to the specified output device. The beginning address of the string is passed. For example:

```
Text(0, "This is a string");
String:= "HELLO";
Text(0, String);        \Display HELLO on the screen
```

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 file
```

15: Close(device);

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

```
Close(3);               \Flush output buffer to the file
```

16: Abort;

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

17: Trap(bits);

This intrinsic determines which run-time errors abort the program and display error messages. The default is to trap (abort on) 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 *
        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) *

    * Not implemented
```

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 (2147483647).

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
```

19: boolean:= Rerun;

This intrinsic returns the value of the Rerun flag, which is either true or false. The Rerun flag is false when a program starts. It's set to "true" by the intrinsic Restart (6), and it can be set to "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: error:= GetErr;

This intrinsic returns the number of the most recently detected un-trapped 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 sent out appears at this location. X is the horizontal position, with 0 being the left column; and Y is the vertical position, with 0 being the top row. For example:

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

24: FSet(descriptor, mode);

This intrinsic assigns the file descriptor that is to be used by device 3. The "descriptor" is normally gotten from FOpen (29). The "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 are in the range $0000_0000 through $FFFF_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, thus a minus sign will be 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 are terminated by a non-hex character, such as a carriage return. This intrinsic also returns after reading eight hex digits, which enables continuous sequences of hex digits to be read 32-bits at a time.

27: HexOut(device, value);

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

```
HexOut(0, $a12);        \Displays: "00000A12" on the screen
```

29: descriptor:= FOpen("/path/filename.ext", mode);

This intrinsic opens a file and returns its descriptor. The file is specified by a string containing the file name and an optional path name. Note that file names are case-sensitive under Linux. No wild cards (* or ?) are allowed. Any extra space characters are ignored. The string must be less than 256 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 comma after the last character
- A space or control character (<=$20), such carriage return

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

FOpen is typically used with other intrinsics as shown here:

```
FD:= FOpen("/boot/config.txt", 0);
FSet(FD, ^I);
OpenI(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 first opening a file with FOpen, Linux sends them to the monitor screen, and no error is detected. (See: 6.3 Device 3).

32: FClose(descriptor);

This intrinsic closes a file descriptor. All internal buffers associated with the file are flushed, and the descriptor is released for possible reuse. If the file was modified, the time, date, and size are updated in the directory.

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

33: boolean:= ChkKey;

This intrinsic returns "true" if a key was struck on the keyboard.

39: Sound(volume, duration, period);

This intrinsic emits a constant tone from the speaker. "Volume" is zero for no sound and non-zero for full sound. "Duration" is the number of seconds times 20. "Period" is in microseconds, thus it's equal to one million divided by the desired frequency. This intrinsic can be used as an approximate time delay by setting "volume" to zero. The optimizing compiler (xx) must be used for the tone to be emitted, however, it will work as a delay when used with either compiler. For example:

```
Sound(1, 20, 3817);     \One second of Middle C (262 Hz)
```

40: Clear;

This intrinsic quickly clears the graphics screen and sets the pen position to the upper-left corner (0,0). Text mode screens should be cleared by sending a form feed like this: ChOut(0, $0C).

41: Point(X, Y, color);

This intrinsic draws 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 depending on the video mode set by SetVid (45) or by the arguments sent by SetFB (84). The way a color gets displayed depends on the depth of the mode, which is specified in bits.

Color bits:

8	Color is specified by a byte that selects one of 256 colors from a palette. The first eight colors are those listed for the Attrib intrinsic (69). For example, $0C selects bright red.
16	Color is specified by intensities of red, green and blue using this bit pattern: rrrr rggg gggb bbbb. For example, $F800 displays bright red.
24	Color is specified by intensities of red, green and blue using this byte pattern: RR GG BB. For example, $FF0000 displays bright red.
32	Color is specified the same as for 24-bit depth, but its transparency is determined by the highest byte. The pattern is: AA RR GG BB. AA specifies (alpha) transparency ranging from 0 being completely transparent (invisible) to $FF being completely opaque. For example $80FF0000 displays red with any pixels that are already on the screen partially showing through.

The default video mode (3) uses 16 bits (unlike the IBM-PC's 4 bits) to specify color.

When a mode with fewer than eight bits is selected, if bit seven of "color" is set then the low six bits of "color" are exclusive-ored with the pixel already on the screen. This provides a simple way to move an image over a background pattern by exclusive-oring its pixels with the background pixels, and then erasing the image and restoring the background by exclusive-oring the image a second time.

42: Line(X, Y, color);

This intrinsic draws a straight line from the last point drawn--or moved to with the Move intrinsic--to the specified X and Y coordinates. "Color" is the same as for Point (41), but the high byte can be used to specify various patterns of dotted and dashed lines.

Pixels are not drawn at locations corresponding to set bits in this high byte. For example, for video modes with eight or fewer bits of color, setting "color" to $7F01 draws a line with widely space dots. For video modes with more than eight bits of color the highest byte is used, thus the same dotted and dashed pattern is specified with "color" set to $7F0000A8 (which sets the blue intensity for 24-bit color to the same intensity as used for palette register 1). There is no dotted and dashed line capability for 32-bit color modes, which instead use the highest byte to specify transparency.

For example:

```
SetVid($101);                   \Set 640x480x8 graphics
Move(10, 50);                   \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 or the location where characters will be displayed by device 6. It sets the graphic pen position (penx, peny).

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 clears the screen and sets the cursor and pen positions to the upper-left corner (0,0). Any window set up by SetWind (70) is expanded to the full screen dimensions. Device 6 attribute colors and 8-bit graphic palette colors are reset to their defaults.

The CGA, EGA, VGA, and VESA modes defined by the IBM-PC are simulated:

Mode	Resolution	Colors	Type
$00	40x25	16	text
$01	40x25	16	text
$02	80x25	16	text
$03	80x25	16	text
$04	320x200	4	graphic
$05	320x200	4	graphic
$06	640x200	2	graphic
$07	80x25	2	text
$0D	320x200	16	graphic
$0E	640x200	16	graphic
$0F	640x350	2	graphic
$10	640x350	16	graphic
$11	640x480	2	graphic
$12	640x480	16	graphic
$13	320x200	256	graphic
$6A	800x600	16	graphic
$100	640x400	256	graphic

Additional graphic modes:

Color Bits	320x200	640x480	800x600	1024x768	1280x1024
4	$0D	$12	$6A/102	$104	$106
8	$13	$101	$103	$105	$107
15/16	$10D	$110	$113	$116	$119
16	$10E	$111	$114	$117	$11A
24	$10F	$112	$115	$118	$11B

The number of colors and the number of color bits shown are the original specification. All modes actually have at least seven color bits and thus can display at least 128 colors. Those modes with fewer than eight color bits use the most significant bit to specify exclusive-oring, as described for the Point intrinsic (41). The modes that specify 15 color bits are actually displayed in the 16-bit format, which is described in Point (41). Modes not listed are set to 640x480 graphics with eight color bits.

Text modes $02 and $03 actually display the resolution and color depth set by the operating system. This is normally much greater than the 80 columns by 25 rows defined by the IBM-PC, and 16-bit color is normally used. Characters can be displayed in graphic as well as text modes, but the flashing cursor is turned off for graphic modes. Points and lines can be drawn in text as well as graphic modes (unlike the IBM-PC).

The Raspberry Pi always displays square pixels, so, for example, mode $06, which is 640x200x2, does not make the pixels taller to fill the screen.

Some versions of Raspbian have the red and blue colors reversed for 24-bit graphic modes. This can be corrected by adding framebuffer_swap=1 to /boot/config.txt.

Here's an example of a graphic program that plots a sine wave:

