User Programs *(version 4.0)*

The data processing available through a user program is based on a robust programming syntax that is a cross between the 'C' and FORTRAN languages. The user script is first compiled into an efficient, stack-based command structure that is then executed each time the main program calls the routine. Most user programs are designed to read from and write to specific data fields within a pre-defined set of data files. The main program typically manages the file input and output and the user program performs calculates based on data records from the files.

User programs can be implemented in both Windows and Linux executables. In a Windows program all of the syntax commands and variables are not case sensitivity (e.g., "a1" is the same as "A1"). A Linux executable is case sensitivity. The syntax keywords and functions must be defined with all-caps in a Linux script. The case for variable and field names is set when they are first defined.

Variables

All variables included in the user-provided data files are automatically defined and available for use. The user program operates on fields in the current data records using the syntax:

```
File_Name.Field_Name
```

For example, the main program defines a user file named "NODE" and the first line of this file or the associated Definition file defines the following field names:

```
X, Y, TAZ, AREA_TYPE
```

The user can reference the fields in the current data record using the following syntax:

```
NODE.X
NODE.Y
NODE.TAZ
NODE.AREA_TYPE
```

In addition to data records, the user can define any number of local variables. Local variables are defined using declaration statements. Three types of variables can be declared. These include integers, decimal numbers, and strings. The declaration syntax is:

```
INTEGER  xxx, x1..x20, Y_100, z  ENDDEF
REAL  R10..R30  ENDDEF
STRING  svalue  ENDDEF
```

Where "xxx, x1..x20, Y_100, z", "R10..R30", and "svalue" represent the variable names defined by the user. Variable names must start with a letter and may include numbers and underscores (i.e., "_"). The list of variable names can continue onto multiple lines of the program. The declaration process continues until the "ENDDEF" keyword is reached.

As above examples demonstrate, variable ranges can be used. "x1..x20" will declare all variables between "x1" and "x20" as integers. The text parts of a range command must be identical. The number parts must increase from the lower value to the upper value.

Local variables are initialized to zero (or blank) before the first program execution, but maintain their current value as each record is processed. This enables the user to accumulate data for a given data field, or pass information from a previous record to a subsequent record. The software also maintains a pre-defined record counter ("RECORD") that the user program can reference. "RECORD" is set to one for the first execution and is automatically incremented for each subsequent execution.

Arithmetic Operators

The purpose of a user program is to manipulate data fields, perform calculations, or implement conditional procedures. To facilitate these tasks, the user program provides a full set of arithmetic operators. These operators include:

Operator	Syntax	Result
+	value1 + value2	Sum of 'value1' and 'value2'
–	value1 – value2	Difference between 'value1' and 'value2'
*	value1 * value2	Product of 'value1' and 'value2'
/	value1 / value2	'Value1' divided by 'value2'
**	value1 ** value2	'Value1' raised to the power of 'value2'
%	value1 % value2	The remainder of 'value1' divided by 'value2'
–	–value	Changes the sign of 'value'
+	string1 + string2	Concatenates 'string2' to the end of 'string1'

The operators can be combined in any order into compound statements. Compound statements will use standard order of operation logic. This means that power, modula, and negative operators will be executed first, multiplication and division is second, and addition and subtraction last. The user can include parenthesis to modify the order of operation. For example, (t1 + t2) * (t3 + t4).

Sample equations include:

```
J = J + 10
Y = A * X ** 2 + B * X + C
S2 = S1 + " MODE"
VMT = VMT + LINK.VOLUME * LINK.LENGTH
```

Logical Statements

In addition to arithmetic operators, a full set of logical operators is available. The operators can be defined using the syntax typically associated with 'C' and FORTRAN programming languages. The syntax can be mixed and matched in any way the user prefers. Parentheses are available for defining logical groups. The following describes the logical syntax:

Operator	Syntax	Result
&& AND	Logic1 && logic2 logic1 AND logic2	True if 'logic1' and 'logic2' are both True (or not zero)
|| OR	Logic1 || logic2 logic1 OR logic2	True if 'logic1' or 'logic2' are True (or not zero)
! NOT	!logic NOT logic	True if 'logic' is False (or zero)
== EQ	Value1 == value2 value1 EQ value2	True if 'value1' equals 'value2'
!= NE	Value1 != value2 value1 NE value2	True if 'value1' is not equal to 'value2'
<= LE	Value1 <= value2 value1 LE value2	True if 'value1' is less than or equal to 'value2'
>= GE	Value1 >= value2 value1 GE value2	True if 'value1' is greater than or equal to 'value2'
< LT	value1 < value2 value1 LT value2	True if 'value1' is less than 'value2'
> GT	value1 > value2 value1 GT value2	True if 'value1' is greater than 'value2'

When variables of different data types are used in arithmetic equations or logical statements, industry standard type conversion rules are used. If the statement includes decimal numbers, the arithmetic operators will use floating point methods. If a floating point statement is assigned to an integer variable, the decimal number is truncated to the next lower integer before the assignment is made.

In general, integer and decimal numbers cannot be combined with string variables in arithmetic or logical statements. The result of a logical statement can, however, be assigned to an integer variable. If the result is True, the integer will be set to 1. If the result if False, the integer will be set to zero. Conversely, a logical test can be based on a number or string. If the number is not zero, the condition is True. If the number is zero, the condition is False.

Sample logical statements include:

```
IF (J < 10) …
C1 = (HHOLD.INCOME < 10000)
IF (J GE 100 AND J LE 1000) …
WHILE (STAT) …
IF (!(D1.TAZ > 10 && D1.TAZ < 25) || J == 1) …
```

Conditional Processing

Logical statements are typically used to implement conditional processing. Two types of conditional processing are available within user programs. Branching procedures are implemented with IF-THEN-ELSE statements. Repetitive procedures are implemented with WHILE-LOOP statements. The software is designed to accommodate up to 100 nested IF statements and 10 nested WHILE statements. The logical keywords include:

Operator	Syntax	Result
IF	IF (logic)	If 'logic' is False (or zero), go to ELSE or ENDIF command
THEN	THEN …	End of logic statement and beginning of True statements
ELSE	… ELSE …	End of True statements and beginning of False statements
ENDIF	… ENDIF	End of True or False statements
WHILE	WHILE (logic)	If 'logic' is False (or zero), go to statement after ENDLOOP
LOOP or {	LOOP … or { …	End of logic statement and beginning of Loop statements
BREAK	BREAK	Go to statement after ENDLOOP command
ENDLOOP or }	… ENDLOOP or … }	End of Loop statements and go to WHILE command

The basic branching syntax is:

```
IF (logical_test) THEN [do-when-true] ELSE [do-when-false] ENDIF
```

The syntax can be provided on one line or multiple lines. The ELSE clause is optional, but an ENDIF is required for each IF statement. Up to 100 branching statements can be nested within each other to form complex conditional tests. Nested conditional processing would look like the following:

```
IF (t1 > 10) THEN
      IF (t1 > 15) THEN
            t2 = 15
      ELSE
            t2 = 10
      ENDIF
ELSE
      IF (t1 > 5) THEN
            t2 = 5
      ELSE
            t2 = 1
      ENDIF
ENDIF
```

The basic repetitive syntax is:

```
WHILE (logical test) LOOP [do-while-true] ENDLOOP
```

The syntax can be provided on one line or multiple lines. Note that the user is responsible for implementing the changes that will cause the logical test to eventually be False. In other words, the loop statements should set a status code or increment a counter that results in a False logic statement. The loop will continue as long as the logical test is True.

