Personal Designer

User Programming Language (UPL)

Revision 6.0

User Reference Guide

Chapter 2

Programm Structure

Program Structure

Program Structure

Program Structure

Syntax and General Rules

Like most programming languages, UPL has a number of reserved words that the compiler recognizes. These have a fixed meaning and cannot be used within a program for any other purpose. Keywords are the words that usually start a UPL statement. Reserved words include keywords, system variables, and intrinsic procedures and functions names. For example, in the statement,

```
PRINT 'What is your Name?'
```

PRINT is a keyword that tells UPL to display output to the screen.

A character string can contain a keyword. For example, in the statement,

```
PRINT 'Do You Know How To Print Your Name'
```

The first PRINT is a keyword that tells UPL to print output to the screen. The second print is simply one word in a character string. Since it is enclosed in quotation marks, it does not act as a keyword and has no special significance.

Keywords can be in any combination of upper- and lowercase. See Appendix A for a list of all UPL reserved words.

Variable, procedure, function, and label names must start with a letter and can be any length, but only the first 16 characters are significant. Upperand lowercase letters and digits can be used in names. The case of letters does not make a name unique (see example below). The underscore character is ignored in variable names and keywords and can therefore be used to make them more readable. The following names are equivalent:

```
ALONGNAME
A_LONG_NAME
ALongName
```

Although a statement is generally contained on one line, it can be continued to the next line by placing a backslash (\) at the end of the line. Two or more statements may be placed on one line by separating them with a semicolon (;).

Comments may be started anywhere on a line, except in a string constant, with a double dash (--). The end of the physical line terminates the comment. Therefore comments that spill over to additional lines must start with a double dash at the beginning of each line. A semicolon does not terminate a comment.

Program Structure

You can make programs more readable by including blank lines because UPL ignores them. You can also indent lines, as in the sample program below, to make your program more readable.

Explanation of UPL Program Structure

This chapter describes the structure of UPL, using three sample programs. Below is one of the simplest UPL programs.

```
Proc Main
     Print 'HELLO'
End Proc
```

This UPL program will print the word HELLO on your screen. All UPL programs must contain the PROC MAIN and END PROC statements. PROC MAIN is the starting point for all UPL programs. In this program UPL first goes to the PROC MAIN section, executes the statements, reads END PROC, and stops.

The following program, Rectangl, creates a rectangle.

Program Structure

```
------------------------------------------------
-- Rectang1.upl

Proc Main

--These next three lines are data declarations.
--They declare the variable names that will be
--used in this program.

string ans:1
Integer Cnt, NumDigs
Coord Points (2), Corners(5)

--These next two lines send output to the screen.

Print "Insert Rectangle: a UPL program."
Print "Digitize the opposite corners:",

--The line below is an intrinsic procedure that
--accepts input in the form of digitizes.

GetDig(2,1,NumDigs,Points(1))

--The next five lines are assignment statements
--that assign a value to a variable. The corners
--of the rectangle are defined in these lines.

Corners(1) = Coord(Points(1).X, Points(1).Y, 0.0)
Corners(2) = Coord(Points(1).X, Points(2).Y, 0.0)
Corners(3) = Coord(Points(2).X, Points(2).Y, 0.0)
Corners(4) = Coord(Points(2).X, Points(1).Y, 0.0)
Corners(5) = Corners(1)

--The following lines perform Personal Designer
--commands from the UPL program. This part of the
--program connects the corners of the rectangle
--with lines and then repaints the screen.

Echo Off All
Send
Send "INSERT STRING:",
Loop Cnt = I to 5
     Send DigStr(Corners(Cnt)),
End  Loop
Send "REPA" -- may need an extra send here
Echo On
End  Proc
------------------------------------------------
```

Program Structure

The next example is a more comprehensive UPL program. While the PRINT HELLO and Rectangle.upl programs contain only one of the UPL program sections, the Centigrade program contains all three of the UPL program sections. These are:

1. The GROUP section

2. The procedure (PROC) and function (FUNC) declaration section

3. The PROC MAIN section

NOTE: In this manual PROC MAIN is referred to as a separate section. It is a procedure declaration, but since it performs a special task in UPL and is different from all other procedure declarations, it is discussed separately.

The sample program ChgTemp locates text strings that contain temperature notations and changes these notations from Fahrenheit to Centigrade. This program is presented here to give you an idea of the scope of a typical UPL program. Do not be concerned if you cannot decipher each line at this time. This program will become clearer later in the manual. Refer back to these programs as you continue through this chapter.