```
int X;
begin
SetVid($101);                           \640x480 with 256 colors
Move(320, 0);   Line(320, 479, 1);      \Draw axes in blue
Move(0, 240);   Line(639, 240, 1);
for X:= 0 to 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;
```

46: real variable:= RlRes(integer);

This intrinsic reserves space for real arrays. RlRes(3) reserves enough memory to hold three real numbers. For example:

```
real Array;
Array:= RlRes(3);      \Reserve elements 0 through 2
```

Array space is normally reserved in the declaration of the array name. For example, this does the same thing as above:

```
real Array(3);
```

47: real variable:= RlIn(device);

This intrinsic gets a real number from the specified input device. It converts the number from its ASCII digits into binary form. After the 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, which returns 0.0) is read in, 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 sends a real value to the specified output device. It converts the real value 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 digits shown after the decimal point can be specified by the Format intrinsic (52).

49: real variable:= Float(integer);

This intrinsic converts an integer value to its equivalent real number. (See: 2.1 Mixed Mode.) For example:

```
RlOut(0, (Float(35));   \Display "35.00000"
```

The command word "float" does the same thing but is faster.

50: integer:= Fix(real);

This intrinsic rounds a real value to its nearest integer. (See: 2.1 Mixed Mode.) For example:

```
IntOut(0, Fix(13.002)); \Display "13"
```

Converting a value outside the range -2147483648.0 through 2147483647.0 causes a fix overflow run-time error.

The command word "fix" (lowercase) does almost the same thing, and it's several times faster. The only difference is that it does not abort with a run-time error if the argument is out of range. Instead, it returns the closest possible signed integer, either -2147483648 or 2147483647.

51: real variable:= RlAbs(real);

This intrinsic takes the absolute value of a real number. For example:

```
X:= RlAbs(X);           \Remove the minus sign from X
```

The command word "abs" does the same thing but is faster. Also, this (lowercase) "abs" works for both reals and integers.

52: Format(integer, integer);

This intrinsic specifies the format of real numbers that RlOut (48) sends to an output device. The first integer is the number of places before the decimal point, including a possible minus sign; and the second integer specifies the number of places after the decimal point. If the first integer is 0 then scientific notation is used. If the first integer is -1 (or any negative value) then engineering notation is used.

One purpose of Format is to align decimal points. However, if the value is too large to fit in the designated places, all 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 digits specified after the decimal point is 0 then a decimal point is not sent out. Truncated values round to the number of displayed digits. For example:

```
define  A = 12345.67;
begin
RlOut(0, A); CrLf(0);   \sends 12345.67000
RlOut(0, -A); CrLf(0);  \sends -12345.67000
Format(6, 2);
RlOut(0, A); CrLf(0);   \sends  12345.67
RlOut(0, -A); CrLf(0);  \sends -12345.67
RlOut(0, 3.14); CrLf(0);\sends      3.14
Format(0, 1);
RlOut(0, A); CrLf(0);   \sends  1.2E+004
RlOut(0, -A); CrLf(0);  \sends -1.2E+004
Format(-1, 4);
RlOut(0, A); CrLf(0);   \sends   12.3457E+003
RlOut(0, -A); CrLf(0);  \sends  -12.3457E+003
Format(6, 0);
RlOut(0, A); CrLf(0);   \sends  12346
RlOut(0, -A); CrLf(0);  \sends -12346
end;
```

53: real variable:= 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
```

The command word "sqrt" (lowercase) does almost the same thing and is faster, and it works for both reals and integers. The only difference is that it does not abort with a run-time error if the argument is negative. If the argument is a negative integer, it returns 0. If the argument is a negative real, it returns a special value called "not a number" (NAN).

54: real variable:= Ln(real);

This intrinsic returns the natural logarithm (base e) of the argument. The argument should be > 0.0, otherwise the returned value is either infinite (INF) or not a number (NAN).

55: real variable:= Exp(real);

This intrinsic computes the exponential function (e^X). This is the inverse operation of Ln (54). For example, RlOut(0, Exp(Ln(123.0))) displays 123.00000. If the argument is larger than 709.0 then INF is returned.

56: real variable:= Sin(real);

This intrinsic computes the sine function. All trig functions use angles represented in radians. To convert from radians to degrees, multiply by 180.0/pi (approximately 57.2957795) degrees/radian. To convert degrees to radians, divide by 180.0/pi. For example:

```
X:= Sin(30.0/57.2957795); \Sine of 30 degrees (=0.5)
```

It may seem surprising that the area under the first hump of a sine curve is exactly equal to two given that one of the dimensions is an irrational number. This little program more or less confirms it:

```
real S, X;
def  DX=0.001;                  \  ∫π
def  Pi=3.141592653589793;      \   sin(x) dx = 2
[S:= 0.0;  X:= 0.0;             \  ∫0
repeat  S:= S + Sin(X)*DX;
        X:= X + DX;
until   X >= Pi;
RlOut(0, S);
]
```

57: real variable:= 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 set to 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 variable:= 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(7.6, 2.5);      \X:= 0.1
X:= Mod(123.456, 1.0);  \Get the fractional part (0.456)
```

59: real variable:= 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 variable:= Cos(real);

This intrinsic computes the cosine function. The argument is the angle in radians. See Sin (56).

61: real variable:= Tan(real);

This intrinsic computes the tangent function. The argument is the angle in radians. See Sin (56).

62: real variable:= ASin(real);

This intrinsic computes the arc-sine function in radians. The argument should be in the range ±1.0, otherwise "not a number" (NAN) is returned.

```
Ang:= ASin(X);          \X: -1.0 to +1.0; Ang: -pi/2 to +pi/2
```

63: real variable:= ACos(real);

This intrinsic computes the arc-cosine function in radians. The argument should be in the range ±1.0, otherwise "not a number" (NAN) is returned.

