SmartWare

Formula Reference

COPYRIGHT INFORMATION

34 St.Patrick Street • Suite 200
Toronto, Canada • M5T 1V1

Table of Contents

Chapter 1: Introduction to Formulas and Functions

This book explains how to create and use formulas with ANGOSS. ***Chapter 1*** provides an introduction to the "language" you will use to write formulas in ANGOSS. *Chapter 2* is a complete alphabetical reference to ANGOSS' built-in functions. ***Chapter 3*** describes how to use the statistical database functions available in ANGOSS Spreadsheet and ANGOSS Database. In the ***Appendices*** you will find a summary of formula error messages and recovery instructions, a table of keyboard keys and their ANGOSS terms, and a comparison of ANGOSS' functions with those of Lotus 1-2-3. For information on Voice Recognition functions, refer to the Voice Recognition documentation.

Formula Basics

A formula is a method of expressing instructions that describe how your computer should perform a specific calculation. When you write formulas in ANGOSS, you express these instructions in a language that ANGOSS understands. Each of the elements that you can use in this language is explained in detail in the following sections.

ANGOSS features powerful, advanced software technology for executing commands and calculating formulas. When you run ANGOSS, a complete calculation system is placed in your computer's memory, ready to accept and translate your commands and instructions. Because this system is memory resident, ANGOSS is exceptionally quick and responsive to your instructions. Also, because the calculation system is used by all modules, you can use the same formula language for all your applications.

You can use formulas in many ways in ANGOSS. These are some of the most important uses for formulas:

- Calculating and displaying results in Spreadsheet cells.
- Calculating formulas with the ANGOSS Calculator.
- Calculating formulas in the Text Editor and the Project Editor.

- Defining calculated fields in Database views.
- Defining query selection criteria in the Database Query Editor.
- Computing formulas typed in Word Processor documents.
- Calculating values and constructing expressions in ANGOSS' Project Development Language.

However you use ANGOSS, you will find that working with formulas increases your power and productivity.

Types of Formulas

Arithmetic and other mathematical calculations involving numbers are important applications for ANGOSS' formulas. However, formulas can also be used to instruct ANGOSS to perform a variety of other operations. Formulas can be classified in five categories, according to the type of data used in the calculation and the kind of operation performed in the calculation.

Formula Type	Type of data	Example
Numeric	Numbers, values, amounts	2 + 2
Text	Text, words, alphanumeric data	“Top” | “Flight”
Date	Dates	DAYNAME ("7/6/89")
Time	Dates and times	ADDMINUTES ("16:13:02", 4)
Logical	All types of data	IF Amount > 0 THEN Amount ELSE "Error"

Numeric. Numeric formulas specify calculations involving mathematical operations on numbers. ANGOSS provides a broad range of special functions for numeric formulas, including statistical, financial, and trigonometric. These functions enhance ANGOSS' ability to realize sophisticated engineering, business, and scientific applications quickly and easily.

Text. Text formulas allow you to process and control textual information. You can use text formulas to combine, split, alter, and manipulate words, phrases, and other strings of text data into useful forms.

Date and Time. ANGOSS can perform calculations on date and time information. Date and Time formulas use specially-formatted text and numeric data that represent specific dates and times. With ANGOSS' date and time functions you can calculate elapsed time and durations in minutes, hours, days, months, or years. You can also derive the day name of a particular date, add or subtract increments of time, and much more.

Logical. A logical formula is an instruction to examine items of data and render a decision. Logical formulas usually cause the system to choose among alternatives based on a specified set of conditions. These conditions are expressed as statements, called logical expressions, which are either true or false. Thus, a logical formula allows you to create "intelligent" applications that can provide for different responses to various situations.

The Elements of Formulas

Formulas are made up of **operators** and **expressions**. An operator specifies a particular operation that is to be performed on data. Expressions provide the data to be operated on. When combined into a properly written formula these two elements provide a complete description of how a calculation is to be performed.

Operators

An operator specifies either a relationship between two expressions or an operation to be performed upon some data. In ANGOSS formulas operators are represented by symbols. There are four categories of operators: numeric, text, relational, and logical.

Numeric Operators

Numeric operators perform mathematical operations on numeric expressions. These operators are used only with numeric expressions or expressions returning numeric data. The ANGOSS numeric operators are summarized in the following table.

Operator	Meaning	Example
+	Addition	1 + 1
-	Subtraction	2 - 1
*	Multiplication	2 * 2
/	Division	4 / 2
^	Exponentiation	2 ^ 3

Text Operators

Text operators perform an operation called text concatenation. Text concatenation is the combination of two text expressions into one. You can concatenate with or without a space between the two original expressions. The ANGOSS text operators are summarized in the following table.

Operator	Meaning	Example	Result
&	Concatenates text with a space separator.	"in"&"to"	"in to"
|	Concatenates text without a space separator	"in"|"to"	"into"

Relational Operators

Relational operators express a relationship between two numeric or text items. When you use a relational operator to describe such a relationship you are creating a logical expression. The ANGOSS relational operators are summarized in the following table.

Operator	Meaning	Example
=	Equal	1 = 1
<>	Not equal	1 <> 0
>	Greater than	1 > 0
<	Less than	0 < 1
>=	Greater than or equal to	1 >= 0
<=	Less than or equal to	1 <= 1

In this table the meaning of the relational operators is illustrated with numbers. In each example the operator causes the two values to be compared. All these examples are true. When you use relational operators with numeric expressions, ANGOSS determines equivalency to a precision of 14 significant digits.

These relational operators can also be used to form logical expressions with text. Text comparisons refer to the ASCII values of the characters.

Examples

"z" <> "Z" "a" < "z" “Ang” | “oss” = “Angoss”

You can use three additional relational operators with text. The operators ! and !! search the first expression for the set of characters in the second expression. The operator == converts all text to lower case for the purpose of comparing text without reference to case. The following examples are true.

Operator	Meaning	Example
!	Contains	"Hello" ! "lo"
!!	Does not contain	"Hello" !! "Hi"
==	Compare ignoring case	"HELLO" == "hello"

Logical Operators

Logical operators are used to connect logical expressions. A logical operator causes each of a pair of logical expressions to be evaluated and results in a value of TRUE (non-zero) or FALSE (zero) for the combination. The following table summarizes ANGOSS' logical operators (the examples are true).

Operator	Meaning	Example
AND	True if both logical expressions are true	"abc" ! "b" AND 1 > 0
OR	True if either logical expression is true	"abc" ! "d" OR 1 > 0

Operators and Priority

Formulas are generally evaluated from left to right. The order of evaluation of a formula may be different, however, depending on the priority of the operators and the use of parentheses.

An intrinsic priority is attributed to every operator by ANGOSS. Expressions are evaluated in an order of precedence according to the priority of the operators they contain. The following list shows the operators in order of priority.

Priority	Operator
1	^ (Exponentiation)
2	* and / (Multiplication and Division)
3	+ and - (Addition and Subtraction)
4	& and | (Text concatenation)
5	All relational operators
6	All logical operators

You can use parentheses to control the order of evaluation of a formula. Placing parentheses around an expression indicates that it is to be evaluated before other expressions in the formula.

NOTE: If parentheses are used within other parentheses (i.e., nested), the innermost level receives the highest priority.

For example, the formula `100 * 2 + 3` instructs ANGOSS to multiply 100 by 2 and then add 3. By inserting parentheses, you can change the formula so that the addition is performed before the multiplication: `100 * (2 + 3)`. The result of the first formula is 203; the second yields 500. The use of parentheses in this example alters the normal left-to-right evaluation of the formula

Expressions

An expression is an item of data in a form suitable for processing or calculating in ANGOSS. There are three general categories of expressions: numeric, text, and logical. Two additional categories of expressions, date and time, are specialized forms of the numeric and text expressions.

When calculated, most ANGOSS formulas return information as one of these types of expressions. Thus, formulas can serve as expressions within other formulas.

Numeric Expressions

A numeric expression is a form in which you must specify a number or a term that is to be interpreted as having a numeric value. A numeric expression can be in either decimal or hexadecimal notation

Decimal. Decimal numbers are numbers expressed in base ten. You can express decimal numbers in either normal or exponential notation. The following characters are valid in decimal numeric expressions.

Operator	Meaning	Example
Numerals: 0-9	Digits (numeric constants)	123
. or ,	Decimal	123.00
E	Exponent	1.23E+02
-	Negative	-123
+	Positive	+123

NOTE: The minus symbol used as a negative "operator" indicates that a numeric value is negative. The plus symbol indicates that a number is positive. In ANGOSS, numbers are assumed to be positive so the plus symbol is optional. When these operators might be confused with the numeric operators represented by the same symbols, you should place parentheses around the expression to clarify its meaning (see **Operators and Priority**)

Hexadecimal. Hexadecimal notation expresses numbers in base sixteen. Hexadecimal numbers do not include decimal fractions and cannot be expressed in exponential notation. In ANGOSS, hexadecimal numbers must be preceded by the characters **0x**. The following characters are valid in hexadecimal numeric expressions.

Character	Meaning	Example
Numerals: 0-9	Digits (dec. values 0-9)	0x10
Letters: A-F	Digits (dec. values 10-15)	0x1B

When working with numbers, you must distinguish between the form in which you express the number in a formula and the form in which you want the number to be displayed. Numeric expressions used in formulas must be expressed as a basic number or a formula that returns such a number. ANGOSS can interpret and display numbers as currency, percentages, dates, or times. Numbers can also be displayed with or without thousands separators or to a specified decimal precision. A variety of negative number formats is also available in the Spreadsheet. To have numbers interpreted or displayed in special formats, you will use the appropriate ANGOSS functions or format display features to control their appearance. The following examples show some typical display formats.

Numeric Exp.	Display Format	Appears As
100	Normal, precision 0	100
1000	Currency, precision 2	$1000.00
.5	Percent, precision 1	50.0%
-10 * 100	E-Notation, precision 0	-1.E+03

COMMENT: Although you can choose to display a number with a comma or another thousands separator, do not type the comma when entering the number.

Text Expressions

A text expression is one or more characters used literally as textual information. Text expressions are sometimes referred to as strings. Each character in a text expression represents itself, not a value, date, or other quantity. For example, the "D" character in the expression "James D. Kempton" represents the letter D (an initial in a name), not the value 0xD hexadecimal (13 decimal).

NOTE: If a text expression is used in a numeric formula it is treated as having the value 0.

COMMENT: Text expressions are sorted alphabetically; numeric expressions are sorted in numeric order.

Unless you are specifically prompted for text information, you must surround a text expression with quotation marks when you enter it. Quotation marks distinguish text from other types of expressions that may employ some of the same set of characters. For example, when a number is surrounded by quotation marks, it is treated as a string of numerals, not as a numeric value. Characters valid in text expressions include (but are not limited to) the following.

- The lower case and upper case letters of the alphabet.
- The numerals 0-9.
- Common punctuation and graphic symbols.

Text expressions are commonly used to store and process names, addresses, serial numbers and codes, zip codes, and telephone numbers. Here are some examples of text expressions.

"66215" "Ad astra per aspera" "$895.00"

NOTE: The quotation marks in these examples are not included in the text expressions. Rather, they delimit the expressions. To display quotation marks as part of a text expression, you must use double quotation marks at both the beginning and the end of the characters you want quoted. The double quotation marks are displayed as a single set of double quotation marks. For example, the expression "I said,""Eureka!""" results in a text expression that looks like *`I said, "Eureka!"`*.

Logical Expressions

A logical expression makes a statement comparing two or more things. A logical expression consists of two expressions joined by a relational operator. The two expressions are compared according to the rules of the operator involved (see ***Relational Operators***).

This kind of statement is evaluated as either true or false---that is, the comparison of the two expressions designated by the relational operator is either correct or incorrect. When a logical expression is calculated in a formula, it returns a non-zero (usually 1) numeric value if it is true. If it is false, it takes the numeric value zero. The following examples show simple logical expressions and their truth values.

10 > 9.9	TRUE
"Z" <> "z"	TRUE
10 * 10 = 100	TRUE
(1 >= 2) = 1	FALSE

Logical expressions can be joined with logical operators. The resulting compound logical expression is also either true or false, depending on the truth values of its parts and the rules of the operator used (see ***Logical Operators***).

10 > 9.9 AND 9.9 > 9.8	TRUE
"Z" <> "z" OR "Z" = "z"	TRUE
(1 >= 2) = 1 AND (2 > 1)	FALSE
(2 < 1 OR "5" = 5) AND 1 + 1 = 2	FALSE

Date Expressions

A date expression is a form in which you must specify a date so that it can be interpreted and calculated in a date formula. ANGOSS allows you the freedom to enter dates in virtually any form you prefer. These date expression forms are categorized into three types.

COMMENT: A number used as a date is interpreted to mean the number of days before or after December 31, 1899.

Type 1. Type 1 is a text expression in any form recognizable as a date. You can punctuate and abbreviate month and day names however you wish. If you specify a series of three numbers as a date, the `Date style` setting in the Global Preferences

menu determines the order of day, month, and year. If you use only two-digits for the year, the twentieth century is assumed; to work with dates in other centuries, you must specify all four digits for the year.

Examples

"September 15, 1989"	"Sep 15 89"	"Sept. 15th, 1989"
"07 06 52"	"07/06/52"	"7--6--1952"

Type 2. Type 2 is a single numeric expression representing a quantity of days before or after the base date, Dec 31, 1899. Positive numbers refer to dates after the base date; negative numbers refer to dates before. The following examples compare dates expressed as types 1 and 2.

Type 2	Type 1
32765	"September 15, 1989"
-3759	"September 15, 1889"
55705	"July 6, 2052"
0	"December 31, 1899"

COMMENT: ANGOSS allows you to use dates ranging from Jan. 1, 100 to Dec. 31, 9999 in formulas.

Type 3. Type 3 is a set of three numeric expressions separated by commas in the form: `yyyy, mm, dd`. If you use only two digits for the year, the twentieth century is assumed; to work with dates in other centuries, you must specify all four digits for the year. The following examples compare dates expressed as types 1 and 3.

Type 3	Type 1
89, 9, 15	"September 15, 1989"
2089, 9, 15	"September 15, 2089"
0, 12, 31	"December 31, 1900"

Time Expressions

A time expression is a form in which you must specify a time of day so that it can be used in a formula involving time calculations. A time expression is a text expression consisting of three two-digit numbers separated by colons in the form `HH:MM:SS`. The digits representing seconds are optional. If you want to express time according to a 24-hour clock, the hours digits can range from 00 to 24. If you prefer to work with a 12-hour clock, you can restrict the hours digits to the range 00 to 12 and include the letters "A" or "P" as the last characters in the expression to indicate AM or PM, respectively.

Examples

"06:00:21" "06:00:21A" "4:30P" "16:30"

Functions

A function is a special keyword that instructs the formula calculation system to perform a particular operation on data. You may think of functions as "prefabricated" formulas that perform useful activities with data. Functions can enhance your fluency in ANGOSS' formula language.

Pi, for example, is a value frequently used in geometrical calculations. It represents the ratio of the circumference of a circle to its diameter and is roughly equivalent to 3.14159265359. In ANGOSS formulas, you can use the function PI to replace this long string of numbers when you want to use pi in a formula (e.g., `PI * radius ^ 2`).

ANGOSS offers over 200 built-in formula functions. In addition, you can create your own functions through Project Processing. You can use your functions much like ANGOSS' built-in functions. First you must create a project file that names and defines your function. Then, whenever you want to perform a calculation that requires your function, you can load the project file that defines it (see ***Project Processing***)

Function Formats

To use a function in a formula you must follow its required "format" or syntax. ***Chapter 2*** provides a complete reference for the specific formats required by the ANGOSS functions. You may type a function name in either upper or lower case letters. Many functions

require arguments. Arguments are independent data that determine the value functions return when calculated.

Suppose you want to determine the average of several numbers. First you must add all these numbers. Then you must count how many numbers you have added and divide the sum by that count. The formula might look like this:

(57 + 62 + 8 + 21 + 10 + 13) / 6

A more efficient way to perform this calculation is to use ANGOSS' AVERAGE function. Your formula now looks like this:

AVERAGE(57,62,8,21,10,13)

The six numbers in parentheses are the AVERAGE function's argument. Unlike PI, a function that always returns the same value, AVERAGE assumes different values depending on its argument. AVERAGE performs three operations on its argument. First, it counts the items and then it sums them. Finally, it divides the sum by the count.

If a function is to operate properly it must have the correct number of arguments in the proper order and of the appropriate type. An argument is usually placed in parentheses following the function name. If a function requires multiple arguments they are usually separated by commas (see the note regarding comma decimal separators following ***Item Lists***)

Expressions and Arguments

Function formats vary according to the data and the operations involved. Most functions accept one or more of the expression types as arguments. These expression types are represented in the function reference in ***Chapter 2*** by the following abbreviations.

Abbreviation	Meaning
numeric	Numeric expression
text	Text expression
logical	Logical expression
date	Date expression

Abbreviation	Meaning
time	Time expression
expression	More than one type of expression

Optional Arguments. Some functions have optional arguments. In the function reference these arguments are placed within angle brackets (e.g., <***text***>). If an argument is optional, it is not required for the function to work properly, although it may alter the results returned by the function if it is supplied.

Item Lists. Some functions can take a set of one or more expressions as a single argument. This type of argument is called an item list and is represented in the function reference like this: ***item list***. The number of expressions or items in an item list can vary according to the data involved. Items are separated by commas.

The argument to the AVERAGE function in the previous example is an item list consisting of six numeric expressions. However, AVERAGE can calculate the average of varying quantities of items, not just sets of six. Functions, like AVERAGE, that use item lists automatically adjust to the number of items in their argument.

NOTE: In the Spreadsheet, when an item list includes a block of cells, blank cells within that block are ignored. However, if the item list includes an item that is a reference to a single cell and that cell is blank, it is interpreted as having a value of zero.

NOTE: If you use a comma as a decimal separator, you must use the combination of characters, space-comma-space, to separate multiple items or arguments. For example, in `AVERAGE(87, 3)` the system would recognize only one item, treating 87,3 as eighty-seven and three tenths. To enter the first item as 87, followed by 3 as a second item, you must enter the formula like this: `AVERAGE(87 , 3)`

Data References

Data references are terms that indicate the locations or addresses of specific data items. Data references in formulas allow you to perform calculations using information that you have entered and stored in an ANGOSS module's data files. When you include a data

reference in a formula, it points to the "object" containing the data that you want used in the calculation. Using data references effectively is the key to utilizing formulas in your ANGOSS applications.

Suppose that you have already entered the numbers you want averaged in some cells in a worksheet. The numbers are located in columns 10 through 15 of row 5. Instead of retyping these numbers in an item list argument to the AVERAGE function, you can refer to them with a data reference. Your formula would look like this:

AVERAGE(r5c10:15)

The data reference `r5c10:15` replaces the list of specific numbers in the argument. The formula now instructs ANGOSS to calculate the average of the values that it finds in the referenced cells. The data reference connects the formula to that particular set of cells. Thus, the formula no longer includes specific values (constants), which can yield only one result. Instead, it depends on the data stored in cells.

In a typical Spreadsheet "what if" application, the numbers involved in a calculation change often. Initially, you might place estimated values into the referenced cells. Then, as real data becomes available, you might enter new figures and recalculate to obtain a more precise or current result. The formula itself remains the same. You simply update the data stored in the referenced cells and recalculate. The referenced cells might actually contain other formulas depending on yet another set of data. By using data references to build relationships between your data you can create powerful applications.

Cell Reference. A reference to a single worksheet cell is a cell reference. A cell reference consists of the row and column coordinates of the cell in the form `rxcy`, where `x` is the row number and `y` is the column number. For example, `r3c20` refers to the cell at row 3 column 20. An alternative way to reference a cell is to assign it a name with the Sheet Name commands.

You can refer to cells in active worksheets other than the current worksheet. Called an external worksheet reference, this kind of cell reference consists of the name of the other worksheet, followed by a period, followed by the cell reference or name. For example `wrksht2.r50c1` refers to the cell at row 50 column 1 in the worksheet named "wrksht2."

When using cell references in formulas, make sure that the cells contain data of the appropriate expression type for your formula. Worksheet cells can contain text, numeric, date, and time data. Remember that date and time cells really contain numeric data but display it in date or time format. Date cells, for example, contain a number that represents a quantity of days before or after December 31, 1899. Cells can also contain formulas returning all these types of data.

Field Reference. A reference to a single view field is a field reference. A field reference consists of the number or title of a field surrounded by brackets (e.g., `[Last name]`).

You can refer to cells in active views other than the current view. Called an external view reference, this kind of field reference consists of the name of the view, followed by a period, followed by the field reference (e.g., `[mailst.Last name]`).

When using field references in formulas, make sure that the fields contain data of the appropriate expression type for your formula. ANGOSS Database allows you to create alphanumeric (text), inverted alphanumeric (text), numeric, date, and time fields. Remember that date and time fields really contain numeric data but display it in a date or time display format. Time fields, for example, contain a decimal fraction representing a time of day as a fraction of the whole day (0.5 is displayed as "12:00:00P").

Block Reference. A block reference is a reference to one or more view fields or contiguous worksheet cells. The reference to row 5, columns 10 through 15 in the AVERAGE example formula discussed previously is a worksheet block reference. It identifies six neighboring cells in a worksheet.

A worksheet block reference is similar to a cell reference, except that it includes the first and last row and/or column numbers separated by colons. For example, `r2:4c1:9` indicates the cells in rows 2 through 4 of columns 1 through 9. You can use names to reference blocks of cells, as well as single cells. External worksheet references to blocks are also possible.

Some Spreadsheet functions require blocks as arguments, even though the calculation involves only a single cell, the upper left or "home" cell of the block. For these functions, ANGOSS interprets a single cell reference as a block reference if you precede the reference with an exclamation point (e.g., !r5c10).

A Database block reference, called a field list, is similar to a field reference, except that it includes multiple field titles or numbers separated by semicolons. Field numbers indicate the input order sequence for the current view. For example, `[Name; Address; City; State]` refers to the fields titled Name, Address, City, and State and `[1;3;5]` denotes fields occurring first, third, and fifth in input order in the current view. You can use both titles and numbers in the same field list. External block references are also possible.

To indicate a sequence of consecutively numbered fields, you can use references to the first and last fields separated by a vertical bar. For example, `[Name|Zip]` identifies the fields Name and Zip and all fields ranging between them in input order.

Variable. A variable is a keyword that identifies a memory location you can use for temporary data storage. A variable can have virtually any name (up to 31 characters long) that uniquely identifies it. A variable can store either text or numeric data.

COMMENT: In ANGOSS' Project Development Language you must declare a variable before you can use it.

Variables are an important part of ANGOSS' Project Development Language (see the ***Project Processing Manual***) and have a variety of uses in application development. However, you can also create and use them in calculations while working interactively with the ANGOSS modules.

To "assign" or store data in a variable, you must calculate a formula using the LET function. You can use variables created in this way, as well as variables created through Project Processing, as terms in formulas, much like any other data reference. Variables created through LET formulas remain available in memory until the module is exited. You can give a variable a constant value (e.g., LET #rate = 0.25) or allow its value to be determined by the result of formulas (e.g., LET #commission = #sales * .25).

NOTE: ANGOSS provides a special type of notation for referencing specific characters stored in a text variable. An expression in the form *V[N]* denotes the Nth character in the variable V. This type of notation yields a numeric value, the ASCII value of the specified character. For example, if the variable $name contains "Washington", then *$name[2]* returns 97, the ASCII value of the second character, "a". You can use the function CHR to obtain the text character from its ASCII value

Array. An array is a list or table of data identified by the name of a variable. Arrays are like variables that contain multiple items of data. Each data item in an array is called an element and is identified by one or a combination of numbers, called a subscript. An array element can be used as an expression in formulas. However, an array can only be created by executing a project that defines it. Refer to ***Project Processing*** for information on defining and using arrays.

A reference to an array element is an expression in the form A[S], where A is the array's name and S is the subscript. The quantity of dimensions in the array determines the quantity of numbers in the subscript. Multiple numbers are separated by commas. For example, `#distance[2,3]` refers to the element located in the second position in the first dimension and the third position in the second dimension of the array #distance.

The notation mentioned in the previous note for referencing characters within text can also be used with array elements containing text. For example, `$address[2,1][2]` refers to the second character in the text expression located in the second position in

the first dimension and the first position in the second dimension of the array $address.

Array Handling Enhancements

These array enhancements are available in version 2.65 and higher

This section discusses the new array functions. Since the first argument in each of the four new array functions, ARRAYSORT, ARRAYFIND, ARRAYSIZE, and ARRAYRESIZE, must be an array pointer, you may want to review the section, ***Pointers to Arrays and Variables*** before continuing here.

Although the size and resize functions are easy enough to understand, the sort and find functions, when operating on multi-dimensional arrays, require some explanation. To do this, the following program is discussed in the next block of text. Note: this program, called array.pf2, can be found in the sample application's home directory.

The array.ptr Program:

```
GLOBAL fill_records()
GLOBAL print_sort()
GLOBAL print_find()
GLOBAL scrprint()

MAIN
LOCAL db[1,1] AP_db
LOCAL #direction #sort_key
LOCAL #type #find_on_field #found $data
     SCREEN CLEAR 0 15
     AP_db = ARRAYPTR( db[1,1] )
     fill_records( AP_db )

     #sort_key = 4
     #direction = 1
     ARRAYSORT( AP_db, #direction, 1, #sort_key )
     print_sort( AP_db )

     #type = 0
     #find_on_field = 4
     $data = "B"
     #found = ARRAYFIND( AP_db, $data, #type, 1, #find_on_field )
     print_find( AP_db, #found )
```

```
    MESSAGE "Press a key"
END MAIN

'****************************************
FUNCTION fill_records( AP_db )
LOCAL #record
    #record = 0

    #record = #record + 1 'RECORD 1
    ARRAYRESIZE( AP_db, #record, 4 )
    WRITEPTR( AP_db, #record, 1, #record )
    WRITEPTR( AP_db, #record, 2, "Glogs" )
    WRITEPTR( AP_db, #record, 3, "Fred" )
    WRITEPTR( AP_db, #record, 4, "C" )

    #record = #record + 1 'RECORD 2
    ARRAYRESIZE( AP_db, #record, 4 )
    WRITEPTR( AP_db, #record, 1, #record )
    WRITEPTR( AP_db, #record, 2, "Carver" )
    WRITEPTR( AP_db, #record, 3, "Tim" )
    WRITEPTR( AP_db, #record, 4, "B" )

    #record = #record + 1 'RECORD 3
    ARRAYRESIZE( AP_db, #record, 4 )
    WRITEPTR( AP_db, #record, 1, #record )
    WRITEPTR( AP_db, #record, 2, "Stern" )
    WRITEPTR( AP_db, #record, 3, "Marge" )
    WRITEPTR( AP_db, #record, 4, "A" )
END FUNCTION

'****************************************
FUNCTION print_sort( AP_db )
LOCAL #number_of_records #rctr
LOCAL #number_of_fields #fctr
    #number_of_records = ARRAYSIZE( AP_db, 1 )
    #number_of_fields = ARRAYSIZE( AP_db, 2 )
    FOR #rctr = 1 TO #number_of_records
        FOR #fctr = 1 TO #number_of_fields
            scrprint( #rctr, (#fctr*10)-9, READPTR(AP_db, #rctr, #fctr) )
        END FOR
    END FOR
END FUNCTION

'****************************************
FUNCTION print_find( AP_db, #found )
```

```
LOCAL #number_of_fields #fctr
    scrprint( 8, 1, "Found in index:" & str(#found) )
    #number_of_fields = ARRAYSIZE( AP_db, 2 )
    FOR #fctr = 1 TO #number_of_fields
        scrprint( 10, (#fctr*10-9), READPTR(AP_db, #found, #fctr) )
    END FOR
END FUNCTION

'*************************************
FUNCTION scrprint( #row, #col, $data )
    SCREEN PRINT #row+1 #col+1 0 15 STR($data)
END FUNCTION
```

The example program above demonstrates how array functions and pointers are used, including a two-dimensional sort and find. This two-dimensional array, db[1,1], is local to MAIN and is referenced thereafter only through the array pointer AP_db.

The first steps are to create and then fill the array. If you look at the fill_records function, you should see that the data is arranged like records in a data base file. Element [1,1] contains the first record's physical number and elements [1,2], [1,3], and [1,4] contain the remaining field data - the second record begins with the element [2,1] and so on.

Notice that the ARRAYRESIZE function is called to increment the size of the array's first dimension each time a record is added (a useful technique when used in a loop with unknown reiterations). Typically though, for efficiency, the array size would only be increased once every 100 records or so.

```
ARRAYRESIZE( AP_db, #record, 4 )
```

The resize function arguments are: the array pointer, the size of the first dimension determined by #record, and the size of the second dimension which, in this case, is fixed at four.

Data is written to each element with WRITEPTR. When complete, the array contains:

Elements	1	2	3	4
1	1	Glogs	Fred	C
2	2	Carver	Tim	B
3	3	Stern	Marge	A

Next, the array is sorted.

```
ARRAYSORT( AP_db, #direction, 1, #sort_key )
```

Here, arguments to the ARRAYSORT function are: the AP_db pointer, an ascending sort direction, the sort dimension, and the index position of the other dimension to determine the sort order (since the sorted dimension is set to one, the index position must be on the second dimension). Think of this as a key field. For instance, with #sort_key equal to four, the array becomes:

Elements	1	2	3	4
1	3	Stern	Marge	A
2	2	Carver	Tim	B
3	1	Glogs	Fred	C

If #sort_key is set to two, the outcome would be:

Elements	1	2	3	4
1	2	Carver	Tim	B
2	1	Glogs	Fred	C
3	3	Stern	Marge	A

The array is then printed to the screen. The most interesting aspect of the print_sort function is that ARRAYSIZE is used to determine the size of each dimension so that looping does not exceed the array bounds.

```
#number_of_records = ARRAYSIZE( AP_db, 1 )

#number_of_fields = ARRAYSIZE( AP_db, 2 )
```

After printing the sorted array, ARRAYFIND is used to search the array for specific data in the fourth field.

```
#found = ARRAYFIND( AP_db, $data, #type, 1, #find_on_field )
```

Arguments to the ARRAYFIND function are: the AP_db pointer, the search data, the search type (binary in this case), the return dimension, and the index position of the other dimension to search (again, since the return dimension is set to one, the index position must be on the second dimension). With search data equal to B and #find_on_field equal to four, the function returns index 2 of the first dimension.

Finally, the print_find function displays this index as:

```
2 Carver Tim B
```

With both ARRAYSORT and ARRAYFIND, the example program basically operates on the array's first dimension (in keeping with the data base format). However, these functions can also operate on the second dimension. You can visualize this in the same way by flipping the array sideways. Further, while three-dimensional arrays are harder to visualize, the same principles are simply extended.

Note that arrays can remain intact when changing modules under version 2.65 and higher. As with variables, the LOCK SYSTEM command works on arrays.

Pointers to Arrays and Variables

Array and variable pointers are available in version 2.65 and higher

Variable Pointers

The concept of pointers is as simple to understand as is the concept of variables. A variable contains data which is stored at a particular address. When using a variable, you don't need to know the actual address of the data or even anything about memory addresses. This is also true of pointers. The difference between a variable and its pointer is that a variable contains data while its pointer contains the address where that data is stored.

To get the address of a variable, the function VARPTR is used. In the following example, VARPTR returns the address of a variable called $var and places it in a second variable called VP_$var.

```
VP_$var = VARPTR( $var )
```

The variable VP_$var is now a pointer. We can find out the data contained in $var through its pointer by using the READPTR function.

```
READPTR( VP_$var )
```

This means that the following IF statement would be true.

```
IF $var = READPTR( VP_$var )
```

We can also assign a new value to $var by using the WRITEPTR function.

```
WRITEPTR( VP_$var, "new value" )
```

On the face of it, pointers would appear to be a needless complication. In fact, they can make your programs much less complicated and much more efficient.

Efficiency can be gained because variables may contain large amounts of data, such as long strings, while pointers contain only addresses. When variables are passed to a function, those

variables are duplicated which increases memory usage by the size of the data passed (and moving around large amounts of data can slow programs down). However, when pointers are passed to a function (known as "passing by reference"), memory usage is increased by the relatively small size of addresses. No copy of the data occurs. Instead, operations on the data are made directly through the READPTR and WRITEPTR functions.

This then brings up two of the most important reasons to use pointers. Because functions can operate directly on data, they can effectively return multiple values and reduce the use of global variables.

For instance, if you have a function that swaps the values of two strings, how would you return the swapped strings? You could use global or even public variables and change them directly, but keeping track of many global or public variables in large systems can be difficult. You could return a joined value of the strings, but they would then need to be separated in the calling function. In this case, the best approach is to pass the variables by reference.

```
swap( VARPTR($str1), VARPTR($str2) )
```

The swap function could then receive the pointers and change the data directly. For example:

```
FUNCTION swap( VP_$str1, VP_$str2 )
LOCAL $hold
    $hold = READPTR( VP_$str1 )
    WRITEPTR( VP_$str1, READPTR(VP_$str2) )
    WRITEPTR( VP_$str2, $hold )
END FUNCTION
```

Another way to look at pointers is that they allow local variables to be visible to other functions. There is one trap though. Like local variables, if a function creates a pointer, that pointer is cleared when the function is terminated (see Example #4 below).

Array Pointers

Array pointers are handled in the same way as variable pointers except that individual elements must be specified when reading or writing to them.

To get the address of an array, the function ARRAYPTR is used.

```
AP_$array = ARRAYPTR( $array[1] )
```

To find the data stored in the tenth element of this array, pass READPTR both the pointer and the element number.

```
$element10 = READPTR( AP_$array, 10 )
```

To change the data stored in the tenth element of this array, pass WRITEPTR the pointer, the element number, and the new data.

```
WRITEPTR( AP_$array, 10, "new data" )
```

Like variable pointers, array pointers provide similar benefits. But they're also useful in cutting down the number of parameters in a function or for allowing functions to effectively accept a variable number of parameters.

Suggested Naming Convention

In order to help keep track of pointers, it is suggested that a naming convention be adopted for variables that contain a pointer.

VP_ Is the recommended prefix for variable pointers.

AP_ Is the recommended prefix for array pointers.

Example #1 - Print a passed array

```
GLOBAL print_array()

MAIN
LOCAL arr[5]
    arr[1] = "A"
    arr[2] = "B"
    arr[3] = "C"
    arr[4] = "D"
    arr[5] = "E"
    print_array( ARRAYPTR(arr[1]) )
END MAIN

FUNCTION print_array(AP_arr)
    LOCAL #ct
    FOR #ct = 1 TO ARRAYSIZE(AP_arr,1)
        SCREEN PRINT #ct 1 15 0 READPTR(AP_arr,#ct)
    END FOR
END FUNCTION
```

Example #2 - Filling a passed array

```
GLOBAL fill_array()

MAIN
LOCAL arr[1]
LOCAL AP_arr
LOCAL #ct #size
    AP_arr = ARRAYPTR(arr[1])
    size = fill_array(AP_arr)
    FOR #ct = 1 TO #size
```

```
        SCREEN PRINT #ct 1 15 0 arr[#ct]
    END FOR
END MAIN


FUNCTION fill_array(AP_arr)
LOCAL #ct
    ARRAYRESIZE( AP_arr, 10 )
    FOR #ct = 1 TO 10
        WRITEPTR( AP_arr, #ct, STR(#ct) )
    END FOR
    RETURN 10
END FUNCTION
```

Example #3 - Returning Multiple Values

```
GLOBAL get_person()

MAIN
LOCAL $name $phone #age
    get_person( VARPTR($name), VARPTR($phone), VARPTR(#age) )
    MESSAGE $name & $phone & STR(#age)
END MAIN

FUNCTION get_person(VP_name,VP_phone,VP_age)
        WRITEPTR( VP_name, "Warren Piece" )
        WRITEPTR( VP_phone, "555-1122" )
        WRITEPTR( VP_age, 29 )
END FUNCTION
```

Example #4: Incorrect Pointer Usage

```
GLOBAL error_example()

MAIN
LOCAL VP_badidea
    VP_badidea = error_example()
    MESSAGE READPTR(VP_badidea)
END MAIN

FUNCTION error_example()
’Don’t do this because the variable $str is local and will be
’destroyed when the function exits"
LOCAL $str
    $str = "error"
    RETURN VARPTR($str)
END FUNCTION
```

DDE Access

Dynamic Data Exchange (DDE) is a Windows feature that facilitates communication between two active Windows applications. This communication can take the form of both simple data sharing and sending commands to each other. Note however that not all Windows applications support DDE.

When an application initiates a DDE conversation, that application is said to be the "client". The application accepting the DDE link is referred to as the "server". This is similar to a modem connection where the machine that places the call might be called the "client" and the machine that answers the phone is its "server".

Note that you can use the SPL function CREATE_PROCESS to start another Windows application before initiating a DDE conversation.

The ANGOSS DDE Client

Initiating a DDE Conversation

Since more than one DDE conversation can be established at a time, a unique channel number is used in subsequent exchanges between a given client/server connection. In ANGOSS SPL, the client initiates a DDE conversation through the DDE_INITIATE function by passing it a server name and a "topic". If the server responds, a non-zero channel number is returned. Otherwise, zero indicates that the server doesn't exist or that it cannot converse on the given topic.

```
#channel = DDE_INITIATE( "server_name", "topic" )
```

Valid server names and topics are dependant on the Windows applications you have installed. In many cases, the server name is the application's name and the topic is a specific file such as a worksheet or document (e.g., letter.doc).

NOTE: Some applications have a special topic called "system" which can be used to determine the topics supported by that application. Typically, the item requested in this case is "topics" (see the DDE_REQUEST information below on requesting data for an item).

Retrieving Data from the Server

Once a connection is established, the client can retrieve information from the server through the DDE_REQUEST function.

```
$string = DDE_REQUEST( #channel, "item" )
```

The channel number identifies the client/server connection (recall that it was returned by the DDE_INITIATE function). The item parameter is application specific: it may be a worksheet

cell or range, perhaps a text block from a document - whatever the server application considers a valid item.

If the request function is successful, a string corresponding to the value of the item is returned. Otherwise, a numeric error code is returned (refer to the section DDE Error Codes).

Sending Data to the Server

Data can be sent to the server through the DDE_POKE function.

```
DDE_POKE( #channel, "item", "data" )
```

As in the DDE_REQUEST function, the channel number identifies the connection. The item is again application specific however, in this case, the item is the location that the data is sent to. If the poke function is unsuccessful, a numeric error code is returned.

Executing a Command in the Server

While the DDE_REQUEST and DDE_POKE functions are used to pass data back and forth, the server can also be made to execute a command.

```
DDE_EXECUTE( #channel, "command_string" )
```

The command is sent as string. It is entirely up to the server to determine whether or not the command string is valid. If it is not, or if some other error occurs, DDE_EXECUTE will return a numeric error code.