Program Structure

```
-------------------------------------------------
-- ChgTemp.upl
-- This program will change all occurrences of
-- a Fahrenheit temperature within a text entity
-- to its equivalent Centigrade notation.

-- The first section in this program is the
-- "Group" section. It holds the declarations of
-- data to be shared between the procedures and
-- functions of the program. (It is here for
-- illustrative purposes only in this example.
-- None of these variables in this program.)

Group
     Integer this_var, that_var
     Real another_var
     String yet_another:20
     Coord point_var
End Group

-- The next section, "Proc ScanDegrees", is a
-- user-defined procedure. It is like a program
-- within a program. It will be called by the
-- Proc Main (below) to perform a specific task.
-- It has its own local data, but can share data
-- with other procedures and functions in the
-- program.

Proc ScanDegrees(IN String Method:1,ScanStr:256; \
                 INOUT String Degrees; \
                 Integer Offset)

     Integer StartIdx, EndIdx, Cnt
     String TestChar:1

     EndIdx = index(ScanStr, Offset, '~' + Method)
     If EndIdx=0 Then
          Offset=0
          Return
     Else
          Loop StartIdx = EndIdx-1 to 1 by -1
               TestChar = extract(ScanStr,StartIdx,1)
               Exit When \
               ((TestChar<'0')OR(TestChar>191))
               AND((TestChar<>'.')OR(TestChar<>'-'))
          End Loop
```

```
          Offset = StartIdx + 1
          Cnt = EndIdx - Offset
          Degrees = extract(ScanStr,Offset,Cnt)
          Return
     EndIf
End Proc

-- The next section, "Func Centigrade", is a
-- user-defined function. It is similar to a
-- procedure except that it may be called from within
-- an expression. Functions return a value through
-- the "Return" statement.

Func Centigrade(IN Integer DegreesFaren) \
                                        Return Real
   Return((5.0/9.0) * (Real(DegreesFaren) - 32.0))
End Func

-- The last section is the "Proc Main". It
-- directs program execution. Every UPL program
-- must have a Proc Main. All other sections are
-- optional and allow you to break a large
-- program up into manageable parts.

Proc Main
     Integer4 Num, Miblist(100)
     Integer Iend, StrIndx, EntCnt
     String OpRangeStr:256
     String TempStr1:256, TempStr2:256
     String NewDeg:12, FarenDeg:10
     Boolean Done = False

     BreakChar=3
     Print 'Digitize text string(s):',
     Getent(100,Num,Miblist(1),Iend)
     Loop EntCnt = 1 to Integer(Num)
          Verify Text EntId(Miblist(EntCnt)) \
                                     TXT(OpRangeStr)
          Loop
               ScanDegrees('F',OpRangeStr,FarenDeg, \
                                          StrIndx)
               Exit When StrIndx = 0
               NewDeg =  String((Centigrade(Integer( \
                         FarenDeg))):5:1) + '~C'
               TempStr1 = extract(OpRangeStr,1, \
                                  StrIndx-1)
               TempStr2 = extract(OpRangeStr,StrIndx \
                                 +FarenDeg.Length+2,256)
```

```
                OpRangeStr = TempStr1+NewDeg+TempStr2
                StrIndx: = StrIndx+FarenDeg.Length+2

            End Loop
            Modify Text EntId(Miblist(EntCnt)) \
                    TXT(OpRangeStr) RPNT(True)
        End Loop
End Proc
-------------------------------------------------
```

General Program Structure

A UPL program may start with the global data definition section or Group section. Here you define variables and constants that will be referenced by all procedures and functions in the program. Data defined in the Group section can also be shared with other programs by using the **Process** statement. See Chapter 4 for a full description of the Process statement. If your program consists of only Proc Main, omit the Group section and declare the variables within Proc Main. The global data definition section begins with the statement Group and ends with the statement End Group.

The next section of the sample program consists of procedure and function declarations. If a program consists of only one procedure, that procedure must be named Proc Main. If a program consists of several procedures, the last one must be called Proc Main. Think of procedures and functions as programs within your program. These 'sub-programs' perform specific tasks. Procedures and functions can contain sequences of code that would otherwise be repeated many times. Data used in these procedures and functions can be shared in the Group section or by passing parameters. Procedures and functions begin with the reserved word Proc or Func followed by the procedure or function name and end with the statement End Proc or End Func. Procedures and functions can contain variable declarations, statements, procedure calls, and function calls.

The final section, Proc Main, is a special procedure. It is the first procedure executed when you run the program. Often. a program consists of only the Proc Main. In more complicated programs the Proc Main directs the flow of the program by calling other procedures and functions. Proc Main is terminated with the statement End Proc.

Variables, Constants, and Expressions

Variables

Variables represent values in a UPL program. When you assign a value to the variable, this value is stored in a reserved location. You can set a variable's value to be the result of calculations or statements in the program. If the variable changes value during the program, UPL replaces it with the new value.

Before using a variable in a program, you must declare it's characteristics. Variables have names, data types, aggregate types, storage types, and attributes. Variable declarations are discussed below.

Names

The names you supply for variables must start with a letter. You can use upper- and lowercase letters and numerals. UPL ignores underscore characters in variable names, so you can use these to make the names more readable.

Variable names are recognized by the first 16 characters only. Any characters appearing after the 16th do not make the name unique. For example, the variable names CENTIMETERLENGTH I and CENTIMETERLENGTH2 describe the same variable.

Single letters and abbreviations are often used as variable names. Use meaningful words or abbreviations as variable names, to make it easier to remember what the variables represent.

Data Types

In UPL, there are several kinds of data you can work with. These data types are described below.

Integer represents whole numbers (without a decimal point) in the range -32,768 to 32,767.

Integer4 represents a larger range of whole numbers, from approximately -2 billion to +2 billion (or -2,147,483,648 to +2,147,483,647). Integer4 is often called long integer or 32-bit integer. Use *Integer4* rather than *integer* for entity counts and MIB identifiers, since these have a range which exceeds the maximum value that can be represented by integer variables.

Real refers to positive or negative numbers that contain a decimal point and start with a digit. Any integer can be written as a real number by adding a decimal point followed by a zero. For example, 10 is an integer but 10.0 is a real number. Real values represent numbers over the range of plus or minus 38 orders of magnitude with 7 places of precision.

Booleans store the result of comparisons or logical tests. There are two values that a Boolean variable can store: True and False.

Coordinates represent a location in 3D space. It is actually three values treated as cartesian coordinate.