```
Ang:= ACos(X);          \X: -1.0 to +1.0; Ang: 0 to pi
```

69: Attrib(colors);

This intrinsic specifies the colors used when sending characters to device 6. (Devices 0 and 1 always display white characters on a black background. The term "attribute" comes from the IBM-PC.) Normally, the high nibble of the argument sets the background color, and the low nibble sets the foreground color. The colors for the nibble values are listed below. For example, Attrib($1F) would display bright white characters on a blue background.

$0: Black	$8: Gray
$1: Blue	$9: Light Blue
$2: Green	$A: Light Green
$3: Cyan	$B: Light Cyan
$4: Red	$C: Light Red
$5: Magenta	$D: Light Magenta
$6: Brown	$E: Yellow
$7: White	$F: Bright White

If the background color is the same as the foreground color then the background color is not written to the screen. This makes the background transparent (and makes characters draw faster). This also means that if a character is written on top of an existing character, it will not completely replace the existing character as is done with normal text. This can be used to show struck-out characters.

The above describes how the Attrib intrinsic works for the default 16-bit color video mode and for modes with fewer than eight color bits. The modes described below all have eight or more color bits, and they don't have the option of making the background transparent. If the background and foreground colors are the same then any character output to device 6 will display a solid colored block and the character will not be visible.

For 8-bit color modes, the low byte specifies the foreground color, and the next higher byte specifies the background color. For example, Attrib($010F) would display bright white characters on a blue background.

For 24-bit color modes, only the foreground color can be specified, and the background is always black. For example, Attrib($FFFFFF) would display bright white characters on a black background.

For 32-bit color modes, only the foreground color can be specified, but transparency can also be specified. The highest byte specifies the amount of (alpha) transparency. It ranges from 0 being completely transparent (invisible) to $FF being completely opaque. For example, Attrib ($80FFFFFF) would display half-bright white characters with any background pixels showing through at half intensity.

Video modes with fewer than eight color bits have one more trick. If the background and foreground colors are the same then the character is written by exclusive-oring the color with what's already on the screen, and the background color is not written. This enables characters to be drawn on top of an existing pattern; and if the character is redrawn in the same location, it gets exclusive-ored a second time, which erases the character and restores the original pattern.

70: SetWind(X0, Y0, X1, Y1, mode, fill);

This intrinsic specifies the window used when sending characters 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 operates.

> 0 = Scroll: When the cursor position moves beyond the right edge of the window, it jumps to the beginning of the next line down. When the cursor position moves beyond the bottom of the window, the text in the window scrolls up and the cursor position jumps to the beginning of the bottom line. (Writing to the bottom-rightmost character cell scrolls the text.)
>
> 1 = Wrap: When the cursor position moves beyond the right edge of the window, it wraps to the beginning of the current line. When the cursor position moves beyond the bottom of the window (with a linefeed, which is the same as a "newline" in Linux), the cursor position wraps to the beginning of the top line.
>
> 2 = Clip: When the cursor position moves beyond the right edge or beyond the bottom of the window then characters are clipped and don't appear.

If the "fill" flag is "true", the window is erased by filling it with the background color specified by the Attrib intrinsic (69). If the "fill" flag is "false", the window is set up without changing any characters already on the screen.

Opening device 6 for output with OpenO(6) resets the window to the full screen size and enables normal scroll mode. No text is erased.

Note: 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 displayed. The terminating space character is not sent out. For example:

```
RawText(6, "╔═══════╗ ");
```

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(6, "╔═══════╗");
RawText(6, "╔═══════╗ ");       \displays terminating space
```

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. These are character coordinates like used with the Cursor intrinsic (23), not graphic coordinates like used with the Move intrinsic (43). "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 then any characters in the specified rectangle will be blotted out (there is no exclusive-or feature).

73: address:= MAlloc(bytes);

This intrinsic returns the starting address of a block of memory. The number of bytes in the block are specified by the argument.

Unlike the Reserve intrinsic (3), MAlloc does not automatically release memory when a procedure returns. If MAlloc is called in a procedure and the procedure is repeatedly executed, more memory is 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. If you write beyond the end of the allocated space, Linux may abort the program with a segmentation fault.

74: Release(address);

This intrinsic deallocates a block of memory that was allocated by MAlloc. The address of the block is passed to indicate which block to deallocate. For example:

```
proc    Demo;
char    Image;
begin
Image:= MAlloc($100000);        \1 megabyte
. . .
Release(Image);
end;
```

Allocated memory is automatically released when a program terminates, so it's unnecessary to release memory allocated in the main procedure.

If a block of memory is released using an address not returned by MAlloc, Linux will likely abort the program with a segmentation fault.

75: TrapC(boolean);

This intrinsic turns control-C trapping on and off. "True" is passed to turn on control-C trapping, which prevents the Ctrl+C key from aborting a program. Control-C trapping is normally off. Any change to the way control-C is handled is restored when a program terminates.

76: boolean:= TestC;

When control-C trapping is on, this intrinsic is used to determine if the Ctrl+C key has been struck. If it has then TestC returns "true".

Each time the Ctrl+C 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".

A control-C cannot be detected until it's read in from the keyboard. However, since the buffered keyboard (device 0) reads keystrokes before they are read in by a program (because the Enter key has not yet been struck) TestC can detect a control-C before the program reads all the characters from the buffer.

77: ShowMouse(boolean);

This intrinsic turns the display of the mouse pointer on and off. The mouse pointer defaults to being off.

78: MoveMouse;

This intrinsic moves the displayed mouse pointer as needed to track the position of the mouse.

79: RanSeed(integer);

This intrinsic sets the seed used by the Ran intrinsic's random number generator (1). This provides millions of different, repeatable random number sequences.

81: Paint(X, Y, W, H, Image, W2);

This intrinsic quickly copies an image to display memory, which is useful for animations. X, Y are the coordinates where the upper-left corner of the image data will be displayed on the screen. W, H are the width and height (in pixels) of the displayed image data. "Image" is the address of the image data array. W2 is the actual width (in pixels) of the image data array.

For 8-bit color modes ($13, $101, etc.) the image array is reserved as a byte array, for example: char Image(640*480).

For 16-bit color modes ($111, $114, etc.) the size of the image array must be doubled, for example: char Image(640*480*2). A pair of bytes must be combined, and the colors must be set in the resulting 16-bits like this:

```
bit:   $F E D C B A 9 8  7 6 5 4 3 2 1 0
color:  r r r r r g g g  g g g b b b b b
```

For the 24-bit color modes ($112, $115, etc.) Image is reserved as an integer array, for example: int Image(640*480). The order of the colors in the 4-byte integer is: $IIRRGGBB, where II is the intensity, or brightness, ranging from 0 (black) to $FF (full brightness). This color arrangement is the same even if SetFB (84) sets the color depth to 32 bits.

WARNING: The brightness feature does not work in some distributions of the operating system. To enable it, remove "framebuffer_ignore_alpha=1" in /boot/config.txt.

82: time:= GetTime;

This intrinsic returns the current time in microseconds. The unsigned 32-bit integer rolls over about every hour and 11 minutes.

GetTime is often used to measure the duration between two events. If this duration is less than half an hour, you don't need to be concerned about the rollover. The duration is simply the time of the second event minus the time of the first event. Any rollovers are automatically handled.

Two consecutive calls to GetTime on the slowest Raspberry Pi (RPi-1 @ 700 MHz) takes less than 4 microseconds.

```
int T0, I;
[T0:= GetTime;          \How long for a billion?
for I:= 1 to 1_000_000_000 do \nothing\;
RlOut(0, float(GetTime-T0) / 1e6);
Text(0, " seconds^J")]
```

83: BackUp;

This intrinsic enables the last byte read from any input device to be reread (like C's ungetc function). It's handy, for instance, in the situation where a user can step through a series of numbers with the Enter key (or Tab key) and change a number merely by typing its new value. In the example below BackUp enables IntIn to be used if the user typed a digit.

```
int Number, Digit;
begin
Number:= 123;                   \default value
IntOut(0, Number);  CrLf(0);
Digit:= ChIn(0);
if Digit>=^0 & Digit<=^9 then   \change it
        [BackUp;  Number:= IntIn(0)];
IntOut(0, Number);  CrLf(0);    \confirm result
end;
```

84: SetFB(width, height, depth);

This intrinsic sets the displayed frame buffer to a specified width, height, and depth. Widths should be multiples of 32 pixels. Heights can be almost anything, but pixels are always square rather than being stretched to fill the screen. Depths can be 8, 16, 24, or 32 bits. A depth of 8 uses a palette for colors (like the PC's VGA mode $13). Depths of 16 and 24 are like the PC's high color and true color modes. A depth of 32 provides a byte that specifies (alpha) transparency that ranges from 0 being completely transparent (invisible) to $FF being completely opaque.