Note that many applications require that square brackets be placed around the command itself but within the quotes (i.e., "[command_string]").

Terminating the Connection

When a DDE connection is no longer needed, it can be terminated through the DDE_TERMINATE function.

```
DDE_TERMINATE( #channel )
```

Cold Links, Warm Links, and Hot Links

As you can see, the DDE client is simple to use. In fact, the most difficult aspect of using DDE is that you must be familiar with the structures and command syntax of both applications.

So far, the discussion has looked at DDE in basically a one-way direction: from the client to the server. This is known as a "cold link". The following is a simple example of this (it tells the Program Manager to create a new program group and add two applications to it).

Cold Link Example:

```
MAIN
LOCAL #pm_chan
    #pm_chan = DDE_INITIATE( "PROGMAN", "PROGMAN" )
    DDE_EXECUTE( #pm_chan, "[CreateGroup(RAD Apps)]" )
    DDE_EXECUTE( #pm_chan, "[AddItem(Customers)]" )
    DDE_EXECUTE( #pm_chan, "[AddItem(GL)]" )
    DDE_TERMINATE( #pm_chan )
END MAIN
```

It is possible, however, to have the server notify the client that the value of a particular item has changed. This is known as a "warm link" because the server communicates back to the client. It's a little more complicated to use than a "cold link" since the client must inspect the standard event queue to discover the server's notification.

A "warm link" is accomplished with the DDE_ADVISE function.

```
DDE_ADVISE( #channel, "item" )
```

DDE_ADVISE tells the server to notify the client when the given item has changed. The item must be valid. Note that any number of these advise statements can be issued so that multiple items can be monitored.

```
DDE_ADVISE( #channel, "item1" )

DDE_ADVISE( #channel, "item2" )

DDE_ADVISE( #channel, "item3" )
```

When the value of an item changes in the server, the ANGOSS client will be handed a {DdeAdvise} event. The identity of the item can then be extracted by the ***EVENTINFO*** function.

Warm Link Example Code Fragment:

```
WHILE TRUE
    CASE INEVENT
    WHEN {DdeAdvise}
        #chan = EVENTINFO( m_dde_handle )
        $item = EVENTINFO( m_dde_item )
        $newval = DDE_REQUEST( #chan, $item )
        EXIT WHILE
    END CASE
END WHILE
```

In the above example, the client inspects the event queue using ***INEVENT*** and then, when a {DdeAdvise} event is found, extracts the channel number and item string with two

EVENTINFO calls. Note that EVENTINFO in both places can be replaced with the equivalent lines:

```
#chan = DDE_CHANNEL

$item = DDE_ITEM
```

The advise status for any item can be shut off with the DDE_UNADVISE function.

```
DDE_UNADVISE( #channel, "item" )
```

A "warm link" differs from a "hot link" in that the server simply sends the notification. It is up to the client to act on that notification. In the "hot link" scenario, the server actually takes control of the client.

NOTE: ANGOSS does not support the "hot link" concept.

The ANGOSS DDE Server

Setting ANGOSS up as a DDE server is slightly more difficult and demanding than is a client. All link information for the server comes in through the event queue. Essentially, the ANGOSS server spins around in an SPL loop, inspecting the queue with INEVENT and acting on requests sent by the client. It must be able to service these events in a reasonable length of time - failure to do so may result in time-outs at the client.

Acknowledging the Client

Creating an ANGOSS server begins with the DDE_ACCEPT function.

```
#channel = DDE_ACCEPT( "server_name", "topic" )
```

Here, the given server and topic names have been announced to the rest of the computer.

There is no restriction on the contents of these strings except that they cannot be null; try to select a server and topic name that "makes sense" and is, hopefully, not already in use. The recommended naming scheme is to use the name of the application as the server and a fundamental input file (e.g., document) as the topic.

The channel number returned by DDE_ACCEPT cannot be used for any DDE I/O until a connection from a client is received. The server is notified of such a connection by the {DdeInitiate} event.

Acting on the Client's Request

Once a connection has been made, the server can expect to receive {DdeRequest} events from the client. The item string is bound to the event and can be extracted by using the ***EVENTINFO*** or DDE_ITEM functions.

```
$item = EVENTINFO( m_dde_item )

$item = DDE_ITEM
```

The server can then respond by sending new data back to the client with the DDE_DATA function.

```
DDE_DATA( #channel, $item, "send data" )
```

If the server cannot provide a value for the requested item, DDE_ERROR should be executed.

```
DDE_ERROR( #channel )
```

When the client is finished with the channel, it will issue a terminate. The server is then notified with a {DdeTerminate} event. The server can also terminate the channel with the DDE_TERMINATE function.

Warm Links

The concept of "advise" also exists for the SPL server, however, it is used somewhat differently. After a channel has been established, the server can define any number of items for "warm link" event processing. A DDE client may only request notification of change on those items - that is, those items must be pre-defined.

```
DDE_ADVISE( #channel, "item1" )

DDE_ADVISE( #channel, "item2" )
```

If a client requests it, the server will be given a {DdeAdvise} event, notifying it that a particular item has been put on alert. A similar event, {DdeUnadvise}, will be given when the client removes the advisory.

In order to maintain some simplicity at the SPL level, however, most DDE servers can probably ignore the advise/unadvise events and instead take the following approach.

Server Example:

```
FUNCTION dde_server()
LOCAL #chan $item #req $hold
    'start the server
    #chan = DDE_ACCEPT( "angossdb", "file" )
    'place items on advise list
    DDE_ADVISE( #chan, "field1" )
    DDE_ADVISE( #chan, "field2" )

    WHILE TRUE
        CASE INEVENT
        WHEN {DdeInitiate}
```

```
    ' handle DDE connect - in most cases, just ignore this event.

WHEN {DdeRequest}
    ' DO NOT ignore this event - at least, acknowledge via DDE_ERROR
    #req = DDE_CHANNEL
    $item = DDE_ITEM
    CASE $item
    WHEN "field1"
        DDE_DATA( #req, $item, [field1] )
    WHEN "field2"
        DDE_DATA( #req, $item, [field2] )
    OTHERWISE
        DDE_ERROR( #req )
    END CASE

WHEN {DdeAdvise}
    ' ignore the DDE advise

WHEN {DdeUnadvise}
    ' ignore DDE unadvise event

WHEN {DdeTerminate}
    ' handle termination by re-starting the server
    #chan = DDE_ACCEPT( "angossdb", "file" )
    DDE_ADVISE( #chan, "field1" )
    DDE_ADVISE( #chan, "field2" )

OTHERWISE
    ' handle other possible events (eg. keystrokes)
    ' in this case, we just shutdown and exit
    DDE_TERMINATE( #chan )
    RETURN

END CASE

' insert other DDE server processing here

' finish the loop with the "advise" item recomputation
$hold = [field1]
change_field( "field1" )
IF [field1] <> $hold
    DDE_ADVISE( #chan, "field1" )
END IF
$hold = [field2]
change_field( "field2" )
```

```
        IF [field2] <> $hold
            DDE_ADVISE( #chan, "field2" )
        END IF

    END WHILE
END FUNCTION
```

With each main loop, new values for "field1" and "field2" are computed. If they've changed from previous values, new "DDE_ADVISE" functions are executed.

This has the effect that, if the client has asked to be notified of the change, the low-level drivers inside ANGOSS will automatically generate {DdeAdvise} messages for the DDE client. The client will, in kind, typically respond with {DdeRequest} messages.

More sophisticated servers can use the Initiate/Advise/Unadvise messages to conditionally compute potentially expensive item values.

Current Limitations

The DDE server does not support the DDE_POKE or DDE_EXECUTE commands from clients. They are presently not acknowledged at the driver-level inside ANGOSS and the SPL is unaware of them.

Both the client and server only transfer string information.

Summary of Functions

Client Functions

DDE_INITIATE

DDE_REQUEST

DDE_POKE

DDE_EXECUTE

Server Functions

DDE_ACCEPT

DDE_DATA

DDE_ERROR

Client/Server Functions and Channel Control

DDE_ADVISE

DDE_UNADVISE

DDE_ITEM

DDE_CHANNEL

DDE_TIMEOUT

DDE_TERMINATE

DDE Error Codes

The following list shows the return values for the DDE_ functions.

Returns	Description
0	In general, a 0 return indicates that things are OK (except for the DDE_INITIATE, DDE_REQUEST, DDE_ACCEPT, DDE_ITEM, and DDE_CHANNEL functions since these return other relevant values.
-1	The server or client did not accept the last request or execute string or similar error.
-2	The server/client was busy doing something else. Possible if it is servicing more than one client.
-3	Invalid channel id.
-4	Out of memory.
-5	The server/client timed out during your last request.

Menu Functions

The MNU_ series of functions are used to manipulate and examine data structures used to store menus. These data structures can be used to generate pulldowns and/or tool bars with the

PULLDOWN INIT and TOOLBAR INIT commands. They may also be used if you are writing programs to display menus in your own style. RAD applications use mnu_ functions.

Briefly, these are the mnu_ functions:

Function	Description
MNU_OPEN	Creates a new menu or sub menu, or loads a menu from disk..
MNU_INSCH	Inserts a choice.
MNU_INSLANG	Inserts a language.
MNU_INSUM	Inserts a usermode (user type).
MNU_INFO	Changes or reads information.
MNU_DELCH	Deletes a choice.
MNU_DELLANG	Deletes a language.
MNU_DELUM	Deletes a usermode.
MNU_WRITE	Saves a menu to disk.
MNU_CLOSE	Release memory used by a menu.

Because menus are hierarchical data structures, each menu has a root menu. Items on a menu may be either a choice or a sub menu. The data structures themselves allow for multilingual use: textual data shown to the end user is indexed by a language ID. Menu data structures also index some data by user types.

All operations, such as inserting, deleting or modifying choices, affect a menu object that is loaded in memory. ***MNU_OPEN***(<file>) and ***MNU_WRITE*** are the only MNU_ functions that interact with the storage device.

Note that ascii menus can be edited manually. Programmers can create ascii menus and use them in their own SPL code. The mnuc program can be used to compile or decompile menus (use the mnuc -x flag for a list of options).

ODBC Access

Since version 2.65, SmartWare has had the ability to access ODBC databases, allowing developers to use any ODBC supported database. A set of SPL functions provides developer support for ODBC access. See also the ***Database*** manual for ODBC server information.

Database Terminology

Of the Structured Query Language (SQL) database terms described below, several are similar to those used by the SmartWare database module but should be understood in a different context.

Database

A collection of information accumulated to meet a particular need. A database consists of one or more "tables".

Tables

The files in which data is stored. The information is arranged in a tabular format of columns and rows. In contrast, SmartWare refers to these files as "data-files".

Columns

The structural components of a table, that is, the components that store specific categories of information. Some database systems refer to columns as "fields".

Rows

The sets of data items that populate a table. Some database systems refer to rows as "records".

Data Types

The categories of information that can be stored in columns. There is one "string" data type and several numeric data types for storing integers and floating point numbers.

Query

A request submitted to a database management system about the contents of a database. The database management system responds by retrieving appropriate rows of data.

SELECT Statement

An SQL command used to perform a query. A SELECT statement must contain a SELECT clause, indicating the columns to select, and a FROM clause, indicating the

tables to search. Among others, it may also contain a WHERE clause specifying selection conditions, an ORDER BY clause specifying sort conditions, and various other clauses.

Joining

Method of retrieving data from several tables as if they were components of a single table. Joining is accomplished by linking columns that contain similar data. The linking columns are called "join columns".

Database Management System

A database management system (DBMS) is comprised of the components necessary to create, maintain, and manipulate a database. The part of the DBMS that receives SQL statements, retrieves data from a database, and returns data to an application is called a "database engine" or "database server".

SPL ODBC Functions

Developer access to ODBC is delivered by the DBC_ series of functions which is available in all modules.

Connecting and Disconnecting

The first step in accessing an SQL database is to call the DBC_CONNECT function. Its complement function, DBC_RELEASE, is used to terminate the connection.

Reading Data

To read data from an SQL database, a "list" is created in memory by issuing an SQL SELECT statement through the DBC_SELECT function. This "list" is essentially made up of one or more columns (somewhat analogous to fields) and one or more rows (records), however, the "list" may actually be drawn from multiple SQL tables.

A "cursor" then points to a position before the first row in the "list". Moving this "cursor" and reading data is accomplished with the DBC_FETCH, DBC_CURRENT, and DBC_EOF functions. In general, the "cursor" can only be moved to the next row, however, because some relational databases allow movement to the previous row, the DBC_FETCHP function may be used.

Writing Data

The three DBC_TRANS functions are used in the process of writing data to the SQL database. Note that these functions are not related to the "list" created by the DBC_SELECT function described above.

By passing an SQL command, the DBC_SQL_EXEC function is used to actually write the data. While the syntax of the command is dependant on the SQL software, typically the commands are: INSERT INTO, UPDATE, and DELETE.

The DBC_SQL_EXEC function can also be used to perform any SQL commands not already covered by the DBC_ series of functions.

Information Retrieval Functions

The DBC_DBTYPE function returns the SQL database type. Because it is the actual database that we are inspecting as opposed to the "list", a "cursor" returned by the DBC_SELECT function is not a required parameter.

The DBC_NUM_COLS, DBC_GET_COLNAMES, and the DBC_COL_INFO functions are used to retrieve information about the "list" created by the DBC_SELECT function. This information includes the number of columns, as well as, the column names, types, and width.

SQL Example:

```
FUNCTION read_db()
LOCAL #db_handle #cursor_handle
    #db_handle = DBC_CONNECT(1,256,"my_db")
    ' Assumes DB open
    #cursor_hnd = DBC_SELECT( #db_handle, "select * from my_db" )
    WHILE NOT( DBC_EOF(#db_handle) )
        MESSAGE DBC_FETCH( #cursor_hnd, "id" )
    END WHILE
END FUNCTION
```

Chapter 2: Function Reference

Function Lists by Group

A function's group identifies the type of formula the function is used to create. ANGOSS provides functions for a broad range of uses, including Business / Financial, Statistical, Trigonometric, and Random.

Business/Financial

Database

Tabular

Date

Graphics

Logical

Miscellaneous

Numeric

Project Processing (General)

Project Processing (Array Handling)

Project Processing (DDE)

Project Processing (Menu)

Project Processing (ODBC)

Project Processing (Pointers)

Random

Spreadsheet

Statistical Database

Statistical

Text

Time

Transcendental

Trigonometric

Word Processing

Address Functions

The following functions return either 0, 16 bit addresses, or 32 bit addresses when executed in the specified mode. Three protect mode functions complement their respective real mode counterparts::

Function or Command	286 Mode	386 Mode	Unix	Comment
OFFSETOF	16	16	32	Real memory
SEGMENTOF	16	16	0	Real memory
POINTEROF	32	32	32	Real memory
OFFSETOFPM	16	32	32	Protected memory
SEGMENTOFPM	16	16	0	Protected memory
POINTEROFPM	32	32	32	Protected memory
DOSOFFSET	16	32	32	
DOSSEG	16	data segment	0	
DOSPTR	32	32	32	

Alphabetical Listing of Functions

This section contains an alphabetical listing of ANGOSS functions. Each function is accompanied by a description and an explanation of its syntax or **Format**. For many functions, examples are provided to show usage in a formula and what result that formula generates. **OS** indicates the operating system that the function will run under. The **Scope** of a function indicates the modules in which the function is available. Most functions are available in all the ANGOSS modules. A few functions are useful in only one of the modules. **Available** indicates the version of SmartWare that can use the function - if this entry does not exist, the function is available in all versions of the software. **Returns** designates the type of data (e.g., text, numeric) that results when the function is calculated. Some functions return different types of data depending on their arguments; these functions are identified as returning "Conditional."

Some functions have optional arguments. In the function reference these arguments are placed within angle brackets (e.g., ***<text>***).

Unlike Lotus 1-2-3, ANGOSS allows but does not require an "@" prefix with many function names. Numerous ANGOSS functions have names matching (except for the @ sign) their equivalent 1-2-3 functions. For like-named functions that differ in result between ANGOSS and 1-2-3, the @ preceding the name distinguishes the two functions. Refer to ***Appendix C*** for a comparison of ANGOSS and Lotus 1-2-3 functions.

Some functions, such as a number of the ANGOSS Spreadsheet Statistical Database functions, require the @ sign.

The following section is an alphabetical listing, with input and usage information, of the functions available in ANGOSS.

Some functions are specific to an operating system (i.e., DOS or Unix) or work differently. This is indicated by the OS listing. Further information on differences is provided in the body of the function description.

ABS

Format: ABS(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

ABS calculates the absolute value of the numeric expression.

Formula	**Result**
ABS(11)	11.0
ABS(--3.3)	3.3:

ACOS

Format: ACOS(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

ACOS calculates the arccosine of the numeric expression. The numeric expression must have a value between --1 and 1, inclusively. The result is a value expressed in radians between 0 and pi.

Formula	**Result**
ACOS(.11)	1.460573

ADATE

Format: ADATE(***date***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date

ADATE converts the date expression into a text expression of the form "month dd, yyyy." If the year is represented by two digits, the twentieth century is assumed.

Formula	**Result**
ADATE("07-06-88")	July 6, 1988
ADATE("09/15/3888")	September 15, 3888

ADDDAYS

Format: ADDDAYS(***date, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date

ADDDAYS returns a date representing the date expression plus the number of days in the numeric expression. If the numeric expression is negative, the result is that number of days subtracted from the date expression.

The first argument must be a date expression type 1 or 2. Type 1 is a text expression representing a date. Type 2 is a numeric expression representing a number of days before or after the beginning of the current century.

Formula	**Result**
ADDDAYS("07-06-52",14)	07-20-52
ADDDAYS("Sept 15, 1984",--28)	18 Aug 84

The date returned will be in the setting defined for the Date1 format in Tools Preferences Global.

ADDHOURS

Format: ADDHOURS(***date or time, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date or time

ADDHOURS adds the number of hours represented by the numeric expression to the time or date indicated by the time or date expression. If the numeric expression is negative, the result is that number of hours subtracted from the date or time.

The first argument for ADDHOURS may be either a date or a time expression. When a date expression is used, ADDHOURS returns a date; the argument must be a date expression type 1 (text) or type 2 (numeric). When a time expression is used as the first argument, a time expression is returned.

Formula	**Result**
ADDHOURS("12/24/84",36)	12/25/84
ADDHOURS("06:13:21P",4)	10:13:21P

ADDMINUTES

Format: ADDMINUTES(***date or time, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date or time

ADDMINUTES adds the number of minutes represented by the numeric expression to the time or date indicated by the time or date expression. If the numeric expression is negative, the result is that number of minutes subtracted from the date or time.

The first argument for ADDMINUTES may be either a date expression or a time expression. When a date expression is used, ADDMINUTES returns a date; the argument must be a date expression type 1 (text) or type 2 (numeric). When a time expression is used as the first argument, a time expression is returned.

Formula	Result
ADDMINUTES("16:13:21",41)	16:54:21
ADDMINUTES("Sept 15, 84",21220)	29 Sep 84

ADDMONTHS

Format: ADDMONTHS(***date, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date

ADDMONTHS returns a date representing the date expression plus the number of months in the numeric expression. If the numeric expression is negative, the result will be that number of months subtracted from the date expression. The argument must be a date expression type 1 (text) or type 2 (numeric).

Formula	Result
ADDMONTHS("12-24-84",16)	04-24-86
ADDMONTHS("08/24/57",--7)	01-24-57

ADDSECONDS

Format: ADDSECONDS(***date or time, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date or time

ADDSECONDS adds the number of seconds represented by the numeric expression to the time or date indicated by the time or date expression. If the numeric expression is negative, the result will be that number of seconds subtracted from the date or time.

The first argument for ADDSECONDS may be either a date expression or a time expression. When a date expression is used, ADDSECONDS returns a date; the

argument must be a date expression type 1 (text) or type 2 (numeric). When a time expression is used, a time expression is returned.

Formula	Result
ADDSECONDS("09/15/84",100350)	09/16/84
ADDSECONDS("06:13:59A",221)	06:17:40A

ADDYEARS

Format: ADDYEARS(***date, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date

ADDYEARS returns a date representing the date expression plus the number of years in the numeric expression. If the numeric expression is negative, the result will be that number of years subtracted from the date expression. The argument must be a date expression type 1 (text) or type 2 (numeric).

Formula	Result
ADDYEARS("July 6, 1952",36)	06 Jul 88
ADDYEARS("06/02/88", --74)	06/02/14

APINFO

Format: APINFO(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Conditional

APINFO returns a variety of information about the current application, based upon the named constant specified in the argument. Valid argument entries and their return values are as follows:

Name	Returns
ap_zoom	1 (TRUE) if the current window is zoomed; 0 (FALSE) if the current window is not zoomed
ap_border	1 (TRUE) if the current window has a border; 0 (FALSE) if the current window has no border
ap_window	The current window number
ap_file	The name of the file displayed in the current window; no extension or path is included
ap_filex	The name, including extension, of the file in the current window; path is not included
ap_filep	The name, including path and extension, of the file in the current window
ap_paper	The name of the currently selected paper profile

In the following examples, APINFO first returned the name of the currently displayed file with no extension. In the second example, APINFO returned the file name and extension.

Formula	Result
APINFO(AP_FILE)	cust
APINFO(AP_FILEP)	cust.vw

ARRAYFIND

Format: ARRAYFIND(***pointer***, ***num/txt***, ***num*** <,***num***> <,***num***> <,***num***>)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Conditional

ARRAYFIND searches the elements of an array for specific data. The search may be binary or sequential. If a match is found, an array index position is returned otherwise, zero is returned.

```
ARRAYFIND( AR_ptr, data, type, dim, index1, index2 )
```

The parameters in the above example are:

Parameter	Explanation
AR_ptr	Pointer to the searched array. Refer to the function ***ARRAYPTR***.
data	Numeric or string value to search for.
type	Search type. To use a binary search, pass 0. To use a sequential search, pass the element starting position.
dim	The dimension to be searched. Used only when the array has more than one dimension. An index position along this dimension is returned when a match is found.
index1	Index position of the dimension not being searched (i.e., a dimension other than dim). Used only when the array has more than one dimension. Its maximum value is the number of elements along this dimension. For example, in array[10,20] where the search dimension is 1, index1 cannot be greater than 20.
index2	Index position of the second dimension not being searched. Used only when the array has three dimension. Note that the dimension referred to by index2 is greater than that of index1. For instance, if dim equals 2, then index1 refers to the first dimension and index2 refers to the third.

For a discussion on this function's usage, refer to ***Array Handling Enhancements***.

ARRAYRESIZE

Format: ARRAYRESIZE(***pointer***, ***numeric*** <,***numeric***> <,***numeric***>)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Nothing

This redimensions an array. Unlike REDIMENSION, if the array size is enlarged, data already in the array will not be destroyed. The first parameter is a pointer to an array (refer to the function ARRAYPTR). The second parameter specifies the size of the first dimension. If required, the third and fourth parameters specify the size of the second and third parameters respectively.

NOTE: the number of dimensions cannot not be changed.

For a discussion on this function's usage, refer to ***Array Handling Enhancements***.

ARRAYSIZE

Format: ARRAYSIZE(***pointer***, ***numeric***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

This returns the size of an array. The first parameter is a pointer to an array (refer to the function ARRAYPTR). The second parameter specifies the dimension.

ARRAYSORT

Format: ARRAYSORT(pointer, numeric <,numeric> <,numeric> <,numeric>)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Nothing

This function sorts the contents of an array.

```
ARRAYSORT( AR_ptr, dir, dim, index1, index2 )
```

The parameters in the above example are:

Parameter	Explanation
AR_ptr	Pointer to the sorted array. Refer to the function ***ARRAYPTR***.
dir	The direction of the sort where 1 equals ascending and -1 equals descending.
dim	The dimension to be sorted. Used only when the array has more than one dimension.
index1	Index position of the dimension not being sorted (i.e., on a dimension other than dim). Used only when the array has more than one dimension. Its maximum value is the number of elements along this dimension. For example, in array[10,20] where the sort dimension is 1, index1 cannot be greater than 20.
index2	Index position of the second dimension not being sorted. Used only when the array has three dimension. The dimension referred to by index2 is greater than that of index1. For instance, if dim equals 2, then index1 refers to first dimension and index2 refers to the third.

For a discussion on this function's usage, refer to ***Array Handling Enhancements***.

ARRAYPTR

Format: ARRAYPTR(***array***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Address

This function returns the address of an array. The passed argument must be a declared array however, the element(s) chosen is irrelevant. The following example gets the address of a two-dimensional array:

```
AP_ptr = ARRAYPTR( #array[1,1] )
```

For a discussion on pointer usage, refer to the section, ***Pointers to Arrays and Variables***.

ASC

Format: ASC(***text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

ASC returns the ANGOSS Character Set value of the first character in the text expression. Refer to ***Appendix B*** in ***Software System Manual*** for a table of ANGOSS Character Set values.

The second example assumes that the variable $var contains the text "[".

Formula	**Result**
ASC("A")	65
ASC($var)	91

ASIN

Format: `ASIN(` ***numeric*** `)`
OS: DOS and Unix
Scope: All
Returns: Numeric

ASIN calculates the arcsine of the numeric expression. The numeric expression must have a value between --1 and 1, inclusive. The result is a value expressed in radians between --pi/2 and pi/2.

Formula	**Result**
ASIN(.17)	0.1708296

ASK

Format: `ASK(` ***text*** `)`
OS: DOS and Unix
Scope: All
Returns: Text

ASK causes the text expression to appear as a prompt on the screen just beneath the window. The program pauses for input from the keyboard. The data entered becomes the text string result returned by ASK.

Formula	**Result**
ASK("Enter greeting:")	"Hello World!"
ASK("Enter Value:")	"2"
VAL(ASK("Enter value:")) * 2	4

When the first formula is calculated, the prompt `Enter greeting:` is displayed. You enter **Hello World!** and this is the result returned.

When the second formula is calculated, the prompt `Enter value:` appears below the window. You enter **2**, and the text expression "2" is returned.

When the third formula is calculated, you are prompted to enter the value. Because the VAL function converts text to a numeric value, the 2 entered in response to the prompt is multiplied by 2 to obtain the result, 4.

ATAN

Format: ATAN(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

ATAN calculates the arctangent of the numeric expression. The result is a value expressed in radians in the range --pi/2 to pi/2.

Formula	**Result**
ATAN(3.23)	1.270558

ATAN2

Format: ATAN2(***numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

ATAN2 returns an angle whose value is the arctangent of the second argument (the sine, or y component of the angle) divided by the first argument (the cosine, or x component of the angle). The value returned is in radians between -pi and pi and correctly reflects the quadrant in which the angle lies.

Formula	**Result**
ATAN2(2,2)	0.785398
ATAN2(15,3)	0.197396

ATIME

Format: ATIME(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a time

ATIME returns a time in AM/PM format by evaluating the numeric expression as a number of minutes elapsed since the beginning of the day (12:00:00A).

Formula	**Result**
ATIME(121)	02:01:00A
ATIME(1022)	05:02:00P

ATIME24

Format: ATIME24(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Text formatted as a time

ATIME24 returns a time in 24-hour format by evaluating the numeric expression as a number of minutes elapsed since the beginning of the day (00:00:00).

Formula	**Result**
ATIME24(13.5*60)	13:30:00
ATIME24(1022)	17:02:00

AVERAGE

Format: AVERAGE(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

AVERAGE calculates the average of the numeric items in the item list, including numeric items with a value of 0. AVERAGE does not include text, null or blank items. AVERAGE is equivalent to:

SUM(***item list***) / COUNT(***item list***).

Formula	**Result**
AVERAGE(4.5,6.0,27.3,0)	9.45
AVERAGE(4.5,6.0,27.3,"0")	12.6

AVG

Format: AVG(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

AVG calculates the average of the items in the item list. AVG does not include null or blank items but does include text items, attributing a value of 0 to them. AVG is equivalent to:

SUM(***item list***) / @COUNT(***item list***).

Formula	**Result**
AVG(4.5,6.0,27.3,0)	9.45
AVG(4.5,6.0,27.3,"0")	9.45

BLOCKMARK

Format: BLOCKMARK
OS: DOS and Unix
Scope: SS
Returns: Text

The BLOCKMARK function, used only within project files, specifies a block of cells in the current worksheet. To use this function, you must first assign a variable with the value of BLOCKMARK, as follows:

```
LOCAL block
block = BLOCKMARK
```

Once this variable assignment is encountered, the worksheet is displayed, and the user can then define a block. After the block is defined, the variable contains a string that defines the location of the block.

This variable containing the block location string can be used as an expression for commands such as Edit Copy From, where you must specify one or more blocks. You can also use this variable as an argument to a function, but you must concatenate it into a string containing that function, as follows:

```
Enter Formula "SUM("|BLOCKMARK|")"
```

You cannot use BLOCKMARK directly in a formula or use the result of BLOCKMARK as an argument representing a range or block to another function. For example, you cannot use the formula

```
total = SUM(BLOCKMARK)
```

BGBACKGROUND, BGDIMPLEASING, BGEDITING, BGERROR, BGHIPLEASING, BGINVPLEASING, BGINVSTANDARD, BGPLEASING, and BGSTANDARD

Format: BGBACKGROUND
OS: DOS and Unix
Scope: All
Returns: Numeric

These functions return the color numbers of standard background colors used in ANGOSS screen displays. The values returned by these functions vary according to the screen driver currently in use. Using these functions for the color entries in SCREEN command statements guarantees acceptable foreground / background contrast regardless of the screen driver or display type that you are using. This practice also allows you to display text in colors consistent

The following table shows the ANGOSS display element whose background color is represented by each function.

Function	Use in ANGOSS Display
BGBACKGROUND	Background of labels on the status line
BGDIMPLEASING	Background of unavailable items on the paper profile menu
BGEDITING	Background of ANGOSS editors and the highlighted portion of the Status Line
BGERROR	Background of an error message
BGHIPLEASING	Background of valid items on the paper profile menu
BGINVPLEASING	Background of selected items in definition menus
BGINVSTANDARD	Background of the highlighted item in an ANGOSS module menu

Function	Use in ANGOSS Display
BGPLEASING	Background of definition menus
BGSTANDARD	Background of an ANGOSS module menu

BITAND, BITOR, and BITXOR

Format: `BITAND(` ***numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

These functions return a number that is the result of a binary operation on the respective bits of the two arguments, evaluated as 32-bit integers. Any fractional part of an argument is discarded.

The following table explains the result of the operation performed on the corresponding bits of the arguments. Examples below are shown in hexadecimal representation for clarity.

Function	Operation
BITAND	Result bit is 1 only if both bits are 1. Using BITAND to compare any bit with 0 produces 0; comparing any bit with 1 retains the value of the bit
BITOR	Result bit is 1 if either bit is 1. Using BITOR to compare any bit with 0 retains the value of the bit; comparing any bit with 1 produces 1
BITXOR	Result bit is 1 if either of the bits is 1, but not both. Using BITXOR to compare any bit with 0 retains the value of the bit. Comparing any bit with 1 complements (swaps) the original bit

Formula	Result
HEX(BITAND(0x00FF, 0x1234))	0x34
HEX(BITAND(0xFF00, 0x1234))	0x1200
HEX(BITOR(0x00FF, 0x1234))	0x12FF
HEX(BITOR(0xFF00, 0x1234))	0xFF34
HEX(BITXOR(0x00FF, 0x1234))	0x12CB
HEX(BITXOR(0xFF00, 0x1234))	0xED34

BLANK

Format: BLANK
OS: DOS and Unix
Scope: All
Returns: Blank

BLANK returns the blank status. When a formula returning BLANK is stored in a cell, field, or variable, any data in that object is discarded and it assumes a blank status.

NOTE: You must use the Edit Blank command in the Spreadsheet to blank a cell. You can assign BLANK status to a variable, but the original status of variables is zero (0), not blank.

BMPINFO

Format: BMPINFO(***constant, bmp_var***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

This function returns information about a bitmap. Note that the bitmap variable bmp_var is created by GRAPHICS BITMAP SAVE or GRAPHICS BITMAP READ.

Value	Constant	Explanation
0	bmp_height	height in pixels
1	bmp_width	width in pixels

CASE

Format: CASE ***expression case list***< ELSE ***expression***>
OS: DOS and Unix
Scope: All
Returns: Conditional

CASE creates a special type of logical formula which provides for a set of possible responses to a list of specified conditions.

CASE examines the first expression and compares it with the case list. The case list consists of pairs of expressions separated by a comma and enclosed in parentheses. Each pair contains an item to be compared with the first expression and an item to be returned if the comparison results in a match. You may include an ELSE statement in the formula to provide a response when no match is found. If none of the items in the case list is found to be equivalent to the first expression and no ELSE statement is included, the formula returns an Error 14.

This example assumes that the variable #FORM equals 1.

Formula	Result
CASE #FORM (1,"a")(2,"b")(3,"c")(4,"d") ELSE "No match"	"a"

A CASE formula is a specialized IF-THEN-ELSE statement. The following example shows how the CASE formula would look if written as an IF-THEN-ELSE formula:

```
IF #FORM = 1 THEN "a"
        ELSE IF #FORM = 2 THEN "b"
        ELSE IF #FORM = 3 THEN "c"
        ELSE IF #FORM = 4 THEN "d"
        ELSE "No match"
```

NOTE: If the first expression contains a function or is composed of multiple expressions, enclose it within parentheses to avoid ambiguity.

CELL

Format: CELL(***text, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Conditional

CELL returns a text or numeric item that provides specific information about the "home" cell referenced in the block argument. Although CELL returns information about a single cell, the second argument specified must be in the form of a block reference. The information returned pertains to the cell in the upper left corner (home) of the block.

COMMENT: You may reference a single cell by using the format !r1c1. The ! preceding the cell reference tells ANGOSS to treat the single cell reference as a block reference.

The first argument determines the kind of information requested. The following table describes the results possible for each type of data. If the result is not dependent upon the data in the cell, the right column is blank.

First Argument	Result Returned	Cell Contains
address	Absolute reference of the specified cell	
col	Column number of the cell	
contents	Value in the cell	
format	F0 to F15	0 to 15 decimal places in fixed decimal format
format	P0 to P15	0 to 15 decimal places in percent format
format	C0 to C15	0 to 15 decimal places in currency format
format	S0 to S15	0 to 15 decimal places in scientific format
format	(space character)	General format
format	D1 to D3	Date formats 1 through 3
format	T1 or T2	Time formats 12 or 24 hour
prefix	’	Left-justified text
prefix	"	Right-justified text
prefix	^	Centered text
prefix	(space character)	Blank cell
protect	1 if the cell is locked, 0 if the cell is not locked	

First Argument	Result Returned	Cell Contains
row	Row number of the cell	
type	b	Blank (empty)
type	v	Number or numeric formula
type	l	Text or text formula
width	Column width of the specified cell	

The second example assumes that r3c20 contains a numeric value. The third example assumes that r1c1 contains left-justified text.

Formula	Result
CELL("row",r3:4c20:21)	3
CELL("type",r3:4c20:21)	v
CELL("prefix",!r1c1)	’

CELLPOINTER

Format: `CELLPOINTER(` ***cell***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

CELLPOINTER is similar to CELL. While CELL yields information concerning the cell whose location or block name you indicate, CELLPOINTER yields information about the currently highlighted cell. The result returned by this function changes according to the location of the current cell when recalculation occurs.

In these examples, the current cell is r10c7:

Formula	Result
CELLPOINTER("row")	10
CELLPOINTER("col")	7

CELLTEXT

Format: CELLTEXT(***text***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Text

CELLTEXT returns a text expression that exactly matches the screen display of the text or text-equivalent contents of one or more specified cells in a row. The argument for this function must be a text expression in the form of a cell or a (horizontal) block reference. The text returned includes leading and/or trailing spaces, if necessary to represent the full cell width; text exceeding the cell width is truncated.

The following examples refer to the worksheet in Figure 3-2 of ***Formula Reference Manual***.

Formula	Result
CELLTEXT("r5c3")	$0.20
CELLTEXT("r5c4:6")	"USA $0.45 400"

CERROR

Format: CERROR
OS: DOS and Unix
Scope: All
Returns: Numeric

CERROR returns the number of any error resulting from the execution of the **immediately preceding command**. CERROR returns 0 if no error occurred during

the execution of the preceding command. You can use CERROR in a logical formula to identify any errors that may occur during the execution of a project and provide for recovery.

NOTE: CERROR returns the number of the "Current" error; LERROR returns the number of the "Last" or most recent error.

The following fragment of a Database project file illustrates how CERROR can be used to test for the failure of a Data Find command preceding it. When the Database cannot locate the search data, CERROR assumes the value 3716. The If statement calls a function called REDO_FIND that provides the necessary recovery information.

```
Data Find "[Last name]" Equal "Kempton" Options "g"
If CERROR = 3716
        REDO_FIND()
End if
```

Refer to ***Project Processing*** for a list of the error numbers assumed by this function.

CHARINFO

Format: CHARINFO(***constant <,param>***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

The CHARINFO function is used to relate the row column coordinates of the character based features to the xy coordinates of the GRAPHICS commands. This is useful if you want to write programs that mix the two components.

Informative and conversional calls can be made. The following is a list of accepted constants:

Value	Constant	Explanation
0	char_x	Combined with char_y, this is the upper left corner of row1 col1 in graphics screen.
1	char_y	See char_x.
2	char_h	Font height in pixels.
3	char_w	Font width in pixels.
4	char_x_to_col	Converts x to column.
5	char_y_to_row	Converts y to row.
6	char_col_to_x	Converts column to x.
7	char_row_to_y	Converts row to y.
8	char_x_to_nextrow	Converts xy to rc. Ensures data to left or above does not get overwritten. If x is the first pixel of a character then the conversion is exact, otherwise it selects the next column (or row).
9	char_y_to_nextcol	See char_x_to_nextrow.

Note:

0 - 4 One parameter only, conversion not allowed.

5 - 9 Second parameter is value to convert.