Strings consist of a series of characters.

Aggregate Types

Aggregate type refers to how the data in variables is grouped as a unit. UPL has three aggregate types.

Scalar variables are single elements.

Arrays contain a series of elements that let you store items of the same data type in an organized, accessible way. In UPL, multi -dimensional arrays can contain up to five dimensions. The first subscript changes the fastest as elements are allocated in memory. This is sometimes referred to as column major form.

Files can contain data of all data types. These files can be stored separate your program,say, on a disk. A file variable represents the file in the program.

Storage I Access Types

There are two ways to store variables: locally and globally. A global variable is declared in the GROUP section of a program. This means that it can be shared by other sections of the program. A local variable can only be used in the procedure or function it is declared in.

Variable Attributes

Variable attributes describe characteristics of variables apart from their cur-rent value. To access attribute information, place a period after the variable name, followed by the name of the attribute. Variable attributes for coordinate, string, file, and array variables are discussed below.

The following table lists variable attributes for coordinate variables.

Attribute	**Data type**	Can be set?	**Description**
X	real	Yes	The X value
Y	real	Yes	The Y value
z	real	Yes	The Z value

Table 2-1. Coordinate variable attributes.

The following table lists attributes for string variables.

Attribute	**Data type**	Can be set?	**Description**
LEN	integer	No	The maximum number of characters the STRING can have.
LENGTH	integer	Yes	Current number of characters in STRING variables.

Table 2-2. String variable attributes.

The following table lists variable attributes for variables with file aggregate type. A READ operation may be either the **Read** statement (for text and binary files) or the **ReadCArray, ReadIArray, ReadRArray** intrinsics (binary files only). An *end-of-file marker is* defined as ASCII 26 in DOS and ASCII 04 in UNIX. The *end-of-line marker is* defined as ASCII 13 followed by ASCII 10 in DOS. It is an ASCII 10 in Unix.

Attribute	**Data type**	Can be set?	**Description**
EOF	Boolean	No	Set TRUE when any READ operation (text or binary, random or sequential) reads past the physical end of file. Also set TRUE when a sequential access Read statement on a text file encounters an *end-of-file marker.* Otherwise it is false0therwise it is false.
EOLN	Boolean	No	Set TRUE when any sequential access Read statement on a text file encounters an *end-of-line marker,* an *end-of-file marker* or the physical end of file. It is reset to FALSE after reading the last item in a Read statement that does not end in a comma. Binary files do not use the EOLN attribute.
OPEN	boolean	No	Set TRUE by the Open statement and reset to false by the Close statement.

POSITION	integer	Yes	Used to set or determine the current position in a random access file. It holds the the **current record. number** The first record in a file is record one. The position of a random access file depends upon this number and the record length attribute. Record numbers can be set in the range I to 32,767 using POSITION. For sequential files, the position attribute always has a value of one and has no effect when set.
POS4	integer4	Yes	Used to set or determine the current position in a random access file. It holds the position of the file pointer as a byte offset from the **beginning of the file** The byte offset ranges from 0 to 2,147,483,646.
RECLEN	integer	No	Contains the file's record length as set by the Open statement.. If the file was not opened with RECLEN, then a value of zero is returned.

Table 2-3. File variable attributes.

The following table lists variable attributes for array variables.

Attribute	Data type	Can be set?	Description
SIZE	integer	No	Gives the number of elements in an array.

Table 2-4. Array variable attributes.

System Variables

Some system variables are predeclared and accessible to every UPL program. These variables either return information about or change the current status of the system. Other system variables are accessible through the intrinsic procedures **SysVarI**, **SysVarR**, and **SysVarS**.

There are three categories of predeclared system variables: file status information, window information, and character information.

File status system variables hold information about data base and file access status. These cannot be set.

Window information system variables poll and change the alpha-numeric windows used by the system's commands. These can be set.

Character information system variables hold data about characters sent between Personal Designer and the UPL program.

The predeclared system variables are discussed in detail in Appendix B,. Other system variables are discussed in Chapter 4 under the intrinsic procedures **SysVarI, SysVarR,** and **SysVarS.**

Constants

Unlike variables, constants do not change when you run the program. There are two types of constants: literal and named. Literal constants consist of numerical, booleans, or string values. You supply these values when you write the program.

Literal constants

Literal coordinate constants are enclosed in brackets and define a specific point in space. If you are working in one or two dimensions, you can omit the y and z coordinates and the system will automatically supply the value zero. Examples of literal coordinate constants are:

```
[6.0,-8,-19.2]
[9.2]  is the same as [9.2,0.0,0.0]
[]  is the same as [0,0,0]
[3.0,4.6]  is the same as [3.0,4.6,0.0]
```

Literal string constants are enclosed in quotation marks. The quotation marks tell the program not to process the information contained within. For example, the UPL statement

```
PRINT 32+54
```

produces the screen output 86. The program performs the addition operation first and prints the result. However, the statement

```
Print '32+54'
```

produces the screen output 32+54. The program prints the exact expression within the quotation marks without processing the information. The quotation marks act as delimiters and are not part of the string.