Since calling this intrinsic is regarded as setting a graphic mode, the flashing cursor is turned off. It can be turned on if desired with ShowCursor (88).

85: boolean:= OpenMouse;

This intrinsic initializes the mouse and sets its position to the center of the screen, although its pointer remains hidden. This intrinsic returns "false" if an error is detected. If a mouse is not connected, this does not detect its absence. Since the mouse is now opened automatically, this intrinsic is no longer needed.

86: address:= GetMouse;

This intrinsic returns the address of an integer array that contains the state of the mouse. The integers are:

```
0: X position (from left edge of screen, in pixels)
1: Y position (down from top of screen, in pixels)
2: buttons: bit 0 set = left button down
            bit 1 set = right button down
            bit 2 set = middle button down
```

For example, if Address(2) = 3, it means that both the left and right buttons are currently being pressed.

87: address:= GetMouseMove;

This intrinsic returns the address of an integer array that contains the distance that the mouse moved (in pixels) since the previous call to this intrinsic. The integers are:

```
0: change in X position (in pixels)
1: change in Y position (in pixels)
```

88: ShowCursor(boolean);

This intrinsic turns the flashing cursor off or on. The flashing cursor defaults to being on for text display modes and off for graphic display modes. If a program turns it off, it normally should be turned back on when the program terminates. For example:

```
ShowCursor(false);      \disable the flashing cursor
```

89: character:= GetKey;

This intrinsic has become mostly obsolete because its features have been added to ChIn(1), which should be used in new code. This intrinsic returns the ASCII value of a character struck on the keyboard. It's similar to ChIn(1) but handles non-ASCII keys a different way. Linux's read(stdin) returns escape sequences for the function keys, arrow keys, and even for the Esc key itself. Since this can complicate things, this intrinsic converts these non-ASCII keys to the negative value of their scan codes (which are listed in appendix A.4). F11 and F12 and Ctrl+ Function keys are not available. Only the Alt values for Alt+A through Alt+Z are available. (The Alt+Function keys are used by Linux to switch terminals.) This intrinsic does not echo characters to the screen, and Ctrl+C does not abort the program. If the Pause key is struck, another key must be struck before this will return (with a zero).

90: SetPalette(register, red, green, blue);

This intrinsic changes a color in the 256-color palette that's used with 8-bit depth and lower graphics. The red, green and blue values range from 0 through 255, but only the high six bits are actually used (just like VGA on the PC). An 8-bit or lower graphic mode must already be set up with either intrinsic SetFB (84) or SetVid (45).

Unfortunately SetPalette has a couple bizarre problems (due to Raspbian's frame buffer driver): color register 15 or 255 must be set to make any register actually change, and (incredibly) the frame buffer memory might be zeroed (erased) as a side effect. This latter problem was fixed in the September 8, 2014 version of Raspbian. (The Linux command "uname -a" displays the version and date.)

91: address:= GetFont(face);

This intrinsic is used to return the address of a 256-character font table. If face = 0 then address points to a 8x16 table, which has 16 bytes per character. If face = 1 then the address of the 8x8 table is returned, and if face = 2 the address of the 8x14 table is returned. One use for accessing these font tables is to make banner programs with giant letters.

92: SetFont(height, address);

This intrinsic changes the character font table used by device 6. Height is the number of bytes per character, which is also the height of a character cell in pixels. Character cells are always 8 pixels wide. Address is the location of the replacement table. There is no way to change the font used by devices $106 and $206.

93: bits:= GetShiftKeys;

This intrinsic returns the current state of some keyboard keys. If bits = 1 then a left or right Shift key is held down. If bits = 4 then a Ctrl key is down, and if bits = 8 then an Alt key is down. If, for example, bits = $D then all three keys are currently held down. Bit 5 = NumLock.

94: DelayUS(duration);

This intrinsic delays "duration" microseconds. For example, DelayUS (54945) would delay about 1/18 of a second, which is the duration of a IBM-PC system clock tick.

95: address:= GetDateTime;

This intrinsic returns the address of a byte (char) array containing the current system date and time. The bytes are:

```
0: year since 1900
1: month (1..12)
2: day (1..31)
3: hour (0..23)
4: minute (0..59)
5: second (0..59)
6: hundredths of seconds (0..99)
7: day of week (0=Sun, 1=Mon, ... 6=Sat)
```

96: InsertKey(character);

This intrinsic inserts a character into the keyboard's input buffer. The next key read from ChIn(1) will be the inserted character. Several characters can be inserted before being read out in sequence by ChIn(1). Keys can also be read out with ChIn(0), but there may already be other characters ahead of the inserted keys in its buffer. This intrinsic is useful for GUI buttons and menu items that have shortcut keys.

97: address:= GetFB;

This intrinsic returns the address of an integer array that contains information about the currently displayed frame buffer. The integers are:

```
0: width (in pixels)
1: height (in pixels)
2: depth (in bits per pixel, e.g: 8, 16, 32)
3: frame buffer's memory address
4: graphic pen horizontal position (penx)
5: graphic pen vertical position (peny)
```

98: WaitForVSync;

This intrinsic waits for the vertical sync signal, which occurs every 1/60th of a second at the beginning of each video frame. It's useful for animations. This signal is available in versions of Raspbian on and after September 8, 2014.

99: ShowPage(page);

This intrinsic sets the page of the video memory that gets displayed. There are two pages: 0 and 1. Page 1 immediately follows page 0 in memory. Images can be built up out of sight on page 1, while page 0 is being displayed, by adding an offset to the Y coordinate equal to the screen height. For example: Point(X, Y+480, $0F). This intrinsic works in versions of Raspbian on and after September 8, 2014.

100: CopyMem(dst, src, size);

This intrinsic copies an array of bytes from the address in "src" to the address in "dst". It's equivalent to, but many times faster than: for I:= 0 to Size-1 do Dst(I):= Src(I); It also works if the source and destination areas overlap.

101: FillMem(address, value, size);

This intrinsic fills an array of bytes. It's equivalent to, but many times faster than: for I:= 0 to Size-1 do Address(I):= Value;

103: PlaySoundFile("/path/filename.ext");

This intrinsic starts playing a sound file. The file extension can be: wav, raw, voc or au. The xpl program continues to run while the sound file is playing.

A . 1 : C O M P I L E E R R O R S

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 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 mark. 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 the compile error messages are listed below along with suggestions on how to avoid them.

1: TOO MANY VARIABLES. This is caused by a variable being declared after arrays that total more than 16 megabytes of space. The optimizing compiler (xx) does not have this limitation, nor do arrays that are set up using the Reserve intrinsic (3). When using the non-optimizing compiler (x), the problem might be avoided by declaring large arrays after other (scalar) variables. Passing more than 64 integers or 32 reals to a procedure also causes this error.