Example:

The following code segment sets up a series of buttons and draws boxes around them. Buttons use row column coordinates, lines use xy graphics coordinates.

```
'************************************
FUNCTION draw_com_boxes( #x,# y )
LOCAL #x1 #y1 #x2 #y2
```

```
LOCAL #ct #row
LOCAL b$settings
    GRAPHICS SET SAVE b$settings
    #top_com_row = CHARINFO( char_y_to_nextrow, #y )
    #left_com_col = CHARINFO( char_x_to_nextcol, #x )
    GRAPHICS SET LINE-WIDTH 1
    GRAPHICS SET FOREGROUND c_black
    FOR #ct = 1 TO #num_com
        #row = #com_row_offset[#ct] + #top_com_row
        BUTTON CREATE AT #row #left_com_col #row \
            #left_com_col+#max_len-1 #com_keys[#ct] $com_names[#ct]
        #x1 = CHARINFO( char_col_to_x, #left_com_col ) - #com_margin
        #y1 = CHARINFO( char_row_to_y, #row )-#com_margin
        #x2 = CHARINFO( char_col_to_x, #left_com_col+#max_len ) + #com_margin
        #y2 = CHARINFO( char_row_to_y, #row ) + CHARINFO(char_h) + #com_margin
        GRAPHICS DRAW RECTANGLE #x1 #y1 #x2 #y2
    END FOR
    BUTTON DRAW ALL
    GRAPHICS SET RESTORE b$settings
END FUNCTION
```

CHOOSE

Format: CHOOSE(***numeric, item list***)
OS: DOS and Unix
Scope: All
Returns: Conditional

CHOOSE creates a logical formula in which the value of the numeric expression determines which item in the item list is returned.

CHOOSE counts the sequence of items in the list and returns the item whose place in the list corresponds to the value of the numeric expression. The count begins with 0. If the numeric expression is equal to 0, the first item in the list is returned. If the value of the numeric expression is less than 0 or greater than the number of items in the list minus 1, the formula returns Error 24 (CHOOSE Failed).

Assume in the following examples that r10c4 contains 2 and that #amt contains 8.

Formula	Result
CHOOSE(r10c4,"Zero","One","Two", "Three","Four")	"Two"
CHOOSE(#amt, 10,9,8,7,6,5,4,3,2,1)	2

CHOOSE is similar to an IF-THEN-ELSE formula. If the first example were approximated in IF-THEN-ELSE form, it would look like this:

```
IF r10c4 = 0 THEN "Zero"
        ELSE IF r10c4 = 1 THEN "One"
        ELSE IF r10c4 = 2 THEN "Two"
        ELSE IF r10c4 = 3 THEN "Three"
        ELSE IF r10c4 = 4 THEN "Four"
        ELSE ERRORTEXT(24)
```

CHR

Format: CHR(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

CHR returns the text character corresponding to the ANGOSS Character Set decimal value of the numeric expression.

Formula	Result
CHR(65)	A
CHR(91)	[

CLICKINFO

Format: CLICKINFO(***constant***)
OS: DOS and Unix
Scope: All
Returns: Numeric

CLICKINFO returns information about the last mouse event. Typically, it is used after a {mouse} keystroke has been found in the keyboard buffer. The following named constants can be used with the CLICKINFO function:

Constant	#	Description
m_leftup	1	Indicates the left button was released.
m_leftdown	2	Indicates the left button was down.
m_rightup	3	Indicates the right button was released.
m_rightdown	4	Indicates the right button was down.
m_middleup	5	Indicates the middle button was released.
m_middledown	6	Indicates the middle button was down.
m_row	7	Row of pointer
m_col	8	Column of pointer
m_x	9	x coordinate pixel of pointer
m_y	10	y coordinate pixel of pointer
m_double	11	Indicates a double click was last up event

For example:

```
#key = INCHAR
IF #key = {mouse}
    IF CLICKINFO(m_leftup)
        MESSAGE “The left button was lifted at row”& \
                STR(CLICKINFO(m_row)) & “column” & \
                STR(CLICKINFO(m_col))
    END IF
END IF
```

See also MOUSEINFO.

Mouse Event Keystroke {mouse}

This keystroke is put into the keyboard buffer when a mouse button is pressed down or lifted up. The CLICKINFO function can be called to inquire about the details of the mouse action. A click produces two (mouse) symbols in the keystroke buffer: one for the mouse down event, and a second for the mouse up event. A ‘double click’ produces four.

COLLATE

Format: COLLATE(***text, text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

This function compares two text expressions according to a collation table. A collation table is a sorting sequence used when sorting or merging alphabetic characters. Unlike the ASCII sequence, collation table comparisons treat upper and lower case versions and accented and non-accented versions of the same character as equivalent. Punctuation symbols are also treated as equal.

If the first expression is greater than the second, COLLATE returns 1. If the expressions are equal, COLLATE returns 0. If the first expression is less than the second, COLLATE returns -1.

Formula	Result
COLLATE("a","b")	-1
COLLATE("?","&")	0
COLLATE("A","a")	0
COLLATE("B","A")	1

COLS

Format: COLS(***block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

COLS returns a numeric value equivalent to the number of columns in the worksheet block.

Formula	Result
COLS(r23:109c12:64)	53
COLS(!r1c1)	1

COLUMN

Format: COLUMN
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

If entered in a formula in a cell, COLUMN returns the number of the worksheet column in which the formula containing it is located. Otherwise, COLUMN returns the number of the column where the cell highlighter is located.

COS

Format: COS(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

COS calculates the cosine for the value of the numeric expression. The argument is expressed in radians.

Formula	**Result**
COS(1.1)	0.453596121426

COSH

Format: COSH(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

COSH calculates the hyperbolic cosine for the value of the numeric expression.

Formula	**Result**
COSH(.24)	1.028938505694

COUNT

Format: COUNT(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

COUNT returns the number of numeric expressions in the item list. Text expressions and blank or empty items in the list are not counted; items with the value 0 are counted.

Formula	**Result**
COUNT(1,2,3,4)	4
COUNT(1,"a",3,4,3+2)	4

@COUNT

Format: @COUNT(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

@COUNT returns the number of expressions in the item list. Text expression items with the value 0 are counted. Blank or empty items in the list are not counted.

Formula	**Result**
@COUNT(1,2,3,4)	4
@COUNT(1,"a",3,4,3+2)	5

CRYPT

Format: CRYPT(***text***, ***text***)
Scope: All
Available: Version 2.65 and Higher
Returns: Text

The first parameter is the password to encrypt, the second parameter is a two character "salt" string which changes the encryption algorithm. The CRYPT function returns a garbled string.

CTERM

Format: CTERM(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

CTERM returns a value equal to the number of compounding periods of an investment, based upon the arguments of the formula:

CTERM (***interest rate, future value, present value***)

The CTERM function is used to determine how long it will take the present value of an investment to reach the future value amount, based on a specified compounding period interest rate.

Formula	**Result**
CTERM(.12 / 12,25000,5000)	161.747

Example:

Suppose you deposited $5000 in an account that pays 12% annual interest, compounded monthly. To determine how long it will be before the $5000 you invested yields $25,000, use the formula CTERM(.12 / 12,25000,5000). A result of 161.7 months (13.5 years) is returned. The 12% annual interest rate has been divided by 12 months to indicate the interest is compounded monthly.

CTERM uses the following formula to compute the term:

(LN ((future value) / (present value)) / LN (1 + (interest rate))

CURRENCY

Format: CURRENCY(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

CURRENCY converts the numeric expression into a text expression representing the number as a currency value. The resulting text expression has a currency symbol and two decimal places.

Formula	**Result**
CURRENCY(41.446)	"$41.45"
CURRENCY(275+15)	"Rs. 290.00"

In the first example, the currency symbol specified is $ (dollar). In the second example, Rs. (Rupees) is used. Refer to the Tools Preferences Global command in ***Software System Manual*** for information on setting a currency symbol.

CURRFILES

Format: CURRFILES(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

CURRFILES returns a text expression consisting of the names of active (loaded) files separated by spaces. The argument must be 0, 1, or 2. The 0 argument causes CURRFILES to return only the file names. A single name appears in the text for active Database views and data-files having identical names. The 1 argument returns the filenames and extensions. The 2 argument returns the complete pathnames of all active files.

Formula	Result
CURRFILES(0)	"week1 week2 week3"
CURRFILES(1)	"week1.ws week2.ws week3.ws"
CURRFILES(2)	"\weekly\week1.ws \weekly\week2.ws\weekly\week3.ws"

DATE

Format: DATE(***numeric, numeric, numeric***)
Format: DATE(***date expression***)
OS: DOS and Unix
Scope: All
Returns: Numeric

DATE returns the number of days between December 31, 1899, and the day of the date expression. The argument can be a date entered as text, using either an alphabetic or numeric format. If three numeric values separated by commas are entered, ANGOSS assumes the date to be a Type 3 date. DATE is functionally equivalent to DAYS.

Formula	Result
DATE("9/15/84")	30939
DATE(57,8,24) -- DAYS("July 6, 1952")	1875

DATE1 DATE2 DATE3

Format: DATE1(***date***) etc.
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date

Each of these functions returns the date expression in the corresponding user-definable date format. For example, DATE1 produces a date in Date1 format. Date1 format is one of three formats you can use to control the appearance of date

information. The Date Format settings in the Tools Preferences Global menu define these formats.

Formula	Result
DATE1("09 15 88")	"15-Sep-88"
DATE2("9/15/88")	"September 15, 1988"
DATE3("September 15, 1988")	"Thursday"

The first example assumes that you have specified the format string "dd-Mon-yy" for the Date1 Format setting. The second assumes you have defined Date2 format as "Month dd, yyyy". The third assumes that Date3 is simply "Day".

NOTE: When you enter a date as a set of numbers (e.g. 9/5/88) ANGOSS references the Date Style setting in the Tools Preferences Global menu to interpret the order of day, month and year.

DATEVALUE

Format: DATEVALUE(***date***)
OS: DOS and Unix
Scope: All
Returns: Numeric

DATEVALUE returns a decimal number representing a unique date. The number returned is the number of days that have passed between Dec. 31, 1899, and the day of the date expression. The argument must be a date expression type 1 (text) or type 2 (numeric).

Formula	Result
DATEVALUE("09/15/84")	30939
DATEVALUE("12/30/1899")	-1

@DAVERAGE

Format: @DAVERAGE(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DAVERAGE returns the average value in the specified field / column of selected records in a worksheet database. @DAVERAGE ignores blank and text cells.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

@DAVG

Format: @DAVG(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DAVG returns the average value in the specified field / column of selected records. @DAVG ignores blank cells and treats text cells as having a value of 0.
All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

DAY

Format: DAY(***date***)
OS: DOS and Unix
Scope: All
Returns: Numeric

DAY returns a number corresponding to the day of the month in the date expression.

Formula	**Result**
DAY("July 6, 1988")	6

DAYNAME

Format: DAYNAME(***date***)
OS: DOS and Unix
Scope: All
Returns: Text

DAYNAME returns the name of the day of the week specified in the date expression.

Formula	**Result**
DAYNAME("07-06-52")	"Sunday"

DAYS

Format: DAYS(***date***)
OS: DOS and Unix
Scope: All
Returns: Numeric

DAYS returns a decimal number representing a unique date. The number returned is the number of days that have passed between December 31, 1899, and the day of the date expression. Dates prior to December 31, 1899, are returned as negative numbers.

Formula	**Result**
DAYS("24 Aug 57")	21055
DAYS("09/15/84")	30939
DAYS(57,08,24)	21055
DAYS("12/30/1899")	-1

DAYS2

Format: DAYS2(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

DAYS2 returns the number of days between December 31, 1899, and the day of the date expression. The argument must be a date expression type 3.

Formula	**Result**
DAYS2(84,3*3,15)	30939
DAYS2(57,8,24) -- DAYS("July 6, 1952")	1875

DBC_CLOSE

Format: DBC_CLOSE(***handle***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Nothing

DBC_CLOSE closes the "cursor" on the indicated database.

```
DBC_CLOSE( #cursor_hnd )
```

DBC_CONNECT

Format: DBC_CONNECT(***numeric***, ***numeric***, ***text***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Handle

DBC_CONNECT makes a connection to the specified database and returns a database handle. In order, the parameters are: connect method, number of "cursors", and the connect string.

```
#db_handle = DBC_CONNECT( #method, #cursors, dbname )
```

The connect method is always 1 (MTH_CHANNEL). The number of cursors to obtain has a minimum of 16 however, this amount is ensured.

DBC_COL_INFO

Format: DBC_COL_INFO(***handle***, ***numeric***, ***constant***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

The DBC_COL_INFO function returns information on a specified column in a "list". In order, the parameters are: the "cursor" handle, the column number, and the information constant.

```
DBC_COL_INFO( #cursor_hnd, #column, #code )
```

The information constants are:

Constant	Description
dbc_colname	Column name.
dbc_coltype	Column type (returns an integer, see below).
dbc_colwidth	Column width.

The list below shows the values returned by DBC_COL_INFO when column type is specified. These integers match the data type parameter constants for the DBC_SQL_EXEC.

Value	**Constant**	**Description**
1	dbc_i1	one-byte integer
2	dbc_i2	two-byte integer
3	dbc_i4	four-byte integer
4	dbc_string	string
5	dbc_f4	four-byte float
6	dbc_f8	eight-byte float

DBC_CURRENT

Formats:
DBC_CURRENT(***handle***, ***text***)
DBC_CURRENT(***handle***, ***numeric***)
DBC_CURRENT(***handle***, ***pointer***, ***pointer***, ***...***)
DBC_CURRENT(***handle***, ***array_pointer***)
OS: DOS and Unix
Scope: All
Returns: Conditional

All formats of the DBC_CURRENT function require a cursor handle returned by aDBC_SELECT function.

The first two forms of the function return the contents of the column specified by the secondparameter. This parameter may be either the column name or number.

```
DBC_CURRENT( #cursor_hnd, "column name")

DBC_CURRENT( #cursor_hnd, #column_number )
```

The last two forms do not return the contents of a column. Instead, the contents of columns read sequentially from the current "cursor" position are placed into either the variable(s) specified by variable pointer(s) or the array elements specified by the array pointer.

```
DBC_CURRENT( #cursor_hnd, VP_1, VP_2, ... VP_n )

DBC_CURRENT( #cursor_hnd, AP_array )
```

DBC_CURRENT is similar to the ***DBC_FETCH*** and ***DBC_FETCHP*** functions.

DBC_DBTYPE

Format: DBC_DBTYPE(***handle***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

DBC_DBTYPE returns a string indicating the type of database. Its single parameter specifies a database currently connected.

```
DBC_DBTYPE( #connect_hnd )
```

Possible return values are

Return	Vendor	Return	Vendor
Not Connect		ODBC	Various
Adabas	Software AG	Oracle	Oracle
Allbase	HP	Rdb	DEC
DB2/2	IBM	SQLDB	Software AG
Informix	Informix	Sybase	Sybase
Ingres	ASK/Ingres		

DBC_EOF

Format: DBC_EOF(***handle***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Boolean

DBC_EOF is used to determine whether the "cursor" position is outside the boundaries of the "list" specified by the handle parameter. That position may be after the last row or before the first one.

In effect, DBC_EOF returns 1 when the previous DBC_FETCH or DBC_FETCHP functions do not return meaningful data.

In the following example note that the data ($column1) is not operated on until after the DBC_EOF test.

```
WHILE TRUE
    $column1 = DBC_FETCH( #cursor_hnd, 1 )
    IF DBC_EOF
        EXIT WHILE
    END IF
```

```
    MESSAGE $column1
END WHILE
```

DBC_FETCH

Formats:
DBC_FETCH(***handle***, ***text***)
DBC_FETCH(***handle***, ***numeric***)
DBC_FETCH(***handle***, ***pointer***, ***pointer***, ***...***)
DBC_FETCH(***handle***, ***array pointer***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Conditional

All formats of the DBC_FETCH function require a cursor handle returned by a DBC_SELECT function. Before obtaining any values, DBC_FETCH advances the "cursor" to the next row.

The first two forms of the function return the contents of the column specified by the second parameter. This parameter may be either the column name or number.

```
DBC_FETCH( #cursor_hnd, "column name")

DBC_FETCH( #cursor_hnd, #column_number )
```

The last two forms do not return the contents of a column. Instead, the contents of columns read sequentially from the "cursor" position are placed into either the variable(s) specified by variable pointer(s) or the array elements specified by the array pointer.

```
DBC_FETCH( #cursor_hnd, VP_1, VP_2, ... VP_n )

DBC_FETCH( #cursor_hnd, AP_array )
```

DBC_FETCH is similar to the ***DBC_FETCHP*** and ***DBC_CURRENT*** functions.

DBC_FETCHP

Formats:
DBC_FETCHP(***handle***, ***text***)
DBC_FETCHP(***handle***, ***numeric***)
DBC_FETCHP(***handle***, ***pointer***, ***pointer***, ***...***)
DBC_FETCHP(***handle***, ***array pointer***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Conditional

All formats of the DBC_FETCHP function require a cursor handle returned by a DBC_SELECT function. Before obtaining any values, DBC_FETCHP moves the "cursor" to the previous row.

Note that not all relational database systems can perform this function.

The first two forms of the function return the contents of the column specified by the second parameter. This parameter may be either the column name or number.

```
DBC_FETCHP( #cursor_hnd, "column name")

DBC_FETCHP( #cursor_hnd, #column_number )
```

The last two forms do not return the contents of a column. Instead, the contents of columns read sequentially from the "cursor" position are placed into either the variable(s) specified by variable pointer(s) or the array elements specified by the array pointer.

```
DBC_FETCHP( #cursor_hnd, VP_1, VP_2, ... VP_n )

DBC_FETCHP( #cursor_hnd, AP_array )
```

DBC_FETCH is similar to the ***DBC_FETCHP*** and ***DBC_CURRENT*** functions.

DBC_GET_COLNAMES

Format: DBC_GET_COLNAMES(***handle***, ***pointer***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Nothing

This function places the column names of a "list" into an array. The first parameter is the "cursor" handle and the second is a pointer to an array.

```
DBC_GET_COLNAMES( #cursor_hnd, ARRAYPTR($columns[1]) )
```

DBC_NUM_COLS

Format: DBC_NUM_COLS(***handle***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

This function returns the number of columns in a "list".

```
#columns = DBC_NUM_COLS( #cursor_hnd )
```

DBC_RELEASE

Format: DBC_RELEASE(***handle***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Nothing

DBC_RELEASE closes the connection to the indicated database. A value of -1 closes all connections.

DBC_SELECT

Format: DBC_SELECT(***handle***, ***string*** <,***hostvar***, ***constant***> <, ***hostvar***, ***constant...***>)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Handle

DBC_SELECT performs an SQL SELECT statement, creates a "list", and returns a "cursor" handle. The first parameter is the database connect handle and the second parameter is the SQL SELECT statement.

The remaining parameters must be paired as host variable and type constant. The host variables are substituted into the SQL SELECT statement where the first host variable substitutes ":1" in the SQL SELECT statement, the second host variable substitutes ":2", and so on.

```
DBC_SELECT( #db_hnd, "select * from "|$dbname, "data", dbc_string )
```

The following table describes the type constants. These constants match the values returned by the DBC_COL_INFO function when inquiring about the column type.

Value	Constant	Description
1	dbc_i1	one-byte integer
2	dbc_i2	two-byte integer
3	dbc_i4	four-byte integer
4	dbc_string	string
5	dbc_f4	four-byte float
6	dbc_f8	eight-byte float

DBC_SQL_ERROR

Format: DBC_SQL_ERROR(***numeric***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Conditional

DBC_SQL_ERROR returns the message of the last SQL error that has occurred. When LERROR or CERROR is set to 452 ("DBC: SQL Error"), DBC_SQL_ERROR can be used to determine the actual SQL error. If the return type argument is 1, DBC_SQL_ERROR returns the SQL message, otherwise the SQL error code is returned.

```
DBC_SQL_ERROR( #returntype )
```

DBC_SQL_EXEC

Format: DBC_SQL_EXEC(***handle***, ***str*** <, ***hostvar***, ***constant***> <, ***hostvar***, ***constant...***>)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Nothing

DBC_SQL_EXEC executes SQL statements. The first parameter is the database connect handle and the second parameter is the SQL statement.

The remaining parameters must be paired as host variable and type constant. The host variables are substituted into the SQL statement where the first host variable substitutes ":1" in the SQL statement, the second host variable substitutes ":2", and so on.

```
DBC_SQL_EXEC( #db_hnd, "SQL statement :1", "data", dbc_string )
```

The following table describes the type constants. These constants match the values returned by the DBC_COL_INFO function when inquiring about the column type..

Value	Constant	Description
1	dbc_i1	one-byte integer

Value	Constant	Description
2	dbc_i2	two-byte integer
3	dbc_i4	four-byte integer
4	dbc_string	string
5	dbc_f4	four-byte float
6	dbc_f8	eight-byte float

NOTE: DBC_SQL_EXEC cannot be used as a replacement for the ***DBC_SQL_SELECT*** function because it does not return a "cursor".

DBC_TRANS_START, DBC_TRANS_COMMIT, DBC_TRANS_ABORT

Formats:
DBC_TRANS_START(***handle***)
DBC_TRANS_COMMIT(***handle***)
DBC_TRANS_ABORT(***handle***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Nothing

These functions are somewhat analogous to the ***LOCK-RECORD***, ***WRITE-RECORD***, and ***CANCEL-RECORD*** commands. Data is written to the database with an SQL statement issued through the ***DBC_SQL_EXEC*** function.

DBFLDAT

Format: DBFLDAT(***numeric, numeric***)
OS: DOS and Unix
Scope: Database
Returns: Text

DBFLDAT returns the field name at the location specified. If the coordinates do not match any field position, null is returned. The arguments are as follows:

DBFLDAT(row, column)

DBFLDINFO

Format: `DBFLDINFO(` ***field, flag***)
OS: DOS and Unix
Scope: Database
Returns: Conditional

DBFLDINFO returns information about a database field. The first argument, ***field***, is either a numeric value representing the field number or a string representing a field name in the current view. The second argument is a flag or constant to specify the type of data to be returned, as follows:

	Constant	Returns
0	dbf_name	View field name.
1	dbf_type	Field type: 1 (inverted) 2 (alpha) 3 (numeric) 4 (counter) 5 (date) 6 (time).
2	dbf_dbwidth	Field storage width.
3	dbf_scrwidth	Field screen width.
4	dbf_iskey	Key field: 0 (no) 1 (yes).
5	dbf_ontable	On table: 0 (no) 1 (yes).

	Constant	Returns
6	dbf_table	Table name.
7	dbf_startrow	Start row. Returns -1 if not on the screen.
8	dbf_startcol	Start column. Returns -1 if not on the screen.
9	dbf_endrow	End row. If the file is in browse mode or the field is on a table, this is the row location of its bottom right corner.
10	dbf_endcol	End column. If the file is in browse mode or the field is on a table, this is the column location of its bottom right corner.
11	dbf_filenum	Data file number of the field (or view only field): 0 (driver data file) 1-127 (driven data file) 128 (view only field). To find the actual name of the file, see the ***DBINFO*** function.
12	dbf_attrib	Field attribute: 0 (read/write) 1 (read only) 2 (mandatory entry) 3 (project write).
13	dbf_dbname	Data file field name.
14	dbf_relstartrow	Combined with the following three constants, this refers to the position of a field relative to the view's top left corner (not the screen coordinates). A position is returned even if the field is not currently displayed.
15	dbf_relstartcol	See dbf_relstartrow.
16	dbf_relendrow	See dbf_relstartrow.
17	dbf_relendcol	See dbf_relstartrow.
18	dbf_dispform	Display format.
19	dbf_inpmask	Input mask.
20	dbf_message	Data entry message.
21	dbf_default	Default equation.

	Constant	Returns
22	dbf_menutype	Menu type: 0 (none) 1 (bar menu) 2 (pop-up) 3 (data file).
23	dbf_menuchoices	String of menu choices. If the menu is a data file type, field names are returned.
24	dbf_calctype	Calculation type: 0 (none) 1 (immediate) 2 (wait) 3 (manual).
25	dbf_calculation	Calculation.
26	dbf_autoadv	Automatic advance field 0 (no) 1 (yes).
27	dbf_titleplace	Location of field title 0 (none) 1 (above) 2 (left).
28	dbf_datafg	Data foreground color.
29	dbf_databg	Data background color.
30	dbf_titlefg	Title foreground color.
31	dbf_titlebg	Title background color.
32	dbf_popupdb	Pop-up menu data file name.
33	dbf_popupreturn	Field returned in pop-up menu.

In the following examples, DBFLDINFO first returns an alpha field type for the field [name]. The second example returns the field name for field 1.

Formula	Result
DBFLDINFO("[name]", dbf_type)	2
DBFLDINFO(1, dbf_name)	name

DBGET

Format: DBGET(***text***)
OS: DOS and Unix
Scope: Database
Returns: Conditional

DBGET returns the contents of a field. The argument is a text expression in the form of a field reference. The view name may be included, if you are referring to an active view other than the current view.

The following examples assume that the field [amt] contains 1000, and [ytd.tax] contains 362.00.

Formula	**Result**
DBGET("[amt]")	1000
DBGET("[ytd.tax]")	362.00

DBINFO

Format: DBINFO (***named constant***)
OS: DOS and Unix
Scope: Database
Returns: Conditional

DBINFO returns a variety of information about the current view, depending upon the named constant specified in the argument. Valid arguments and their return values are as follows:

Name	Return Value
db_browse	1 (TRUE) if ANGOSS is in Browse Mode; 0 (FALSE) if ANGOSS is not in Browse Mode
db_dvrlinks	Version 2.65 and higher.The driver link fields (names on view). While its value is 4000, it actually returns nothing unless a value is added to it
db_dvnlinks	Version 2.65 and higher.The driven link fields (names on data file - key fields). Similar to db_dvrlinks
db_fields	The total number of fields on the current view
db_order	A number representing the current view order: 0 = physical, 1 = key, 2 = index
db_orderfld	A string containing the name of the key field by which the view is currently ordered. If the view is not in key order, an empty string ("") is returned
db_orderidx	A string containing the name of the index by which the view is currently ordered. If the view is not in index order, an empty string ("") is returned
db_paths	Version 2.65 and higher. The path of the data files. Its value is 2000 and, like the db_driver + db_driven constants combined, each number added to it refers to the next file. If same as view path, null is returned
db_pathexps	Version 2.65 and higher. The path expression of the data files. With a value of 3000, it behaves similarly to db_path. If no expression is used, null is returned
db_tables	A string containing the names of the tables on the current view. The table names are enclosed in angle brackets, <>, and separated by spaces. If the view contains no tables, an empty string ("") is returned
db_curtable	A string containing the name of the current table. If no table is present, an empty string ("") is returned

Name	Return Value
db_enter	1 (TRUE) if ANGOSS is in Enter / Update Mode; 0 (FALSE) if ANGOSS is not in Enter / Update Mode.
db_recalc	1 (TRUE) if the recalculation being called is the recalculation performed when a record is saved; 0 (FALSE) if the recalculation being called is not the recalculation performed when a record is saved (i.e., the recalculation performed when the cursor passes through a calculated field, or when no calculation at all is being executed.)
db_orderidxpf	A string that contains the full name, including path and extension, of the current index.
db_top	The db_top constant combined with db_left, db_bottom, and db_right refer to the current position of a view within its window (not the screen coordinates). For instance, on a multi-paged view that has been scrolled down two lines, DBINFO(db_top) would return 3 regardless of the view's screen coordinates. Since these returned values are relative, they can also be used to determine the size of the window.
db_left	See db_top.
db_bottom	See db_top.
db_right	See db_top.
db_height	The height of a view regardless of screen size.
db_width	The width of a view regardless of screen size.
db_row	The top row on the screen. Border on/off can affect this value.
db_col	The leftmost column on the screen. Border on/off can affect this value.
db_driver	A string containing the name of the driver data file. The value of this constant is 1000.

Name	Return Value
db_driven	A string containing the name of the first driven data file. The value of this constant is 1001. Though there are no constants for the remaining driven files, these names can be returned by passing a numeric value to DBINFO rather than a constant. Add one to db_driven (maximum of 1127) for each subsequent file. See ***Example 2***.
db_files	The number of data files on a view.

Example 1:

```
‘use DBINFO(db_recalc) to control the execution of ASK.
IF DBINFO(db_recalc) <> TRUE
    THEN ASK("Enter amount")
ELSE NOCHANGE
```

Example 2:

```
’Display the name of each driven data file
#filesdriven = DBINFO(db_files)-1 ’driver not included
FOR #ct = 1 TO #filesdriven
    #driven = db_driven + #ct - 1      ’db_driven+0 is 1st file
    $file = DBINFO( #driven )
    MESSAGE "Driven file" & str(#driven) & "is:" & $file
END FOR
```

DBKEY

Format: DBKEY(***numeric***)

OS: DOS and Unix

Scope: Database

Returns: Numeric

DBKEY is used by project processing programs to control cursor movement. It is a good alternative to SUSPEND/KEYS statements since unnecessary screen flashing does not occur and performance is improved.

This function passes a given keystroke to the underlying module. If a cursor movement keystroke was processed, 1 is returned, otherwise, the keystroke was not a cursor movement key and 0 is returned.

DBKEY handles all movement keys and mouse activity.

Example 1:

```
DBKEY({down})  'Move the cursor down one line
```

Example 2:

```
'Control navigation by cursor keys, mouse clicks and scroll bar
WHILE #key<>{esc}
    #key = INCHAR
    IF DBKEY(#key)    'Key was cursor movement
        '(process command)
    ELSE              'Key was NOT cursor movement
        '(process command)
    END IF
END WHILE
```

DBPUT

Format: DBPUT(***text, expression***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

DBPUT places data into a specific field, returning TRUE (1) if the procedure is successful, or FALSE (0) if the procedure is not successful. The first argument must be a text expression in the form of a field reference. The second argument is the data being placed in the field.

The procedure fails if the target field is defined as read only. If the target field is defined as mandatory entry and you attempt to place BLANK into it, failure also occurs. Finally, DBPUT fails when the data does not conform to rules defined for the field contents in the view definition. When failure occurs, data previously in the field is retained.

The following examples indicate that the DBPUT function executed successfully. In the first example, 2 was put into field [exemp]. In the second example, the sum of [ytd] and [curr] was put into [tot.tax].

Formula	**Result**
DBPUT("[exemp]",2)	1
DBPUT("[tot.tax]",[ytd]+[curr])	1

DCOUNT

Format: DCOUNT(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

DCOUNT returns the number of value entries in the specified field / column of selected records. DCOUNT ignores blank and text cells.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

@DCOUNT

Format: `@DCOUNT(` ***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DCOUNT returns the number of value and text entries in the specified field / column of selected records. @DCOUNT ignores blank cells.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

DDB

Format: `DDB(` ***numeric, numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

DDB returns a double declining balance depreciation value, which is determined by the following arguments:

DDB (***cost value, salvage value, useful life, term***)

DDB accelerates the rate of depreciation so that a greater depreciation expense occurs early in the term. When the book value of the asset reaches the salvage value, depreciation stops. The book value in any period is defined as the total cost of the asset minus the total depreciation over all prior periods.

The double-declining balance depreciation in any period is computed as:

(Book value) * 2 / (life of asset)

The result of the formula is adjusted as necessary to ensure that total depreciation for the life of the asset equals the asset's cost minus its salvage value

Formula	**Result**
DDB(15000,2000,7,1)	4285.71
DDB(15000,2000,7,3)	2186.59
DDB(15000,2000,7,6)	789.02

Example:

Suppose you purchase a $15,000 automobile which has a useful life of seven years and a salvage value of $2000 at the end of the seven years. The first formula listed above is used to determine what the depreciation rate would be for the first year of the car's useful life and returns a result of $4285.71. The second formula computes the depreciation rate for the third year of the car's useful life and returns a result of $2186.59. The third formula returns a result of $789.02, indicating the depreciation rate for the sixth year of the car's useful life.

DDE_ACCEPT

Format: DDE_ACCEPT(***text***, ***text***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

ANGOSS DDE servers are set up with the DDE_ACCEPT function. It announces to the rest of the computer the given server and topic names.

```
#channel = DDE_ACCEPT( "server_name", "topic" )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_ADVISE

Format: DDE_ADVISE(***channel***, ***text***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

DDE_ADVISE performs two functions depending on whether the ANGOSS application is a DDE client or server. When the application is a client, DDE_ADVISE tells the server to notify it when a specified item is changed. When the application is an ANGOSS server, DDE_ADVISE announces that a specified item is available for "warm link" processing.

Assuming the server is an ANGOSS application, a client may only request notification of change on those available items. In either case, any number of DDE_ADVISE calls can be made.

```
DDE_ADVISE( #channel, "item" )
```

If a client requests it, the server will be given a {DdeAdvise} event.

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_CHANNEL

Format: DDE_CHANNEL
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

This function returns a channel number when the DDE server has received a {DdeRequest} event. An alternative is to use EVENTINFO with the m_dde_handle constant.

```
#channel = DDE_CHANNEL
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_DATA

Format: DDE_DATA(***channel***, ***text***, ***text***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

DDE_DATA is used by a server to respond to a client's request by sending new data back. The channel and item can be determined with the DDE_CHANNEL and DDE_ITEM functions.

```
DDE_DATA( #channel, $item, "send data" )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_ERROR

Format: DDE_ERROR(***channel***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

DDE_ERROR is used by a server to respond to a client's request when the server cannot provide a value for the requested item. The channel can be determined with the DDE_CHANNEL function.

```
DDE_ERROR( #channel )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_EXECUTE

Format: DDE_EXECUTE(***channel***, ***text***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

DDE_EXECUTE sends a command string to the server for "execution". Whether or not this string uses acceptable syntax is determined by the server.

```
DDE_EXECUTE( #channel, "command string" )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_INITIATE

Format: DDE_INITIATE(***text***, ***text***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

DDE_INITIATE starts a conversation with the named server about the given topic. The returned integer is the channel number used in subsequent exchanges. A returned value of zero indicates that the server does not exist or that it cannot converse on the topic.

```
#channel = DDE_INITIATE( "server_name", "topic" )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_ITEM

Format: DDE_ITEM
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Text

This function returns an item name when the DDE server has received a {DdeRequest} event. An alternative is to use EVENTINFO with the m_dde_item constant.

```
$item = DDE_ITEM
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_POKE

Format: DDE_POKE(***channel***, ***text***, ***text***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

DDE_POKE attempts to poke a new value into the server - that is, it tries to assign a new value to the given item. Note that not all servers will accept these requests.

```
DDE_POKE( #channel, "item", "new value" )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_REQUEST

Format: DDE_REQUEST(***channel***, ***text***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Text

DDE_REQUEST sends a request to the server for the value of a specified item. If successful, a string value corresponding to the value of the "item" is returned.

```
DDE_REQUEST( #channel, "item" )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_TERMINATE

Format: DDE_TERMINATE(***channel***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

This terminates the link to the server.

```
DDE_TERMINATE( #channel )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_TIMEOUT

Format: DDE_TIMEOUT(***channel***, ***numeric***)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

This sets the time-out value used by the DDE client.

```
DDE_TIMEOUT( #channel, #timeout )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DDE_UNADVISE

Format: DDE_UNADVISE(channel, text)
Scope: All (Windows)
Available: Version 2.65 and Higher
Returns: Numeric

DDE_UNADVISE removes a DDE_ADVISE item.