String constants can include letters, numerals, and special characters in any combination. You can use either single or double quotes to enclose the string. If you want to include a single quote within a string, use double quotes as delimiters. Examples of string constants are:

```
'A String Constant'
"The letter 'A' is for able."
"the conversion factor is 2.54"
''
```
(a null string)

To use control character sequences or a carriage return in a string, enter a pound sign (#) followed by the ASCII character code that represents the character followed by another optional pound sign. The second pound sign should be used if the character immediately following the ASCII character code is a digit from zero through nine. Some of the most frequently used ASCII character codes are 13(carriage return), 3(^C), 2(A B), and I O(line feed). To represent a carriage return in a string, use '# 13'. To use the pound sign as a character in a string, put in two pound signs. See Appendix F for a complete listing of the ASCII codes.

Named constants

You can give a constant a name and use this name in the program in place of the literal value. Named constants make a program easier to understand and modify. For example, if the maximum value is 500, you can use MAX_VALUE, as the named constant, to replace each occurrence of 500. If you want to increase the maximum value to 800, only one line in your program has to be changed. After you reset MAX-VALUE to 800 and compile your program, the system updates all occurrences of MAX-VALUE in the program code.

Declaring Variables and Constants

In UPL, you must declare all variables and constants before they can be used. You can do this in the GROUP section or within procedures. To declare a variable, list its data type, aggregate type, and variable name. In the following examples of variable declarations, RI is declared as a real variable and B I is declared as a Boolean variable.

```
REAL R1
BOOLEAN B1
INTEGER I1
```

Once you declare a variable as belonging to one data type, you cannot give it a value belonging to another data type. In the above example, you cannot give B I the value 10 because 10 is an integer.

The exception to this rule is with integer, integer4, and real constants. You may assign an integer or an integer4 constant to a real variable. The result is simply an integer constant treated as a real. For example

```
REAL R1 = 10
```

results in assigning 10.0 to R1.

You may also assign a real constant to an integer or integer4 variable. The real constant is rounded and then assigned as an integer value to the variable. For instance,

```
INTEGER I1 = 10.6
```

results in the assignment of 11 to I1.

To declare more than one variable of the same data type, separate the variables with commas. The following example declares the variables A, J, and K as integers.

```
INTEGER A, J, K
```

To give a variable an initial value, follow the variable with an equal sign and the initial value. The data type of the variable and the initial value must match. In the examples below, the integer variable J is initialized to the value 32. The coordinate variable C3 is initialized to the value [-5.5, -4.0, 9.0] and the real variable R1 is initialized to the real value 10.0.

```
INTEGER A,J=32,K
COORD C3 = [-5.5, -4.0, 9.0]
REAL R1 = 10.0
```

When declaring variables as strings, you must give the maximum number of characters in the string by following the variable with a colon and an integer constant. The maximum length tells the system how many bytes of storage to reserve for the string. Here is an example: STRING NAME:40, DRWNM:8 = ' NEW.DRW '.

This declares the variable NAME as a string with a maximum length of 40 and the variable DRWNM as a string with a maximum length of 8 and an initial value of NEW.DRW.

To declare an array, you give the array name followed in parentheses by the array dimensions. You can declare arrays of up to five dimensions. The following example declares D as a one-dimensional array with 10 elements:

```
INTEGER D(10)
```

You cannot initialize an array variable in its declaration.

The following example declares a two-dimensional coordinate array, C1, with four rows and five columns containing coordinate data:

```
COORD C1(4,5)
```

When you declare a named constant, write the keyword CONST first, followed by the data type, and then the name that you equate to the value it represents. Once you declare a named constant, you cannot change its value throughout the program. The following are some examples of named constant declarations:

```
CONST INTEGER HIGHEST = 90, AVERAGE = 54
CONST REAL INTEREST_RATE = 0.165
```

Program Structure

Expressions

UPL programs contain expressions that evaluate or take action upon data. There are four kinds of expressions: arithmetic, string, relational, and Boolean. An expression must have two operands of the same data type and an operator.

In the expression, 4.0*2.0, 4.0 and 2.0 are operands and * is an operator. 4.0 and 2.0 are both real data types. In order to use two different data types in an expression, you need to use intrinsic functions to convert the operands to the same data types.

For example, to add a real number R1 to an integer I2, use the intrinsic function **Real** to convert the integer to a real number,

```
R1 + Real(I2)
```

See Chapter 4, Statements and Intrinsics, for more information on intrinsic functions.

The operands in expressions can be any of the following:

```
Constants Variables variable attributes Function
calls Other expressions
```

When arithmetic, relational, and Boolean operators appear in the same expression, the order in which they are evaluated follows the guidelines in Table 2-5. One has first precedence and eight has last precedence.

Operator	Operation	Precedence
	Identity	
+ or -	Positive or negative value	1
	Arithmetic	
** or ^	Exponentiation	2
/	Division	3
*	Multiplication	3
-	Subtraction or Negation	4

+	Addition or Identity	4
	Relational	
<	Lessthan	5
<= or =<	Less than or equal to	5
	Equal to	5
<>	Not equal to	5
>	Greater than	5
>= or =>	Greater than or equal to	5
	Boolean	
NOT	Negation	6
AND	Conjunction	7
OR	Inclusive Disjunction	8

Table 2-5. Relative Precedence of Operator Classes and Expressions.

Operator Precedence

If an expression contains more than one operator from the same level of precedence, those operations are evaluated from left to right.

Parentheses can be used to change or clarify the order of precedence. Operations within parentheses are evaluated first. If you use multiple pairs of parentheses, calculations are performed from the inside out.

The following expressions are identical except for parentheses. Because of the parentheses they are evaluated differently.

2 + (50 + 10) *6 equals 362
2 + 50 + (10*6) equals 112

Arithmetic Expression

Arithmetic expressions consist of numbers or variables connected with a mathematical symbol.