Up to 10 sets of repetitive statements can be nested within each other to form multi-dimensional loops. Each loop can include up to 10 BREAK statements to exit the loop without returning to the WHILE logic test. A BREAK statement, however, only has meaning if it is included within a branching statement. The following example demonstrates a nested loop with BREAK statements:

```
rec = 0

WHILE (1) LOOP
      rec = rec + 1
      stat = INPUT (A1, rec)

      IF (stat != 0) THEN BREAK ENDIF

      total = 0
      t1 = 1

      WHILE (t1 < 80) {
            data = READ (A1, t1, "%10", data)

            IF (data == 0) THEN BREAK ENDIF

            t1 = t1 + 10
            total = total + data

      }
      PRINT ("\n\tRecord %", rec)
      PRINT (" Total = %", total)
ENDLOOP
```

Arithmetic Functions

User programs also provide a number of arithmetic functions. These functions return a value that can be used in standard calculations or logic statements. In most cases the return value is a decimal number. These functions include:

Functions	Syntax	Result
MIN	MIN (value1, value2)	Minimum of 'value1' and 'value2'
MAX	MAX (value1, value2)	Maximum of 'value1' and 'value2'
ABS	ABS (value)	Absolute value of 'value'
SQRT	SQRT (value)	Square root of 'value'
EXP	EXP (value)	The exponential of 'value'
LOG	LOG (value)	The natural logarithm of 'value'
POWER	POWER (value1, value2)	'Value1' raised to the 'value2' power
LOG10	LOG10 (value)	Base 10 logarithm of 'value'
RANDOM	RANDOM ()	Random number between 0 and 1

A few example function applications are shown below:

```
C1 = MIN (C2, C3 - 30)
U1 = EXP (-0.0225 * OD1.TTIME + 1.2008)
IF (ABS (D1) > 100) …
PROB = RANDOM ()
R1 = LOG (OD1.TDIST)
```

Conversion Functions

In addition to arithmetic functions, the software provides a full set of type conversion functions. These functions typically convert data between string fields and integer or decimal numbers. They include:

Functions	Syntax	Result
ATOI	ATOI (value)	The string 'value' converted to an integer number
ATOF	ATOF (value)	The string 'value' converted to a decimal number
ITOA	ITOA (value)	The integer 'value' is converted to a string
FTOA	FTOA (value)	The decimal number 'value' is converted to a string
INT	INT (value)	'Value' truncated to an integer
FLOAT	FLOAT (value)	'Value' changed to a decimal number
ROUND	ROUND (value)	'Value' rounded to the nearest integer
SUBSTR	SUBSTR (string, i1, i2)	The substring of 'string' between characters 'i1' and 'i2'

A few conversion examples are shown below:

```
I1 = ATOI (string)
S1 = "NUMBER = " + FTOA (number)
J = INT (number + 0.5)
S = SUBSTR (string, 5, 10)
IF (ROUND (number) > 10) …
R1 = 10.5 + i1 / FLOAT (total)
```

Date-Time Functions

Date and time fields often require special processing for logic statements or conversion programs. User programs work with date and time data in "time stamp" format. A "time stamp" is the standard computer representation of a date and time. The data are stored as the integer number of seconds from January 1, 1970. Functions are provided to give the user access to the year, month, day, hour, minute, and second components of this data type. The following table describes the behavior of these functions:

Functions	Syntax	Result
DOW	DOW (time_stamp)	The day-of-the-week (0-6) from the 'time_stamp'
HOUR	HOUR (time_stamp)	Decimal hour (0.00-23.99) from the 'time_stamp'
MONTH	MONTH (time_stamp)	Integer month (1-12) from the 'time_stamp'
YEAR	YEAR (time_stamp)	Integer year (1970+) from the 'time_stamp'
DATE	DATE (year, month, day)	The time stamp equal to the date components
DATE_TIME	DATE_TIME (year, month, day, hour, minute, second)	The time stamp equal to the date and time components
DATE_STR	DATE_STR (time_stamp)	The 'mm/dd/yyyy' string equal to 'time_stamp'
TIME_STR	TIME_STR (time_stamp)	The 'hh:mm:ss' string equal to 'time_stamp'

A few date-time examples are shown below:

```
IF (YEAR (COUNT.DATE) == 1999) …
S1 = "DATE = " + DATE_STR (d1)
D1 = DATE (2003, 11, 25)
IF (DOW (d1) > 0 && DOW (d1) < 6) …
T1 = HOUR (d1) * 3600.0
```

Input-Output Functions

User programs provide a few input and output functions that enable the user to manipulate data files directly, generate reports, or display debug information. These functions are listed below:

Functions	Syntax	Result
PRINT	PRINT (format, value)	Send the formatted 'value' to the report file
LIST	LIST (format, value)	Send the formatted 'value' to the screen
FORMAT	FORMAT (format, value)	A string with the formatted 'value'
READ	READ (file, column, format, value)	The 'value' read from the ASCII 'file' starting at 'column' using 'format' and 'value' data type
WRITE	WRITE (file, column, format, value)	The column after 'value' is written to the ASCII 'file' starting at 'column' using 'format'
INPUT	INPUT (file, record)	The status code from reading 'record' from 'file'
OUTPUT	OUTPUT (file, record)	The status code from writing 'record' to 'file'

The "format" parameter in the input-output functions enables the user to control how multiple statements interact with one another. Each message adds to the end of the previous message until a new line command is included in the format string. The syntax of the format string generally follows the 'C' programming language for a "printf" statement. The primary difference is that the functions operate on only one variable at a time and the data type of that variable is supplied by the program or the user.

The placement of the data variable within the format string is defined with a percent sign (%). The data item may be preceded and followed by a text message. If the user wishes to control the size or format of the data item, the field width and the number of decimal places will follow the percent sign. The following shows a few sample output messages:

```
PRINT  ("%", t1)
PRINT  ("Record Number = %5\n", t0)
PRINT  ("\n\t%10.5", r1)
PRINT  ("\n\nStreet Name: %40 \nUser ID:", ArcView.StreetName)
```

The first message prints a single integer on the report file. The second message prints a message followed by a five-digit integer number and a new line command ("\n"). The new line command can be included at the end of the message, at the beginning of the message, or in the middle of the message. The third message shows the new line command at the beginning of the message followed by a tab command ("\t"). The decimal variable will be displayed using a maximum of ten digits and five decimal places. The fourth message uses double spaces before printing the label and 40 characters of a string variable. It then starts a new line and prints a label message in

anticipation of the next PRINT command. This message prints the variable called "StreetName" from a user-provided data file named "ArcView".

As implemented with the FORMAT command, string formatting can also be used to convert numbers to strings or truncate strings. For example:

```
S1 = FORMAT ("%10.1", R1)
S3 = FORMAT ("%10", S2)
```

The first example converts a decimal number to a string with one decimal point. The second example saves the first 10 characters from "S2" in "S3".

READ and WRITE are designed to process ASCII files that don't contain fields or a field header. These functions enable the user to read or write information from the data record using column positions and field widths. For example:

```
R2 = READ (A1, 20, "%10", R1) + 100
C1 = WRITE (A2, C1, "%,", R3)
```

The first example reads 10 characters starting at column 20 from a file named "A1", converts this data to a variable of type "R1", adds 100 to the result, and assigns the sum to the variable "R2". The second example writes the variable "R3" followed by a comma to a file named "A2" starting at column "C1". The number of characters written depends on the data type of "R3" and the value of "R3". The WRITE function returns the column number after the data are written to the record. This column number can be used in subsequent WRITE statements to append additional data to the end of the record.

INPUT and OUTPUT are designed to read and write records to a file. In most user program applications the main program controls the file input and output. The main program reads the records in the file one at a time and executes the user program to manipulate the data fields in the current record. When the execution is finished, the main program writes the results to the output file and reads the next record from the input file.

In certain situations, like unformatted ASCII files or multi-line custom reports, the user may want direct control over data input and output. The INPUT and OUTPUT functions enable the user to read and write records to the file. For example:

```
stat = 0
record = 0

WHILE (stat EQ 0) LOOP
      record = record + 1
      IF (INPUT (A1, record)) THEN BREAK ENDIF
      A2 = A1
      stat = OUTPUT (A2, record)
ENDLOOP
```

This example reads records from file "A1" until an end-of-file (non-zero return code) is found or the OUTPUT command returns an error (non-zero return code). Each record from file "A1" is read sequentially, copied to the data record for file "A2", and file "A2" is written to the disk.