```
DDE_UNADVISE( #channel, "item" )
```

For a general discussion on DDE, refer to the section ***DDE Access***.

DELETED

Format: DELETED
OS: DOS and Unix
Scope: Database
Returns: Numeric

DELETED returns TRUE (1) if the current record is deleted and FALSE (0) if the current record is not deleted.

DICTCOMP

Format: DICTCOMP ***(text, text)***
OS: DOS and Unix
Scope: All
Returns: Numeric

This function compares two text expressions with a true dictionary comparison. The case and accent of a letter will be taken into consideration if the comparison is otherwise equal. Results returned by DICTCOMP are affected by the currently selected character comparison table. If you will be comparing foreign language characters, it is important that you select the proper character comparison table for that language. The character comparison table is specified on the Global Preferences menu.

If the first expression is greater than the second, DICTCOMP returns 1. If the expressions are equal, DICTCOMP returns 0. If the first expression is less than the second, DICTCOMP returns -1.

DIRPROMPT

Format: DIRPROMPT
OS: DOS and Unix
Scope: All
Returns: Text

DIRPROMPT calls the ANGOSS file prompter in directory mode. A single directory may be selected. No parameters are required.

See also ***FILEPROMPT***.

@DMAX

Format: `@DMAX(` ***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DMAX returns the maximum value among the value entries in the specified field / column of selected records.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

@DMIN

Format: `@DMIN(` ***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DMIN returns the minimum value among the value entries in the specified field / column of selected records.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

DOSOFFSET

Format: DOSOFFSET(***variable***)
OS: Differences
Scope: All
Returns: Numeric

DOSOFFSET returns the offset (lower two bytes) of the memory address of the data referenced by a variable. The argument is the name of an existing variable. For more information, see the discussion of the Buffer command in the Project Processing manual.

This function returns:

286 mode	386 mode	Unix
16 bit	32 bit	32 bit

DOSPTR

Format: DOSPTR(***variable***)
OS: Differences
Scope: All
Returns: Numeric

The DOSPTR function returns segment and offset (four bytes) of the memory address of the data referenced by a variable. The argument is the name of an existing variable. For more information, see the discussion of the Buffer command in the Project Processing manual.

If the variable contains a low memory buffer and a real mode address is necessary for use with PEEK or POKE, REAL FARCALL, or REAL INTERRUPT, use the REALPTR function. This function returns a 32 bit pointer in 286 mode, 386 mode and Unix.

DOSSEG

Format: DOSSEG(***variable***)
Scope: All
OS: Differences
Returns: Numeric

DOSSEG returns the segment (upper two bytes) of the memory address of the data referenced by a variable. The argument is the name of an existing variable. For more information, see the discussion of the Buffer command in the Project Processing manual.

If the variable contains a low memory buffer and a real mode address is necessary for use with PEEK or POKE, REAL FARCALL, or REAL INTERRUPT, use the REALSEG function.

This function returns:

286 mode	386 mode	Unix
16 bit	data segment	0

@DSTD

Format: @DSTD(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DSTD returns the standard deviation of a population for the values in the specified field / column of selected records. See STD for further information.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

@DSTDEV

Format: @DSTDEV(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DSTDEV returns the standard deviation of a sample for the values in the specified field / column of selected records. See STDEV for further information.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

@DSUM

Format: @DSUM(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DSUM returns the sum of values in the specified field / column of selected records.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

@DSUMSQ

Format: @DSUMSQ(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DSUMSQ returns the sum of the square of the values in the specified field / column of selected records.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

DVAR

Format: DVAR(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

DVAR returns the sample variance of the values in the specified field/column of selected records.
All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

@DVAR

Format: @DVAR(***block, numeric, block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

@DVAR returns the population variance of the values in the specified field/column of selected records.

All Spreadsheet SDb functions use arguments as follows:

FUNCTION(***database block, column offset, criteria block***)

EOF

Format: EOF(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

EOF returns TRUE (1) when ANGOSS detects the end of the file during processing with the Fread and Fseek project commands. Otherwise, EOF returns FALSE (0). The argument is the file number assigned to the file when it is opened with the Fopen command.

When Fread's Length option is specified, EOF returns TRUE upon encountering the end of the file. When Fread's Length option is not specified, EOF returns TRUE when detecting the actual end of the file or an end-of-file character (ASCII 26). Consult ***Project Processing*** for more information on processing data files with these commands.

ERROR

Format: ERROR
OS: DOS and Unix
Scope: All
Returns: Error

ERROR returns Error 35 (User error). ERROR is used in a logical formula to indicate a condition that you want to identify as an error but that would not otherwise be treated as an error by ANGOSS.

Formula	**Result**
IF #SCORE <= 100 THEN #SCORE ELSE ERROR	Error 35
IF r1c1 > 1 THEN ERROR ELSE r1c1 * sales	100

The first example assumes #SCORE to be greater than 100.

The second example illustrates the use of ERROR to check cell entries in the Spreadsheet. In this example the named cell "sales" contains a dollar amount of sales.

Cell r1c1 contains a value that is a percentage of sales for an item. You are using ERROR to determine that the percentage of sales figures is not over 1 (100%). If r1c1 is greater than 1 (100%) "Error 35" will be displayed in the formula cell.

ERRORTEXT

Format: ERRORTEXT(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

The ERRORTEXT function returns the error message of the error specified (by number) in the argument. See ***Project Processing*** for a list of error numbers and messages used by ANGOSS.

Formula	**Result**
ERRORTEXT(35)	"User error"
ERRORTEXT(4)	"Bad syntax"

EVENTINFO

Format: EVENTINFO(***constant***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

EVENTINFO is identical to the function CLICKINFO. The function has been renamed because it is used for more than just mouse click events (notably after the {MouseMoved} function). In future releases it may return information on scroll bars, resizing events, etc.

Aside from the original CLICKINFO constants, new constants have been added:

m_mnu_id	Indicates the last menu handle an event took place on.

m_dde_handle	Indicates the channel number bound to a {DdeRequest} event.
m_dde_item	Indicates the item name bound to a {DdeRequest} event.

EVENTINFO returns information about the last event. Typically, it is used after a {mouse} keystroke has been found in the keyboard buffer. The following named constants can be used with the EVENTINFO function:

Constant	Description
m_dde_handle	Indicates the channel number bound to a {DdeRequest} event.
m_dde_item	Indicates the item name bound to a {DdeRequest} event.
m_leftup	Indicates the left button was released.
m_leftdown	Indicates the left button was down.
m_rightup	Indicates the right button was released.
m_rightdown	Indicates the right button was down.
m_middleup	Indicates the middle button was released.
m_middledown	Indicates the middle button was down.
m_mnu_id	Indicates the last menu handle an event took place on.
m_row	Row of pointer
m_col	Column of pointer
m_x	x coordinate pixel of pointer
m_y	y coordinate pixel of pointer
m_double	Indicates a double click was last up event

For example:

```
#key = INCHAR
IF #key = {mouse}
    IF EVENTINFO(m_leftup)
        MESSAGE “The left button was lifted at row”& \
                STR(EVENTINFO(m_row)) & “column” & \
                STR(EVENTINFO(m_col))
    END IF
END IF
```

See also MOUSEINFO and CLICKINFO.

Mouse Event Keystroke {mouse}

This keystroke is put into the keyboard buffer when a mouse button is pressed down or lifted up. The CLICKINFO function can be called to inquire about the details of the mouse action. A click produces two (mouse) symbols in the keystroke buffer: one for the mouse down event, and a second for the mouse up event. A ‘double click’ produces four.

EXACT

Format: EXACT(***text, text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

The EXACT function can be used as an alternative to the = operator in a formula, to test whether two text strings contain exactly the same characters. EXACT returns TRUE (1) if the two text strings are identical and FALSE (0) if they are not.

The second example assumes that r6c12 contains "Toronto".

Formula	Result
EXACT("John","john")	0
EXACT(r6c12,"Toronto")	1
EXACT("1234","12345")	0

EXP

Format: EXP(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

EXP calculates the value of "e" (the natural logarithm base: 2.7182818 . . .) taken to the power of the numeric expression.

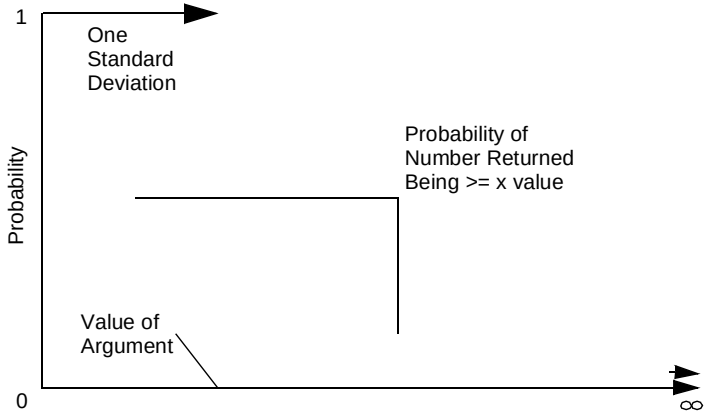
Formula	Result
EXP(1.1)	3.00416602395

EXPONENTIAL

Format: EXPONENTIAL(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

EXPONENTIAL returns a number randomly selected from an exponential distribution. The value of the numeric expression specifies the standard deviation of the distribution.

Figure 2-1



FACTORIAL

Format: `FACTORIAL(` ***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

FACTORIAL converts the argument to an integer and returns the value according to the following equation. The argument must be a positive number. A negative number returns a value of 1.

FACTORIAL(n) = n*(n-1)*(n-2) . . . *(2)*(1)

For example, `FACTORIAL(5)` is the same as `5 * 4 * 3 * 2 * 1`.

Formula	**Result**
FACTORIAL(5)	120

FACTUAL

Format: FACTUAL
OS: DOS and Unix
Scope: All
Returns: Numeric

FACTUAL returns the number of bytes read in the most recent Fread project command. Consult the Project Processing manual for more information on processing data files with this command.

FALSE

Format: FALSE
OS: DOS and Unix
Scope: All
Returns: Numeric

FALSE returns the value 0. FALSE can be used in logical formulas to signal that a condition does not exist.

FETCHFIELD

Format: FETCHFIELD(***field***)
OS: DOS and Unix
Scope: Database
Returns: Conditional

FETCHFIELD returns the contents of the specified field in the last record accessed. The field must be a data-file field. FETCHFIELD offers the ability to accomplish in a formula a result similar to that produced by pressing F9 in Enter/Update mode.

NOTE: FETCHFIELD returns the NA status for the first record accessed. However, if FETCHFIELD returns the contents for a record other than record 1 and then you move the cursor to record 1, the other record becomes the last record accessed. At that point, a FETCHFIELD on record 1 will return the contents, not the NA status. Note, also, that

reordering the file, with the Order Change command, for example, resets the last record accessed.

FGBACKGROUND, FGDIMPLEASING, FGEDITING, FGERROR, FGHIPLEASING, FGINVPLEASING, FGINVSTANDARD, FGPLEASING, and FGSTANDARD

Format: FGBACKGROUND
OS: DOS and Unix
Scope: All
Returns: Numeric

These functions return the color numbers of standard foreground colors used in ANGOSS screen displays. The values returned by these functions vary according to the screen driver currently in use. Using these functions for the color entries in Screen command statements guarantees acceptable foreground/background contrast regardless of the screen driver or display type that you are using. This practice also allows you to display text in colors consistent with various elements of ANGOSS' display.

The following table shows the ANGOSS display element whose foreground color is represented by each function.

Function	Use in ANGOSS Display
FGBACKGROUND	Foreground of labels on the status line
FGBACKGROUND	Foreground of labels on the status line
FGDIMPLEASING	Foreground of the unavailable items on the paper profile menu
FGEDITING	Foreground of ANGOSS editors and the highlighted portion of the status line
FGERROR	Foreground of an error message

Function	Use in ANGOSS Display
FHIPLEASING	Foreground of valid items on the paper profile menu
FGINVPLEASING	Foreground of selected items in definition menus
FGINVSTANDARD	Foreground of the highlighted item in an ANGOSS module menu
FGPLEASING	Foreground of definition menus
FGSTANDARD	Foreground of an ANGOSS module menu

FIELDTEXT

Format: `FIELDTEXT(` ***text***)

OS: DOS and Unix

Scope: Database

Returns: Text

FIELDTEXT returns a text expression that represents the exact displayed content of an ANGOSS Database field. The argument for this function must be a text expression in the form of a field name.

Any special formatting attached to the field will be included. This function is similar to the CELLTEXT function in ANGOSS Spreadsheet.

The following example is derived from Figure 3-6 in Chapter 3 of the Formula Reference Manual.

FIELDTEXT([Stone])

Notice that if your data is displayed in Browse format, the text returned is the content of the highlighted field. If your data is displayed in standard format, the text returned is the content of the currently displayed field.

FILE

Format: FILE(***text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

FILE returns TRUE (1) if a file named by the argument exists in the directory. Otherwise, FILE returns FALSE (0). The argument must include the filename and extension and may include a pathname appropriate for the operating system you are using.

Formula	**Result**
FILE("letter.doc")	1
FILE("\mydir\data\income.ws")	0
FILE("d:\skram.pf2")	1

The above examples indicate that letter.doc and d:\skram.pf2 were found; \mydir\data\income.ws was not found.

FILEAVERAGE

Format: FILEAVERAGE(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

FILEAVERAGE computes the average value in the specified field for all records in a view. Only nonblank fields are included in the calculation.

NOTE: Refer to Chapter 3 of the Formula Reference Manual for more information on using Statistical Database (SDb) functions in the Database.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILEAVERAGE([ytd]) FILEAVERAGE([ytd],[yr]>1987)

The first example computes the average of all non-blank values in the [ytd] field. The second example computes the average of all [ytd] values in records where [yr] is greater than 1987.

FILECOUNT

Format: `FILECOUNT(`***field <, logical>***)

OS: DOS and Unix

Scope: Database

Returns: Numeric

FILECOUNT counts the number of non-blank entries in the specified field for all records in a view.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILECOUNT([ytd]) FILECOUNT([ytd],[yr]>1987)

The first example counts the number of records having entries (non-blank) in [ytd]. The second example counts [ytd] entries in records where [yr] is greater than 1987.

FILEINFO

Format: FILEINFO(***filename, named constant***)
OS: DOS and Unix
Scope: All
Returns: Numeric

FILEINFO returns system information regarding the specified file or directory. The table below lists the constants that indicate the type of information returned:

	Named Constant	Return Value
0	f_type	0 = file does not exist 1 = file 2 = directory
1	f_datetime	Integer portion = date Fraction portion = time Like the NOW function (e.g., 32678.2138465)
2	f_size	File size in bytes
3	f_uid	An integer representing the user id that owns the file (DOS = 0)
4	f_gid	An integer representing the group id that owns the file (DOS = 0)
5	f_writable	1 for writable, otherwise 0
6	f_readable	1 for readable, otherwise 0

Example:

```
IF FILEINFO(“C:\DOS”,f_type)=2
   MESSAGE “Directory C:\DOS exists.”
END IF
#age = NOW - FILEINFO(“C:\B.BAT”,1)
MESSAGE “B.BAT modified” & str(seconds(#age)) & ”seconds ago.”
```

FILELOOKUP

Format: FILELOOKUP(***field, field, expression <,expressions...>)***
OS: DOS and Unix
Scope: Database
Returns: Conditional

FILELOOKUP searches a file for a data item in a specified field. When it encounters a record containing the search item, FILELOOKUP returns data from another specified field of that same record. If the search item exists in multiple records, FILELOOKUP returns data from the first record it encounters. The search field must be a key field, however, the file does not have to be in order by that field. The searched file must not be in index order.

The arguments are used as follows:

FILELOOKUP (***field searched, field returned, search data***)

Formula	**Result**
FILELOOKUP ([City], [State], "Greenville")	"Alabama"

In the formula above, FILELOOKUP searches the key field [City] for the data "Greenville". The first record in which a match occurs contains the data "Alabama" in the field [State].

FILELOOKUP supports searching of a minor key field. To specify the data for the minor key fields, place the data as the fourth parameter, separated by commas, in the order of the minor key fields. For example:

FILELOOKUP([State], [Population], “Alabama”, “Greenville”)

The above example returns the population of Greenville, Alabama. It assumes that [State] is a key field and that [State]’s minor key field is the city.

NOTE: FILELOOKUP and the File SDb functions cannot be nested.

FILEMAX

Format: `FILEMAX(`***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

FILEMAX returns the largest value in the specified field for all records in a view.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILEMAX([ytd])

FILEMAX([ytd],[yr]>1987)

The first example returns the maximum value occurring in the [ytd] field. The second example computes maximum [ytd] value in records where field [yr] contains data greater than 1987.

FILEMIN

Format: FILEMIN(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

FILEMIN returns the smallest value in the specified field for all records in a view. Only non-blank fields are included in the calculation.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILEMIN([ytd])

FILEMIN([ytd],[yr]>1987)

The first example returns the minimum value occurring in the [ytd] field. The second example computes minimum [ytd] value in records where field [yr] contains data greater than 1987.

FILEPROMPT

Format: FILEPROMPT(text, item list, numeric, numeric, numeric)
OS: DOS and Unix
Scope: All
Returns: Text

This function calls the ANGOSS file prompter and returns a string representing one or more user-selected files. The arguments are as follows:

FILEPROMPT(“prompt”, “extension”, display default, display extensions, multi-file)

The first parameter is a text string used for the prompt. The second parameter is a space separated list of file name extensions (i.e.,"doc txt") that causes fileprompt to initially list only files with those extensions.

The remaining parameters are boolean. If display default is TRUE, [default] will appear as the upper left file (like the PRINT DOCUMENT file prompter). If display extensions is TRUE, file names are displayed with their extensions. If multi-file is TRUE, then users may select more than one file.

The following rules should be observed when using FILEPROMPT:

- If [default] is selected, a string containing both square brackets (“[]”) is returned.
- The current directory is initially displayed, though users may move to an other directory.
- If multiple files are selected by the user, the file names are returned in a space separated list.
- If multiple files are selected from a different directory, only the first file name will contain the full path.
- All selected files must reside in the same directory.
- Users may type a file that does not exist.

See also ***DIRPROMPT***.

FILESTD

Format: `FILESTD(`***field*** **<, *logical*>**)
OS: DOS and Unix
Scope: Database
Returns: Numeric

FILESTD computes the standard deviation of a population for the specified field for all records in a view. Only non-blank values are used in the calculation.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILESTD([ytd])

FILESTD([ytd],[yr]>1978)

The first example computes the standard deviation of a population using all non-blank [ytd] values. The second example computes standard deviation of a population using all [ytd] values in records where [yr] is greater than 1987.

FILESTDEV

Format: `FILESTDEV(`***field <, logical>***`)`
OS: DOS and Unix
Scope: Database
Returns: Numeric

FILESTDEV computes the standard deviation of a sample for the specified field for all records in a view. Only non-blank values are used in the calculation.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILESTDEV([ytd])

FILESTDEV([ytd],[yr]>1978)

The first example computes the standard deviation of a sample using all non-blank [ytd] values. The second example computes standard deviation of a sample using all [ytd] values in records where [yr] is greater than 1987.

FILESUM

Format: `FILESUM(`***field <, logical>***`)`
OS: DOS and Unix
Scope: Database
Returns: Numeric

FILESUM calculates the sum of values in the specified field for all records in a view.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILESUM([ytd])

FILESUM([ytd],[yr]>1987)

The first example calculates the sum of values in [ytd]. The second example calculates the sum of values in [ytd] where [yr] is greater than 1987.

NOTE: FILESUM includes deleted records in its calculation. To prevent the use of deleted records, use a qualifier as the second argument, as shown in the following example.

FILESUM([ytd],NOT(DELETED))

FILESUMSQ

Format: `FILESUMSQ(`***field <, logical>***`)`
OS: DOS and Unix
Scope: Database
Returns: Numeric

The FILESUMSQ function computes the sum of the squares of the values for the specified field for all records in a view.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILESUMSQ([ytd])

FILESUMSQ([ytd],[yr]=1987)

The first example computes the sum of the squares of all entries in [ytd]. The second example computes the sum of the squares of all entries in [ytd] where [yr] is equal to 1987.

FILEVAR

Format: FILEVAR(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

The FILEVAR function computes the sample variance of the values in the specified field for all records in a view.

You can enter a logical expression as an optional second argument. Each record is evaluated to determine if it fits the condition specified in the logical expression. If the expression is true for a record, the record is included in the calculation. If the expression is false, the record is not included.

Examples:

FILEVAR([ytd])

FILEVAR([ytd],[month]>=7)

The first example computes the variance of all [ytd] values. The second example computes the variance for all [ytd] values in records where [month] is greater than or equal to 7.

FIND

Format: FIND(***text, text, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

FIND returns a number representing the sequential character position at which a text item is found to begin within a larger text expression. For example, "str" begins at position 3 in "substring."

The arguments are used as follows:

FIND (***search text, text expression, start value***)

The first character in the text string is assigned a value of 0, the second character is assigned a value of 1, the third character a value of 2, and so forth.

The search begins at the position specified by the start value argument. If the search text is not found within the text string, or if the start value entered is at a character position beyond the occurrence of the search text, the result is Error 18 (Bad argument). Negative start values also yield Error 18.

In the first example, the cell r3c5 contains the text string "Sales Totals".

In the second example, the field [name1] contains the text string "Mr. and Mrs. Kempton." The result is the position of the second period (following "Mrs") because the search starts at position 4, past the first period.

The third example illustrates how the @MID function can be used with FIND to extract a text string when its starting position is located. The expression "913-492-3800" is contained in [phone].

Formula	**Result**
FIND("ls",r3c5,0)	11
FIND(".",[name1],4)	11
@MID([phone],FIND("492",[phone], 0),8)	"492-3800"

FIXED

Format: FIXED(***numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

FIXED returns a text expression representing a number. The first argument is the number FIXED operates on. The second argument determines the number of decimal places in the text representation of the number.

Formula	**Result**
FIXED(200.3648,3)	"200.365"
FIXED(1.9,2)	"1.90"

FORMAT

Format: FORMAT(***expression, text <, numeric>***)
OS: DOS and Unix
Scope: All
Returns: Text

FORMAT is a versatile function that allows you to specify the format, or appearance, of data in a variety of ways. FORMAT returns a text expression in which a data item is formatted according to a formatting specification. If a width is specified, the data is aligned within that width. The arguments are used as follows:

FORMAT (***data item, format string <, width>***)

The first argument, the data item to be formatted, can be either text or numeric. If it is numeric, FORMAT follows ANGOSS' formatting conventions for numeric data. For example, if you specify currency format, the data is formatted according to the Currency symbol: and Currency symbol location: settings in the Global Preferences menu. If you specify a date format, the number is interpreted as a date expression type 2. If you specify a time format, any fractional part of the number is interpreted as an amount of elapsed time expressed as a fraction of the whole day.

If the first argument is text, FORMAT interprets the expression according to the data-type format code you specify. If you specify a date format, FORMAT interprets the text as a date expression (type 1 or 2). For date expressions type 1, FORMAT follows the `Date style:` setting in the Global Preferences menu to determine the order of numeric day, month and year values. If you specify a time format, FORMAT interprets the text as a time expression. If you specify one of the other data-type codes or if you specify a precision, FORMAT converts the text to a number. With no data type and no precision, FORMAT simply aligns and "pads" the text with blanks as specified in the format string.

The format string is a text expression consisting of one or more types of codes that specify the data format. Use of each type of code is optional, depending on the data being formatted and the result you want to obtain. Those that are used must be placed in the following order:

<precision><alignment><width><options><data-type>

NOTE: The entire format string must be a single sequence of text characters with no intervening spaces.

Precision. A precision code is a number from 0 to 15 that indicates a fixed number of decimal places for numeric data. If no precision code and no data-type are specified, numeric data is returned in the "general" format (decimal precision matches the precision of the numeric value).

Alignment. The alignment code specifies the alignment of the data within a specified width. You can use the following alignment codes.

Code	Meaning
L	Left alignment
R	Right alignment
M	Center (middle) alignment

If no alignment is specified, FORMAT assumes left alignment. Alignment of data has no effect if the width is not specified.

Width. A width code is a number from 1 to 255 that specifies the length (number of characters) of the text expression that FORMAT returns. Space characters (or asterisks if the F option code is specified) are used to "pad" to the desired width that portion of the resulting text not occupied by the formatted data. If the data exceeds the specified width, a string of asterisks filling that width is returned. If no width is specified, FORMAT returns exactly the number of characters required to produce the desired format.

NOTE: Width can also be specified as a third argument to FORMAT. If this third argument is included, it takes precedence over any width specification in the format string. When specified as a third argument, the width can range from 1 to 511 characters.

Options. Option codes provide special formatting capabilities. You can use the following option codes:

Code	Meaning
F	Fill with asterisks (*)
Z	Blank if 0
P	Parentheses around negative numbers
B	"cr" following negative & "db" following positive numbers
C	"cr" following negative numbers
,	Thousands separator; you must also specify a precision with this code

The F option code causes the text returned to be padded to the specified width with asterisks. This option is useful for check amount protection. The F code has no effect if

the width is not specified or if the width is less than or equal to the number of characters occupied by the data in the specified format.

The Z code returns a single space character if the data item is a numeric 0.

The P, B, and C codes provide various options for the treatment of negative numbers. FORMAT recognizes these codes only if you also specify a precision code or an appropriate data-type (i.e., $, %, or E).

Data-type. A data-type code specifies one of several data formats available in ANGOSS. You can use the following data-type codes:

Code	Meaning
$	Currency
%	Percent
E	Exponential notation
H	Histogram
D1	Date1 format
D2	Date2 format
D3	Date3 format
T1	12-hour time format
T2	24-hour time format

You may also create your own date format by specifying a data-type code consisting of the capital letter D followed by a custom date mask including one or a combination of the following characters:

Code	Meaning
d	Numeric day (omits leading 0 on single digit)
dd	Numeric day (includes leading 0)
day	Text day name
m	Numeric month (omits leading 0 on single digit)
mm	Numeric month (includes leading 0)
mon	Text month (3-character abbreviation)
month	Text month name
yy	Numeric year (2-digit)
yyyy	Numeric year (4-digit)

You can include other characters, such as hyphens (-) and slashes (/), within a custom date mask. Such characters are placed literally in the text expression returned, producing the familiar punctuation in date formats. If you want to include a term that serves as a code (e.g., the word "Month"), or the numbers 1, 2, or 3, as literal characters in a custom date mask, you must precede them with the backslash (\) character.

NOTE: If you inadvertently enter more than one data-type code, FORMAT uses the last (rightmost) one in the format string.

The optional width argument is a number from 1 to 511 that indicates the number of characters in the formatted text expression returned by FORMAT. This width specification supersedes any width specified in the format string.

Formula	Result
FORMAT(.1,"%")	"10%"
FORMAT(100,"R10")	" 100 "
FORMAT(100,"2R10B")	" 100.00db"
FORMAT("100","2M$F",12)	"**$100.00**"
FORMAT("-100","0P")	"(100)"
FORMAT(1.634,"T2")	"15:12:58"
FORMAT("32694","Dd/m/yy")	"6/7/89"
FORMAT("07/06/89", "Dday month d, yyyy")	"Thursday July 6, 1989"
FORMAT("11/22/63", "D\Month: month")	"Month: November"
FORMAT(5,"H")	"+++++"

FV

Format: FV(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

FV calculates the future value of a current lump sum payment at a given interest rate over a specified period of time. The three arguments are used as follows:

FV (***principal amount, term, interest rate***)

Formula	Result
FV(100,2,.12)	125.44

Example:

Using the values listed, if you invested $100 for a two-year term, compounded at an annual interest rate of 12%, the value of your original investment at the end of the 2 year term would be $125.44 ($100.00 + first year's 12% interest = $112.00; $112.00 + second year's 12% interest = $125.44).

FVA

Format: FVA(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

FVA calculates the future value of a stream of equal payments (an annuity) at a given interest rate over a specified period of time. The arguments are used as follows:

FVA (payment amount, term, interest rate)

Formula	**Result**
FVA(250,12,.09)	5035.18

Example:

If you made an annual investment of $250 for a 12 year term, compounded at an annual interest rate of 9%, at the end of the 12 year term, your $3000 total investment would be worth $5035.18.

GETENV

Format: GETENV(***text***)
OS: Differences
Scope: All
Returns: Text

GETENV searches the operating system environment for the specified environment variable name. The argument is a text expression representing the environment variable name.

If the variable is found, GETENV returns a text string associated with that variable. The text string may be a null (empty) string. If the variable is not found, GETENV returns numeric zero (0). The numeric zero allows you to test for whether the variable is found, as follows:

Example:

```
LOCAL X
X = GETENV("xyzzy")

IF(ISNUMBER(X))
        MESSAGE "Variable not found"
ELSE
        .
        .
        .
END IF
```

NOTE: Under DOS, the argument for GETENV is not case sensitive so that GETENV(“ANGOSS”) and GETENV(“angoss”) produce the same result. Under Unix, the argument is case-sensitive since the operating system supports this.

GETFNAMES

Format: GETFNAMES(***text, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

GETFNAMES returns a text expression consisting of a list of filenames separated by space characters. The first argument is a text expression representing the file specification of the files you want listed. You can use "wild card" characters, as in DOS. The second argument is a numeric expression: 0 causes GETFNAMES to return filenames only; any other number returns both filenames and their extensions.

The first two examples assume that expense.rdf and expense.ws are the only files in the current directory. The third example refers to a directory containing three worksheets beginning with the word "week." The last example assumes that the directory specified has two ".rdf" files.

Formula	**Result**
GETFNAMES("*.rdf",0)	"expense"
GETFNAMES("expense.*",1)	"expense.ws expense.rdf"
GETFNAMES("week?.ws",1)	"week1.ws week2.ws week3.ws"
GETFNAMES("c:\prim\data*.rdf",1)	"qtr1.rdf qtr2.rdf"

You can use the GROUP function to extract individual file names from the list.

NOTE: GETFNAME returns up to approximately 1,024 characters before truncating.

GETREG

Format: GETREG(***register***)
OS: DOS Only
Scope: All
Returns: Numeric

GETREG returns the CPU register value from the last Interrupt or Farcall command. Use the register name indicated to specify the register. The following example returns the value in the AX register:

GETREG(AX)

This function is designed for use with the ANGOSS Programming Language's Interrupt and Farcall commands. Consult ***Project Processing Manual*** for details on using these commands.

NOTE: GETREG and the Interrupt command were designed for use in a DOS environment. A run-time error message will be produced under Unix.

Register names include the following:

AX	BX	CX	DX	SI
DI	DS	ES	FLAGS	

GOAL

Format: GOAL(***numeric, numeric, GUESS formula***)
OS: DOS and Unix
Scope: All
Returns: Numeric

GOAL attempts to find a value for a term in an equation that will make the equation true. You supply the equation, an initial estimated value, and a result. The arguments are used as follows:

GOAL (***estimated value, desired result, formula using GUESS***)

The first argument is an approximation of or "guess" at the value you think will solve the equation. This value is initially represented in the formula (the third argument) by the variable GUESS. The second argument is the result that should be obtained when the true result value is substituted for GUESS in the formula. The formula can be an algebraic expression and can use ANGOSS functions.

Example:

To solve 270 = X^4 using 3 as an initial guess, the formula is:

GOAL(3,270,GUESS^4)

NOTE: Remember to check the result GOAL returns by calculating the formula with that result entered as a constant.

The value 3 is substituted in the formula for the variable GUESS and the result of the equation is calculated. The result is compared with the desired result (given in the second argument). Based upon the difference between the actual and desired results, a new value is obtained for GUESS. This second value is substituted into the formula and a new result is calculated. The process is repeated until two successive results differ less than 1E-7 or the maximum of 20 iterations (recalculations) is reached.

When using GOAL you should keep the following in mind:

- The initial guess should be as close as possible to the expected value, since the process ceases after a total of 20 iterations. If the initial guess is too far from the true value, the successive results will not converge before the maximum number of iterations is reached.
- If the values involved are extremely small or the initial guess is so far out of range as to result in an extremely small result, the difference between the actual and desired results may be less than 1E-7 and the calculation process will cease. In the first case you may need to multiply all values by a large constant (1,000; 1,000,000; etc.). In the second case, you should use a more reasonable initial guess.

Formula	Result
GOAL(1,4,GUESS*GUESS)	2
GOAL(--1,4,GUESS*GUESS)	-2
GOAL(3,270,GUESS^4)	4.053600464
GOAL(500,5035.18,FVA(GUESS,12,.0 9))	250.000002512

The first and second examples show how the estimated value effects the result returned in formulas where more than one true result can be derived. The fourth example calculates the amount of the payments where the desired future value is 5035.18, 12 payments are to be made and the interest rate is 9%.

GROUP

Format: GROUP(***text, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

GROUP returns a group of text characters occupying a particular position in a list. GROUP assumes that the first argument is a text expression listing a series of items separated by spaces. Each item can be one or more text characters, called a "group." The second argument is the position number of the group to be returned. If the position number specified is less than or greater than the number of groups, GROUP returns the null text character.

The second example assumes that the GETFNAMES expression returns the text "moe.doc larry.doc curly.doc".

Formula	Result
GROUP("define edit undefine ",2)	"edit"
GROUP(GETFNAMES("*.doc",1),3)	"curly.doc"

GR_FONTOPEN

Format: GR_FONTOPEN(***text, numeric, constant***)

OS: DOS and Unix

Scope: All (Graphics Environment)

Available: Version 2.65 and Higher

Returns: Numeric

This function requests the use of a font and returns an identifier that can be used from that point on.

```
GR_FONTOPEN( "type face", point_size, attributes )
```

The "type face" string, (e.g. helvetica) names the font. Point size is the size of the font. Note that a point is 1/72 of an inch and is not the number of pixels. The attributes argument is a bit flag field indicating what attributes the font should have. Legal attributes are:

fn_attr_bold

fn_attr_italic

fn_attr_prop

See also ***GR_FONTATTRIB***.

Multiple attributes can be added together (such as fn_attr_bold + fn_attri_italic).

Example:

```
#font_id = GR_FONTOPEN( "helvetica", 20, 0 )
GRAPHICS SET TEXT-FONT #font_id
```

GR_FONTCLOSE

Format: GR_FONTCLOSE(***font_id***)

OS: DOS and Unix

Scope: All (Graphics Environment)

Available: Version 2.65 and Higher

Returns: Numeric

This closes the given font and releases resources for other use.

GR_FONTATTRIB

Format: GR_FONTATTRIB(***text, constant***)
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Numeric

This function returns whether an attribute is supported by a given font family.

```
GR_FONTATTRIB( "type face", attribute )
```

Value	Constant	Explanation
1	fn_attr_bold	bold face
2	fn_attr_italic	italic
4	fn_attr_prop	proportional

The function returns 1 if the attribute is available or 0 if it is not available.

GR_FONTAVAILABLE

Format: GR_FONTAVAILABLE
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Numeric

This returns the number of font families that are available. It is used in conjunction with ***GR_FONTFAMILY***.

GR_FONTFAMILY

Format: GR_FONTFAMILY(***index***)
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Text

This returns the family name of the font given by some index number. The index runs from 1 to gr_fontavailable. The order of the list is completely arbitrary.

GR_FOREGROUND, GR_BACKGROUND, GR_LINETYPE, GR_LINEWIDTH, GR_FILLTYPE, GR_TEXTFONT, and GR_TEXTHEIGHT

Format: GR_FOREGROUND
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.6 and Higher
Returns: Numeric

These functions are used in a graphics environment and determine the current values for color, line type, line width, fill type, text font, and text height. The returned value is always numeric rather than the possible constant name. Refer to the GRAPHICS SET commands in the ***Project Processing*** manual.

GR_GETPIXEL

Format 1: GR_GETPIXEL(***x, y***)
Format 2: GR_GETPIXEL(***bmp_var, x, y***)
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Numeric

This function returns the color of pixel specified by x and y on the screen or specified bitmap. However, GR_GETPIXEL will only work on some graphics devices. Use GR_GETPIXEL after the following open commands:

```
GRAPHICS OPEN SCREEN

GRAPHICS OPEN BMP-FILE
```

When GRAPHICS OPEN CGM-FILE is used, GR_GETPIXEL will always return 0.

See also ***GR_SETPIXEL***.

GR_FILE

Format: GR_FILE
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Text

This indicates the name of the file that the graphics device currently has open when GR_MODE returns 1 or 3 (CGM or BMP file).

GR_MODE

Format: GR_MODE
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Numeric

This indicates the type of graphics device currently open. Return values:

Returns	Explanation
0	no graphics device open
1	CGM file
2	screen
3	BMP file

GR_MAPCOLOR

Format: GR_MAPCOLOR(***color***)
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Numeric

GR_MAPCOLOR takes a color and maps it to the closest color being displayed.

GR_RGBCOLOR

Format: GR_RGBCOLOR(***red, green, blue***)
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Numeric

This creates a new color, where you specify the RGB values. All values are limited from 0.0 to 1.0

GR_SETPIXEL

Format 1: GR_SETPIXEL(**x, y, color**)
Format 2: GR_SETPIXEL(**bmp_var, x, y, color**)
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Numeric

This function sets the color of pixel specified by x and y on the screen or specified bitmap. However, GR_SETPIXEL will only work on some graphics devices. Use GR_GETPIXEL after the following open commands:

```
GRAPHICS OPEN SCREEN

GRAPHICS OPEN BMP-FILE
```

GR_SETPIXEL returns the color actually set on the screen or in the bitmap. When GRAPHICS OPEN CGM-FILE is used, GR_SETPIXEL will always return 0.

See also ***GR_GETPIXEL***.

GR_TEXTASCENT and GR_TEXTDESCENT

Format: GR_TEXTASCENT
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.6 and Higher
Returns: Numeric

These functions are used in a graphics environment. GR_TEXTASCENT returns the distance from the baseline to the top of capital letters, when text is drawn using the current text font and height. GR_TEXTDESCENT returns the distance from the baseline to the bottom of lowercase letters with descenders (like y and g), when text is drawn using the current text font and height. Refer to the GRAPHICS commands in the ***Project Processing*** manual.

GR_TEXT_WIDTH

Format: GR_TEXT_WIDTH(***string***)
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.6 and Higher
Returns: Numeric

This returns the width of the specified string when drawn using the current text font and height in a graphics environment. Refer to the GRAPHICS commands in the ***Project Processing*** manual.

GR_X1, GR_Y1, GR_X2, and GR_Y2

Format: GR_X1
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.6 and Higher
Returns: Numeric

These functions are used in a graphics environment. GR_X1 and GR_Y1 return the top left screen coordinate. GR_X2 and GR_Y2 return the bottom right screen coordinate. Refer to the GRAPHICS commands in the ***Project Processing*** manual.

GR_XDPI and GR_YDPI

Format: GR_XDPI
OS: DOS and Unix
Scope: All (Graphics Environment)
Available: Version 2.65 and Higher
Returns: Numeric

These functions return an estimate of the dots per inch for the current graphics screen. It can be used so that graphics can be approximately the same physical size on many different graphic screens. These functions can also be used to convert font point sizes (which are 1/72 of an inch) to pixel coordinates.

HEX

Format: HEX(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

HEX returns a text expression representing the hexadecimal notation of the numeric argument. HEX can represent up to 32-bit numbers in hexadecimal format.

Formula	Result
HEX(32767)	"7FFF"
HEX(16)	"10"

HLOOKUP

Format: HLOOKUP(***expression, block <, numeric>***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Conditional

HLOOKUP searches a lookup table and returns data from a target cell in the table. A lookup table is a block of contiguous cells containing related data items. The top row usually contains information categorizing the data in the columns of the table. HLOOKUP searches the top row for a cell containing data matching the first argument. If a match is found, HLOOKUP returns the contents of a target cell in the same column.

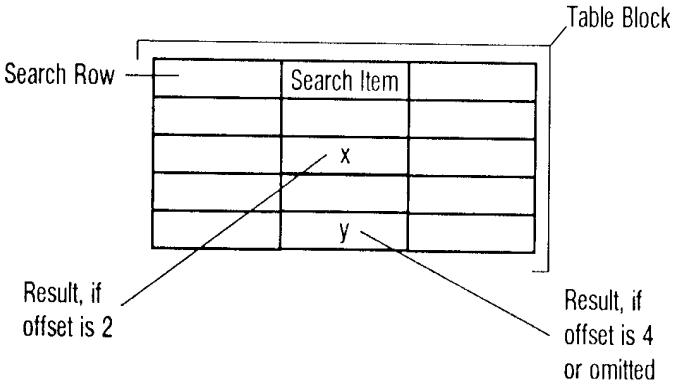
The arguments of this function are used as follows:

HLOOKUP (***search item, table block <, offset>***)

The search item can be any valid numeric or text expression. The block can be any valid block name or reference. The block defines the lookup table. HLOOKUP searches for data identical to the search item in the first row of the block. If the data is found, the contents of a target cell in this "lookup column" are returned. If the data is not found, HLOOKUP returns an error.

If no offset is specified, the cell in the last row of the lookup column is the target cell. If you include an offset, that number specifies the number of rows down from top of the block to the target cell. The search row is considered row 0.

Figure 2-2



@HLOOKUP

Format: @HLOOKUP(***expression, block <, numeric>***)

OS: DOS and Unix

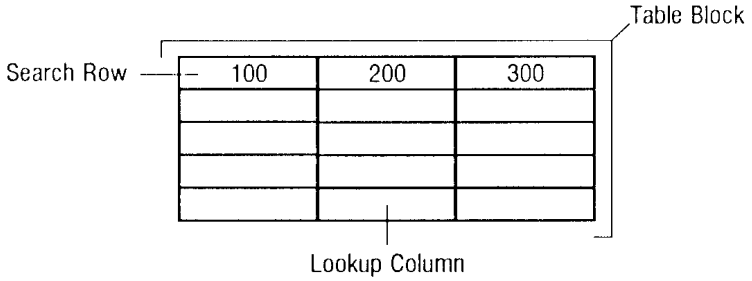
Scope: Spreadsheet

Returns: Conditional

@HLOOKUP is similar to HLOOKUP, except that it searches the top row in the block for the cell containing the largest value less than or equal to the first expression. This function requires that data in the search row (the top row) be in ascending numeric order.

For example, assume that the search item is 250. Since 250 falls between 200 and 300, the column containing 200 in the search row is the lookup column (See Figure 2-3)

Figure 2-3



The worksheet in Figure 2-4 of ***Formula Reference Manual*** illustrates examples of both the HLOOKUP and @HLOOKUP functions. The formula in the first example is located at r7c2. In the second example it is at r7c5 and in the third at r8c5.

Figure 2-4

	1	2	3	4	5	6	7
1	100.00	200.00	300.00	400.00			
2	10.00	20.00	30.00	40.00			
3	25.00	50.00	75.00	100.00			
4	40.00	80.00	120.00	160.00			
5	55.00	110.00	165.00	220.00			
6							
7	Hlookup 1	80.00		Hlookup 2	Error 8		
8				@Hlookup	80.00		

Formula	Result
HLOOKUP(200,r1:5c1:4,3)	80.00
HLOOKUP(250,r1:5c1:4,3)	Error 8
@HLOOKUP(250,r1:5c1:4,3)	80.00

HOUR

Format: HOUR(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

HOUR extracts the hour of the day from a decimal number representing the time of day as a fraction of the whole day, and returns a number between 0 (midnight) and 23 (23:00 or 11:00 p.m.). The fractional part of any number can be interpreted as a time of day and assigned a time value.

Formula	**Result**
HOUR(.05)	1
HOUR(0.99)	23
HOUR(0.488136574074)	11

HOURS

Format A: HOURS(***time***)
Format B: HOURS(***date <, time>***)
OS: DOS and Unix
Scope: All
Returns: Numeric

If you use Format A, HOURS returns the number of hours elapsed between the time represented by the time expression and the beginning of the day (00:00:00).

If you use Format B, HOURS returns the number of hours elapsed between the date (and time, if included) and 00:00:00 on 1/1/00 (the beginning of the century).

NOTE: To find the hours elapsed between two times, subtract the value returned by HOURS for the earlier time from the value returned for the later time.

HOURS accepts only date expression types 1 and 2. Type 1 is a text expression representing a date. Type 2 is a numeric expression representing a number of days before or after the beginning of the current century.

Formula	Result
HOURS("06:19:11a")	6
HOURS("Jan 5, 1988")	771504
HOURS("12/24/84","04:16:00") -- HOURS("12/23/84","22:00")	6

IF-THEN-ELSE

Format: `IF` ***logical*** `THEN` ***expression*** `ELSE`***expression***
OS: DOS and Unix
Scope: All
Returns: Conditional

IF-THEN-ELSE is used to construct a logical formula that tests for a condition described by the logical expression. If the condition exists (the logical expression is true), the expression in the "then clause" is returned. If the logical expression is false, the expression in the "else clause" is returned.

IF-THEN-ELSE formulas can be nested by placing one in the else clause of another. You may use logical operators to join logical expressions to create a highly specific condition description. When formulas are nested, the first logical expression found to be true determines the result. Subsequent logical expressions, which may also be true, are ignored.

The following illustrate possible uses of the IF-THEN-ELSE formula in various applications.

Example:

```
IF #POINTS >= 1000 THEN "Winner!" ELSE #POINTS + #BONUS

IF SUM([12;15;19]) <> [20] THEN ERROR ELSE SUM([12;15;19])

IF (r1c1 = 100) AND (r5c5 = 1000) OR (r2c1<1000)
        THEN (sum(r1c1:10))
        ELSE (sum(r11c1:10))
```

```
#amount * (IF #amount < 1000 THEN .06
        ELSE IF #amount < 2000 THEN .07
        ELSE IF #amount < 3000 THEN .08
        ELSE IF #amount < 4000 THEN .09
        ELSE 1)
```

The first example formula returns the text expression "Winner!" if the variable #POINTS contains a value that is greater than or equal to 1000. Otherwise, it returns a value obtained by multiplying #POINTS times the variable #BONUS.

The second example formula returns Error 35 (generated by the function ERROR) if the sum of fields 12, 15, and 19 is not equal to the contents of field 20. Otherwise it returns the sum of fields 12, 15, and 19.

The third formula returns the sum of values in r1c1:10 if either of the following two conditions is true:

- The cell r1c1 contains 100 and the cell r5c5 contains 1000.
- The cell r2c1 contains a value less than 1000.

Otherwise, the third example returns the sum of r11c1:10.

The fourth example returns a value calculated by multiplying the variable #amount by a multiplier obtained from an IF-THEN-ELSE formula. The IF-THEN-ELSE formula is calculated as follows:

1. If the variable #amount contains a value less than 1000, then the multiplier is 0.06.
2. If #amount contains a value greater than or equal to 1000 but less than 2000, then the multiplier is 0.07.
3. If #amount is greater than or equal to 2000 but less than 3000, then the multiplier is 0.08.
4. If #amount is greater than or equal to 3000 but less than 4000, then the multiplier is 0.09.
5. If #amount is greater than or equal to 4000, then the multiplier is 1

@IF

Format: `@IF(` ***logical, expression, expression***)
OS: DOS and Unix
Scope: All
Returns: Conditional

If the logical expression is true, the first expression following the logical expression will be calculated and returned. If it is not true, the second expression will be calculated and returned.

The second example following assumes that #POINTS is greater than 100.

Formula	**Result**
@IF(2*2>3,100,0)	100
@IF(#POINTS>=100,"Winner!", #POINTS + #BONUS)	"Winner"

INCHAR

Format: `INCHAR`
OS: DOS and Unix
Scope: All
Returns: Numeric

INCHAR returns the ANGOSS key value of the next key pressed after it is calculated. Program execution pauses until a key is pressed .

The simplest way to test a returned value from INCHAR is to use a key constant. A key constant is made up of a key code (see the table in ***Appendix B***) placed between a pair of braces. For instance, the constant for function key 10 is: {F10}.

Example:

```
WHILE TRUE
    CASE INCHAR
        WHEN {F5}
                DATA GOTO RECORD NEXT
        WHEN {F6}
                DATA GOTO RECORD PREVIOUS
        WHEN {esc}
                EXIT WHILE
END WHILE
```

INDEX

Format: INDEX(***block, numeric, numeric***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Conditional

INDEX returns the contents of a cell whose position is determined by the arguments of the function:

INDEX (***block reference, row offset, column offset***)

INDEX provides a means of using a calculated cell reference. The location of the cell from which the data will be taken is calculated as an offset from the cell in the upper left corner of the block. The row offset is the number of rows below the top row in the block. The column offset is the number of columns to the right of the leftmost column in the block.

Examples for INDEX are provided with @INDEX.

@INDEX

Format: `@INDEX(` ***block, numeric, numeric***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Conditional

@INDEX returns the contents of a cell whose position is determined by the arguments of the function:

@INDEX (***block reference, column offset, row offset***)

@INDEX provides a means of using a calculated cell reference. The location of the cell from which the data will be taken is calculated as an offset from the cell in the upper left corner of the block. The column offset is the number of columns to the right of the leftmost column in the block (the leftmost column is numbered 0) and the row offset is the number of rows below the top row in the block (the top row is numbered 0).

If either argument is too large for the block, or if the argument is a negative value, the formula returns Error 32 (Cell out of range).

The following examples refer to the worksheet in Figure 2-5 of ***Formula Reference Manual***.

Figure 2-5

	1	2	3	4	5	6	7
1	26.50	73.45	62.60	96.03	906.50	1,009.62	
2	345.00	800.63	102.23	227.10	927.24	1,120.56	
3	116.30	299.95	44.55	234.11	1,208.27	538.16	
4	92.92	4,223.71	571.17	78.40	911.84	527.24	
5	620.19	0.96	275.50	6,110.97	816.53	849.11	
6	2,911.56	17.10	4,329.60	1,904.53	724.52	2,712.00	
7							
8							

Formula	Result
@INDEX(r1:6c1:6,4,4)	816.53
@INDEX(r1:6c1:6,3,4)	6118.97
INDEX(r1:6c1:6,3,4)	911.84
INDEX(r1:6c1:6,2,5)	538.16

NOTE: While the @INDEX and INDEX functions return similar results, you cannot use them interchangeably. @INDEX's arguments must be specified in an order different from those of INDEX.

INDIRECT

Format: `INDIRECT(` ***text***)
OS: DOS and Unix
Scope: All
Returns: Conditional

INDIRECT evaluates almost any valid expression. It is particularly useful for returning the contents of a cell or field whose location is specified in another cell or field. The argument, which must be text, can be in any of the following forms:

- A cell reference
- A field reference
- A public variable name
- The name of a user-defined project function that has been declared public
- A valid formula expression

INDIRECT cannot be used to evaluate local or global variables or non-public project functions.

The second example assumes r1c1 contains the value $125.00 and r2c2 contains the text entry "r1c1". The third example, which uses the INDIRECT function to calculate a cell reference with MAKECELL, assumes that cell r6c7 contains the cell reference r20c10 and that the cell r20c10 contains 13295.

Formula	Result
INDIRECT("2*2")	4
INDIRECT(r2c2)	125.00
INDIRECT(MAKECELL(6,7))	13295

INEVENT

Format: INCHAR
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

The INEVENT function is just like INCHAR except that INEVENT may also return new event constants.

Note that in version 2.65 and higher, INCHAR and NEXTKEY work as they did in version 2.61. The {mouse} constant is returned on down and up mouse events.

The following constants can be returned by INEVENT:

LeftDown	Mouse event.
RightDown	Mouse event.
Middleman	Mouse event.
LeftUp	Mouse event.
RightUp	Mouse event.
Middle	Mouse event.
LeftDoubleDown	Mouse event.
RightDoubleDown	Mouse event.
MiddleDoubleDown	Mouse event.

LeftDoubleUp	Mouse event.
RightDoubleUp	Mouse event.
MiddleDoubleUp	Mouse event.
LeftHold	Mouse event. When held down, acts like a repeating key on a keyboard.
RightHold	Mouse event. When held down, acts like a repeating key on a keyboard.
MiddleHold	Mouse event. When held down, acts like a repeating key on a keyboard.
MouseMoved	Mouse event. Only when a button is down.
WinResize	Occurs when user has changed window size.
EndUtterance	Occurs after last keystroke sent when the keystrokes are coming from a verbal utterance (DragonDictate 3.0 only and dependant on compatibility delay=0).
DdeAdvise	DDE server places this in the DDE client's event queue when a specified item has changed.
DdeInitiate	DDE initiate notification to the DDE server.
DdeRequest	DDE request notification to the DDE server.
DdeTerminate	DDE terminate link notification to the DDE server.
DdeUnadvise	DDE notification to the DDE server that the client has removed an item's advisory status.

INT

Format: `INT(` ***numeric*** `)`
OS: DOS and Unix
Scope: All
Returns: Numeric

INT returns the largest integer less than or equal to the value of the numeric expression.

Formula	**Result**
INT(410.98)	410
INT(--2.5)	-3

@INT

Format: `@INT(` ***numeric*** `)`
OS: DOS and Unix
Scope: All
Returns: Numeric

@INT, like INT, returns the integer value of a number. Unlike INT, @INT returns the next larger integer for negative numbers.

Formula	**Result**
@INT(410.98)	410
@INT(--2.5)	-2

INTEREST

Format: `INTEREST(` ***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

INTEREST calculates the interest rate paid on a specified amount, given a fixed payment over a specified period of time. The arguments are used as follows:

INTEREST (***principal amount, payment amount, term***)

Formula	**Result**
INTEREST(65988.40,7000,30)	.10
INTEREST(5000,175,36)	.013

Example:

If you financed a $5000 automobile for 36 months, with monthly payments of $175.00, your interest rate would be 15.6%. The formula for this calculation, `INTEREST(5000,175,36),` returns a result of 1.3%, which represents your monthly interest rate (based on your entry of the expressions in monthly increments: 36 monthly payments of $175 each). To calculate your yearly interest rate, you must multiply the monthly interest rate of 1.3% by 12 months to yield a 15.6% annual interest rate.

NOTE: To obtain accurate results with this function, you must use consistent increments of time in the formula. If the term is expressed as a monthly value, the payment must also be expressed as a monthly value. If you choose to express the term and payment as yearly values, you will receive a result which is based on only one payment being made annually, which yields a different result from a like formula based on monthly payments.

INVERT

Format: INVERT(***text***)
OS: DOS and Unix
Scope: Database
Returns: Text

INVERT returns the text expression with the last (rightmost) group of characters placed first (leftmost). A group is one or more text characters separated by spaces in a single text expression. INVERT is useful when you need to convert a name so that the last name comes first. ANGOSS Database uses this method for sorting and performing comparisons on data in inverted name type fields.

The backslash (\) character forces the inversion to occur at a specific point in the text expression. In cases where the last group is not the group by which you want to sort (e.g., Tom Jones Jr.), you can insert the backslash to mark another position to be used for sorting purposes.

The following examples assume that [name] is an alphanumeric field, currently containing "James Kempton".

Formula	**Result**
INVERT([name])	"Kempton James"
INVERT("Fran~c,ois-Marie Arouet de Voltaire")	"Voltaire Fran~c,ois-Marie Arouet de"
INVERT("Tom \Jones Jr.")	"Jones Jr. \Tom"

IRR

Format: IRR(***numeric, item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

IRR calculates an approximate internal rate of return for a sequence of regular payments. The internal rate of return is the discount rate at which the present value of a series of payments made at constant intervals equals the value of the investment.

IRR finds an interest rate for which the net present value of a stream of payments equals 0. The arguments are used as follows:

IRR (***1st approximation, income flow .__.__.***)

IRR uses 20 iterations (20 subsequent recalculations) in order to produce the most accurate expression of the internal rate of return. The first approximation is used to calculate the first result. This result is then used in the second calculation. The second result is used in the third calculation, and so forth. An error message is issued if no convergence occurs in 20 iterations using the rate entered.

At least one number in the item list (usually the first) is negative, representing an outlay of funds. All empty cells / fields in a block referenced by the item list are interpreted as having a value of 0.

NOTE: Any worksheet blocks referenced in the item list must be one-dimensional (i.e., all cells lie in the same row or the same column).

In the following example, the block r1:11c2 contains -220000; 0; 25,000; 40,000; 40,000; 40,000; 40,000; 40,000; 40,000; 40,000; 40,000.

Formula	**Result**
IRR(.10,r1:11c2)	.0787

Example:

Suppose you are considering expanding your business by purchasing equipment at the cost of $220,000. By purchasing this equipment, you estimate that you will generate no additional business the first year, but the second year you estimate a return of $25,000. Returns of $40,000 per year are projected for the next eight years.

To calculate the internal rate of return on your investment over the next ten years, the formula would be `IRR(.10,-220000, 0, 25000, 40000, 40000, 40000, 40000, 40000, 40000, 40000, 40000)`, which returns a result of 0.0787. The 10% figure at the beginning of the formula indicates an estimated 10% interest rate return. The $220,000 is entered as a negative number because it represents the initial cash outlay. The result translates into a 7.87% internal rate of return on your original investment over a 10 year term.

ISBLANK

Format: ISBLANK(***expression***)
OS: DOS and Unix
Scope: Spreadsheet, Database
Returns: Numeric

ISBLANK returns TRUE (1) if the expression is blank or FALSE (0) if the expression is not blank. Normally, the argument should be a data reference.

ISCALC

Format: ISCALC
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

ISCALC returns 1 (TRUE) if the current worksheet has been changed since the last recalculation or 0 (FALSE) if the current worksheet does not need to be recalculated.

In manual recalculation order, the Spreadsheet calculates and displays results only for newly entered and edited formulas. When you edit a worksheet or enter new cells, you must press the recalculation key to have the entire worksheet recalculated and the new results displayed. ISCALC returns 1 when you have made changes to a worksheet without recalculating. During iterative recalculation ISCALC returns FALSE after the first iteration.

ISDATE

Format: ISDATE(*"text"*)
OS: DOS and Unix
Scope: All
Returns: Numeric

The ISDATE function evaluates a string, and returns 1 (TRUE) if the entry is a date, or 0 (FALSE) if the entry is not a date. ISDATE accepts only date expressions type 1 and 2.

Formula	Result
ISDATE("Jan 20, 1987")	1
ISDATE("Jan Smith")	0
ISDATE("01/01/87")	1

ISERR

Format: ISERR(*expression*)
OS: DOS and Unix
Scope: All
Returns: Numeric

ISERR returns 1 (TRUE) if the object referenced in the argument returns an error upon calculation. Normally, the argument is a data reference. The value of ISERR is 0 (FALSE) if there is no error in the object. ISERR can be used in a logical expression to determine if a formula error has occurred in a cell, field, or variable.

The example following assumes that r10c2 contains a formula that returns an error.

Formula	Result
IF ISERR(r10c2) THEN "Oops" ELSE r10c2	"Oops"

ISNA

Format: `ISNA(` ***expression*** `)`
OS: DOS and Unix
Scope: All
Returns: Numeric

ISNA returns 1 (TRUE) if the object specified in the argument returns N/A and 0 (FALSE) if it does not. The NA function is used in a formula to return the N/A (Not available) status. Normally, the argument is a data reference.

ISNUMBER

Format: `ISNUMBER(` ***expression*** `)`
OS: DOS and Unix
Scope: All
Returns: Numeric

ISNUMBER returns 1 (TRUE) if the expression is a number or returns a number (value). If the expression is not a number ISNUMBER returns 0 (FALSE).

Formula	**Result**
ISNUMBER(5)	1
ISNUMBER("July")	0

ISSTRING

Format: `ISSTRING(` ***expression*** `)`
OS: DOS and Unix
Scope: All
Returns: Numeric

ISSTRING returns 1 (TRUE) if the expression is text or returns text. If the expression is not text ISSTRING returns 0 (FALSE).

Formula	Result
ISSTRING(5)	0
ISSTRING("July")	1

ISVAR

Format: ISVAR(***text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

ISVAR evaluates the argument and returns TRUE (1) if it is the name of a public variable, or FALSE (0) if it is not the name of a public variable.

The following examples assume that #AMT1 has been declared a public variable and #AMOUNT has not.

Formula	Result
ISVAR("#AMT1")	1
ISVAR("#AMOUNT")	0

KEYVALUE

Format: KEYVALUE(***text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

KEYVALUE returns the ANGOSS key value of the key specified in the argument. The argument must be an ANGOSS key term (i.e., an activity control term or character term used in macro definition or with the Keys project commands). If the argument does not contain a valid key term, KEYVALUE returns 0. Refer to ***Appendix B*** for a table of keys and their associated key terms.

Formula	Result
KEYVALUE("F5")	319

LASTKEY_SOURCE

Format: LASTKEY_SOURCE
OS: DOS and Unix
Scope: All
Returns: Numeric

This function indicates the source of the last keystroke. It returns an integer where:

1 = From keyboard

2 = From macro (quick keys)

3 = From voice (some voice recognition software only)

4 = From mouse

5 = From button

See also NEXTKEY_SOURCE.

LEFT

Format: LEFT(***text, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

LEFT returns a string of text characters from the text expression starting at the first character and including the number of characters specified in the numeric expression. The first (leftmost) character is at position 1.

Formula	Result
LEFT("07/06/88",3)	"07/"
LEFT("James D. Kempton",5)	"James"

LEN

Format: LEN(***text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

LEN returns the length (number of text characters) of the text expression. LEN counts all text characters, including spaces. LEN returns 0 if the expression is null or numeric.

Formula	Result
LEN("1.1")	3
LEN("sales totals")	12
LEN(1.1)	0

LERROR

Format: LERROR
OS: DOS and Unix
Scope: All
Returns: Numeric

LERROR returns the numeric code of the last error that occurred, or 0 if no error has occurred during the current session. The numbers are the same as those returned by CERROR, which returns the number of the "current" error (i.e., the error resulting from the command just executed). The Clearerror project command resets LERROR to 0.

LET

Format: <LET> ***expression*** = ***expression***
OS: DOS and Unix
Scope: All
Returns: Numeric

LET assigns the result of the second expression (right of the equal sign) to a target data reference specified in the first expression (left of the equal sign). LET then returns the value of the second expression.

LET is a tool for moving data from one place to another. LET allows you to store data in variables, array elements, cells, and fields; thus, the first expression must be one of these data references, appropriate to the module in which you are working.

In ANGOSS' Project Development Language, you must first declare a variable before you can use LET to assign a value to it. Elsewhere, if the target is the name of a variable that does not exist, LET creates a new public variable to serve as the target.

Examples:

LET #amount = 0

LET $get_the_character = inchar

LET r5c6 = (LET #subtotal = 60)

The first example formula stores the value 0 in the variable #amount and returns 0. The second example accepts the next keystroke, stores its ANGOSS key value in the variable $get_the_character and returns that value. The third example assigns the value 60 to both the variable #subtotal and the cell r5c6. It also returns 60.

IMPORTANT: The use of LET within formulas to perform assignments involving cells and fields should be undertaken carefully. Several limitations on the use of this feature are described below.

LET formulas are designed to fail in cases where assignments may cause undesirable effects. Failure is usually signaled by an error condition. LET fails when you attempt to assign new contents to a Spreadsheet formula cell, except during project file execution. For example, you cannot use a LET formula in a cell to overwrite the cell itself (e.g., `LET rc = 3`) or another formula cell. This restriction also applies to user-defined functions executed from formula cells. Also, you cannot use LET to assign data to locked cells or read-only fields.

The results of LET formulas used with cells and calculated fields may depend on the recalculation order of the worksheet(s) or view(s) involved. These modules allow you to establish a calculation order for formulas referencing their data. However, because LET can alter data items while calculation is in progress, such applications of the function may have unexpected consequences.

LN

Format: `LN(` ***numeric*** `)`
OS: DOS and Unix
Scope: All
Returns: Numeric

LN calculates the natural logarithm (base e) of the numeric expression. The numeric expression must be greater than 0.

Formula	**Result**
LN(7)	1.94591014906

LOG10

Format: `LOG10(` ***numeric*** `)`
OS: DOS and Unix
Scope: All
Returns: Numeric

LOG10 calculates the common logarithm (base 10) of the expression. The argument must be greater than 0.

Formula	**Result**
LOG10(7)	0.84509804

LOGICAL

Format: `LOGICAL(` ***logical***)
OS: DOS and Unix
Scope: All
Returns: Numeric

LOGICAL evaluates the logical expression and returns 1 (TRUE) if the expression is true or 0 (FALSE) if the expression is false.

Formula	**Result**
LOGICAL(2 > 1)	1
LOGICAL(10 = 100)	0

LOWER

Format: `LOWER(` ***text***)
OS: DOS and Unix
Scope: All
Returns: Text

LOWER returns the text expression with all characters converted to lower case.

Formula	**Result**
LOWER("LOWER CASE")	"lower case"
LOWER("Toronto Ont.")	"toronto ont."

MAKEBLOCK

Format: MAKEBLOCK(***numeric, numeric, numeric, numeric***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Text

MAKEBLOCK creates a string in the form of a block reference. This function is useful when you want to use a variable or calculated block reference.

Specifically, MAKEBLOCK returns a text string in the format "rW:XcY:Z", where W and X are row numbers, and Y and Z are column numbers. The arguments are used as follows:

MAKEBLOCK (***row #1, row #2, column #1, column #2***)

In the second example, the current row is assumed to be 4 and the current column is 6.

Formula	**Result**
MAKEBLOCK(1,2,5,6)	"r1:2c5:6"
MAKEBLOCK(ROW,ROW + 1,COLUMN,COLUMN + 3)	"r4:5c6:9"

MAKECELL

Format: MAKECELL(***numeric, numeric***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Text

MAKECELL is used to create a string in the form of a cell reference. This function is useful when you wish to use a variable or calculated cell reference.

Specifically, MAKECELL returns a text string in the format "rXcY" where X is a row number and Y is a column number. The arguments are used as follows:

MAKECELL (***row number, column number***)

In the following examples, the current row is 3 and the project variable #STATE equals 11.

Formula	Result
MAKECELL(1,2)	"r1c2"
MAKECELL(ROW,#STATE + 1)	"r3c12"

MATCH

Format: MATCH(***text, text <, numeric>***)
OS: DOS and Unix
Scope: All
Returns: Numeric

MATCH searches the first expression for the string of characters specified in the second expression. The arguments are used as follows:

MATCH (***text searched in, text searched for <, starting position>***)

If a match is found, MATCH returns the number of the position where the first character of the second expression occurs in the first expression. If the starting position is included, MATCH will start searching at the character position corresponding to that number, starting at position 0. If no match is found, MATCH returns 0.

Formula	Result
MATCH("(913) 555-2089","555")	7
MATCH("Mississippi","iss",4)	5

MAX

Format: MAX(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

MAX returns the value of the largest numeric item in the item list. Text items and blanks are ignored.

Formula	**Result**
MAX(1,3,32,12,6,17,19)	32
MAX(4,"A",5,3*2,10/2)	6
MAX(--4,0)	0

MEMLEFT

Format: MEMLEFT
OS: DOS Only
Scope: All
Returns: Numeric

MEMLEFT returns the amount, in bytes, of available memory remaining in the system.

NOTE: This function was designed for use in a DOS environment. A run-time error message will be produced under Unix.

MID

Format: `MID(` ***text, numeric <, numeric>***)
OS: DOS and Unix
Scope: All
Returns: Text

MID returns the character or set of characters beginning at a specified position within a string of text. The arguments are used as follows:

MID (***text, starting position <, length>***)

The first character in the text is located at position 1. If the length is specified, MID returns that number of characters, beginning with the character at the starting position and extending to the right. If no length is specified, MID returns the character at the starting position and all characters to the right.

You may think of MID as copying characters from the text. The first numeric expression determines where the copy begins. The second numeric expression limits the number of characters copied from the text.

Formula	**Result**
MID("(913) 555-1089",7)	"555-1089"
MID("Overland Park",5,4)	"land"

@MID

Format: `@MID(` ***text, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

@MID returns the character or set of characters beginning at a specified position within a string of text. The arguments are used as follows:

@MID (***text, starting position, length***)

The first character in the text is located at position 0, the second at position 1, and so forth. You can specify a length value that exceeds the length of the text if you want to return the entire remainder of the text. If the length you specify is larger than the remaining characters in the text string, the extra length has no effect on the result returned.

If you enter a start value that is beyond the length of the text string, a null string (0 characters) is returned. A length value of 0 also returns a null string.

The LEFT and RIGHT functions may be used in conjunction with @MID to extract characters from either end of a text string.

Formula	**Result**
@MID("0123456789",0,5)	"01234"
@MID("0123456789abc",10,2)	"ab"

NOTE: While the @MID and MID functions return similar results, you cannot use them interchangeably. MID allows you to eliminate the length argument; @MID requires that all three arguments be entered. Also, MID assigns a value of 1 to the first character in the text string, while @MID assigns a value of 0 to the first character in the text string.

MIN

Format: `MIN(` ***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

MIN returns the value of the smallest numeric item in the item list. Any text items or blanks in the list are ignored.

Formula	**Result**
MIN(3,32,12,600,1,17,19)	1
MIN(4,"A",5,3*2,10/2)	4

MINUTE

Format: `MINUTE(` ***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

MINUTE extracts the minute from a time decimal number, such as that returned by @TIME, and returns a numeric value between 0 and 59. The fractional part of any number can be interpreted as a time of day and assigned a time decimal number.

Formula	**Result**
MINUTE(0.4881365740)	42
MINUTE(0.92190972)	07
MINUTE(3/25)	52

MINUTES

Format A: `MINUTES(` ***time***)
Format B: `MINUTES(` ***date*** <, ***time***>)
OS: DOS and Unix
Scope: All
Returns: Numeric

If you use Format A, MINUTES returns the number of minutes elapsed between the time represented by the time expression and the beginning of the day (00:00:00).

If you use Format B, MINUTES returns the number of minutes elapsed between the date (and time, if included) and 00:00:00 on 01/01/1900 (the beginning of the 20th century).

NOTE: You can use MINUTES to determine the number of minutes elapsed between two times. Subtract the value returned by the function for the earlier time from the value returned for the later time.

MINUTES accepts only date expression types 1 and 2. Type 1 is a text expression representing a date. Type 2 is a numeric expression representing a number of days before or after the beginning of the current century.

Formula	**Result**
MINUTES("06:19:11a")	379
MINUTES("July 6, 1988")	46553760

MNU_CLOSE

Format: MNU_CLOSE(***numeric***)
Scope: All
Available: Version 2.65 and Higher
Returns: nothing

This function removes a menu from memory. Its only parameter is the menu handle identifying the menu being closed.

MNU_DELCH

Format: MNU_DELCH(***numeric***, ***numeric***)
Scope: All
Available: Version 2.65 and Higher
Returns: nothing

This function is used to delete a choice from a menu. If the choice referred to is a sub menu, then all choices on that sub menu are also deleted. The format of MNU_DELCH is:

```
MNU_DELCH( menu, choice # )
```

where menu is the menu handle returned by ***MNU_OPEN***. The choice number identifies which choice is being deleted.

MNU_DELLANG

Format: MNU_DELLANG(***numeric***, ***num/txt***)
Scope: All
Available: Version 2.65 and Higher
Returns: nothing

The MNU_DELLANG function is used to remove all information pertaining to the identified language. The arguments are:

```
MNU_DELLANG(menu,language)
```

where menu is the menu handle returned by ***MNU_OPEN***. The language can be a number or language id text string.

MNU_DELUM

Format: MNU_DELUM(***numeric***, ***num/txt***)
Scope: All
Available: Version 2.65 and Higher
Returns: nothing

The MNU_DELUM function is used to remove all information pertaining to the identified usermode. The arguments are:

```
MNU_DELUM(menu,usermode)
```

where menu is the menu handle returned by ***MNU_OPEN***. The language can be a number or usermode id text string.

MNU_INFO

Format: MNU_INFO(***numeric***, ***numeric***, ***num/txt***, ***num/txt***, ***constant***, ***num/txt***)
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric or string

The MNU_INFO function is used for reading information from the menu structure or for changing information. The format is:

```
MNU_INFO( menu, choice, usermode, language, constant, <new val> )
```

menu

The menu is the menu handle which is returned by the ***MNU_OPEN*** function.

choice

The choice is the choice number that you want to read. The first choice of a menu is number 1. If the information you are reading or changing is not related to a choice, then this parameter is ignored.

Normally, the choice number will only operate on one level therefore to read choices on sub menus you must get the sub menu's handle (***MNU_CH_MENU***) first.

The meaning of choice can be changed with ***MNU_ORDINAL_REF***. If MNU_ORDINAL_REF is set to true, then the referenced choice (and menu) is calculated by the number of terminal nodes encountered in a recursive scan. This allows any terminal choice within the menu hierarchy to be accessed. The following chart shows a sample menu with the ordinal reference numbers in brackets.

File	Edit	Help(8)
Open (1)	Copy (4)	
Print (2)	Paste (5)	
Exit (3)	Delete (6)	
	Move (7)	

The same menu without ***MNU_ORDINAL_REF*** set (which is the default) would be

numbered:.

File(1)	Edit(2)	Help(3)
Open (1)	Copy (1)	
Print (2)	Paste (2)	
Exit (3)	Delete (3)	
	Move (4)	

usermode

The usermode is a number or string. It is used if the field you are accessing is indexed by usermode. If the field is not index by usermode, it is ignored.

language

This argument is also a number or string. It is used if the field you are accessing is indexed by language. If the field is not index by language, it is ignored.

constant

The constant indicates what data item you are reading or changing. See the constant table and descriptions below for more details.

new val

If a sixth parameter is included, its value is assigned to the identified field. If it is not present, the operation is considered to be read only. Note that is you are changing a value, then the old value is returned.

The following table lists the mnu_constants:

Constant	**Where**	**Type**	**Max**	**Save**	**UM**	**Lang**
MNU_LTITLE	ROOT	TEXT	40	Y		Y
MNU_STITLE	ROOT	TEXT	40	Y		Y
MNU_MNU_TYPE	ROOT	VAL		Y		
MNU_HELP	ROOT	TEXT	8	Y		

Constant	Where	Type	Max	Save	UM	Lang
MNU_CLASS	ROOT	VAL				
MNU_MODULE	ROOT	VAL		Y		
MNU_AUDIT	ROOT	VAL		Y		
MNU_TXT_BG	ROOT	VAL		Y		
MNU_TXT_FG	ROOT	VAL		Y		
MNU_HL_FG	ROOT	VAL		Y		
MNU_HL_BG	ROOT	VAL		Y		
MNU_BORDER_FG	ROOT	VAL		Y		
MNU_NUM_ITEMS	MENU	VAL				
MNU_NUM_USERMODES	ROOT	VAL				
MNU_UM_ID	ROOT	TEXT	8	Y	Y	
MNU_LANG_ID	ROOT	TEXT	8	Y		Y
MNU_NUM_LANGS	ROOT	VAL				
MNU_MNU_X1	MENU	VAL				
MNU_MNU_Y1	MENU	VAL				
MNU_MNU_X2	MENU	VAL				
MNU_MNU_Y2	MENU	VAL				
MNU_ISROOT	MENU	VAL				
MNU_TOT_NUM_ITEMS	ROOT	VAL				
MNU_ORDINAL_REF	ROOT	VAL				
MNU_USER_SELECT	MENU	VAL				
MNU_ROOT_MNU	MENU	MNU				
MNU_DESC	CHOICE	TEXT	40	Y		Y

Constant	**Where**	**Type**	**Max**	**Save**	**UM**	**Lang**
MNU_CH_TYPE	CHOICE	MNU		Y		
MNU_CH_MENU	CHOICE	MNU		Y		
MNU_ACT	CHOICE	TEXT	8	Y		
MNU_OBJ	CHOICE	TEXT	1000	Y		
MNU_PARAM	CHOICE	TEXT	1000	Y		
MNU_LINEHELP	CHOICE	TEXT	1000	Y		Y
MNU_CH_HELP	CHOICE	TEXT	8	Y		
MNU_ICON	CHOICE	TEXT	8	Y		
MNU_PASSWORD	CHOICE	TEXT	20	Y	Y	
MNU_CH_X1	CHOICE	VAL				
MNU_CH_Y1	CHOICE	VAL				
MNU_CH_X2	CHOICE	VAL				
MNU_CH_Y2	CHOICE	VAL				
MNU_CH_KEY	CHOICE	VAL				
MNU_GET_ORDREF	CHOICE	VAL				
MNU_CH_MISCSTR	CHOICE	TEXT	1000			
MNU_PARENT	MENU	MNU				
MNU_EXTRAS	ROOT	TEXT	1000	Y		
MNU_UM_INFO	ROOT	TEXT	1000	Y	Y	
MNU_MNU_MISCSTR	MENU	TEXT	1000			
MNU_EVENT_FCT	ROOT	TEXT	1000	Y		
MNU_HIER_MNU	CHOICE	MNU				
MNU_HIER_CH	CHOICE	VAL				

Constant	Where	Type	Max	Save	UM	Lang
MNU_MNU_MISCVAL	MENU	VAL				
MNU_TOOLBAR_POS	ROOT	VAL		Y		
MNU_LIBRARY	ROOT	TEXT	1000	Y		
MNU_BMPLIB	ROOT	VAL				
MNU_CH_JUSTIFY	CHOICE	VAL				
MNU_CUR_MNU		MNU				

General Informative Constants

Constant	Description
MNU_NUM_ITEMS	Number of items on the menu or submenu.
MNU_NUM_USERMODES	Number of usermodes on menu
MNU_UM_ID	The ID of the specified usermode
MNU_LANG_ID	The ID of the specified language
MNU_NUM_LANGS	Number of languages on the menu
MNU_ISROOT	Is the specified menu the root menu
MNU_TOT_NUM_ITEMS	The total number of choices in the entire hierarchical menu structure. Only terminal nodes are counted. In otherwords, choices that lead to sub menus are not counted.
MNU_ORDINAL_REF	Used to change the access mode. See discussion of the choice parameter.
MNU_ROOT_MNU	Returns the root menu handle.
MNU_CH_TYPE	Returns whether the choice is a terminal node (MNU_CHOICE or MNU_SUB_MENU).
MNU_CH_MENU	Returns the menu handle of the sub menu for the specified choice.

Constant	Description
MNU_GET_ORDREF	Given a menu id (which may be a sub menu) and a choice number, MNU_GET_ORDREF returns the number to use to access the same choice in ORDINAL_REF mode. MNU_ORDINAL_REF should be false when this is used.
MNU_PARENT	Returns the parent menu of the given sub menu.
MNU_HIER_MNU	This choice returns the menu handle of a given choice. It is used to convert an ordinal reference to a hierarchical reference. MNU_ORDINAL_REF should by true when it is used.
MNU_HIER_CH	This choice is just like MNU_HIER_MNU, except it returns the choice number.

Constants Relevant to Pulldown Menus and Tool Bars

Constant	Description
MNU_DESC	The description field is displayed as text when used with pulldown menus. If it is blank pulldown menus will ignore the choice.
MNU_ICON	The icon is the bitmap file name used with the TOOLBAR command. If the bitmap field is blank, the TOOLBAR command ignores the choice. Note that the menu must be bound to a bitmap library with the MNU_BMPLIB constant.
MNU_CH_KEY	This is the event that occurs when either a menu item is selected or a toolbar button is used. Usually, {} constants should be used. If {Alt-L} was assigned, then the {Alt-L} keystroke is placed in the queue when item is selected. The {MenuSelect} is often used.
MNU_TOOLBAR_POS	This indicates where the toolbar is placed.

Constant	Description
MNU_BMPLIB	This field binds a menu to a bitmap library. A bitmap library is created andmaintained with the GRAPHICS BMPLIB series of commands. The GRAPHICS BMPLIB CREATE command, sets a library handle. Setting MNU_BMPLIB to this handle number, binds the menu to that bitmap library. Icons are then read from that library when drawing the toolbar.
MNU_CH_JUSTIFY	This field indicates whether a choice will be left or right justified. It is used when the menu is displayed in the form of a toolbar.
MNU_CUR_MNU	This constant returns the menu id of the menu that is currently shown as a pulldown menu.
MNU_USER_SELECT	This field is set when a user selects a choice via the pulldown menus or toolbar. It is generally used in conjunction with EVENTINFO(m_mnu_id), which returns the menu handle of the last event. See the example under ***MNU_INSCH***.

RAD'S use of MNU_INFO constants

This table explains how RAD makes use of the various ***MNU_INFO*** fields. If a menu is not used by a RAD application, any of the constants described below may be used freely for whatever purposes you desire.

Constants	Description
MNU_LTITLE	Long title, used as menu titles and section titles in documentation.
MNU_STITLE	Short title, used as the left hand prompt of character based menus.
MNU_MNU_TYPE	0=menu, 1 ssview, 2=docview, 3=dataview, 4=commandlist.
MNU_HELP	The overview file name for the menu
MNU_CLASS	Not used

Constants	Description
MNU_MODULE	The SmartWare module that should be active when the menu is displayed. (1=ss, 2=wp, 3=db, 4=com, 5=doesn't matter).
MNU_AUDIT	Should activities on the menu be audited (0,1)
MNU_TXT_BG	Text background color.
MNU_TXT_FG	Text foreground color.
MNU_HL_FG	Highlighted choice foreground color.
MNU_HL_BG	Highlighted choice background color.
MNU_BORDER_FG	Graphics color.
MNU_DESC	Choice shown to the end user.
MNU_ACT	Action.
MNU_OBJ	Object (receiver of action).
MNU_PARAM	Function (General parameters).
MNU_LINEHELP	Autohelp line.
MNU_CH_HELP	Help file name for choice.
MNU_ICON	Icon name (some times defaults).
MNU_PASSWORD	An encrypted password for the choice. Absence of a password disabled the menu item.
MNU_CH_KEY	RAD menus always set this to {MenuSelect}.
MNU_EXTRAS	Used to store various extra information about a menu.
MNU_UM_INFO	Used to store information relevant to a usermode and menu.
MNU_EVENT_FCT	An SPL function to call to process events when the menu is current.
MNU_LIBRARY	Used to bind an SPL library to a menu.

Temporary Usage Constants

The following constants are temporary storage units and may be used as required. RAD uses them for its own purposes.

Constant	Description
MNU_MNU_X1	Used for temporary storage of positional information.
MNU_MNU_Y1	Used for temporary storage of positional information.
MNU_MNU_X2	Used for temporary storage of positional information.
MNU_MNU_Y2	Used for temporary storage of positional information.
MNU_CH_X1	Used for temporary storage of positional information.
MNU_CH_Y1	Used for temporary storage of positional information.
MNU_CH_X2	Used for temporary storage of positional information.
MNU_CH_Y2	Used for temporary storage of positional information.
MNU_MNU_MISCSTR	Miscellaneous string.
MNU_MNU_MISCVAL	Miscellaneous value.
MNU_CH_MISCSTR	Miscellaneous string.

MNU_INSCH

Format: MNU_INSCH(***numeric***, ***numeric***, ***text***, ***constant***, ***numeric***)
Scope: All
Available: Version 2.65 and Higher
Returns: numeric (TRUE for success, FALSE fail)

MNU_INSCH is used to add a choice to a menu in memory. The arguments are as follows:

```
MNU_INSCH(menu,position,description,type,key/sub menu)
```

menu

Is a menu handle as returned by ***MNU_OPEN***.

position

Is the position in the menu that the new choice will occupy. One (1) is the first choice.

description

Is the text that will be displayed when the menu is shown to the user. The DEFAULT language is assumed.

constant

The constant is either MNU_SUB_MENU or MNU_CHOICE and indicates whether the choice being created is a terminal node (MNU_CHOICE) or leads to a sub menu (MNU_SUB_MENU).

key/sub menu

If the type is MNU_CHOICE, then the last parameter is the event that will occur when the end user selects the choice. This often {MenuSelect} which is a general event meaning that a menu item was selected. It can also be a keystroke, such as {F10}, {Alt-C} etc.

If the type is MNU_SUB_MENU, then the last parameter is the menu handle, created with ***MNU_OPEN***(#), that will be appended to the menu structure as a sub menu.

Example:

The following code segment creates a menu and sub menu with one choice on it, displays it to the user, and prints the choice description when selected.