Expressions are evaluated according to operator precedence rules. For example, 45 + 15*2, is evaluated to 75. The program first multiplies 15*2, and then adds 45 since multiplication takes precedence over addition.

The arithmetic operators and their order of precedence are shown in Table 2-6.

As the table below indicates, exponentiation has the highest level of precedence, followed by multiplication and division, and then addition and subtraction. Exponentiation is always evaluated from right to left.

Operator	Operation	Precedence
** or ^	Exponentiation	Highest
/	Division	Intermediate
*	Multiplication	Intermediate
-	Subtraction	Lowest
+	Addition	Lowest

Table 2-6. Arithmetic Operators and Their Order of Precedence

You can also use the operators, + and -, with a single unary operand as shown in the following examples. These expressions have only one operand and indicate a positive or negative value. Unary operators have the highest possible precedence.

```
-5
-3.21
+30001
```

The division of two integers results in the mathematical quotient of the two values truncated toward zero. Thus, 7/3 evaluates to two, and (-7)/3 evaluates to negative two. Both 9/10 and 9/(-10) evaluate to zero.

String Expressions

An expression that evaluates to a string is called a string expression. String variables and string constants are examples of string expressions. The intrinsic function **String(R),** that converts a real number to a string, is also a string expression. You can form more complex string expressions by linking two or more string expressions together with a + sign. The following are examples of string expressions:

```
S1
"A STRING"
S1 + 'END'
("BEG" + S1)
```

Relational Expressions

Relational expressions compare the values of two arithmetic or string expressions using a relational operator. When you evaluate a relational expression, the result will be Boolean: True or False. These operators are listed in Table 2-7:

Operator	Operation
<	Lessthan
<= or =<	Less than or equal to
=	Equal to
<>	Not equal to
>	Greater than
>= or =>	Greater than or equal to

Table 2-7. Relational Operators

Relational expressions let you compare integer, real, string, and coordinate data. (You may compare two coordinates for equality or inequality only. To get further information, between two coordinates, compare the x, y, and z attributes of one coordinate with those of the second coordinate.)

To compare strings, UPL compares the ASCII value of the first character in the first string with the ASCII value of the first character in the second string. Characters include letters, numerals, spaces, and special symbols. If the ASCII values of the two characters are the same, UPL compares the second character in each string and so on. If the strings are unequal in length, the shorter one is considered as if it were extended to the length of the longer one by the addition of spaces. The system continues the comparison until it finds an inequality in the ASCII codes.

Relational expressions are often used in conditional statements such as If and Exit When. In many cases relational expressions are combined with the Boolean operators, NOT, AND, and OR, for example:.

```
  IF X>Y AND Y=2 THEN
       X=2
  ENDIF
EXIT WHEN NUM>5 OR NUM<=0
```

Boolean Expressions

A Boolean expression is any expression that is evaluated to true or false. Boolean expressions include Boolean constants, Boolean variables, relational expressions, and intrinsic functions such as **Exist(FN)** that yield a value of data type Boolean.

You can use Boolean operators to combine simple Boolean expressions into more complex Boolean expressions. The Boolean operators are listed in Table 2-8.

Operator	Operation	Precedence
NOT	Negation	Highest
AND	Conjunction	Intermediate
OR	Inclusive Disjunction	Lowest

Table 2-8. Boolean Operators

The AND and OR operators require two operands. The NOT operator is placed in front of its single operand and thus is a unary operator.

As with arithmetic expressions, you can use parentheses to change the order in which operations are performed. If a Boolean expression contains operations in parentheses, these are evaluated first.

Boolean expressions are most often used within conditional statements and as flags. In the following examples, the Boolean expressions are capitalized for clarity.

```
exit when NOT A OR B AND C
exit when (NOT A) AND (B OR C)
if LENGTH<=0.0 OR WIDTH<=0.0 then exit; endif
exit when A
```

There is no Boolean operator that tests for equality like there is for relational expressions. You can assign a true or false value to a Boolean variable, but you cannot test for an equality in a Boolean expression by using an equal sign. In the example, EXIT WHEN A, the implicit meaning is Exit when A is 'equal' to true. In the following example the first statement is not a legal statement since you cannot use an equal sign to test for equality in a Boolean expression. The second statement is a legal UPL statement.

```
exit when bvar = true      --incorrect
exit when bvar             --correct
```

Program Structure

Statements

This section explains the general capabilities of each type of statement class in UPL. The individual statements that belong to these classes are discussed in more detail in Chapter 3, "Functional Listing," and Chapter 4, "Statements and Intrinsics".

Declaration Statements declare the data, aggregate and storage types of variables available in each program. For more information refer to the heading, Declaring Variables and Constants, in this chapter.

Assignment Statements are the most basic statements in UPL. Use these statements to change the value of a variable, variable attribute, or array element within a program.

Program Structure Statements define the structure of a program. They are the **Proc, Func, and Group** statements. **Proc and Func** contain the code portions of the programs and Group contains the variable declarations. These statements are described in the chapter, "Program Structure."

Flow Control Statements control the flow of the program by testing for conditions in the code. Flow control statements direct the program's path.

`If Then Else Endif`	executes a series of statements according to a Boolean result. If statements can be nested.
`Loop EndLoop`	executes a series of statements repetitively. A loop can execute a specified number of times, or until a specific condition or event occurs.
`Exit`	causes control to jump from this statement to the end of a surrounding **Loop or If** statement.
`GoTo`	causes control to jump from this statement to a specified label in the code.
`Return`	causes control to return from a called procedure or function to a calling procedure or function. **When Return** is used with a function it specifies a return value.
`Process`	causes control to pass to the beginning of another UPL program.
Subroutine call	executes a procedure or function.