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 1600. 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 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 "loop"s that are nested inside any 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 -2147483648 through +2147483647.

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 because 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 or be declared as a "real".

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 with 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 (or an underline).

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 or an array or an array element.

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 (or an underline).

46: MIXED MODE. Reals and integers cannot be mixed within an expression without explicitly doing the type conversions using the intrinsics Fix and Float (or the command words fix and float). This message can occur if a variable is undefined. Also, a forward-function declaration and its function must be the same data type ("integer" or "real").

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.

54: ILLEGAL INTRINSIC CALL. Perhaps the wrong number of arguments are being passed. Is an integer value being passed instead of a real value, or vice versa? Perhaps the intrinsic is returning a value that is not used, or vice versa.

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 quote mark (")? A caret (^) can cause a quote mark to not be seen (for example: ^").

65: 'FPROC' & ITS 'PROC' NOT AT SAME LEVEL. A forward procedure declaration and its corresponding procedure declaration must be at the same static nesting level and must be in scope with each other.

66: 'FPROC' REFERENCE NOT FOUND. Unresolved forward procedure or forward 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 procedure).

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: /path/filename.ext; Everything but the file name is optional. The semicolon is required.

71: FILE NOT FOUND. Perhaps the file has the wrong case letters or it's 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. The script for the optimizing compiler (xx) uses short-circuit booleans.

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 aborting for selected run-time errors.

These are the possible run-time error messages:

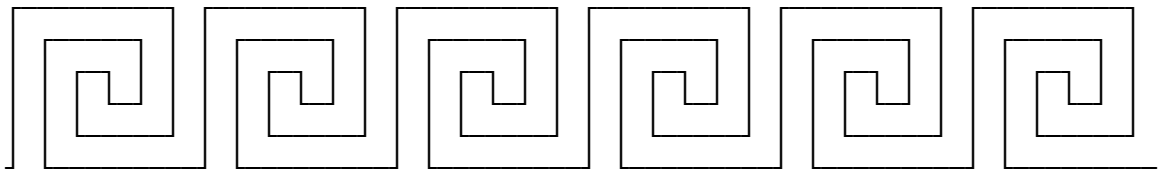
1: DIV BY 0. Division by zero for an integer. If this is untrapped, 2147483647 is returned for the quotient and 0 is returned for the remainder.

2: OUT OF MEMORY. Memory space not available. An array declaration or a Reserve tried to exceed the allotted heap memory space of 64 megabytes. This error can also be caused by MAlloc (73) if the operating system cannot provide the requested amount of memory.

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. File names in Linux are case-sensitive. Perhaps a device number in an intrinsic call is missing.

9: FIX OVERFLOW. Fixed-point overflow. Attempted to Fix (50) too large or too small a value (either greater than 2147483647.0 or less than -2147483648.0). If untrapped, it returns the closest possible integer, either 2147483647 or -2147483648.

10: SQRT < 0. Square-root error. Attempted to take the square root of a negative value. If untrapped, "not a number" (NAN) is returned.



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 newcomers 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 can cause a stack imbalance that results in a segmentation fault.

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 in a long program. 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

OPERATORS

The operator precedence (priority) is shown in parentheses; 1 is highest.

SPECIAL CHARACTERS

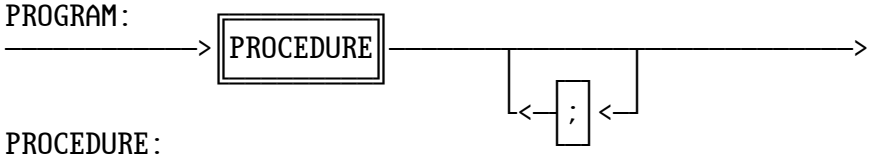
STATEMENTS

```
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 to 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
```