```
MAIN
LOCAL #mnu #sub_mnu #sel_mnu #sel_ch
    ' CREATE AN EMPTY MENU IN-MEMORY
    #mnu = MNU_OPEN("")
    IF #mnu = 0
        MESSAGE "Out of memory allocating new menu."
        EXIT MAIN
    END IF

    ' ALLOCATE A SUB MENU AS A CHILD OF #mnu
    #sub_mnu = MNU_OPEN(#mnu)
    IF #sub_mnu = 0
        MESSAGE "Out of memory allocating new menu."
        EXIT MAIN
    END IF

    ' ATTACH SUB MENU NAMED "File"
    MNU_INSCH(#mnu,1,"File",MNU_SUB_MENU,#sub_mnu)
```

```
    ' CREATE A CHOICE CALLED TEST ON MENU "File"
    MNU_INSCH(#sub_mnu,1,"Test",MNU_CHOICE,{MenuSelect})

    ' DISPLAY MENU IN PULLDOWN FORMAT
    PULLDOWN INIT #mnu "DEV" "DEFAULT"
    WHILE TRUE
        CASE INEVENT ' WAIT FOR AN EVENT
        WHEN {MenuSelect} ' MENU ITEM SELECTED
            ' DETERMINE MENU AND CHOICE # SELECTED
            #sel_mnu = EVENTINFO(m_mnu_id)
            #sel_ch = MNU_INFO(#sel_mnu,1,1,1,MNU_USER_SELECT)
            ' DISPLAY DESC OF SELECTED CHOICE
            MESSAGE MNU_INFO(#sel_mnu,#sel_ch,1,1,MNU_DESC)
        WHEN {Esc}
            EXIT WHILE ' END TEST
        END CASE
    END WHILE

    MNU_CLOSE(#mnu) ' FREE MEMORY USED BY MENU
END MAIN
```

MNU_INSLANG

Format: MNU_INSLANG(***numeric***, ***numeric***, ***text***)
Scope: All
Available: Version 2.65 and Higher
Returns numeric (TRUE for success, FALSE fail)

This function adds a language to the specified menu. The arguments for MNU_INSLANG are:

```
MNU_INSLANG(menu,position,language id)
```

Menu is the menu handle as returned by ***MNU_OPEN***. The position is the ordinal number which the new language will occupy. The language id is a text string (maximum eight characters) that will identify the language. Refer to ***MNU_INFO*** for a table of which data elements are indexed by language.

MNU_INSUM

Format: MNU_INSUM(***numeric***, ***numeric***, ***text***)

Scope: All

Available: Version 2.65 and Higher

Returns numeric (TRUE for success, FALSE fail)

This function adds a usermode to the specified menu. The arguments for MNU_INSUM are:

```
MNU_INSUM(menu,position,usermode id)
```

As with all MNU_ functions, menu is the menu handle as returned by ***MNU_OPEN***. The usermode id is a text string (maximum eight characters) that will identify the usermode. Refer to ***MNU_INFO*** for a table of which data elements are indexed by usermode.

MNU_OPEN

Format: MNU_OPEN(***text*** <,***text***> <,***text***>)

Scope: All

Available: Version 2.65 and Higher

Returns: numeric (menu handle)

MNU_OPEN has three distinct forms:

```
MNU_OPEN( "" )
```

Creates a new menu in memory

```
MNU_OPEN( #handle )
```

Creates a new sub menu where the parent menu is #handle

```
MNU_OPEN( "file", ["usermode"], ["language"] )
```

Reads a menu from disk. If a usermode or language is specified only information pertaining to them is loaded. If no usermode or language is specified all are loaded.

MNU_OPEN returns a menu handle (a number greater than 0) if it is successful. The menu handle is used by all other menu functions to identify the menu. If MNU_OPEN fails 0 is returned.

When a menu is created with MNU_OPEN("") the usermode "DEV" and language "DEFAULT" are automatically inserted.

MNU_WRITE

Format: MNU_WRITE(***numeric***, ***text***, ***constant***)
Scope: All
Available: Version 2.65 and Higher
Returns: numeric (true success,false fail)

The MNU_WRITE function is used to write a menu structure to disk. The arguments are:

```
MNU_WRITE( menu, filename, filetype )
```

where menu is the menu handle as returned by ***MNU_OPEN***. The file type must be the constant MNU_ASCII or MNU_BINARY. The ascii format is human readable, but takes a longer time to open. The binary format is not human readable.

MOD

Format: `MOD(` ***numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

MOD returns the integer value of the remainder when the first argument is divided by the second argument. If the second argument is 0, MOD returns the first argument.

Formula	**Result**
MOD(22,6)	4

MONTH

Format: MONTH(***date***)
OS: DOS and Unix
Scope: All
Returns: Numeric

MONTH returns a number corresponding to the month of the year in the date expression.

Formula	**Result**
MONTH("September 15, 1988")	9.0

MONTHNAME

Format: MONTHNAME(***date***)
OS: DOS and Unix
Scope: All
Returns: Text

MONTHNAME returns the name of the month in the date expression.

Formula	**Result**
MONTHNAME("09-15-88")	"September"
MONTHNAME(19180)	"July"

MOUSEINFO

Format: MOUSEINFO(***constant <,numeric>***)
OS: DOS and Unix
Scope: All
Returns: Numeric

This function returns, and sometimes changes, information about the current mouse status. The first parameter is a named constant indicating the value read or changed.

The arguments are as follows:

MOUSEINFO(constant, new value)

If the second parameter is used to change the mouse status, MOUSEINFO returns the original setting for subsequent restoration. Note that only some values can be changed, see the table below:

Constant	#	Description	Use
m_buttons	0	Returns bits indicating which buttons are pressed (see the table below). This is an alternative to using MOUSEINFO with the following six button constants.	R
m_leftup	1	Indicates the left button is not held down.	R
m_leftdown	2	Indicates the left button is held down.	R
m_rightup	3	Indicates the right button is not held down.	R
m_rightdown	4	Indicates the right button is held down.	R
m_middleup	5	Indicates the middle button is not held down.	R
m_middledown	6	Indicates the middle button is held down.	R
m_row	7	Row of pointer	R
m_col	8	Column of pointer	R
Use: R = Read, W = Write			

Constant	#	Description	Use
m_x	9	x coordinate pixel of pointer	R
m_y	10	y coordinate pixel of pointer	R
m_active	12	Mouse active	RW
m_nobuttons	13	Number of buttons	R
m_cursoron	14	Mouse cursor shown	RW
m_keyright	15	Value of right mouse button	RW
Use: R = Read, W = Write			

MOUSEINFO(m_buttons) will return one of the following values:

0000001 - Left button down

0000010 - Left button up

0000100 - Middle button down

0001000 - Middle button up

0010000 - Right button up

0100000 - Right button down

1000000 - Double button

Example:

```
WHILE TRUE
    if MOUSEINFO(m_leftdown)
        SCREEN PRINT 1 1 15 0 “Left Mouse Down”
    ELSEIF MOUSEINFO(m_leftup)
        SCREEN PRINT 1 1 15 0 “Left Mouse Up “
    ELSEIF MOUSEINFO(m_rightdown)
        SCREEN PRINT 2 1 15 0 “Right Mouse Down”
    ELSEIF MOUSEINFO(m_rightup)
        SCREEN PRINT 2 1 15 0 “Right Mouse Up “
    END IF
    SCREEN PRINT 3 1 15 0 “Mouse row =”&str(mousinfo(m_row))
    SCREEN PRINT 4 1 15 0 “Mouse col =”&str(mousinfo(m_col))
    IF NEXTKEY>0 AND INCHAR={esc}
        EXIT WHILE
    END IF
END WHILE
```

N

Format: N(***block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

N returns the value in the cell in the upper left corner of the block referenced in the argument. If the contents of that cell is text, N returns 0.

Although N returns the result of a single cell, the argument specified must be in the form of a block reference. You can use the ! character at the beginning of a cell reference to force ANGOSS to interpret it as a block reference.

In the following examples, the value in cell r6c4 assumed to be $255.19.

Formula	Result
N(r6:15c4:5)	$255.19
N(!r6c4)	$255.19

NA

Format: NA
OS: DOS and Unix
Scope: All
Returns: Text

NA returns the message N/A (Not available). NA is often used during worksheet development to indicate that some information has not yet been entered or determined. Formulas referencing a cell returning NA also return NA. Thus, NA allows you to indicate when not all data upon which a result is dependent are available.

NEXTKEY

Format: NEXTKEY
OS: DOS and Unix
Scope: All
Returns: Numeric

NEXTKEY returns the ANGOSS key value of the next keystroke waiting in the keyboard buffer, or 0 if no key has been pressed. Unlike INCHAR, NEXTKEY does not remove this character from the buffer. Program execution does not pause when NEXTKEY is calculated.

NEXTKEY_SOURCE

Formats: NEXTKEY_SOURCE
OS: DOS and Unix
Scope: All
Returns: Numeric

This function indicates the source of the next keystroke waiting in the keyboard buffer. It returns an integer where:

1 = From keyboard

2 = From macro (quick keys)

3 = From voice (some voice recognition software only)

4 = From mouse

5 = From button

See also LASTKEY_SOURCE.

NOCHANGE

Format: NOCHANGE
OS: DOS and Unix
Scope: All
Returns: Conditional

NOCHANGE is often used in logical formulas to retain the current contents of a data reference. NOCHANGE provides that no change is made to the data.

For example, you can use NOCHANGE to create a formula that causes data to be entered into a data-file field only if the field currently has no data in it. The following example stores a record's date of initial entry but provides that field will not be altered when the record is updated.

Example:

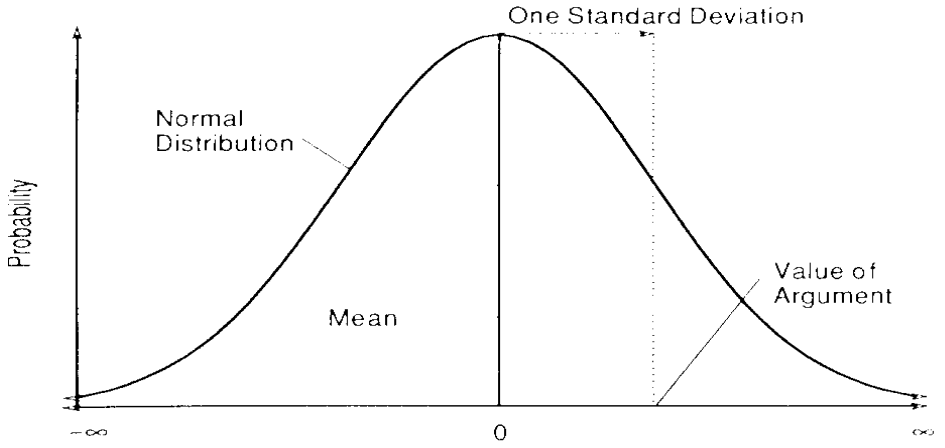
The formula IF [Datefield] = BLANK THEN TODAY ELSE NOCHANGE returns today's date if no date has yet been entered for field 3. Otherwise, it makes no change to the data in the field.

NORMAL

Format: NORMAL(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

NORMAL returns a number randomly selected from a normal distribution. The numeric expression specifies the standard deviation of the distribution.

Figure 2-6



NOT

Format: NOT(***logical***)
OS: DOS and Unix
Scope: All
Returns: Numeric

NOT evaluates a logical expression and returns 0 if the expression is true or 1 if the expression is false.

Formula	**Result**
NOT(2 > 1)	0
NOT(10 = 100)	1

NOW

Format: NOW
OS: DOS and Unix
Scope: All
Returns: Numeric

NOW returns a decimal number (integer and fraction) which represents the current date and time. The whole number returned identifies the date as the number of days from the start of the century to the current date. The fraction returned identifies the time of day as a fraction of the 24-hour day.

NOW uses the operating system's current date and time values. The accuracy of the result is based on whether the correct date and time were entered into the system when it was started. ANGOSS automatically updates NOW formulas whenever recalculations are performed.

In the following example, the current date and time are 10:06 a.m., October 6, 1988.

Formula	**Result**
NOW	32421.42

NPV

Format: NPV(***numeric, item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

NPV (net present value) calculates the present value of a group of future returns, discounted at a specified rate, and reduced by the amount of the initial investment. "Net present value" is often used to project whether a proposed investment will return not only the initial investment and the projected cash flow requirements, but additional funds as well.

NPV is calculated from the beginning of the time periods, rather than the end. The arguments are used as follows:

NPV (***interest rate, cash flow***)

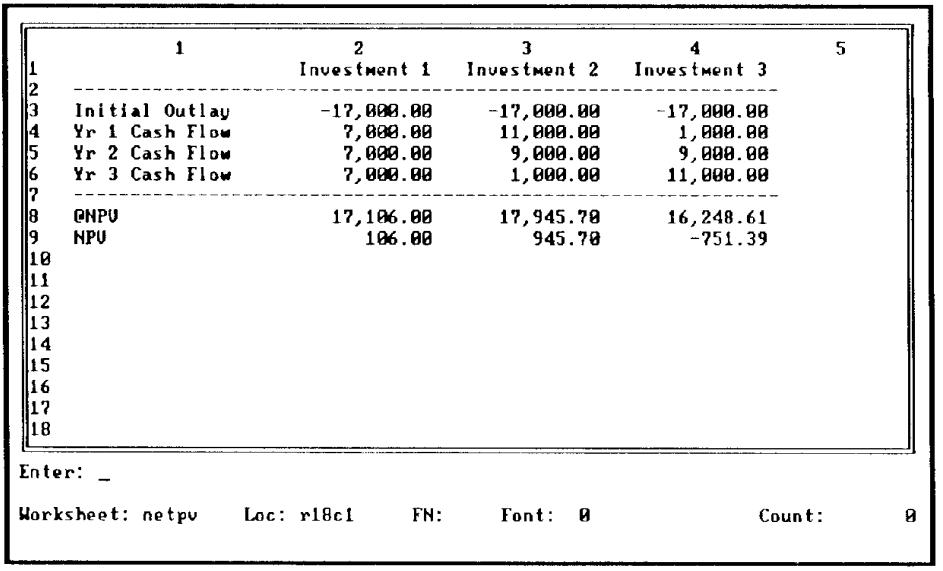
If the item list is defined as a worksheet block reference, the block must be one-dimensional (i.e., all cells in a single row or a single column).

NOTE: Empty cells/fields in a block referenced by the item list are interpreted as 0.

Formula	**Result**
NPV(.11,r3:6c2)	106.00
NPV(.11,r3:6c3	945.70
NPV(.11,r3:6c4) -	751.39

These examples assume that the formulas refer to the worksheet in Figure 2-7 of ***Formula Reference Manual***.

Figure 2-7



	1	2	3	4	5
1		Investment 1	Investment 2	Investment 3	
2					
3	Initial Outlay	-17,000.00	-17,000.00	-17,000.00	
4	Yr 1 Cash Flow	7,000.00	11,000.00	1,000.00	
5	Yr 2 Cash Flow	7,000.00	9,000.00	9,000.00	
6	Yr 3 Cash Flow	7,000.00	1,000.00	11,000.00	
7					
8	@NPV	17,106.00	17,945.70	16,248.61	
9	NPV	106.00	945.70	-751.39	
10					
11					
12					
13					
14					
15					
16					
17					
18					

Enter: _

Worksheet: netpv Loc: r18c1 FN: Font: 0 Count: 0

Example:

Suppose that you are selecting an investment from three possible plans. All three plans require the same initial $17,000 investment, and all three return the same total cash flow. The annual cash flows from the first plan are equal. Cash flows from the second plan are larger in the earlier years and become smaller. The third plan's cash flows are small at the beginning and grow larger.

Based upon the results of the formulas (and assuming all other considerations are equal), the second plan appears to be the best choice because it produces the largest net present value. The first plan produces a smaller but still positive net present value, while the third plan results in a negative net present value.

@NPV

Format: @NPV(***numeric, item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

@NPV is used to calculate net present value using a formula slightly different from NPV. @NPV assumes that each return is calculated and received at the end of each time period.

The arguments for @NPV are:

@NPV(***interest rate, cash flow***)

If the item list is defined as a block reference, the block must be one-dimensional, i.e., a single row or a single column.

Formula	**Result**
@NPV(.11,r4:6c2)	17,106.00
@NPV(.11,r4:6c3)	17,945.70
@NPV(.11,r4:6c4)	16,248.61

The examples assume amounts corresponding to those in Figure 2-7 of ***Formula Reference Manual***. Notice that the initial investment amount (--17000) is not included in the block references in the @NPV formulas.

As in the example for NPV, suppose that you are selecting an investment from three possible plans. All three plans require the same $17,000 investment, and all three return the same total cash flow. The annual cash flows from the first plan are equal. Cash flows from the second plan are larger in the earlier years and become smaller. The third plan's cash flows are small at the beginning and grow larger.

Since @NPV by itself does not return net present value, you must insert an additional calculation into the formula. To obtain net present value using @NPV, reduce the amount returned by the amount of the initial investment.

Formula	Result
@NPV(.11,r4:6c2)+r3c2	106.00
@NPV(.11,r4:6c3)+r3c3	945.70
@NPV(.11,r4:6c4)+r3c4	-751.39

Based upon the results of the adjusted formulas, you can then determine which investment provides the greatest positive net present value.

NULL

Format: NULL
OS: DOS and Unix
Scope: All
Returns: Text

NULL returns the null text character, ASCII character number 0. NULL is sometimes described as the "empty string." NULL is not equal to a space character (ASCII 32) or to the numeric value 0, nor are blank objects necessarily equal to NULL.

NOTE: Another way of specifying NULL is to type "", two double quotes with no intervening space (e.g., $var = "" is the same as $var = NULL.

OFFSETOF
OFFSETOFPM

Format: OFFSETOF(***pointer***)
OS: Differences
Scope: All
Returns: Numeric

OFFSETOF and OFFSETOFPM extract the offset portion of a real memory pointer and a protect mode pointer respectively. The argument is the passed address. For example, with a real memory address, this function is equivalent to the following formula:

OFFSETOF(***pointer***) == BITAND(***pointer***, 0xFFFF)

These functions return:

Function	286 mode	386 mode	Unix
OFFSETOF	16 bit	16 bit	32 bit
OFFSETOFPM	16 bit	32 bit	32 bit

OLDKEY

Format: `OLDKEY(` ***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

OLDKEY returns a numeric key code equivalent to that returned for a particular key in SmartWare 3.10. The argument should be an ANGOSS numeric key value, such as that returned by INCHAR, NEXTKEY, and KEYVALUE.

In ANGOSS the values of some keys and key combinations have changed from SmartWare 3.10. Changes to key values fall into three categories. For keys whose old value was below 32, the ANGOSS value equals the old value plus 736. Keys whose old value was between 32 and 255 have the same value in ANGOSS. For keys whose value was greater than 255, the ANGOSS value equals 256 plus the old value divided by 256.

The main purpose of OLDKEY is to allow you to update project files created under SmartWare 3.10 with a minimum of alteration. If your projects contain lines comparing the value of keys typed against specific constant values, you may wish to use OLDKEY to convert the input to SmartWare 3.10's value so that you do not have to revise the constants.

For example, if your project contains the statement `LET $X = INCHAR`, followed by a series of IF statements testing for particular key values in $X, you may want to revise the line as follows: `LET $X = OLDKEY(INCHAR)`. With this change, subsequent lines should not have to be changed to reflect the new ANGOSS key values.

PATH

Format: PATH(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

PATH returns the path requested in the argument, as specified with a named constant, as follows:

PATH(datapath)

Named constants include the following:

DATAPATH	SYSPATH	PAGEPATH
NOPATH	HOMEPATH	DEFPATH
CMDPATH	CURPATH	

DATAPATH. The data path is the path currently used for all data files. The current data path is set by the Tools New-Directory command. The default data path is established in the Tools Preferences menus.

SYSPATH. The system path is the path where the ANGOSS program files, such as SMART.AIF, are located. The system path is the directory where ANGOSS was installed, thus it does not change.

PAGEPATH. The paging path is the path where a temporary file is written if ANGOSS has to overflow to disk.

NOPATH. Nopath always returns null text. Specifying nopath allows ANGOSS to use the default path for any particular file operation.

HOMEPATH. The home path is the user's home directory. Under DOS, homepath is the same as curpath.

DEFPATH. The default path is the data path for the Main Menu. It is also used by the ANGOSS modules if no other path has been set using a module's Tools Preferences command. When ANGOSS is first invoked, defpath refers to the current directory, unless it is set otherwise by the -D switch on the command line or specified with Tools Preferences Global.

CMDPATH. The command path is set by the -D command line switch.

CURPATH. The current path refers to the directory ANGOSS is in when invoking ANGOSS. It may be used for some temporary files under DOS. It is also used (the same as defpath) if no other data path is specified by -D or the Preferences commands.

PHONEX

Format: PHONEX(***text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

PHONEX evaluates a text expression and returns a number representing its sound when pronounced. In general, words that sound the same are likely to return equal or similar values.

In the following example, "Coal" and "Cole" are pronounced identically and PHONEX returns the same number for each.

Formula	**Result**
PHONEX("Coal") = PHONEX("Cole")	1

PHONEX is useful for locating a name when you know the pronunciation but are uncertain of the exact spelling. For example, you might use the following formula in a database query:

```
PHONEX([lastname]) > PHONEX("mark") - 20 and
        PHONEX([lastname]) < PHONEX("mark") + 20
```

Since the word "mark" has a PHONEX value of 14224, this query formula selects names with values in a range from 14204 (14224--20) to 14244 (14224+20). The name "Markley," having a value of 14228, is selected, while the name "Manning," having the value 14186, is not.

NOTE: Experiment with various numeric ranges until you become familiar with the results returned by PHONEX. Words that you pronounce identically may not yield the same PHONEX value.

PI

Format: `PI`
OS: DOS and Unix
Scope: All
Returns: Numeric

PI returns the value of pi (3.14159265358979).

PMT

Format: `PMT(` ***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

PMT calculates the payment required over a given term at a fixed interest rate to equal the specified principal amount. The arguments are used as follows:

PMT (***principal amount, term, interest rate***)

Formula	**Result**
PMT(65988.40,30,.1)	7000.00
PMT(5000,36,.18/12)	180.76

Example:

If you financed a $5000 automobile for 36 months at an annual interest rate of 18%, your monthly payment would be $180.76. The formula for this calculation is `PMT(5000,36,.18/12)`, with the 18% annual interest rate being divided by 12 months to maintain the consistency of monthly increments in the formula.

NOTE: The interest rate is expressed as the rate per term. The term may be any unit of measure (month, week, year, etc.). To calculate the monthly payment given an annual interest rate use:

PMT (***principal, number of months, annual interest rate divided by 12***)

@PMT

Format: `@PMT(` ***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

@PMT calculates the payment required over a given term at a fixed interest rate to equal the specified principal amount. @PMT returns the same value as PMT, but the arguments are entered in a different order. The arguments are used as follows:

@PMT (***principal amount, interest rate, term***)

Formula	**Result**
@PMT(65988.40,.1,30)	7000.00
@PMT(5000,.18/12,36)	180.76

NOTE: The interest rate is expressed as the rate per term. The term may be any unit of measure (month, week, year, etc.). To calculate the monthly payment given an annual interest rate use:

@PMT (***principal, annual interest rate divided by 12, number of months***)

POINTEROF
POINTEROFPM

Format: POINTEROF(***segment, offset***)
OS: DOS and Unix
Scope: All
Returns: Numeric

POINTEROF creates and returns a pointer from a passed real memory segment and offset. POINTEROFPM is the protect mode version of this function. For example, the real memory function is equivalent to the following formula:

POINTEROF(***segment, offset***) == (***segment*** * 65536) + ***offset***

These functions return:

Function	286 mode	386 mode	Unix
POINTEROF	32 bit	32 bit	32
POINTEROFPM	32 bit	32 bit	32

POWER

Format: POWER(***numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

POWER raises the value of the first expression to the power represented by the value of the second expression. Using a fractional power results in a root. If the power includes a fraction, the first expression must be an integer.

NOTE: You may also use the ^ operator to raise a number to a power (e.g., 10^2 = 100).

Formula	Result
POWER(10,2)	100
POWER(100,.5)	10

PRECORD

Format: PRECORD
OS: DOS and Unix
Scope: Database
Returns: Numeric

PRECORD returns the physical record number of the current record in the main data-file of the view. The physical record number is the absolute number of the record in the sequential order in which it was entered.

PRECORDS

Format: PRECORDS
OS: DOS and Unix
Scope: Database
Returns: Numeric

PRECORDS returns the total number of records contained in the main data-file of the view.

PRINCIPAL

Format: PRINCIPAL(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

PRINCIPAL calculates the principal amount which would produce a fixed, regular payment over a given term at a specified interest rate. The arguments are used as follows:

PRINCIPAL(***payment amount, term, interest rate***)

Formula	**Result**
PRINCIPAL(7000,30,.1)	65988.40
PRINCIPAL(150,36,.18 / 12)	4149.10

Example:

If you made monthly payments of $150 for a term of 36 months at an annual interest rate of 18%, the principal amount that you would have paid at the end of the 36 month term would be $4149.10. The formula for this calculation is `PRINCIPAL(150,36,.18/12)`, with the 18% annual interest rate being divided by 12 months to maintain the consistency of monthly increments in the formula.

PROCESS_CREATE

Format: PROCESS_CREATE(***text***, ***text*** <, ***text***, ...>)
Scope: All
Available: Version 2.65 and Higher
Returns: Numeric

PROCESS_CREATE executes an external program. The first parameter is the full path of the program (including the program name), the second parameter is the program name, the

remaining optional parameters are command line arguments to the executing program. All parameters must be strings.

For example, the following function will start the Windows Paintbrush utility from an SPL program. Because Paintbrush can take a bitmap file as a command line parameter, this function call will load the cars.bmp file into Paintbrush as well.

```
process_create( "c:\windows\pbrush", "pbrush", "c:\windows\cars.bmp" )
```

If the function is successful, PROCESS_CREATE will return a process number in Unix otherwise, zero is returned.

PROPER

Format: PROPER(***text***)
OS: DOS and Unix
Scope: All
Returns: Text

PROPER converts each word in the text expression so that the first letter of the word is capitalized and the rest of the word is in lower case characters.

Formula	**Result**
PROPER("jamEs d. kemPtOn")	"James D. Kempton"

PV

Format: PV(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

PV calculates the present value today of a future lump sum payment at a given interest rate. The arguments are used as follows:

PV(***principal amount, term, interest rate***)

Formula	Result
PV(5000,5,.1)	3104.61

Example:

Suppose you were given the option of receiving a lump sum payment today on an investment asset in lieu of receiving a specified rate of interest over a specified term with the lump sum payment being made at the end of the term. To determine what the amount of the lump sum payment today should be, you would use a PV formula. Using the values listed above, if you were to receive 10% annual interest for five years, with a lump sum payment of $5000 being made to you at the end of the five years, the formula to determine the current value of the $5000 would be `PV(5000,5,.1)`. The result of $3104.61 is the amount you should receive if you elect to take the lump sum payment today.

@PV

Format: `@PV(` ***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

@PV calculates the present value today of a stream of equal payments (an annuity) at a given interest rate over a specified period of time. @PV returns the same value as PVA, but the arguments are entered in a different order. The arguments are used as follows:

@PV(***payment amount, interest rate, term***)

Formula	Result
@PV(5000,.12,10)	28251.12

PVA

Format: PVA(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

PVA calculates the present value today of a stream of equal payments (an annuity) at a given interest rate over a specified period of time. The arguments are used as follows:

PVA(***payment amount, term, interest rate***)

Formula	**Result**
PVA(5000,10,.12)	28251.12

Example:

Suppose you were given the option of receiving a lump sum payment today on an investment asset in lieu of receiving scheduled equal payments at a specified interest rate for a specified period of time. To determine what the present value of the investment asset would be, you would use PVA. For instance, if you were to receive annual payments of $5000 for a term of ten years, with an annual interest rate of 12%, the formula to determine the present value would be `PVA(5000,10,.12)`. The result of $28,251.12 would be the lump sum payment you should accept today in lieu of accepting the annuity payments.

RAND

Format: RAND

OS: DOS and Unix

Scope: All

Returns: Numeric

RAND returns a number selected randomly from between 0 and 1. RAND is equivalent to UNIFORM(1).

NOTE: Each of ANGOSS' random number functions, EXPONENTIAL, NORMAL, RAND, and UNIFORM, returns numbers from a different distribution.

RATE

Format: RATE(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

RATE returns the periodic interest rate for an investment, based on the arguments of the formula:

RATE(***future value, present value, term***)

The rate returned represents the interest rate for the period specified, not necessarily an annual interest rate. If the interest is compounded quarterly, for example, the value returned must be multiplied by four to compute the annual rate.

Formula	**Result**
RATE(20000,5000,10*12)	0.011619
RATE(20000,5000,10*4)	0.035265

Example:

Suppose you invested $5000 in an IRA that matures in 10 years with a value of $20,000. In the first formula above, you are seeking a monthly interest rate return (indicated by the 10 year term being multiplied by 12 months), with the result being 1.16%. In the second formula, you are seeking a quarterly interest rate return (indicated by the 10 year term being multiplied by 4 quarters), with the result being 3.53%.

READPTR

Format: READPTR(***pointer*** <,***index***> <,***index***> <,***index***>)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Conditional

If the pointer passed to READPTR is the address of a variable, the value of that variable is returned. With an array, the address plus the index arguments (one for each dimension) are used to return the value of a specific element.

```
$value = READPTR( VP_variable )
$element = READPTR( AP_array, 3 )
```

For a discussion on pointer usage, refer to the section, ***Pointers to Arrays and Variables***.

RECORD

Format: RECORD
OS: DOS and Unix
Scope: Database
Returns: Numeric

RECORD returns the logical record number of the current record. If the file is in index order, the logical record number will usually differ from the physical record number. If the file is in key order, RECORD returns the physical record number.

RECORDS

Format: RECORDS

OS: DOS and Unix

Scope: Database

Returns: Numeric

RECORDS returns the total number of logical records in the file for the current order of the file. If the file is in an index order, the number of logical records may be smaller than the number of physical records.

REPEAT

Format: REPEAT(***text, numeric***)

OS: DOS and Unix

Scope: All

Returns: Text

REPEAT returns the text expression repeated the number of times indicated in the numeric expression. The maximum length of the resulting expression is 1000 characters.

Formula	**Result**
REPEAT("*",5)	"*****"
REPEAT("+--",5)	"+--+--+--+--+--"

REPLACE

Format: REPLACE(***text, numeric, numeric, text***)
OS: DOS and Unix
Scope: All
Returns: Text

REPLACE removes a specified number of characters from an existing text string and replaces those characters with new text, as determined by the arguments of the function:

REPLACE(***existing text, start value, length value, new text***)

REPLACE deletes the number of characters specified as the length, starting with the character at the starting position, and replaces these characters with the new text. The first character is located at position 0. If you specify a start position that is greater than the number of existing text characters, the new text will be added to the end of the existing text, as illustrated in the third example following.

In the first three examples, r7c2 contains the text string "The difference is speed." In the fourth example, r4c1 contains the text string "The difference is either power or speed."

Formula	**Result**
REPLACE(r7c2,18,5,"power")	"The difference is power."
REPLACE(r7c2,15,2,"should be")	"The difference should be speed."
REPLACE("The difference is speed.", 14,21,"s are speed and power.")	"The differences are speed and power."
REPLACE(r4c1,18,16,"")	"The difference is speed."

REPLACESTR

Format: REPLACESTR(***text, text, text***)
OS: DOS and Unix
Scope: All
Returns: Text

This function performs a 'find and replace' on the passed string (parameter 1). All occurrences of the find string (parameter 2) are replaced with the replacement string (parameter 3). The following example sets $str to "1B3B1"

```
$str = replaceall("12321","2","B")
```

RIGHT

Format: `RIGHT(` ***text, numeric***)
OS: DOS and Unix
Scope: All
Returns: Text

RIGHT returns a line of text characters from the text expression consisting of the rightmost number of characters specified by the numeric expression.

Formula	**Result**
RIGHT("Sept 15, 1989",4)	"1989"
RIGHT("James D. Kempton",3)	"ton"

ROUND

Format: ROUND(***numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

ROUND returns the value of the first numeric expression rounded to the number of decimal places specified in the second argument. If the second argument is negative, the number is rounded to that power of ten.

Formula	**Result**
ROUND(1.1234567,5)	1.12346
ROUND(32.55,1)	32.6
ROUND(1959.8,-2)	2000

ROW

Format: ROW
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

ROW returns the integer number of the worksheet row in which the formula containing the function is located.

ROWS

Format: ROWS(***block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

ROWS returns a number equivalent to the number of rows in a worksheet block.

Formula	**Result**
ROWS(r23:109c2:4)	87
ROWS(!r2c2)	1

S

Format: S(***block***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Text

S returns the text contents of the cell in the upper left corner of the block referenced in the argument. If that cell contains a numeric value, S returns NULL.

NOTE: ANGOSS interprets a reference to a single cell as a block reference if you precede it with the ! character.

In the following examples, SM is a named cell at r10c1 containing the text "Toronto".

Formula	**Result**
S(r10:15c1:3)	"Toronto"
S(!SM)	"Toronto"

SCR_TEXT

Format: SCR_TEXT(***row, col, length***)
OS: DOS and Unix
Scope: Spreadsheet
Available: Version 2.65 and Higher
Returns: Text

This returns the string of characters found on the screen at the specified row, column, and maximum length. Trailing spaces are trimmed.

SCRCOLUMN, SCRHEIGHT, SCRLINE, SCRMODE, and SCRWIDTH

Format: `SCRCOLUMN`
OS: Differences (SCRMODE)
Scope: All
Returns: Numeric

These functions return information about the screen and the location of data printed on the screen by the Message and Screen Print project commands. With these functions you can quickly calculate locations for printing data on the screen through project processing.

SCRCOLUMN returns the number of the column following the last (rightmost) character printed on the screen. SCRLINE returns the line number of the last character printed on the screen. These functions are affected only by the Message and Screen Print commands.

SCRHEIGHT returns the screen height in lines. SCRWIDTH returns the screen width in columns. SCRMODE returns 1 for a graphics display mode or 0 for a character-based display mode.

The following examples assume the use of an IBM CGA display adapter in character mode.

Formula	Result
SCRHEIGHT	25
SCRWIDTH	80

NOTE: SCRMODE will always return 0 under UNIX.

SECOND

Format: SECOND(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

SECOND extracts the second from a time decimal number, such as that generated by @TIME, and returns a numeric value between 0 and 59. The fractional portion of any number can be interpreted as a time of day and assigned a time decimal number.

Formula	Result
SECOND(0.488136574074)	55
SECOND(0.921909722222)	33
SECOND(3/25)	48

SECONDS

Format A: SECONDS(***time***)
Format B: SECONDS(***date*** <, ***time***>)
OS: DOS and Unix
Scope: All
Returns: Numeric

If you use Format A, SECONDS returns the number of seconds elapsed between the time represented by the time expression and the beginning of the day (00:00:00).

If you use Format B, SECONDS returns the number of seconds elapsed between the date (and time, if included) and 00:00:00 on 01/01/00 (the beginning of the 20th century).

Formula	Result
SECONDS("06:19:11a")	22751
SECONDS("Feb 22, 1988", "11:55:23")	2781604523

SECONDS accepts only date expression types 1 and 2. Type 1 is a text expression representing a date. Type 2 is a numeric expression representing the number of days before or after the beginning of the century.

SEGMENTOF
SEGMENTOFPM

Format: SEGMENTOF(***pointer***)
OS: Differences
Scope: All
Returns: Numeric

SEGMENTOF and SEGMENTOFPM extract the segment portion of a real memory pointer and a protect mode pointer respectively. The argument is the passed address. For example, with a real memory address, this function is equivalent to the following formula:

SEGMENTOF(***pointer***) == BITAND(***pointer***/65536, 0xFFFF)

These functions return:

Function	286 mode	386 mode	Unix
SEGMENTOF	16 bit	16 bit	0
SEGMENTOFPM	16 bit	16 bit	0

SELECT

Format: SELECT(***select list***) ELSE ***expression***
OS: DOS and Unix
Scope: All
Returns: Conditional

SELECT creates a special type of logical formula which provides for a set of possible responses to a list of specified conditions.

The select list consists of pairs of expressions separated by a comma and enclosed in parentheses. Each pair contains a logical expression to be evaluated and an item to be returned if the logical expression it is paired with is the first in the list to be found true. You may include an ELSE statement in the formula to provide a response if none of the logical expressions in the select list are true. SELECT returns an error when no ELSE statement is included and none of the logical expressions are true.

In the following example the field [Qty] is assumed to contain 75.

Formula	**Result**
SELECT([Qty]<50,0)([Qty]<100,.05) ([Qty]<500,.1) ([Qty]<1000,.25) ELSE ERROR	.05

The function proceeds through the select list and returns the item paired with the first logical expression which is evaluated as true. Subsequent logical expressions in the list, which may also be true, are ignored.

A SELECT formula is a special type of IF-THEN-ELSE statement. The following example shows how the SELECT formula would look if written as an IF-THEN-ELSE formula:

```
IF [Qty] < 50 THEN 0
        ELSE IF [Qty] < 100 THEN .05
```

```
ELSE IF [Qty] < 500 THEN .1
ELSE IF [Qty] < 1000 THEN .25
ELSE ERROR
```

SETREG

Format: SETREG(***numeric, numeric expression***)
OS: DOS Only
Scope: All
Returns: Numeric

The SETREG function presets values to be loaded into CPU registers prior to an Interrupt command. The first argument is a register name indicated below. The second argument is the value to be placed in the register. The following example sets the segment address of the variable #var1 as the value to be loaded in the AX register by the next Interrupt command.