Program Structure

Input / Output Statements control the input and output to and from the graphics screen, keyboard, digitizer, and DOS or UNIX files. These statements interface to and from files and windows in UPL.

The Window Input / Output statements are:

`Window`	defines characteristics of a window and assigns it to a number.
`Accept`	accepts input from the user and stores it in a variable.
`Print`	sends output to a window.
`Display`	displays blocks of text to a window.
`Clear`	erases the contents of a window.
`Send`	ends commands to the system.
`Echo`	controls echoing of commands sent to the system.

The File Input / Output Statements are:

`Open`	opens a file for input or output. Controls text and binary data access to files. Also determines sequential or random access.
`Read`	reads data from a file.
`Write`	writes data to a file.
`Close`	closes access to a file.

Database Access Statements manipulate the drawing database. These statements act upon the entities in the current drawing file. See Appendix G for more information on these statements. The database access statements are:

`Insert`	lets entities be inserted directly into the part database.
`Verify`	returns information about entities in the database.
`Modify`	allows modification of existing entities in the database.

Compiler Directive Statements control the actions of the compiler. They are:

`$Include`	lets the compiler use code from a text file other than the UPL source code file.
`$Codesize`	tells the system how much external memory to allocate for UPL code.

Procedures and Functions

UPL supports procedures and functions. Procedures and functions are sometimes known as sub-routines or sub-programs. You may want to think of them as programs within your program. They perform some task or calculation and let you break down your program into small, manageable pieces.

Procedures

Procedures are invoked by entering their name as a statement in your program. This is known as *calling* a procedure.

When the program is run and a procedure call is encountered, the following happens. First, the flow of control moves to the procedure. Next, the statements which make up the procedure are executed. Finally, the flow of control returns to the main line of your program, to the statement following the procedure call.

Procedures can have many input and output values known as parameters. Your program specifies these values to the procedure when it is invoked. The procedure returns values to your program when it has finished executing. This is known as *passing parameters.* (see *Parameters,* below)

Functions

Functions are invoked by entering their name in an expression which is part of a statement in your program. This is known as 'calling' a function.

A function operates in a slightly different way from a procedure. A function call may be encountered anywhere, even in the middle of executing a statement. The flow of control moves from your program to the function. The statements which make up the function are executed.

These statements always calculate a value. When the flow of control returns to your program, the value is also returned. This value takes the place of the function call in within the original statement. The rest of the statement is then executed.

Only input parameters can be passed to functions; functions return only one value. (see *Parameters,* below.)

Intrinsic Procedures and Functions

Functions and procedures (that is, subroutines) may be one of two kinds: *intrinsics* available as part of UPL language and *user-defined* subroutines developed by the user.

UPL supplies the intrinsic procedures and functions described in Chapter 4. These procedures and functions are built into the language and do not need to be defined in program code.

There are intrinsic procedures for a variety of tasks such as inputting coordinate data, drawing graphics on the screen, and manipulating the database. Intrinsic functions perform many basic calculations such as taking the square root of a number, finding the sine of a number, and converting a real number to its nearest integer equivalent.

UPL:s intrinsics can be grouped into several classes to make them easier to remember. These classes are listed below. For a complete list of intrinsic procedures and functions grouped by the tasks they perform see Chapter 3, *Functional Listing.*

Intrinsic Classes

UPL has many different intrinsic functions and procedures that make programming easier. These intrinsics can be grouped into the following classes.

Graphics Intrinsics let the programmer define the, menus, and text, display entities, and shade regions.

Input/Output Intrinsics let the programmer define the graphic windows and perform some file operations.

User Interface Intrinsics let the programmer create a command structure identical to Personal Designer's.

Geometric Intrinsics eliminate the need for the user to extract data about entities and calculate their relationship.

Database Access Intrinsics give programmers direct database access and the ability to manipulate the drawing entities.

Operating System Intrinsics allow access to operating system functions.

Arithmetic Intrinsics provide basic arithmetic and mathematical capabilities.

Trigonometric Intrinsics perform trigonometric functions on variables.

Datatype Conversion Intrinsics perform the necessary conversions on the variables in expressions and statements.

String Handling Intrinsics perform manipulations on string variables.

User-Defined Procedures and Functions

User-defined procedures and functions are sub-programs which you, the programmer, create. They must be 'defined' before they can be called. That is, the UPL statements which make up the user-defined procedure or function must occur in your program before the statement or expression that calls them. These definitions must occur in the program after the Group section and before the Proc Main (see *General Program Structure,* earlier in this chapter.)

User-defined procedures and functions may, of course, be called from the Proc Main. They may also be called from within other user-defined procedures and functions as long as the one being called is defined before it is called. For example, if procedure A calls procedure or function B, then B must be defined before A. If A calls B, then B cannot call A.

A user-defined procedure or function can be called from within itself. This is known as recursion.

User-defined procedures and functions pass parameters in the same manner as intrinsic procedures and functions. (see *Parameters,* below)

Defining User-Defined Procedures

The **Proc** statement must begin a user-defined procedure. It is followed by the procedure name which will be used to call the procedure. An optional parameter list follows the procedure name. Following this comes the body of the procedure. This includes declarations, assignments, flow of control statements, calls to other procedures and functions, etc. The procedure must end with the statement **End Proc.** See the entries for **Proc** and **End Proc** in Chapter 4, "Statements and Intrinsics," for more information.