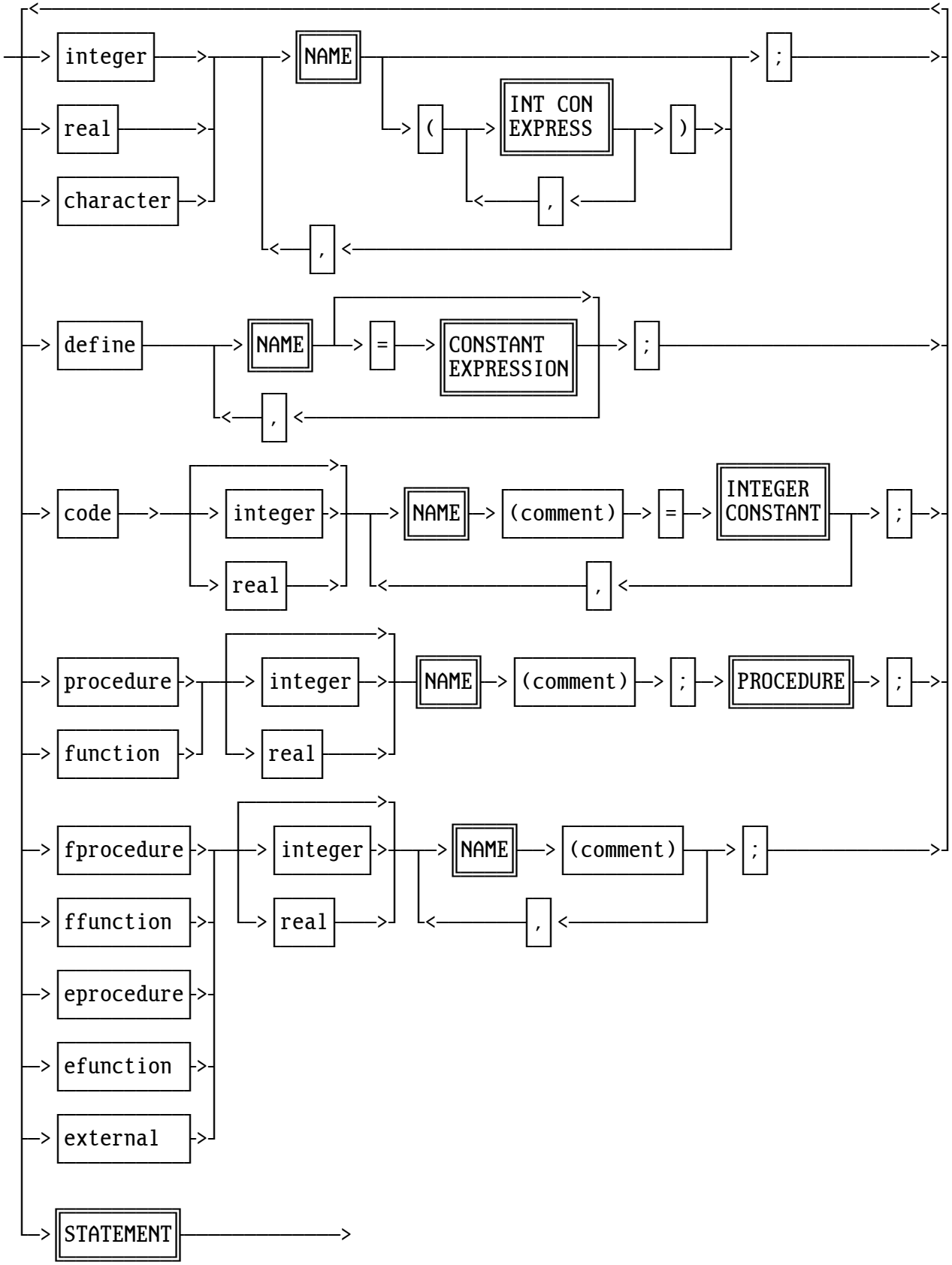
DECLARATIONS

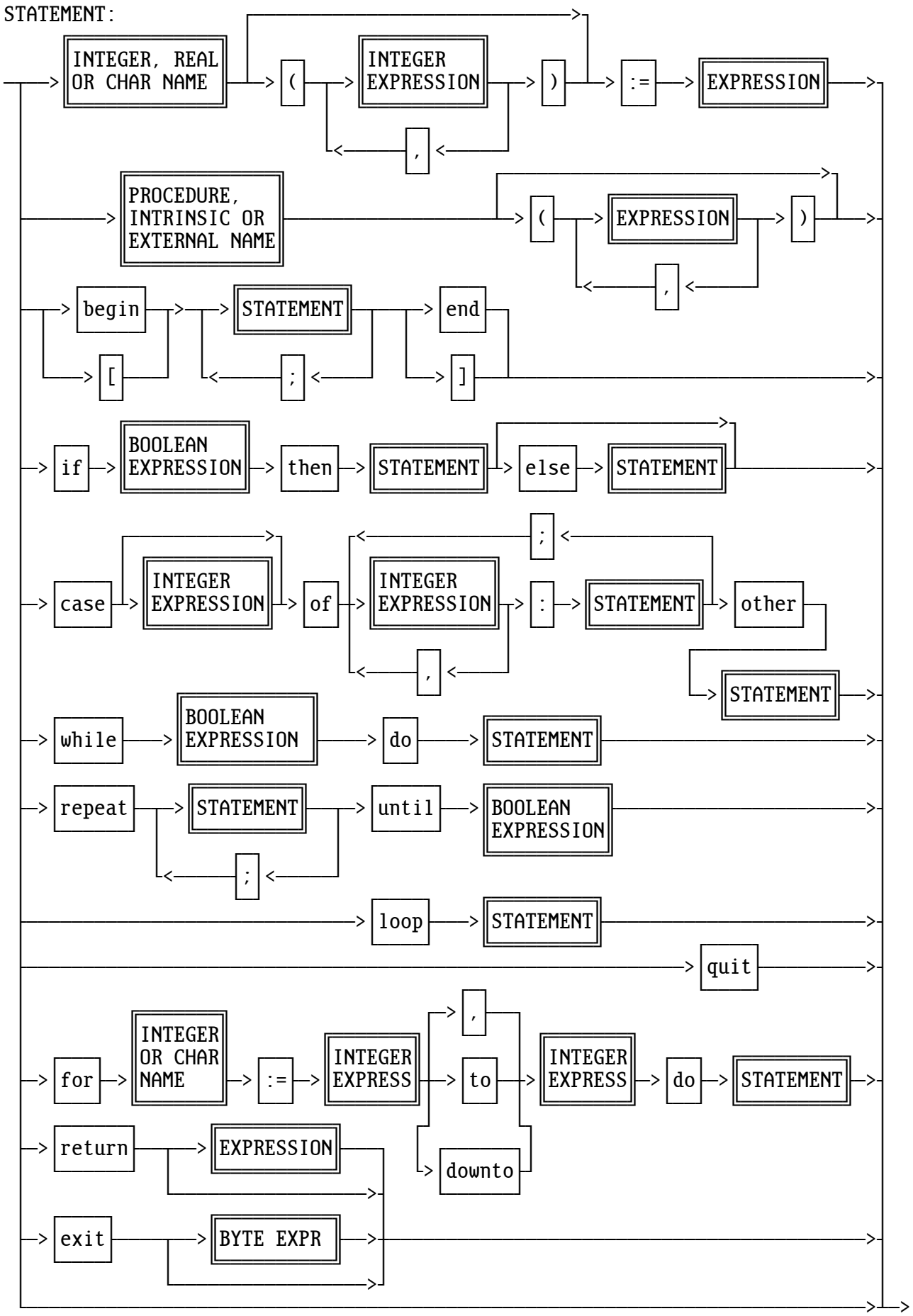
```
integer NAME, NAME, ... NAME;  . . . . . 1.5
real NAME, NAME, ... NAME;  . . . . . 1.5
define NAME=CONSTANT, ... NAME=CONSTANT;  . . . . . 1.6
define NAME, NAME, ... NAME;  . . . . . 1.6
procedure NAME(COMMENT);  . . . . . 4.0
function TYPE NAME(COMMENT);  . . . . . 4.5
code TYPE NAME(COMMENT)=INTEGER, ... NAME(COMMENT)=INTEGER;  . 4.6
fprocedure NAME(COMMENT), NAME(COMMENT), ... NAME(COMMENT);  . 4.9
ffunction TYPE NAME(COMMENT), NAME(COMMENT), ... NAME(COMMENT); 4.10
integer NAME(DIMENSIONS), ... NAME(DIMENSIONS);  . . . . . 5
real NAME(DIMENSIONS), ... NAME(DIMENSIONS);  . . . . . 5
character NAME(DIMENSIONS), ... NAME(DIMENSIONS);  . . . . 5
```

PROGRAM:

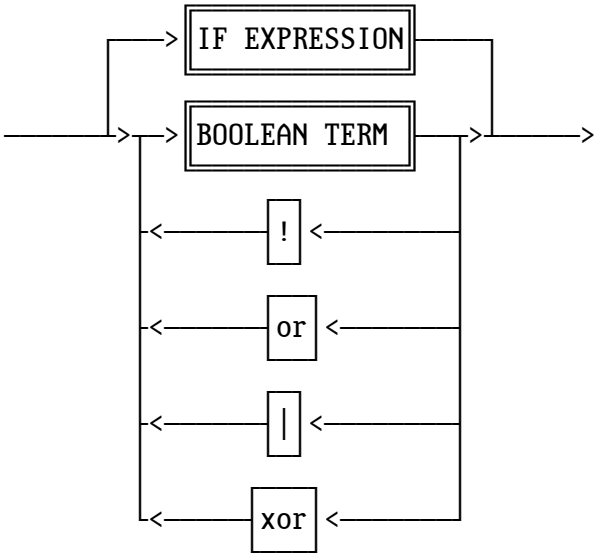


PROCEDURE:

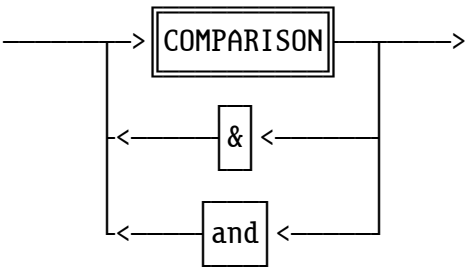



STATEMENT:
INTEGER, REAL OR CHAR NAME
(
INTEGER EXPRESSION
,
)
:=
EXPRESSION
PROCEDURE, INTRINSIC OR EXTERNAL NAME
(
EXPRESSION
,
)
begin
[
STATEMENT
;
end
]
if
BOOLEAN EXPRESSION
then
STATEMENT
else
STATEMENT
case
INTEGER EXPRESSION
of
;
INTEGER EXPRESSION
,
:
STATEMENT
other
STATEMENT
while
BOOLEAN 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:

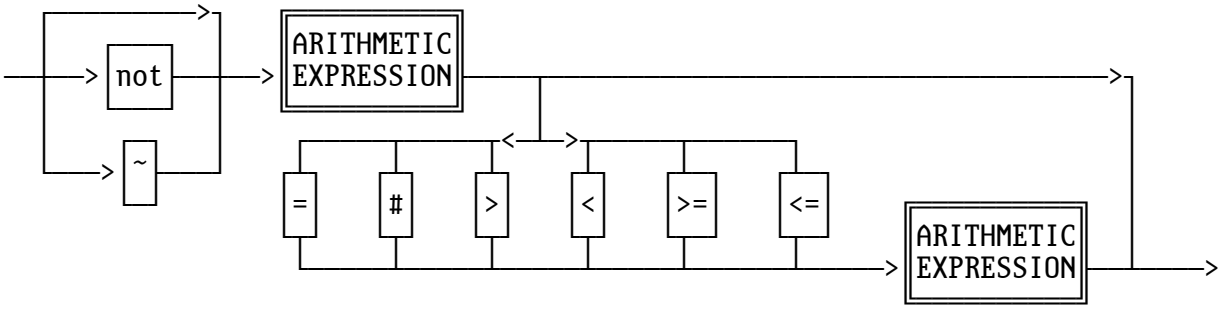


BOOLEAN TERM:

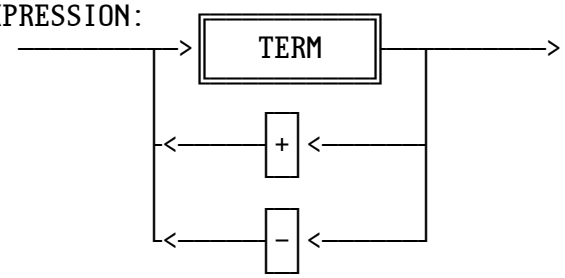


IF EXPRESSION:

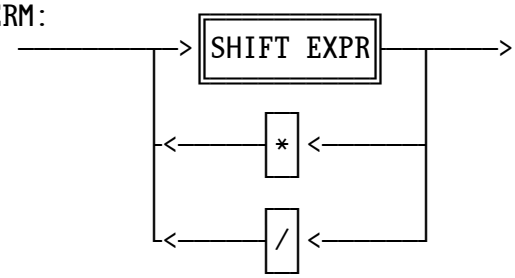
COMPARISON:



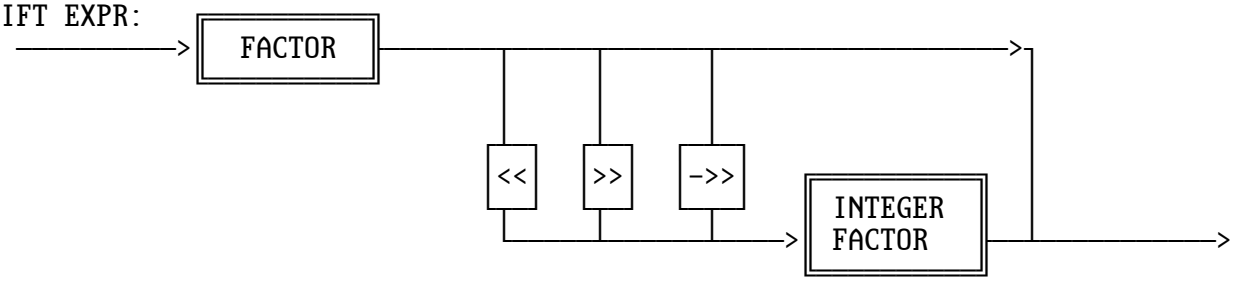
ARITHMETIC EXPRESSION:



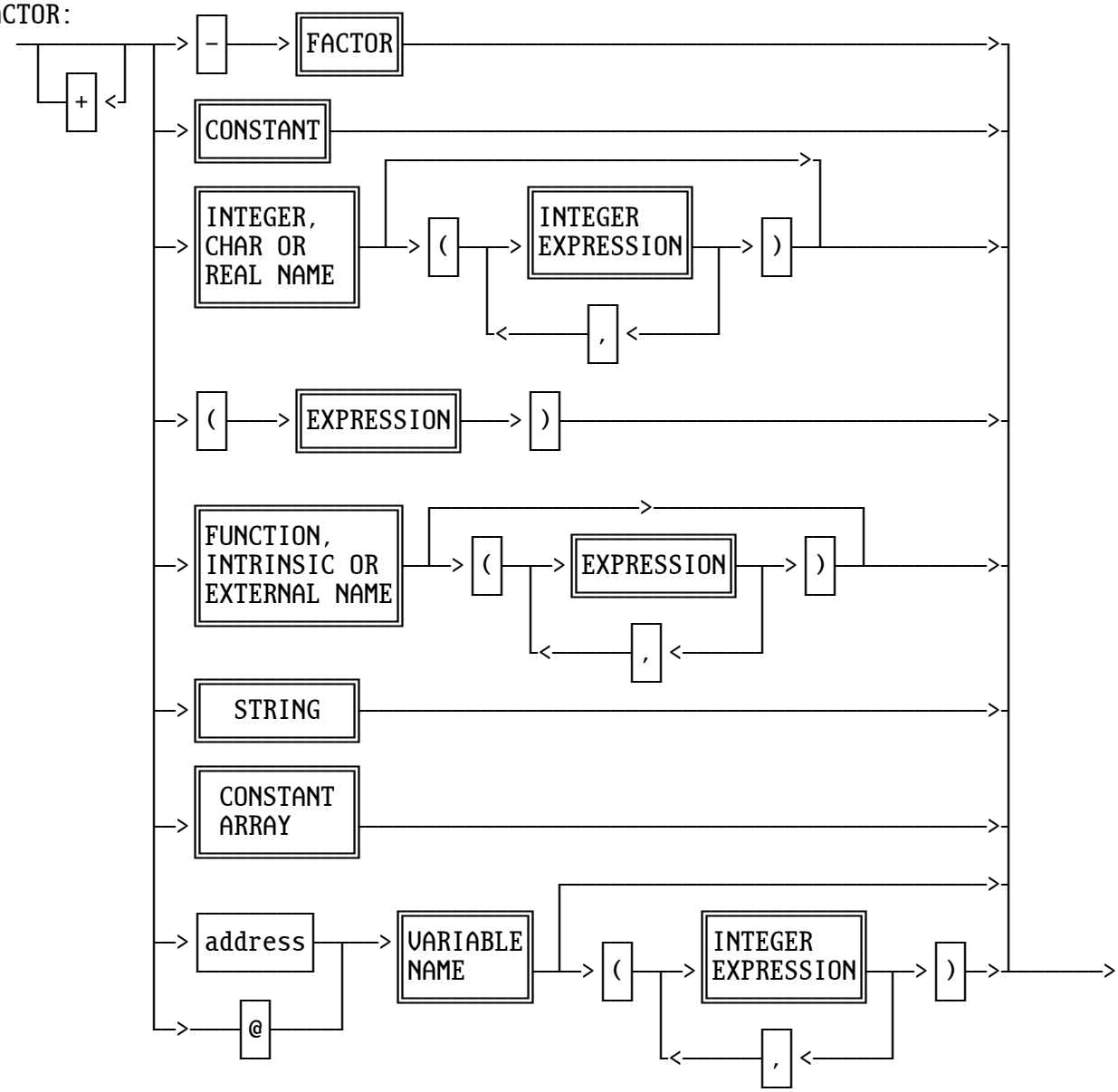
TERM:



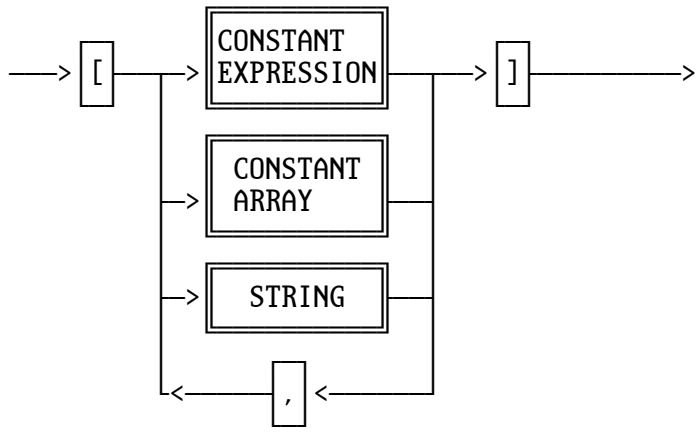
SHIFT EXPR:



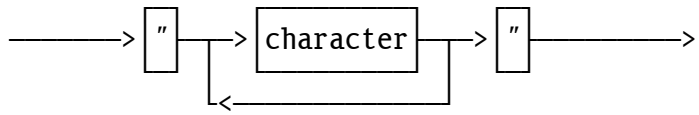
FACTOR:



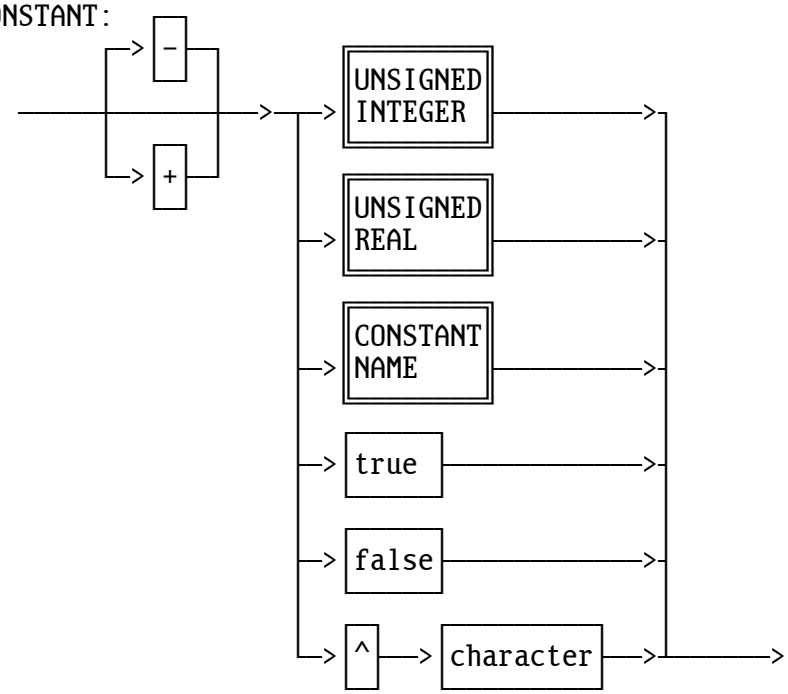
CONSTANT ARRAY:



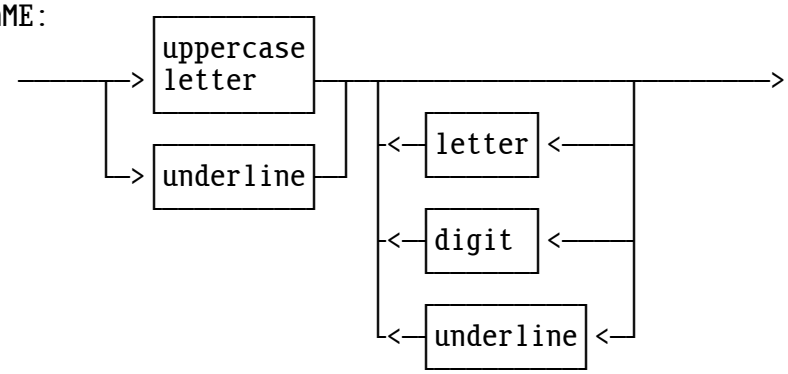
STRING:



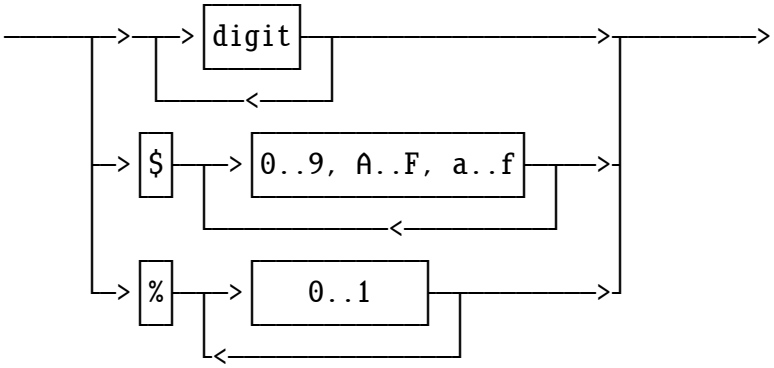
CONSTANT:



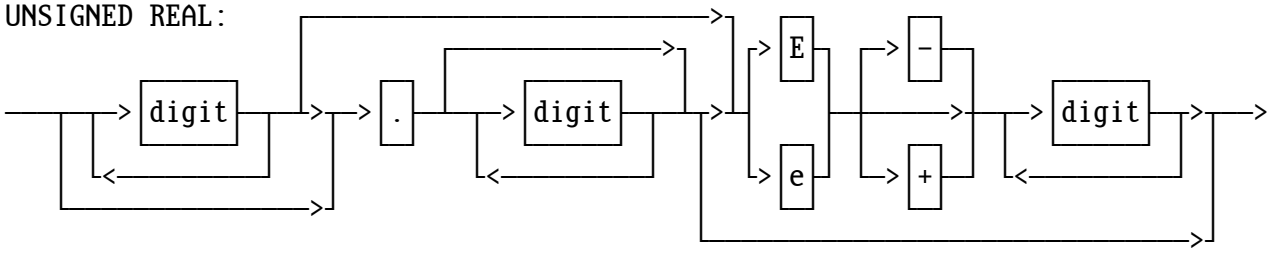
NAME:



UNSIGNED INTEGER:



UNSIGNED REAL:



I N D E X