```
SETREG(AX,DOSSEG(#var1))
```

This function is designed for use with the Project Development Language's Interrupt command. Consult ***Project Processing*** for details on using the Interrupt command.

NOTE: SETREG and the Interrupt command were designed for use in a DOS environment. SETREG and the . . . DOS environment. A run-time error message will be produced under Unix.

AX	BX	CX	DX
DI	SI	DS	ES
FLAGS			

SIN

Format: `SIN(` ***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

SIN calculates the sine of the numeric expression. The numeric expression is a value expressed in radians.

Formula	**Result**
SIN(0.1)	.0998334166468

SINH

Format: `SINH(` ***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

SINH calculates the hyperbolic sine of the numeric expression.

Formula	**Result**
SINH(0.1)	.1001667500198

SLN

Format: SLN(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

SLN computes the straight-line yearly depreciation of an asset. The arguments are used as follows:

SLN(***cost, salvage value, life***)

Formula	**Result**
SLN(10000,1000,7)	1285.71

Example:

To determine the annual depreciation expense of a $10,000 automobile which has a useful life of seven years and a salvage value of $1000 at the end of the seven years, the formula is ***SLN(10000,1000,7)***. The result, $1285.71, is the uniform depreciation expense for each of the seven years during which the car is depreciated.

SQRT

Format: SQRT(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

SQRT calculates the square root of the numeric expression. The numeric expression must be greater than or equal to 0.

NOTE: To obtain roots other than the square root, use the function POWER with a fractional second argument.

Formula	Result
SQRT(100)	10

SSGET

Format: SSGET(*<text,> numeric, numeric*)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Conditional

SSGET returns the result stored in a cell. The data retrieved is from the current worksheet unless another worksheet is specified in the optional first argument. The arguments are used as follows:

SSGET(*<worksheet name,> row number, column number*)

The example shown assumes that r4c1 contains the value 12.

Formula	Result
SSGET(4,1)	12

SSGET is one means of creating a "calculated" cell reference because its arguments can be formulas returning the numbers of a target row and column. It is primarily intended for use in project processing for applications other than user-defined functions. Because SSGET is not encompassed in the natural or minimal worksheet recalculation orders, its use in formula cells and user-defined formula functions is not recommended. Another method of using a calculated cell reference is to use a MAKECELL formula as an argument for INDIRECT.

SSKEY

Formats: SSKEY(***numeric***)

OS: DOS and Unix

Scope: Spreadsheet

Returns: Numeric

SSKEY is used by project processing programs to control cursor movement. It is a good alternative to SUSPEND/KEYS statements since unnecessary screen flashing does not occur and performance is improved.

This function passes a given keystroke to the underlying module. If a cursor movement keystroke was processed, 1 is returned, otherwise, the keystroke was not a cursor movement key and 0 is returned.

SSKEY handles all movement keys and mouse activity.

Example 1:

```
SSKEY({down})  'Move the cursor down one line
```

Example 2:

```
'Control navigation by cursor keys, mouse clicks and scroll bar
WHILE #key<>{esc}
    #key = INCHAR
    IF SSKEY(#key)    'Key was cursor movement
        '(process command)
    END IF
END WHILE
```

SSPOS

Format: SSPOS(***numeric, numeric***)

OS: DOS and Unix

Scope: Spreadsheet

Returns: Text

SSPOS returns the worksheet name and cell that occupies the passed screen coordinates in the form of a string: <work sheet>.r<row>c<col> (e.g., “mysheet.r51c313”). The arguments are as follows:

SSPOS(row, column)

Example:

```
$d = SSPOS(CLICKINFO(m_row), CLICKINFO(m_col))
message “The data you clicked on is: “|CELLTEXT($d)
```

Note that if the screen coordinates supplied are on a border, status line, or other non worksheet locations, a null string (“”) is returned.

SSPOSROW and SSPOSCOL

Formats:

SSPOSROW(***numeric, numeric***)

SSPOSCOL(***numeric, numeric***)

OS: DOS and Unix

Scope: Spreadsheet

Returns: Numeric

The SSPOSROW and SSPOSCOL functions return an integer representing the worksheet row or worksheet column that occupies the passed screen coordinates.The arguments are as follows:

SSPOSROW(row, column) and SSPOSCOL(row, column)

Example 1:

```
‘Put “Hello” into the ‘clicked’ cell
$ws = SSPOSWS( CLICKINFO(m_row), CLICKINFO(m_col) )
#r = SSPOSROW( CLICKINFO(m_row), CLICKINFO(m_col) )
#c = SSPOSCOL( CLICKINFO(m_row), CLICKINFO(m_col) )
IF #r > 0 ‘Check if user clicked a cell
    SSPUT( “Hello”, $ws, #r, #c )
END IF
```

Example 2:

```
’Calculate the top displayed row of the worksheet assuming
’the border and numbers are on.
#r = SSPOSROW(3,5)
MESSAGE “The first row displayed is row # “|str(#r)
```

Note that if the screen coordinates supplied are on a border, status line, or other non worksheet locations, 0 is returned.

SSPOSWS

Format: SSPOSWS(***numeric, numeric***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Text

SSPOSWS returns a string containing the worksheet name that occupies the passed screen coordinates. The arguments are as follows:

SSPOSWS(row, column)

Note that if the screen coordinates supplied are on a border, status line, or other non worksheet locations, a null string (“”) is returned.

SSPUT

Format: SSPUT(***expression, <text,> numeric, numeric***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Numeric

SSPUT places the data returned by the first expression into a cell identified in the remaining arguments, returning TRUE (1) if the procedure is successful or FALSE (0) if the procedure is not successful. SSPUT operates on the current worksheet unless the name of another active worksheet is specified. The arguments are used as follows:

SSPUT(***data, <worksheet name,> row number, column number***)

SSPUT fails (and returns 0) if you attempt an assignment to a locked cell or a cell containing a formula. If the expression results in an error or in NA, the result is entered into the cell as text.

IMPORTANT: Use of SSPUT in formula cells is subject to limitations similar to those described for LET. See the paragraph marked IMPORTANT under ***LET*** for more information.

STD

Format: STD(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

STD calculates the standard deviation of a population for the items in the item list. Text items are considered to have a value of 0. Blanks are ignored. The list must contain at least 2 entries.

STD returns a value equivalent to the square root of the variance as calculated by the @VAR function.

Formula	Result
STD(4.5,6.7,27.2,32.4)	12.2635639192

STDEV

Format: STDEV(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

STDEV calculates the standard deviation of a sample for the numeric items in the item list. All text items or blanks in the list are ignored. The list must contain at least 1 entry.

STDEV returns a value equivalent to the square root of the variance as calculated by the VAR function.

Formula	Result
STDEV(4.5,6.7,27.2,32.4)	14.1607438599

STR

Format: STR(***expression <, numeric>***)
OS: DOS and Unix
Scope: All
Returns: Text

STR returns a text representation of the numeric expression in the first argument. The number of decimal places represented is the same as that of the numeric expression. The optional second argument allows you to specify the number of significant digits to be returned. Nonsignificant trailing zeros are omitted. STR converts the number to E-

notation when it would exceed 15 places to the left or right of the decimal. If the expression is already text, it is returned unchanged.

Formula	Result
STR(11.2300349)	"11.2300349"
STR(.68100,2)	"0.68"

SUM

Format: SUM(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

SUM calculates the sum of the numeric items in the item list. Any text items or blanks in the list are ignored.

Formula	Result
SUM(2,2)	4
SUM(1,3,5,7,9,9)	34
SUM(2+2,"18",3*7,3^2)	34

SUMSQ

Format: SUMSQ(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

SUMSQ calculates the sum of the squares of the numeric items in the item list. Any text items or blanks in the list are ignored.

Formula	Result
SUMSQ(2,3,10)	113

SWAPCASE

Format: SWAPCASE(***text***)
OS: DOS and Unix
Scope: All
Returns: Text

SWAPCASE will swap the case of each character in the argument. Upper case characters will be converted to lower case and lower case characters will be converted to upper case.

Formula	Result
SWAPCASE("Swap Case")	"sWAP cASE"
SWAPCASE("Toronto")	"tORONTO"

SYD

Format: SYD(***numeric, numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

SYD calculates depreciation using the sum-of-the-years' digits method. The depreciation value is determined by the arguments of the function:

SYD(***cost value, salvage value, useful life, term***)

This function computes depreciation at an accelerated rate, whereby a greater depreciation expense occurs early in the term. The depreciable cost is the actual cost less the salvage value. The useful life is the term over which the asset is depreciated.

Formula	Result
SYD(15000,2000,7,1)	3250
SYD(15000,2000,7,3)	2321.43
SYD(15000,2000,7,6)	928.57

Example:

Suppose you purchase a $15,000 automobile which has a useful life of seven years and a salvage value of $2000 at the end of the seven years. The first formula listed above is used to determine what the depreciation rate would be for the first year of the car's useful life and returns a result of $3250.00. The second formula computes the depreciation rate for the third year of the car's useful life and returns a result of $2321.43. The third formula returns a result of $928.57, indicating the depreciation rate for the sixth year of the car's useful life.

SYSVAR

Format: SYSVAR(***constant <,value>***)
OS: DOS and Unix
Scope: All
Returns: Original value

The SYSVAR function will read and/or change internal system variables. Essentially, it is a convenient way of issuing SMARTPEEK and SMARTPOKE statements. The first parameter specifies the system variable to inspect or change. The second parameter, which is optional and only allowed in some cases, is the new value to assign. For a detailed listing of constants, refer

to the documentation on SMARTPEEK and SMARTPOKE in the ***Project Processing*** manual.

Example 1:

```
‘Store current value of INSERT and set it to 0
#ins = SYSVAR($_ins,0)
‘Commands that require INSERT to be off
...
‘Restore original INSERT value
SYSVAR($_ins,#ins)
```

Example 2:

```
‘Remember repaint mode
#rep = SYSVAR($_paint)
REPAINT OFF
‘Commands the require repaint off
...
‘Put repaint on, (if it was off)
IF #rep
   REPAINT ON
END IF
```

TABLEAVERAGE

Format: TABLEAVERAGE(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

TABLEAVERAGE computes the average value in the specified field for all table records in a view record. The field must be in a table in the view in which it is referenced. Only non-blank fields are included in the calculation.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

TABLEAVERAGE([price])

TABLEAVERAGE([price], [quantity] > 10)

The first example computes the average of all non-blank values in the table's [price] field. The second example computes the average of all the table's [price] values in records where another table field, [quantity], is greater than 10.

TABLECOUNT

Format: TABLECOUNT(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

TABLECOUNT counts the number of (non-blank) entries in the specified field for all table records in a view record. The field must be in a table in the view in which it is referenced.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

TABLECOUNT([price])

TABLECOUNT([price], [quantity] > 10)

The first example returns the count of entries in the table's [price] field. The second example returns the count of the table's [price] entries in records where another table field, [quantity], is greater than 10.

TABLELOOKUP

Format: TABLELOOKUP(***field, logical***)
OS: DOS and Unix
Scope: Database
Returns: Conditional

TABLELOOKUP returns the contents of a specified field in a table. The field must be in a table in the view in which it is referenced. The arguments are used as follows:

TABLELOOKUP(***result field, selection criterion***)

TABLELOOKUP searches the table beginning with the first record. It returns the contents of the result field in the first record it encounters where the selection criterion evaluates to a non-zero value.

The TABLELOOKUP examples are based on the following data. The column headings are the field titles and each row represents a table record. This table is similar to the example in Figure 3-8 of ***Formula Reference Manual***.

Fund	Return	Shares
Money Market	11.36	850
Kempton Mutual	10.92	1000
Canadian Superior Fund	10.12	5000
Hazel Group Securities	10.30	1000

The first formula returns the value in [Return] from the record containing "Money Market" in [Fund]. The second example returns the contents of [Fund] for the first table record containing a value greater than or equal to 1000 in [Shares].

Formula	**Result**
TABLELOOKUP([Return], [Fund] = "Money Market")	11.36
TABLELOOKUP([Fund], [Shares] >= 1000)	"Kempton Mutual"

NOTE: TABLELOOKUP and the Table SDb functions cannot be nested.

Example:

If your view contains a table, you may want to select records based upon data in the table. When you base a query upon table data matching the selection conditions you have specified, if a match is found, all of the table records associated with a particular view record are returned, not just the table records that contain matching data.

In the following query example, ANGOSS is instructed to search the table records for values greater than 100 in a table field titled [quantity]:

Invoice	Quantity	Unit Price	Total
	>100		

To enter a full formula expression selecting records from a table, enter an expression using the TABLELOOKUP function in conjunction with the NOT and ISERR functions. The first argument for TABLELOOKUP, in this example, is any field in the table; the second argument is the selection equation, as follows:

NOT(ISERR(TABLELOOKUP([quantity],[quantity]>100)))

NOTE: If this formula cannot find a value greater than 100 in the table during a query, it will fail. The NOT and ISERR functions are included to handle the resulting error message.

TABLEMAX

Format: TABLEMAX(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

TABLEMAX returns the largest value in the specified field for all table records in a view record. The field must be in a table in the view in which it is referenced.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

Example:

TABLEMAX([price])

TABLEMAX([price], [quantity] > 10)

The first example returns the maximum value in the table's [price] field. The second example returns the maximum value in the table's [price] field from among records where another table field, [quantity], is greater than 10.

TABLEMIN

Format: TABLEMIN(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

TABLEMIN returns the smallest value in the specified field for all table records in a view record. The field must be in a table in the view in which it is referenced.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

Example:

TABLEMIN([price])

TABLEMIN([price], [quantity] > 10)

The first example returns the minimum value in the table's [price] field. The second example returns the minimum value in the table's [price] field from among records where another table field, [quantity], is greater than 10.

TABLEREC

Format: TABLEREC(***field***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

The TABLEREC function returns the relative position of the current table record. The argument must be a text expression in the form of a field reference to a field in the table. The first record displayed in the table has position one, the second record has position two, etc.

TABLESTD

Format: TABLESTD(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

TABLESTD computes the standard deviation of a population for all table records in a view record. The field must be in a table in the view in which it is referenced. Only non-blank fields are included in the calculation.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

Example:

TABLESTD([scores])

TABLESTD([scores], [age] > 18)

The first example computes the standard deviation of a population for all non-blank values in the table's [scores] field. The second example computes the standard deviation of the table's [scores] field in records where another table field, [age], is greater than eighteen.

TABLESTDEV

Format: TABLESTDEV(***field <, logical>***)

OS: DOS and Unix

Scope: Database

Returns: Numeric

TABLESTDEV computes the standard deviation of a sample for all table records in a view record. The field must be in a table in the view in which it is referenced. Only non-blank fields are included in the calculation.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

Example:

TABLESTDEV([scores])

TABLESTDEV([scores], [age] > 18)

The first example computes the standard deviation of a sample for all non-blank values in the table's [scores] field. The second example computes the standard deviation of the table's [scores] field in records where another table field, [age], is greater than eighteen.

TABLESUM

Format: TABLESUM(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

TABLESUM returns the sum of the values in the specified field for all table records in a view record. The field must be in a table in the view in which it is referenced.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

Example:

TABLESUM([price])

TABLESUM([price], [quantity] > 10)

The first example returns the sum of all values in the table's [price] field. The second example returns the sum of all the table's [price] values in records where another table field, [quantity], is greater than 10.

NOTE: TABLESUM does not include deleted records when calculating the sum.

TABLESUMSQ

Format: TABLESUMSQ(***field <, logical>***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

TABLESUMSQ computes the sum of the squares for all table records in a view record. The field must be in a table in the view in which it is referenced. Only non-blank fields are included in the calculation.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

Example:

TABLESUMSQ([scores])

TABLESUMSQ([scores], [age] > 18)

The first example computes the sum of the squares for all non-blank values in the table's [scores] field. The second example computes the sum of the squares of the table's [scores] field in records where another table field, [age], is greater than eighteen.

TABLEVAR

Format: TABLEVAR(***field <, logical >***)
OS: DOS and Unix
Scope: Database
Returns: Numeric

TABLEVAR computes the sample variance for all table records in a view record. The field must be in a table in the view in which it is referenced. Only non-blank fields are included in the calculation.

You can enter a logical expression as an optional second argument. Each table record for which the expression returns TRUE is included in the calculation.

Example:

TABLEVAR([scores])

TABLEVAR([scores], [age] > 18)

The first example computes the variance for all non-blank values in the table's [scores] field. The second example computes the variance of the table's [scores] field in records where another table field, [age], is greater than eighteen.

TAN

Format: TAN(***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

TAN returns the tangent of the numeric expression. The argument is a value expressed in radians.

Formula	**Result**
TAN(.12)	.120579337211

TERM

Format: TERM(***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

TERM calculates the term over which a fixed payment must be made in order to equal a principal amount, assuming a given interest rate. The arguments are used as follows:

TERM(***principal amount, payment amount, interest rate***)

Formula	**Result**
TERM(65988.40,7000,.1)	30
TERM(5000,250,.18/12)	23.96

Example:

If you financed a $5000 car at an annual 18% interest rate and set your monthly payment at $250, the formula TERM(5000,250,.18/12) would return a result of 23.96 months as the term of the loan. The annual 18% interest rate is divided by 12 months to maintain the consistency of monthly increments in the formula. Remember

that the time periods for the interest rate and the payments must be the same, i.e., both must be in months, or both must be in years, etc. The first example includes a yearly payment schedule, while the second example shows a monthly schedule.

NOTE: The payment amount cannot be less than the interest amount for a given time period. The interest amount is calculated by dividing the principal by 100 and multiplying the results by the interest rate, as in the following example:

interest amount = 5000 / 100 * 18/12 = 75

Note that 75 is less than the $250 payment.

@TERM

Format: `@TERM(` ***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

@TERM calculates the term over which a fixed annuity payment, made at the end of every month, must be made to equal a specified future value, assuming a specified interest rate. The arguments are used as follows:

@TERM(***payment amount, interest rate, future value amount***)

Formula	**Result**
@TERM(150,.10/12,20000)	90.04

Example:

Suppose you started an investment account with the intention of using the funds at a future time to finance your child's college education. Having established $20,000 as the amount you will need, you plan to deposit $150 at the end of each month into an account earning 10% annual interest. To determine how many months it will take for you to accrue the $20,000 needed, use the formula `@TERM(150,.10/12,20000)`. The result returned is 90.04 months. The 10% annual interest rate has been divided by 12 months to maintain monthly increments in the formula expressions.

To compute the term of an annuity when the payment is made at the beginning of each month, rather than the end of each month, you should enter the formula as follows:

@TERM(***payment amount, interest rate, future value divided by (interest plus 1***))

Formula	**Result**
@TERM(150,.10 / 12,20000 / (1+.10 / 12))	89.51

TIME

Format: `TIME`
OS: DOS and Unix
Scope: All
Returns: Text formatted as a time

TIME returns the current system time as a time expression. The time expression is formatted in 12- or 24-hour format according to the Time Format setting in the Tools Preferences Global menu.

@TIME

Format: `@TIME(` ***numeric, numeric, numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

@TIME returns a decimal number that represents a time of day as a fraction of the whole day. The arguments are used as follows:

@TIME(***hours, minutes, seconds***)

The hours argument represents the number of hours since the beginning of the day (i.e., to indicate 6:00PM you must specify a value of 18 for the hours argument). @TIME uses only the integer part of any argument which includes a fraction.

Formula	Result
@TIME(11,42,55)	0.488136574074
@TIME(11.9,42,55)	0.488136574074
@TIME(25,16,70)	1.05358796296

TIME24

Format: `TIME24`:
OS: DOS and Unix
Scope: All
Returns: Text formatted as a time

TIME24 returns the current system time as a time expression in 24-hour format.

TIMEVALUE

Format: `TIMEVALUE(` ***time***)
OS: DOS and Unix
Scope: All
Returns: Numeric

TIMEVALUE returns the decimal number fraction (between 0 and 1) that corresponds to a particular time of day. The argument can be in either 12- or 24-hour time display format.

Formula	Result
TIMEVALUE("9:27:50")	0.394328703704
TIMEVALUE("21:16:00")	0.886111111111
TIMEVALUE("09:09:09a")	0.381354166667
TIMEVALUE("09:09:09p")	0.881354166667

TODAY

Format: TODAY
OS: DOS and Unix
Scope: All
Returns: Text formatted as a date

TODAY returns the current system date in Date2 format. Date2 format is one of three formats you can use to control the appearance of date information. The Date Format settings in the Tools Preferences Global menu define these formats. TODAY returns the equivalent of the formula DATE2(@TODAY). For an example, see @TODAY.

@TODAY

Format:@TODAY
OS: DOS and Unix
Scope: All
Returns: Numeric

This function returns a whole number representing the number of days since the beginning of the twentieth century (12/31/1899). @TODAY returns the equivalent of DAYS(TODAY). The following examples assume that the current system date is July 6, 1991.

Formula	Result
TODAY	"07-06-91"
@TODAY	33424

TRIM

Format: TRIM(***text***)
OS: DOS and Unix
Scope: All
Returns: Text

TRIM removes all leading and trailing spaces and removes extra spaces from between words in the text expression, leaving one space between each word.

Formula	Result
TRIM("un deux trois")	"un deux trois"
TRIM("J.D. ")&TRIM(" Kempton")	"J.D. Kempton"

TRUE

Format: TRUE
OS: DOS and Unix
Scope: All
Returns: Numeric

TRUE returns the value 1. TRUE can be used in a logical formula to signal that a condition does exist.

The following example assumes that the variables CUSTZIP and OURZIP contain the same values.

Formula	Result
IF CUSTZIP = OURZIP THEN TRUE ELSE FALSE	1

UNIFORM

Format: `UNIFORM(` ***numeric***)
OS: DOS and Unix
Scope: All
Returns: Numeric

UNIFORM returns a real number selected randomly from a uniform distribution between 0 and the value of the numeric expression. The following example simulates the roll of a six-sided die.

```
INT(UNIFORM(6)+1)
```

UPPER

Format: `UPPER(` ***text***)
OS: DOS and Unix
Scope: All
Returns: Text

UPPER returns the text expression with all characters converted to upper case.

Formula	Result
UPPER("upPer caSe")	"UPPER CASE"

VAL

Format: VAL(***text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

VAL converts a text expression to a numeric value. Conversion proceeds from left to right in the text expression and stops with the first character which cannot be interpreted as part of a number. If the expression is numeric or contains only non-numerals, VAL returns 0. The number may be in the form of a decimal number, such as 2300, or a number in exponential notation, such as 23E2.

To yield a negative result, the argument must begin with a minus sign. Leading and trailing space characters do not affect the result.

Formula	**Result**
VAL("100")	100
VAL("2.56E002")	256
VAL("a")	0
VAL(100)	0

NOTE: VALUE and VAL are not identical. If the argument is a numeric expression, VAL returns 0 but VALUE returns the value of the numeric expression.

VALUE

Format: VALUE(***text***)
OS: DOS and Unix
Scope: All
Returns: Numeric

VALUE, like VAL, converts a text expression to a numeric value. Conversion proceeds from left to right in the text expression and stops with the first character that cannot be interpreted as part of a number.

Formula	Result
VALUE("15 September")	15
VALUE(100)	100

VAR

Format: VAR(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

VAR calculates the statistical variance for the numeric items in the item list. All text items or blanks are ignored.

Formula	Result
VAR(4.5,6.7,27.2,32.4)	200.527

@VAR

Format: @VAR(***item list***)
OS: DOS and Unix
Scope: All
Returns: Numeric

@VAR calculates the population variance for the items in the item list. The item list must contain at least two items. Text items are treated as 0; blanks are ignored.

Formula	Result
@VAR(4.5,6.7,27.2,32.4)	150.395

VARLENGTH

Format: VARLENGTH(***variable***)
OS: DOS and Unix
Scope: All
Returns: Numeric

VARLENGTH returns the length of the buffer allocated to a variable by the Buffer project command. This function also returns the length of any text the variable holds and is faster than using the LEN() function, for this specific purpose. VARLENGTH must be used to obtain the buffer length when the buffer contains binary data. Using the LEN function would not work because it counts only the characters up to the first NULL.

VARPTR

Format: VARPTR(***variable***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Address

This function returns the address of a variable. The passed argument must be a declared variable. For example:

```
VP_ptr = VARPTR( $variable )
```

For a discussion on pointer usage, refer to the section, ***Pointers to Arrays and Variables***.

VLOOKUP

Format: VLOOKUP(***expression, block <, numeric>***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Conditional

VLOOKUP searches a lookup table and returns data from a target cell in the table. A lookup table is a block of contiguous cells containing related data items. The left column usually contains information categorizing the data in the rows of the table. VLOOKUP searches the left column for a cell containing data matching the first argument. If a match is found, VLOOKUP returns the contents of a target cell in the same row.

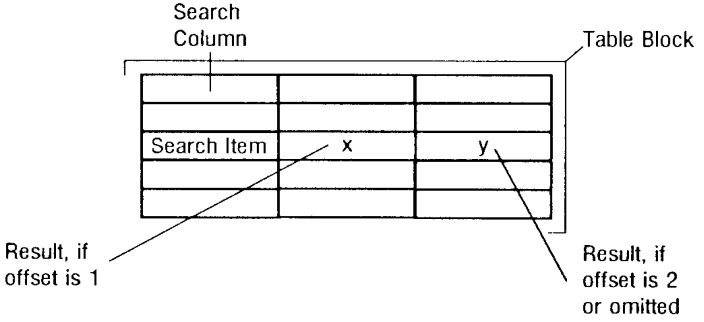
The arguments of this function are used as follows:

VLOOKUP(***search item, table block <, offset>***)

The search item can be any valid numeric or text expression. The block can be any valid block name or reference. The block defines the lookup table. VLOOKUP searches for data identical to the search item in the left column of the block. If the data is found, the contents of a target cell in this "lookup row" are returned. If the data is not found, VLOOKUP returns Error 8, LOOKUP failed.

If no offset is specified, the cell in the rightmost column of the lookup row is the target cell. If you include an offset, that number specifies the number of columns that the target cell falls to the right of the search column. The search column is considered column 0.

Figure 2-8



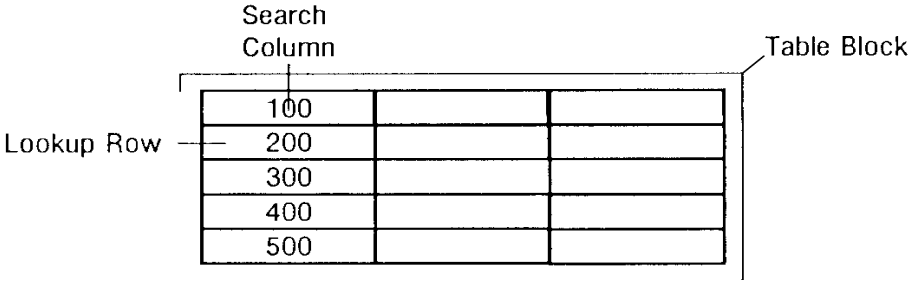
@VLOOKUP

Format: @VLOOKUP(***numeric, block <, numeric>***)
OS: DOS and Unix
Scope: Spreadsheet
Returns: Conditional

@VLOOKUP is similar to VLOOKUP, except that it searches the left column in the lookup table for the cell containing the largest value less than or equal to the first expression. This function requires that data in the search row (the left row) be in ascending numeric order (the values increase from the top to the bottom of the block).

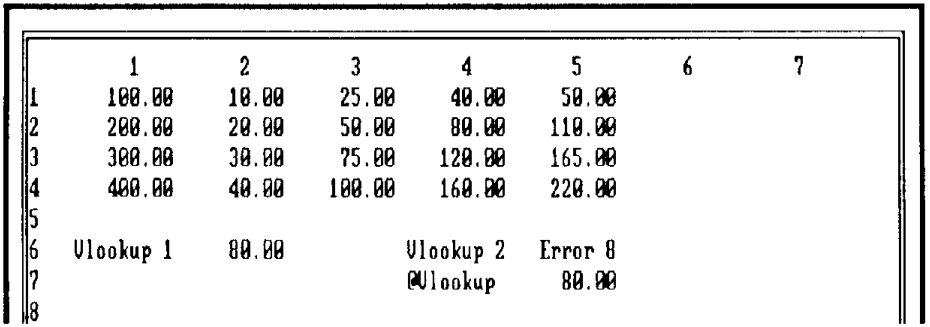
For example, assume that the search item is 250. Since 250 falls between 200 and 300, the row containing 200 in the search column is the lookup row. (See Figure 2-9 in ***Formula Reference Manual***.)

Figure 2-9



The worksheet in Figure 2-10 of ***Formula Reference Manual*** shows examples of both the VLOOKUP and @VLOOKUP commands. The formula for the first example is found in r6c2. The second formula is at r6c5 and the third formula is at r7c5.

Figure 2-10



	1	2	3	4	5	6	7
1	100.00	10.00	25.00	40.00	50.00		
2	200.00	20.00	50.00	80.00	110.00		
3	300.00	30.00	75.00	120.00	165.00		
4	400.00	40.00	100.00	160.00	220.00		
5							
6	Vlookup 1	80.00		Vlookup 2	Error 8		
7				@Vlookup	80.00		
8							

Formula	Result
VLOOKUP(200,r1:4c1:5,3)	80.00
VLOOKUP(250,r1:4c1:5,3)	Error 8
@VLOOKUP(250,r1:4c1:5,3)	80.00

WPGET

Format: `WPGET(` ***numeric, numeric***)
OS: DOS and Unix
Scope: Word Processor
Returns: Text

WPGET returns up to 240 characters of text from a document. The first argument specifies the number of characters to be returned. The count begins at the current cursor position. Reform spaces, which are used only for alignment purposes, are not included in the count. A -1 value for the number of characters will read to the end of the paragraph. If the second argument is 0, only the text is returned. If the second argument is other than 0, font and special character information are included in the returned text. Font and special character information are not included in the character count.

Fonts are indicated by %[x], where x is the font number. Boldface and underscore are indicated by B and U, respectively. The returned text may also include the following special characters.

Spec. Character	Meaning
%t	tab
%i	indent tab
%p	paragraph mark
%e	end of document
%-	soft hyphen
%%	percent sign, as text

See also WPREAD.

WPINFO

Format: `WPINFO(` ***numeric*** `)`
OS: DOS and Unix
Scope: Word Processor
Returns: Conditional

WPINFO returns a variety of information about the current document, based upon the named constant specified in the argument. Valid argument entries and their uses are as follows:

Named Constant	Returns
wp_cursect	Current section number
wp_curpage	Current page number
wp_curline	Current line number
wp_curcol	Current screen column position
wp_filename	Filename (name.ext only)
wp_filespec	File specification
wp_curfont	Font of the current character
wp_newfont	Font for new text

Formula	Result
wp_mcacol	Current MCA column number (or 0 if not in MCA)
wp_mcaent	Current MCA entry (or 0 if not in MCA)

WPKEY

Formats: WPKEY(***numeric***)

OS: DOS and Unix

Scope: Word Processor

Returns: Numeric

WPKEY is used by project processing programs to control cursor movement. It is a good alternative to SUSPEND/KEYS statements since unnecessary screen flashing does not occur and performance is improved.

This function passes a given keystroke to the underlying module. If a cursor movement keystroke was processed, 1 is returned, otherwise, the keystroke was not a cursor movement key and 0 is returned.

WPKEY handles all movement keys and mouse activity.

Example 1:

```
WPKEY({down})  'Move the cursor down one line
```

Example 2:

```
'Control navigation by cursor keys, mouse clicks and scroll bar
WHILE #key<>{esc}
    #key = INCHAR
    IF WPKEY(#key)    'Key was cursor movement
        '(process command)
    ELSE              'Key was NOT cursor movement
        '(process command)
    END IF
END WHILE
```

WPPUT

Format: `WPPUT(` ***text***)
OS: DOS and Unix
Scope: Word Processor
Returns: Numeric

WPPUT enters text with optional special character and font codes into a document at the cursor position. WPPUT returns 1 if the operation is successful, or 0 if the operation is not successful. WPPUT leaves the cursor on the first character of the "put" data.

Text is entered using the specified font, or if no font is specified, the currently selected font. You can change the font within the text by inserting a code in the form`%[fontnumber]`. To specify boldface and/or underscore, include "B" and/or "U" after the font number. If the font specified does not exist, Font 0 is used. Fonts changes effected by WPPUT have no effect upon existing text in the document.

You can also include the following special characters in the text expression.

Special Character	Meaning
%t	tab
%i	indent tab
%p	paragraph mark
%-	soft hyphen
%%	percent sign, as text

Examples:

WPPUT("This is text.")

WPPUT("%[4U]This%[4]__is text.")

The first example inserts "This is text." at the cursor position, using the current font. The second example inserts "This is text." in font 4, with the word "This" underscored.

WPREAD

Format: WPREAD(***numeric, numeric***)
OS: DOS and Unix
Scope: Word Processor
Returns: Text

WPREAD works like the WPGET function except that it moves the cursor to the first character that has not yet been read. This makes it easy to read through a document in project processing programs (similar to FREAD). The arguments are as follows:

WPREAD(characters, fonts)

The first parameter indicates the number of characters to read. If the second parameter is 0, only text is returned, otherwise, font and special characters will be included.
See also WPGET

WRITEPTR

Format: WRITEPTR(***pointer***, <,***index***> <,***index***> <,***index***>, ***expression***)
OS: DOS and Unix
Scope: All
Available: Version 2.65 and Higher
Returns: Boolean

Using the expression argument, WRITEPTR changes the value in the variable or array element specified by the pointer and index arguments.