Defining User-Defined Functions

The **Func** statement must begin a user-defined function. It is followed by the function name which will be used to call the function. An optional ,parameter list follows the function name. Next, the keyword **Return** must appear followed by the data type of the value to be returned. Next comes the body of the function. This includes declarations, assignments, flow of control statements, calls to other procedures and functions, etc. The body of the function must contain a **Return** statement. This statement defines the value returned by the function. The function must end with the statement **End Func.** See the entries for **Func , End Func ,** and **Return (for Functions)** in Chapter 4, "Statements and Intrinsics", for more information.

Parameters

In UPL, you can share data in the Group section with all procedures and functions in your program. Such data is often known as global data. It is usually more advantageous to declare variables locally in a procedure. Unless passed to another procedure as a parameter, local data is accessible only in the procedure in which you declare it. This makes it easier to maintain and debug a program.

You can then share data exclusively between certain procedures and functions by passing parameters. The data that you pass can be variables, constants, or expressions. In large programs, passing parameters lets you keep track of where data is being changed.

Parameters, when declared, act as place holders. They represent the data from your program to the called subroutine.

Passing Parameters

There are two modes for passing parameters, *input* and *inputloutput. Two* important differences exist between input and input/output parameters. First, if the value of a variable passed as an input parameter is changed in the called procedure or function, the value of that variable in the calling procedure does not change. In contrast, when the value of a variable passed as an input/output parameter changes, the value of that variable in the calling procedure also changes.

The second difference is that you can pass either expressions or variables as input parameters, while you can only pass a variable as an input/output parameter.

Use input parameters unless you are expecting the value to change. This protects you from inadvertently changing a variable's value in a procedure and then using that variable when you leave that procedure. Functions can only have input parameters.

There must be a one to one correspondence between the variables in the procedure or function call and the parameters in the called procedure or function definition.

To pass parameters to an intrinsic procedure orfunction, refer to the appropriate page in Chapter 4 for the particular intrinsic procedure or function. The mode is listed along with the data and aggregate type of the parameters, either ' (input) ' or ' (input/output) '. If the parameter is listed as an expression, you can pass either an expression or a variable. If the parameter is listed as an array or variable, you must give the array or scalar variable name.

If the description of the parameter includes the term "specifies", you must initialize your input value or variable. If the description includes the term "returns", the value may change during execution of the intrinsic procedure or function.

The return type of an intrinsic function is given under the *Purpose* heading.

Call the intrinsic procedure or function by specifying its name. Substitute an expression or the name of the variable or constant from your program into each parameter in the parameter list. The data type of the parameters and the values or variables you substitute must be the same. Surround the parameters in parentheses.

To pass parameters to a user-defined procedure orfunction, refertothe **Proc** or **Func** statement in your procedure or function definition. The mode is listed along with the data and aggregate type of the parameters, either '**In** ' for input or ' **InOut** ' for input/output. If the parameter's mode is **In**, you can pass either an expression or a variable. If the parameter 's mode is **InOut** you must give the array or scalar variable name.

Call the user-defined procedure or function by specifying its name. Substitute an expression or the name of the variable or constant from your program into each parameter in the parameter list. The data type of the parameters and the values or variables you substitute **must** be the same. Surround the parameters in parentheses.

Defining Parameters to User-Defined Procedures and Functions

The steps below illustrate how to define the parameters to a user-defined procedure or function. Note that user-defined functions can only have input parameters, thus their mode must always be input. Function input parameters cannot be arrays.

1. List the mode of the parameter, either `in` for (Input) or `inout` for (Input/Output).

2. List the data type of the parameter.

3. List the parameter names, separating each with a comma.

4. To change the data type or mode, use a semicolon followed by the new data type or mode. If you change mode, you must also give the data type even if it has not changed. If the mode does not change you need not repeat it.

5. Make sure that the entire parameter list is enclosed in parentheses.

```
Proc P1  (INOUT Integer I1(3,5), I2; \
          IN Integer I3; Real X; String S1:20;\
          INOUT String S2(10,50))
```

This example has 6 parameters:

Parameter I	Integer array I1 (3,5)	Input/Output
Parameter 2	Integer I2	Input/Output
Parameter 3	Integer I3	Input
Parameter 4	Real X	Input
Parameter 5	String S1:20	Input
Parameter 6	String array S2(10,50)	Input/Output

In the example above, when you declared Real X an Input variable, you did not need to repeat the mode keyword In even though the data type changed.

The program below, CalcLen, illustrates calls to both user-defined and intrinsic procedures. Following this is an explanation of the parameters used.

```
------------------------------------------------
-- CalcLen
Proc FindLength(In Coord End1, End2; \
                Inout Real Length1)
Length1=Sqrt( ((End1.X-End2.X)**2.0) \
            + ((End1.Y-End2.Y)**2.0) \
            + ((End1.Z-End2.Z)**2.0))
End Proc
Proc Main
   Integer NumEnds
   coord EndPoints(2)
   Real TheLength

   Print "This program determines ",
   Print "the length of a line."

   Print "digitize the endpoints:"

   GetEnd(2,1,NumEnds,EndPoints(1))

   FindLength(EndPoints(1),EndPoints(2),TheLength)
Print   "The length of the line from ", \
         EndPoints(1), " to ", EndPoints(2), \
         " is ", TheLength
End Proc
------------------------------------------------
```

Proc FindLength is the *called* procedure. Proc Main is the *calling* procedure. When FindLength is called from Proc Main, the variables EndPoints(1), EndPoints(2), and TheLength are passed as parameters to the procedure FindLength.