Lookup Tables

The software also supports integer, decimal, and string lookup tables using the following syntax:

Functions	Syntax	Result
ITABn	ITABn (index)	the integer at 'index' in integer lookup table 'n'
RTABn	RTABn (index)	the number at 'index' in decimal lookup table 'n'
STABn	STABn (index)	the string at 'index' in string lookup table 'n'
ENDTAB	tTABn (max) = … ENDTAB	end lookup table initialization

Up to 10 sets of lookup tables can be defined. The tables are numbered 0 to 9 using keywords ITAB, RTAB, and STAB (e.g., ITAB2, RTAB5, STAB0). If the table keyword is specified without a number (e.g., ITAB), it is interpreted as table zero.

Lookup tables must be defined before they are used. The table is defined the first time the variable name is encountered. The user program interprets everything following the table name until an "ENDTAB" is reached as table initialization data. The initialization data may span any number of command records. The initialization syntax is:

```
tTABn (max) = x, y, z, … ENDTAB
```

where: `t` = table type (I, R, or S)
`n` = table number (blank, 0..9)
`max` = the maximum number of elements in the table
`x, y, z` = comma delimited table values of type 't' started at element 1.

The following example demonstrates how a table is defined and used:

```
ITAB1 (10) = 1, 2, 3, 4, 5,
             6, 7, 8, 9, 10  ENDTAB

TOTAL = 0
T1 = 0

WHILE (T1 < 10) LOOP
      T1 = T1 + 1
      TOTAL = TOTAL + ITAB1 (T1) * 10
ENDLOOP
```

Return Values

A user program returns to the calling program when a RETURN or END statement is encounters. The END statement terminates the process with a return value of zero. If the user needs to return information to the calling program, the RETURN function should be used. The result of a RETURN command is always an integer value. The syntax is:

RETURN (…)

Any type of calculation can be included within the parentheses, but the result will be interpreted as an integer.

Reports

The user program interface provides two types of printout reports. These are typically identified in the program report list as a "SCRIPT" report and a "STACK" report. A script report prints a copy of the commands sent to the compiler to the printout for the main program. These reports typically look like the following:

```
Household Type Script

      IF (Household.Persons > 2) THEN
           IF (Household.Persons > 3) THEN
                RETURN (13)
           ELSE
                IF (Household.AgeLT5 == 1) THEN
                        RETURN (12)
                ELSE
                      IF (household.Age5to17 == 1) THEN
                           RETURN (11)
                      ELSE
                           RETURN (10)
                      ENDIF
                ENDIF
           ENDIF
      ELSE
           IF (Household.Persons == 2) THEN
                IF (Household.AgeLT5 == 1) THEN
                     RETURN (6)
                ELSE
                     RETURN (5)
                ENDIF
           ELSE
                RETURN (3)
           ENDIF
      ENDIF
```

The stack report lists the compiled commands as they are executed by the user program. This represents sequence of steps the program uses to process the information. It looks similar to the

order of operation used by “reverse polish” calculators. The stack for the script show above is as follows:

```
Household Type Stack

 1) Integer      Household.PERSONS
 2) Integer      2
 3) Relation     GT
 4) Logical      If False, Jump to 29
 5) Integer      Household.PERSONS
 6) Integer      3
 7) Relation     GT
 8) Logical      If False, Jump to 12
 9) Integer      13
10) Return       Integer
11) Logical      Jump to 28
12) Integer      Household.AGELT5
13) Integer      1
14) Relation     EQ
15) Logical      If False, Jump to 19
16) Integer      12
17) Return       Integer
18) Logical      Jump to 28
19) Integer      Household.AGE5TO17
20) Integer      1
21) Relation     EQ
22) Logical      If False, Jump to 26
23) Integer      11
24) Return       Integer
25) Logical      Jump to 28
26) Integer      10
27) Return       Integer
28) Logical      Jump to 45
29) Integer      Household.PERSONS
30) Integer      2
31) Relation     EQ
32) Logical      If False, Jump to 43
33) Integer      Household.AGELT5
34) Integer      1
35) Relation     EQ
36) Logical      If False, Jump to 40
37) Integer      6
38) Return       Integer
39) Logical      Jump to 42
40) Integer      5
41) Return       Integer
42) Logical      Jump to 45
43) Integer      3
44) Return       Integer
45) End
```