```
WRITEPTR( VP_variable, $newvalue )
```

```
WRITEPTR( AP_variable, 10, 2, "newvalue" )
```

WRITEPTR returns 1 if the operation is successful and 0 for failure.

For a discussion on pointer usage, refer to the section, ***Pointers to Arrays and Variables***.

YEAR

Format: YEAR(***date***)
OS: DOS and Unix
Scope: All
Returns: Numeric

YEAR returns a value corresponding to the number of the year in the date expression.

Formula	**Result**
YEAR("July 6, 1952")	1952
YEAR("09/15/2099")	2099

@YEAR

Format: @YEAR(***date***)
OS: DOS and Unix
Scope: All
Returns: Numeric

@YEAR returns a value represented by the last two digits of the year in the date expression.

Formula	**Result**
@YEAR("July 6, 1952")	52

Chapter 3: Statistical Database Functions

Statistical Database (SDb) functions for the Spreadsheet (designed for use with worksheet data organized in "database" format) and for the Database are included in this chapter. Application examples and additional information on using these functions are also provided.

Spreadsheet SDb Functions

The following list summarizes ANGOSS Spreadsheet SDb functions.

Function	Returns
@DAVERAGE	Average value; ignores blank and text cells
@DAVG	Average Value; ignores blanks, treats text as 0
DCOUNT	Number of value entries; ignores blanks but counts text cells
@DCOUNT	Number of value entries; ignores blanks but counts text cells
@DMAX	Maximum value
@DMIN	Minimum value
@DSTD	Standard deviation of a population
@DSTDEV	Standard deviation of a sample
@DSUM	Sum of values
@DSUMSQ	Sum of the squares of values
DVAR	Sample variance
@DVAR	Population variance

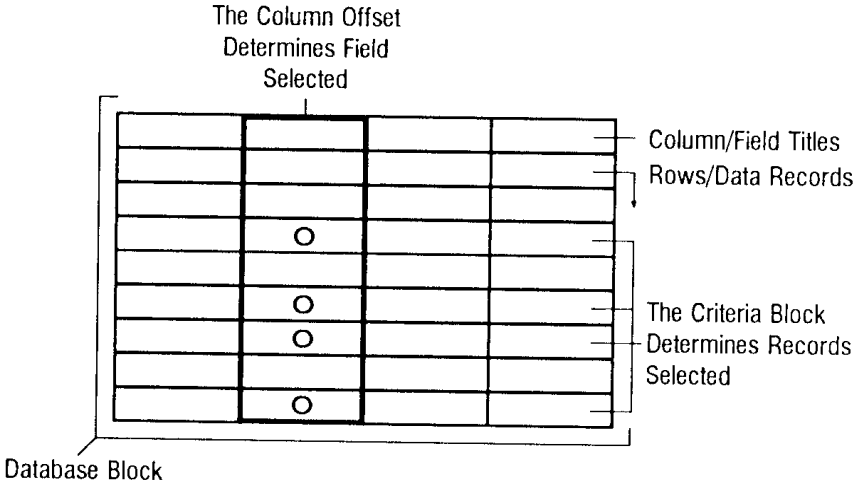
The Spreadsheet SDb functions complement other statistical functions, such as AVERAGE and VAR, but are designed for use with worksheet data organized in a database format.

Spreadsheet SDb Arguments

All Spreadsheet SDb functions employ arguments in the same way, as follows:

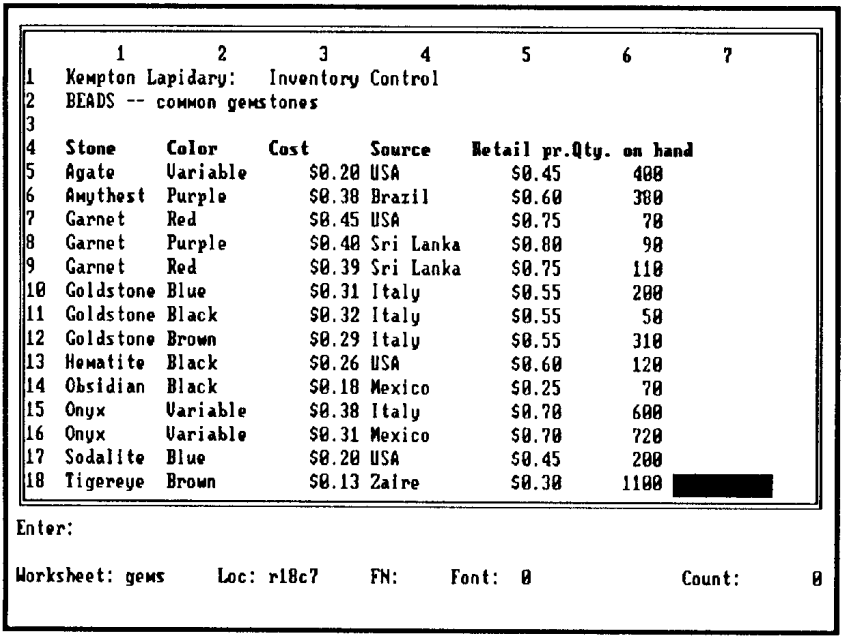
@DAVERAGE(***Database block, column offset, criteria block***)

Figure 3.1.



Database Block. The database block argument identifies the area of the worksheet containing the source data, i.e., the "database." Figure 3.2 shows a simple worksheet database in r4:18c1:6. Each row represents a record; each column is a field. The first row contains field / column titles.

Figure 3.2



	1	2	3	4	5	6	7
1	Kempton Lapidary:		Inventory Control				
2	BEADS -- common gemstones						
3							
4	Stone	Color	Cost	Source	Retail pr.	Qty. on hand	
5	Agate	Variable	$0.20	USA	$0.45	400	
6	Amythest	Purple	$0.30	Brazil	$0.60	300	
7	Garnet	Red	$0.45	USA	$0.75	70	
8	Garnet	Purple	$0.40	Sri Lanka	$0.80	90	
9	Garnet	Red	$0.39	Sri Lanka	$0.75	110	
10	Goldstone	Blue	$0.31	Italy	$0.55	200	
11	Goldstone	Black	$0.32	Italy	$0.55	50	
12	Goldstone	Brown	$0.29	Italy	$0.55	310	
13	Hematite	Black	$0.26	USA	$0.60	120	
14	Obsidian	Black	$0.10	Mexico	$0.25	70	
15	Onyx	Variable	$0.30	Italy	$0.70	600	
16	Onyx	Variable	$0.31	Mexico	$0.70	720	
17	Sodalite	Blue	$0.20	USA	$0.45	200	
18	Tigereye	Brown	$0.13	Zaire	$0.30	1100	

Enter:

Worksheet: gems Loc: r18c7 FN: Font: 0 Count: 0

Column Offset. The column offset argument specifies the column within the database block upon which the statistical calculation is to be performed. The leftmost column is column 0, the next is column 1, etc. In the “Gems” worksheet in Figure 3.2 the offset of the Cost column is 2.

Criteria Block. The criteria block argument identifies an area of the worksheet containing criteria for selecting the records to be included in the statistical calculation. The first row of the criteria block must contain the exact headings of the database columns to be selected. Remaining rows contain specific selection requirements. If two criteria are in the same row, a record must meet both criteria to be selected. If the criteria are in separate rows, a record may fit either one to be selected. Blank cells in the criteria block are treated as if they match all entries. Thus, a criteria block containing headings but no selection criteria returns information from all records.

NOTE: Cell references that refer to database fields must be relative. However, cell references that refer to values outside the database must be absolute. See the ***Spreadsheet Manual*** for information on absolute and relative cell references.

If you enter more than one criterion in the same row of the block, all must match or return TRUE in order for a record to be selected. If you enter criteria in more than one row, each row specifies an independent set of criteria that can cause a record to be selected. In terms of logical operators, criteria in the same row are jointed by AND; criteria in different rows are joined by OR.

Blank cells in the criteria block are treated as if they match all entries. Thus, a criteria block that contains one or more of the field/column headings of a database, but no selection criteria, returns information from all records. Alternatively, you can omit fields for which no criteria are needed.

The following wild card characters can be used to match multiple items in the same criteria.

- **?** matches any single character.
- ***** matches all remaining characters from the point the asterisk is inserted.
- **~** (tilde) at the beginning of the criterion instructs ANGOSS to accept any record entry **except** the one preceded by the tilde character.

If the criterion is a formula, it must be written as a test of the first record in the database. If the formula returns TRUE, the record is selected. If the formula returns FALSE, the record is not included. ANGOSS sequentially replaces the cell reference in the formula with the appropriate cell reference from each record.

If the criterion is a number, the number in the corresponding database block record must match exactly for the record to be selected.

Database File SDb Functions

The following list summarizes ANGOSS File SDb functions.

Function	Purpose
FILEAVERAGE	Average value
FILECOUNT	Number of items (text or value)
FILEMAX	Maximum value
FILEMIN	Minimum value

Function	Purpose
FILESTD	Standard deviation of a population
FILESTDEV	Standard deviation of a sample
FILESUM	Sum of values
FILESUMSQ	Sum of the squares of values
FILEVAR	Sample variance

The Database File SDb functions complement other statistical functions, such as AVERAGE and VAR, but are designed for performing calculations on a specific field for every (logical) record in a data-file. The function FILEAVERAGE, for example, returns the average value in a field from among all the records in a data-file. All File SDb functions use arguments in the same way, as follows:

FILEAVERAGE(***return field <, optional selection criterion>***)

The return field identifies the field containing the data to be used in the calculation. The optional selection criterion is a logical expression specifying conditions under which a record should be included in the calculation. If no selection expression is included, the result is returned from all records.

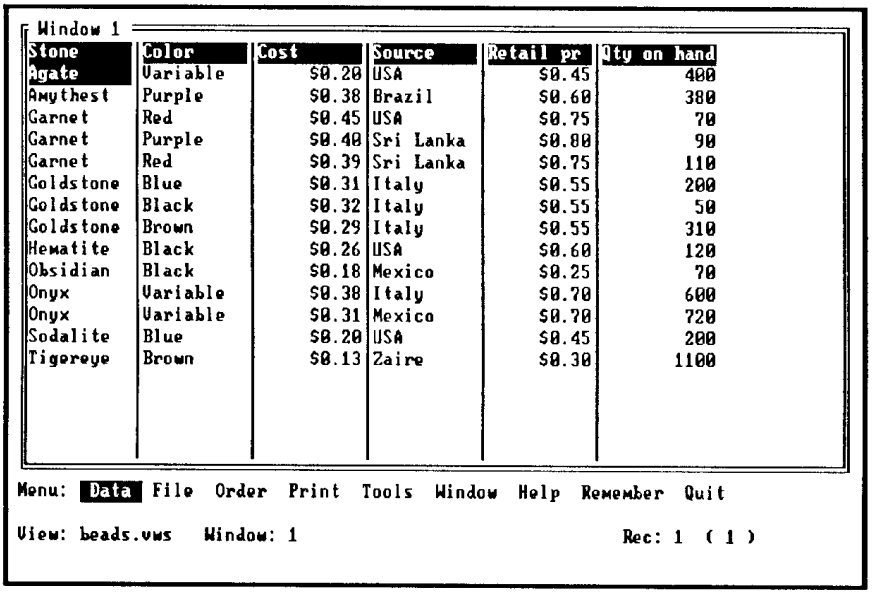
NOTE: When calculating a formula containing a File SDb function, ANGOSS accesses every record in a file. The following paragraphs describe some restrictions that apply to the use of these functions.

File SDb functions cannot be calculated when you are entering or updating a record. If you are using these functions with very large files, the calculation may take a significant amount of time. File SDb functions, as well as FILELOOKUP, cannot be “nested” (made directly dependent on results returned from other File SDb functions) in formulas.

If the return field is within a table, the File SDb functions process all table records associated with all view records in the file, not just the table records in the current view record. In contrast, Table SDb functions operate only on the table records in the current view record.

You can use FILECOUNT, FILEMAX, and FILEMIN to perform calculations on text type fields (i.e., alpha, inverted). FILECOUNT counts text as well as numeric data. FILEMAX and FILEMIN compare text items according to the ANGOSS Character Set values of their characters. Figure 3-3 below is an example from a simple data-file in a standard view.

Figure 3.3



The following examples refer to the file displayed in Figure 3.3.

Formula	Result
FILESUM ([Qty on hand])	4420
FILESUM ([Qty on hand] , [Color]=”Black” OR [Color]=“Brown”	1650
FILECOUNT ([Stone], [Source]=“Italy” AND [Retail pr]<.60	3

Database Table SDb Functions

Function	Purpose
TABLEAVERAGE	Average value
TABLECOUNT	Number of items (text or value)
TABLEMAX	Maximum value
TABLEMIN	Minimum value
TABLESTD	Standard deviation of a population
TABLESTDEV	Standard deviation of a sample
TABLESUM	Sum of values
TABLESUMSQ	Sum of the squares of values
TABLEVAR	Sample variance

The Database Table SDb functions are designed for performing calculations on data displayed in tables (repeating areas) in Database custom views. The function TABLEAVERAGE, for example, returns the average value in a field from among the records displayed in a table in a single view record. All Table SDb functions use arguments in the same way, as follows:

TABLEAVERAGE(***return field <, optional selection criterion>***)

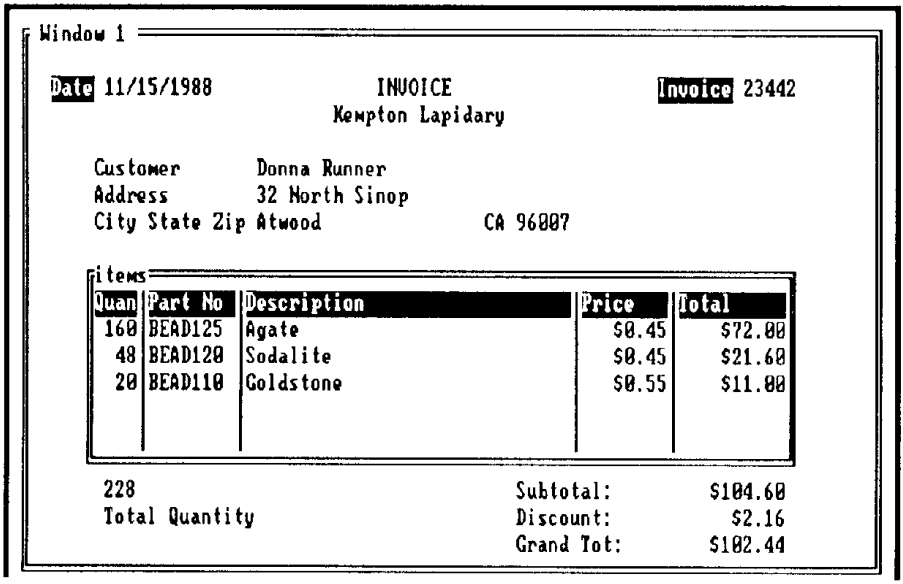
The return field identifies the field containing the data to be used in the calculation. The field must be within a table in the view in which it is referenced. The optional selection criterion is a logical expression specifying conditions under which a table record should be included in the calculation. If no selection expression is included, the result is returned from all table records.

NOTE: When creating views, do not include a field in a table if it is a calculated field using a Table SDb function in its formula. Under such circumstances, the calculation will fail. It is also important to know that Table SDb functions and TABLELOOKUP cannot be nested.

You can use TABLECOUNT, TABLEMAX, and TABLEMIN to perform calculations on text type fields (i.e., alpha, inverted). TABLECOUNT counts text as well as numeric data. TABLEMAX and TABLEMIN compare text items according to the ANGOSS Character Set values of their characters.

Figure 3-4 shows an example of a common application for tables and Table SDb functions.

Figure 3.4



In this application, Table SDb functions help to provide the figures labelled in Total Quantity, Subtotal, and Discount. The filed calculations are defined with the following formulas:

Field	**Formula**
[Total Quantity]	TABLESUM ([Quantity])
[Subtotal]	TABLESUM ([Total])
[Discount]	TABLESUM ([Total] , [Quantity] >= 144) * .03

Appendix A: Formula Error Messages

Introduction to Formula Errors

When ANGOSS detects an error that prevents it from calculating a formula, an error message or error code number is displayed. If the error is detected while you are entering or calculating a formula in an ANGOSS editor or calculator, you are informed by the display of a brief message and given the opportunity to edit the formula. If the error is detected during the calculation of a worksheet or view, the error's code number is displayed in the affected cell or field. When an expression that returns an error is assigned to a variable, the variable assumes the error status and that same error message occurs whenever the variable is subsequently used, unless something else is assigned to it.

This Appendix lists the most common error messages related to formula use, arranged according to their code numbers. A basic description of conditions that may cause most errors is included, along with suggestions for recovery or correction. This information is provided to assist you in using formulas and is not intended to be a comprehensive reference for all error conditions associated with formulas.

0 No Error

The formula contains an error function, i.e., ERRORTEXT(), whose argument is zero. If you are using LERROR or CERROR as the argument, make sure that the correct data is being passed to your error function. LERROR returns the value of the last error that occurred, even when later commands were executed without error. CERROR returns the value of any error occurring during the previous command, or zero if no error occurred.

1 Missing '('

The formula lacks a left parenthesis. For each left parenthesis in the formula you must supply a corresponding right parenthesis, and vice versa. Check your formula to make sure that the number of left parentheses matches the number of right parentheses.

2 Missing ')'

The formula lacks a right parenthesis. For each left parenthesis in the formula you must supply a corresponding right parenthesis, and vice versa. Check your formula to make sure that the number of right parentheses matches the number of left parentheses.

3 Incorrect number of arguments

A function has not received the correct number of arguments. Consult the ***Formula Reference Manual*** to determine the proper number of arguments for the functions you are using.

4 Bad syntax

The formula is improperly written and cannot be calculated. The formula may have an improper operator for the type of expression used, an unrecognizable function or variable name, a function whose scope does not include the current module, or invalid characters. Make certain that function arguments are placed in parentheses, if necessary.

5 Missing 'THEN'

The formula lacks a portion of an IF-THEN-ELSE statement. Every IF-THEN-ELSE formula requires a THEN clause to indicate what to do when the conditions tested for in the IF clause are met. You must supply the word "THEN" to identify the THEN clause. Consult the alphabetical listing of functions for help in writing logical formulas using this function.

6 Missing 'ELSE'

The formula lacks a portion of an IF-THEN-ELSE statement. Every IF-THEN-ELSE formula requires an ELSE clause to determine what happens when the IF expression is false. You must supply the word "ELSE" to identify the ELSE clause. Consult the alphabetical listing of functions for help in writing logical formulas using this function.

7 ERROR

The formula contains a reference to another formula which returns an error. Correcting the error in the referenced formula will eliminate this one.

8 LOOKUP failed

The search item for an HLOOKUP, VLOOKUP, FILELOOKUP, or TABLELOOKUP formula is not found in the search block or field(s). These functions search for data matching the search item argument(s). Make sure you have specified the correct search block or field list and search item or items. If you are searching for worksheet data that falls within a particular numeric range, use @HLOOKUP/@VLOOKUP.

9 Serious error

Calculation of a File SDb function has been interrupted because one or more data-file records cannot be accessed. This may indicate that hardware failure or damage to the file has occurred. Examine the file to evaluate its condition. It may be necessary to restore from backup.

10 Insufficient data

Not enough information is provided in the item list to perform the calculation or some of the information is unavailable due to other errors. @VAR, @COUNT, and STD require at least values; SUM, MIN, MAX, AVERAGE, STDEV, SUMSQ, VAR, and COUNT require at least 1 value in the item list.

11 Division by zero

Calculation of the formula entails dividing a number by zero. Mathematically, division by zero is undefined. ANGOSS allows you the alternatives of returning either zero or Error 11 when this condition occurs. The `Division by zero is:` setting in the Tools Preferences Global menu controls this feature.

12 Bad cell reference

The formula contains a term interpreted as a cell or block reference that cannot be resolved. The formula may reference a named cell or block that is no longer defined. The problem may lie with an external worksheet reference that consists of a valid worksheet name but an invalid or missing cell or block reference. Verify the correctness of the target of any assignment statements and/or the argument of any CELLTEXT functions in your formula.

13 Function or assignment not valid in this context

A formula or statement uses a function in a manner for which it is not designed. For example, use of BLOCKMARK is restricted to project commands and statements

where it permits interactive selection of a block during project execution. BLOCKMARK cannot be used directly in a formula cell or a user-defined function called by a worksheet or calculator formula. Likewise, assignment functions, such as LET and SSPUT, cannot be used directly in a cell.

14 CASE failed

No match is found for the search item in a formula using the CASE function and no ELSE statement is included in the CASE formula. Make sure that you have specified the correct item to be matched, included all appropriate items to be compared in the case list, and included an ELSE statement if there is a possibility that no match will be found.

15 SELECT failed

None of the logical expressions in the select list is true and no ELSE statement is included in the SELECT formula. Make sure that you have specified the correct terms in your select list, used the correct relational operators in the logical expressions, and included an ELSE statement if there is a possibility that none of the conditions is true.

16 Function not available in current module

The formula contains a function whose scope does not include the ANGOSS module in which you are currently working. Make sure that you have entered the function name correctly.

17 Expression too complex

Calculation of the formula requires manipulation of expressions or terms that are too lengthy or too complex. Revise the formula to simplify the operations or calculations involved. Use data references where appropriate. Try performing the calculation in two or more steps.

18 Bad argument

The formula contains an argument that is of the wrong expression type or is out of the range required for the function with which it is used. Make certain that all functions receive arguments of the appropriate type and value.

19 Missing worksheet

The formula contains a term interpreted as a data reference to a worksheet that is not active. You must load or activate the referenced worksheet before recalculating this formula.

20 Missing name

The formula contains a term interpreted as a reference to a named cell or block in an external worksheet. No such name is defined for that worksheet or the worksheet is unavailable. Load the referenced worksheet and use Sheet Name Edit to check the defined names. Then edit the formula to include the correct name.

21 First argument must be date

When you use two arguments with this time function, the first argument must be a date expression.

22 Bad date

A date expression is either improperly formatted or results in a date that is impossible (e.g., "02/31/87") or out of the range of dates supported by ANGOSS.

23 Bad time

A time expression is improperly formatted, results in a time that is impossible, or is out of the range of times supported by ANGOSS (e.g., "25:00:00").

24 CHOOSE failed

The value of the first argument of the CHOOSE function is less than 0 or greater than the number of items in the item list minus 1. Make sure that you have specified the correct terms in your formula and that the item list includes all the appropriate items.

25 Missing ',' or ')'

Within a function, either a comma is missing, as in the example ATAN2(3 15), or a right parenthesis is missing, as in ATAN2(3,15.

26 Date expression expected

A formula expects a date expression but another type of expression has been encountered. Make certain that all date expressions are properly formatted and that all functions and operators are used with expressions of the appropriate type.

27 Inconsistent number of arguments

The formula employs a user-defined function that has not received the correct number of arguments. Make sure that you are using the correct function name and the number of arguments it requires.

28 Out of string space

The formula requires manipulation of a text expression that is too long. Revise the formula or the text expressions involved in the calculation.

29 Expected block

The formula contains a function that requires one argument to be a block reference. Correct the formula so that, where necessary, the function receives a reference to a block of cells or fields.

30 No convergence

The data contained in the arguments for a recursive function (e.g., IRR, GOAL, INTEREST) do not converge to yield a specific result. For IRR, at least one number in the block must be negative. For GOAL, check and revise your initial estimate to make it more closely approximate the final result.

31 Invalid field

The formula contains an item interpreted as a field reference (i.e., within brackets) that is not a valid field title or number in the current view. Make sure that you are working with the correct view and check all field references in the formula.

32 Cell out of range

The offset argument for an HLOOKUP, VLOOKUP or INDEX function is negative or refers to a cell that is outside of the defined block.

33 Can't execute project file function. Overlay conflict.

Using one of the text editors, you have attempted to recalculate using a function from a loaded function library. There is an overlay conflict between the function library and the text editor you are using.

34 Variable not found

The formula contains an item interpreted as a public variable that has not been declared. Check the formula for errors or ambiguous expressions. Public variables must be declared with a Public statement.

35 User error

You have calculated a formula using the ERROR function. This function allows you to generate a formula error based on conditions you specify.

36 Invalid variable assignment

The formula contains a LET statement that attempts to assign a literal data reference (i.e., a cell, field, or variable address) to a target rather than the data denoted by the data reference. Verify the format of your formula and make sure that all functions receive the correct expression types as arguments.

37 Not text variable or character offset outside text

The formula contains an item interpreted as a text variable subscript that cannot be resolved. Character subscripts can only be used with text variables. The subscript must be numeric and must not be greater than the length of the text expression referenced by the variable nor less than one.

38 Bad index value to array

The formula contains an item interpreted as an array reference that cannot be resolved. The array must contain the correct number of subscripts, corresponding to the number of array dimensions. The subscripts must be numeric and must not have a value larger than the number of array elements nor less than one.

39 Invalid LET or command target

The formula contains an expression interpreted as an assignment statement that attempts to place data in an invalid LET or command target. You can use LET to

assign data to variables, fields, and cells (e.g., LET variable1 = 2). You cannot assign data to a formula, constant, or data reference (e.g., LET 2 = #variable1). You cannot use LET or SSPUT to assign data to a formula cell during worksheet recalculation.

40 Missing equal sign in LET

Your formula contains a LET statement that lacks the equal (=) sign. The correct format is `LET target =` ***expression***.

42 Incomplete formula

The formula is improperly written and cannot be calculated. Check the syntax to make sure that all expressions and data references have the correct format and are complete.

43 Calc stack bad

The source of a calculation has somehow been corrupted. Restore the file from backup.

44 Calc token unknown

Some damage has occurred to the formulas in your data file or project file. Save the current file under a new name and exit the module, when possible. It may be necessary to recover your files from backup.

45 Data is not available

The formula describes a calculation that cannot currently be performed. A variety of conditions may produce this error, including those mentioned in the following paragraph. If you receive this error, examine your formula carefully to determine whether or not it describes a meaningful or realistic calculation.

During view definition, you cannot calculate a formula containing a field reference to the view being defined. In Enter/Update mode, you cannot calculate a formula involving a file SDb function. Formulas using FETCHFIELD return this error when no record has previously been accessed since the file was loaded.

48 No expression found

The formula contains an item interpreted as an expression or a data reference that cannot be located. Check the syntax to make sure that all expressions and data references have the correct format and are complete.

49 No closing array bracket

The formula contains an array reference but lacks the closing (]) character for the array subscript. The correct format for a reference to an array element is A[S], where A is an array name and S is the subscript number or numbers.

50 Missing or bad array index

The formula contains an item interpreted as an array reference that lacks or has an invalid subscript. The array must specify the correct number of subscripts, corresponding to the number of array dimensions. The subscripts must be numeric and must not have a value larger than the number of array elements nor less than one.

51 Hex constant too large

The formula contains a hexadecimal value greater than the maximum supported by ANGOSS: FFFF FFFF (32 bits).

52 Project file function not loaded

The formula or statement calls for a user-defined function that is not available. You must load the project file that defines the function before attempting to calculate the formula or execute the statement. The project file that defines the function must contain a Public statement allowing the function to be called externally.

53 Undefined variable or function

The formula or statement contains an unrecognized term. Check the syntax for errors and ambiguous expressions. All variables and arrays used in projects must be declared by means of a Local, Global, Public, or external statement. All functions must be declared by using a Global, Public, or external statement.

54 Array already exists with variable name

ANGOSS' Project Development Language requires that all variables, including arrays, have unique names. Project files currently in use contain variables and/or arrays identified by the same name. Edit your project files to change the name(s) as necessary.

55 Too many functions deep - danger of stack overflow

A command or statement in a user-defined function calls for another user-defined function that calls for yet another. The number of such "nested" or recursive function calls is approaching the maximum (50). Edit your projects to reduce the number of successive function calls. Main counts as the first "level."

56 Unknown compiler token - try recompiling

ANGOSS is unable to continue executing the project or user-defined function, possibly due to corruption of the file or a memory fault. Try recompiling the project file(s) or restore from backup.

57 Error in FOR loop

The expression used to specify the initial counter value, loop length, or step increment in a For statement is invalid. These entries must be numeric.

58 Insufficient memory

Execution of the project or calculation requires more memory than is currently available in the system.

59 Error reading project file

ANGOSS is unable to continue executing the project or calculation, possibly due to corruption of the file or a disk error. It may be necessary to restore from backup.

60 Invalid file number

An expression used to specify the file number in a project data file command (e.g., FOPEN, FREAD . . .) falls outside the allowable range, 1 through 20.

63 Bad search field

The search field specified in a FILELOOKUP formula references a field that cannot be identified or is not a data-file field. All views used in the lookup must be active. Make sure that you have entered all field references correctly, including the view name where needed.

64 Search field is not a key field

The search field specified in a FILELOOKUP formula references a field that is not a key field. Make sure that you have entered all field references correctly and that the search field is a key field.

65 Bad return field

The return field specified in a FILELOOKUP formula references a field that cannot be identified or is not a data-file field. All views used in the lookup must be active. Make sure that you have entered all field references correctly, including the view name where needed.

66 Search and return fields are not from the same view

The search and return fields specified in a FILELOOKUP formula must be located on the same view. Make sure that you have entered all field references correctly, including the view name where needed.

67 Bad search data

One or more expressions used to specify the search data in a FILELOOKUP formula return data invalid for a search. Make sure that each search argument is a valid expression and does not return an error, NA, or BLANK.

68 Cannot execute module commands recursively

A formula or project calls a user-defined function that contains a module command. Unlike project commands, module commands cannot be used in functions that are executed by formulas or other module commands.

Appendix B: Key Codes

The following table lists the key terms for keys and key combinations that are recognized by ANGOSS. These terms are the correct form of the arguments for the function KEYVALUE. They are also used in macro definition and in Keys project commands as character terms and activity control terms (refer to Keys in the chapter entitled Project Command Reference in Project Processing.

Terms for other keys and shifted keys that are used to type most letters, numbers, punctuation marks, and symbols correspond exactly to their characters. For example, the correct term for **Shift = or +**.

Terms for other keys and key combinations typically used for various program-related activities, such as cursor movement and command selection, usually correspond to the keytop symbol for the key. In key combinations, Alt- represents **Alt**, s represents **Shift**, and ^ represents **Ctrl**. For example, the term for **Ctrl X** is ^X.

This table incorporates many character keys as ranges of characters. These ranges derive from an alphabetical or numeric ordering of the characters, rather than the physical positions of the keys on a keyboard. For example, **A** to **Z** refers to the set of alphabetical character keys. An exception is **!** to **>**, which refers to the set of alphabetical number keys on the top row of the keyboard.

Key	Term	Key	Term
A to Z	a to z	**Shift A to Shift Z**	A to Z
0 to 9	0 to 9	**! to >**	! to)
‘ and ~	‘ and ~	**/ and ?**	/ and ?
-	-	**-**	-
=	=	**+**	+
****	\	**|**	|

Key	Term	Key	Term
[and]	[and]	**{ and }**	{ and }
; and :	; and :	**‘ and “**	‘ and “
, and <	, and <	**. and >**	. and >
Spacebar	Space	**Backspace**	Bs
Tab	Tab	**Shift Tab**	sTab
Home	Home	**End**	End
Pgup	PgUp	**PgDn**	PgDn
Up Arrow	Up	**Down Arrow**	Down
Left Arrow	Left	**Right Arrow**	Right
Ins	Ins	**Del**	Del
Enter	Cr or Enter	**Esc**	Esc
Alt A to Alt Z	Alt-A to Alt Z	**Alt 1 to Alt 0**	Alt 1 to Alt 0
Alt -	Alt-	**Alt =**	Alt=
Shift F1 to F12	F1 to F12	**Shift F1 to Shift F12**	sF1 to sF12
Alt F1 to Alt F12	Alt-F1 to Alt F12	**Ctrl F1 to Ctrl F12**	^F1 to ^F12
Ctrl A to Ctrl G	^A to ^G	**Ctrl K**	^K
Ctrl L	^L	**Ctrl N to Ctrl Z**	^N to ^Z
Ctrl 2	^2	**Ctrl 6**	^6
Ctrl -	^-	**Ctrl Backspace**	^Bs
**Ctrl **	^\	**Ctrl]**	^]
Ctrl Home	^Home	**Ctrl End**	^End

Key	Term	Key	Term
Ctrl PgUP	^PgUp	**Ctrl PgDn**	^PgDn
Ctrl Left Arrow	^Left	**Ctrl Right Arrow**	^Right
Ctrl Enter	^Cr or ^Enter	**Mouse Click**	mouse

Appendix C: ANGOSS Functions for 1-2-3 Users

ANGOSS features an equivalent or similar function for all functions available in Lotus 1-2-3 (Release 2). A table of 1-2-3 functions and their ANGOSS counterparts is provided below. A brief explanation of some of the differences follows the list.

The @ Symbol

Unlike 1-2-3 ANGOSS does not require an "@" prefix with most function names. Many ANGOSS functions have names matching their equivalent 1-2-3 functions, except that they do not begin with the @ sign. The @ is optional in these cases; the function returns the same value whether or not you include the symbol. When you enter a formula or read a 1-2-3 worksheet, ANGOSS strips the @ from these functions.

NOTE: If you are accustomed to entering functions preceded by @, you can continue this practice in ANGOSS. Including the @ also ensures that your formulas will calculate as much as possible like you would expect them to in 1-2-3.

For like-named functions that differ between ANGOSS and 1-2-3, the @ preceding the 1-2-3 form distinguishes the two functions. The @ is retained by ANGOSS when you enter or read one of these functions.

There is also a small number of ANGOSS functions (i.e., Spreadsheet SDb functions) that begin with the @ sign but have no exact counter part in 1-2-3.

Comparison of Functions

If you are familiar with 1-2-3, you can use the following table to identify ANGOSS functions that are equivalent or comparable to Lotus 1-2-3 functions. Functions listed in the "ANGOSS Equivalent" column accept arguments and perform calculations like the corresponding 1-2-3 functions. Functions listed in the "Comparable ANGOSS Function" column perform similarly but calculate differently or employ a different set of arguments. Consult Chapter 2 for further information about these functions.

1-2-3 Function	ANGOSS Equivalent	Comparable ANGOSS Function
@ABS	ABS	
@ACOS	ACOS	
@ASIN	ASIN	
@ATAN	ATAN	
@ATAN2	ATAN2	
@AVG	AVG	AVERAGE
@CELL		CELL
@CELLPOINTER		CELLPOINTER
@CHAR		CHR
@CHOOSE	CHOOSE	
@CODE		ASC
@COLS	COLS	
@COUNT	@COUNT	COUNT
@CTERM	CTERM	
@DATEVALUE		DATEVALUE
@DATE	DATE	
@DAVG	@DAVG	@DAVERAGE
@DAY	DAY	
@DCOUNT	@DCOUNT	DCOUNT
@DDB	DDB	

1-2-3 Function	ANGOSS Equivalent	Comparable ANGOSS Function
@DMAX		@DMAX
@DMIN		@DMIN
@DSTD	@DSTD	@DSTDEV
@SUM	@DSUM	
@DVAR	@DVAR	DVAR
@ERR	ERR OR ERROR	
@EXACT	EXACT	
@EXP	EXP	
@FALSE	FALSE	
@FIND	FIND	
@FV	@FV	FV
@HLOOKUP	@HLOOKUP	HLOOKUP
@HOUR	HOUR	
@IF	@IF	IF THEN ELSE
@INDEX	@INDEX	INDEX
@INT	@INT	INT
@IRR	IRR	
@ISERR	ISERR	
@ISNA	ISNA	
@ISNUMBER	ISNUMBER	
@ISSTRING	ISSTRING	

1-2-3 Function	ANGOSS Equivalent	Comparable ANGOSS Function
@LEFT	LEFT	
@LENGTH	LEN	
@LN	LN	
@LOG	LOG10	
@LOWER	LOWER	
@MAX		MAX
@MID	@MID	MID
@MIN		MIN
@MINUTTE	MINUTE	
@MOD	MOD	
@MONTH	MONTH	
@N	N	
@NA	NA	
@NOW	NOW	
@NPV	@NPV	NPV
@PI	PI	
@PMT	@PMT	PMT
@PROPER	PROPER	
@PV	@PV	PVA
@RAND	RAND	
@RATE	RATE	

1-2-3 Function	ANGOSS Equivalent	Comparable ANGOSS Function
@REPEAT	REPEAT	
@REPLACE	REPLACE	
@RIGHT	RIGHT	
@ROUND	ROUND	
@ROWS	ROWS	
@S	S	
@SECOND	SECOND	
@SIN	SIN	
@SLN	SLN	
@SQRT	SQRT	
@STD	STD	STDEV
@STRING		STR
@SUM	SUM	
@SYD	SYD	
@TAN	TAN	
@TERM	@TERM	TERM
@TIME	@TIME	TIME
@TIMEVALUE	TIMEVALUE	
@TODAY	@TODAY	
@TRIM	TRIM	
@TRUE	TRUE	

1-2-3 Function	ANGOSS Equivalent	Comparable ANGOSS Function
@UPPER	UPPER	
@VALUE	VALUE	
@VAR	@VAR	VAR
@VLOOKUP	@VLOOKUP	VLOOKUP
@YEAR	@YEAR	YEAR

Function Differences

AVG and AVERAGE

AVG is the equivalent of SUM divided by @COUNT. AVERAGE is the equivalent of SUM divided by COUNT. @COUNT counts text items included in the item list; COUNT does not.

@CELL and CELL

Because the two spreadsheet programs have different cell formatting options, 1-2-3's @CELL returns values different from ANGOSS' CELL function when used with the "prefix," "format," and "width" arguments.

@CELLPOINTER AND CELLPOINTER

See @CELL and CELL.

@CHAR and CHR

1-2-3's @CHAR returns the text character corresponding to Lotus International Character Set (LICS) decimal value of the numeric expression. ANGOSS' CHR returns the text character corresponding to the Smart Character Set (SCS) decimal value of the numbered expression.

@CODE and ASC

1-2-3's @CODE returns the Lotus International Character Set (LICS) value of the first character in the text expression. ASC returns the Smart Character Set SCS value of the character in the text expression.

@COUNT and COUNT

@COUNT enumerates all text items. COUNT does not include text items.

@DATEVALUE and DATEVALUE

Both functions calculate the number of days between December 31, 1899 and they'd specified in the argument. However 1-2-3 treats the year 1900 as a leap year, inserting a count for February 29. This date did not actually exist on the calendar. Thus ANGOSS' DATEVALUE returns a value 1 less than the value returned by 1-2-3's @DATEVALUE for dates after February 28, 1900.

@DAVG and @DAVERAGE

See AVG and AVERAGE

@DCOUNT and DCOUNT

See @COUNT and COUNT

@DMAX and ANGOSS' @DMAX

1-2-3's @DMAX treats text items as having a numeric value of zero. ANGOSS' @DMAX ignores text items.

@DMIN and ANGOSS' @DMIN

1-2-3's @DMIN treats text items as having a numeric value of zero. ANGOSS' @DMIN ignores text items.

@DSTD and @DSTDEV

See STD and STDEV

@DVAR and DVAR

See @VAR and VAR

@FV and FVA

The syntax for these functions differs slightly as follows:

@FV (payment amount, interest, term)

FVA (payment amount, term, interest)

@HLOOKUP and HLOOKUP

@HLOOKUP requires that the numeric data in the search row (the first row) of the table be in ascending order. If no exact match for the search value is found, @HLOOKUP determines the lookup column by finding the largest value in the search row that is less than or equal to the lookup value. This is sometimes termed a "tax table" lookup.

@LOOKUP searches for an exact match for the search item in the search row. HLOOKUP can be used to match both values and text. The order of information in the search row does not affect processing.

@IF and IF THEN ELSE

Both of these functions return the first value if the logical expression is true, otherwise, the second value is returned. However, the syntax differs as follows:

@IF (*logical, "true" expression, "false" expression*)

IF *logical* THEN *"true" expression* ELSE *"false" expression*

@INDEX and INDEX

The arguments for @INDEX must be specified in a sequence different from those of INDEX:

@INDEX (block, column offset, row offset)

INDEX (block, row offset, column offset)

@INT and INT

@INT returns the next largest integer for negative numbers; INT returns the next smallest.

Formula	Result
@INT(-2.5)	-2
INT(-2.5)	-3

@MAX and MAX

@MAX treats text items as having a numeric value of zero. MAX ignores text items.

@MID and MID

ANGOSS' @MID function assigns a value of 0 to the first character in the text string. ANGOSS' MID function assigns a value of 1 to the first character in the text string. @MID requires the third (length argument; MID returns the remainder of the text if no length is specified.

@MIN and MIN

@MIN treats text items as having a numeric value of zero. MIN ignores text items.

@NPV and NPV

NPV assumes an initial negative cash flow representing an invested sum and calculates periodical returns from that point. For @NPV, the first value represents a return (or payment) at the end of the first time period.

@PMT and PMT

These functions return the same value, but the syntax for them differs slightly as shown below.

@PMT (principle, term, interest)

PMT (principle, term, interest)

@PV and PVA

These functions return the same value, but the syntax differs as follows:

@PV (payment amount, interest, term)

PVA (payment amount, term, interest)

STD and STDEV

STD calculates the standard deviation based on a population variance; STDEV calculates the standard deviation based on a sample variance. STD treats text items as having a value of zero; STDEV ignores text items.

@STRING and STR

The second argument of 1-2-3's @STRING function specifies the number of decimal places. The optional second argument of STR specifies the number of significant digits to be returned.

@TERM and TERM

The TERM function calculates the term over which a fixed, regular payment must be made in order to equal a principal amount. @TERM calculates the term over which a periodical annuity payment must be made to equal a specified future value.

@TIME and TIME

@TIME returns a decimal fraction representing the time of day as a fraction of the whole day. TIME returns the current system time as a time expression (text).

@VAR and VAR

@VAR returns the population variance of the items in a list. VAR returns the statistical variance for the numeric items in a list. @VAR includes text cells, giving them a value of 0, in the calculation; VAR ignores text cells in the list.

@VLOOKUP and VLOOKUP

@VLOOKUP requires that the numeric data in the search column (the first column) of the table be in ascending order. If no exact match for the search value is found,

@VLOOKUP determines the lookup row by finding the largest value in the search column that is less than or equal to the lookup value. This is sometimes termed a “tax table” lookup.

VLOOKUP searches for an exact match for the search item in the search column. VLOOKUP can be used to match both values and text. The order of information in the search column does not affect processing.

@YEAR and YEAR

@YEAR returns a value amounting to the last two digits of the year (e.g., 89 rather than 1989). YEAR returns the number corresponding to the numeric value of the year (e.g., 1989 rather than 89).

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y