The first two variables EndPoint 1 and EndPoint2, will not change since they were passed as input parameters to End 1 and End2. The last variable, TheLength, will change since it was passed as an input/output parameter to Length 1.

CalcLen also contains a call to an intrinsic procedure, **GetEnd.** The data type and mode of each parameter are pre-defined and listed under this intrinsic procedure in Chapter 4. The first two expressions, 2 and 1, are passed as input parameters and thus will not change. The second two variables, NumEnds and EndPoints(1) are passed as input/output parameters and thus can have a new value after **GetEnd** returns.

Passing Strings as Parameters

To pass string variables or expressions, the following rules apply.

1. The length of a string variable, constant, or expression being passed to an input string parameter should not be larger than the declared maximum length of the string parameter.

   In the following parameter list, the IN string has a maximum length of 20.

```
P1(IN String S1:20)
```

If the string variable passed to this parameter is longer, it is truncated to 20.

2. For Input/Output strings, no string length is given in the parameter list of the called procedure. The maximum string length is determined by the declared maximum length of the string in the calling procedure.

Below is an example of passing strings as parameters:

```
------------------------------------------------
-- StrProc1 procedure uses an Input String
-- parameter. It will print out a maximum of
-- 10 characters.
```

```
Proc StrProc1(IN string InpStr1:10)
     print InpStr1
end  Proc

-- StrProc2 procedure uses an Input/Output String
-- parameter.

Proc StrProc2(INOUT string InpStr2)
     print InpStr2
end  Proc

-- The main procedure will pass the same string
-- to the two parameters.

Proc Main
string TestString:20
     TestString = 'This is a test.'
     StrProc1(TestString)
     StrProc2(TestString)
end  Proc
-----------------------------------------------
```

This example would print out:

```
This is a
This is a test.
```

Passing Arrays as Parameters

You can only declare arrays as Input/Output parameters; functions cannot have arrays as parameters. When calling a *user-defined* procedure, the array being passed as a parameter should not be larger than the array declared in the parameter list of the user-defined procedure. When calling the user-defined procedure, you must pass the entire array. Thus you must use a subscript of one for all array indices.

To pass just one element of an array to another procedure, the parameter should be declared as a scalar, not an array.

In *intrinsic* procedures the size of the array variable to be passed as a parameter does not matter. With intrinsic procedures you can pass part of an array. Specify the element before those you want passed and all elements after this one will be passed.

Array parameters for intrinsic procedures are always listed as Input/Output.

Program Structure

When using intrinsic procedures look for the description of the parameter in Chapter 4. Parameters described as specifying something require you to pass an array with data in it. Parameters described as returning something will pass back new data in the array.

The program below shows different methods of passing array parameters.

```
-------------------------------------------------
-- Passing Arrays as Parameters

-- Procedure 'SingleDimArr' may be passed any
-- single dimensioned integer array with less
-- than 100 elements. Although you must pass the
-- first element of the array to (because its a
-- user defined procedure), you may specify an
-- element or subrange of elements with which
-- to work.

Proc SingleDimArr(INOUT integer Arr(100); \
                  IN Integer StartElement, \
                  NumElements)

      integer i
      print 'Single Dim Array = '
      loop i = StartElement to \
                   StartElement+(NumElements-1)
          print Arr(i),'
      end loop print
end proc

-- Procedure 'MultiDimArr' should only be passed
-- an array with dimensions of 3x2. The first
-- subscript varies the fastest as you access
-- elements in storage order. You may have up to
-- 6 dimensions in an array in UPL.

Proc MultiDimArr(INOUT integer Arr(3,2))
    Integer i,j
    print 'Multi Dim Array = '
    loop i = I to 2
        loop j = 1 to 3
            print Arr(j,i),'
        end loop
    end loop
    print
end proc
```

```
-- Procedure SingleElement will accept an single array
-- element as a parameter. Note that the parameter
-- 'Single' is actually a scalar (not an array).

Proc SingleElement(IN Integer Single)
     print 'value passed = ',Single
end proc

-- Procedure Main demonstrates passing arrays as
-- parameters to the user-defined procedures
-- given above. The last procedure call is to
-- the Intrinsic procedure 'Product'. It requires
-- an array of at least 14 elements. In this
-- example it will return data starting at array
-- element 5 and continue to array element 19.

 Proc Main
     integer Data(20)
     integer Iarr(3,2)

     Data(3) = 13
     Data(4) = 14
     Data(5) = 15

     Iarr(1,1) = 1
     Iarr(2,1) = 2
     Iarr(3,1) = 3
     Iarr(1,2) = 4
     Iarr(2,2) = 5
     Iarr(3,2) = 6

     SingleDimArr(Data(1), 3, 3)

     MultiDimArr(Iarr(1,1))

     SingleElement(Iarr(2,1))
     SingleElement(Data(3))

     Product(Data(5))
     print 'Personal Designer version ',
     print Real(Data(6)/100):4:2
end proc
-------------------------------------------------
```

The program *ChgTemp.upl* in the beginning of this chapter shows how to define and use user-defined procedures and functions. This program looks for Fahrenheit temperature notations using the procedure *ScanTemp,* converts them to centigrade using the function *Centigrade,* and then uses the result in an assignment statement. Now that you are more familiar with UPL program structure, go back and examine this sample program.