Advanced Features in QuickCalc

Note: This document contains detailed descriptions of certain functions of QuickCalc BASIC:

- Indirect Name Reference
- **INPUTDIALOG** and **UPDATEDIALOG**
- The **SORT** statement and Sorting Arrays

Please scroll down to those sections if you are searching for those topics.

First of all, QuickCalc BASIC is **BASIC!**

It is designed to bring the experience and ease-of-use of the BASIC programming language to you in an easy-to-use Windows environment. If you have programmed in BASIC before, this is BASIC, and you should have practically nothing to learn. If you never programmed a computer before, QuickCalc is the place to start.

However, this version of BASIC adds a whole lot of features that were never part of the original BASIC language, and does so in a way that integrates seamlessly and allows you to access these new features within the structure of the BASIC language.

What Are The "Advanced Features" in QuickCalc BASIC?

Note: "Advanced" doesn't mean they require advanced programming skills. Many of these features make it EASIER to program in BASIC. Some require more advanced skills , and some just make it more FUN.

It is assumed that you are familiar with BASIC programming before using these "enhancements".

The following features are summarized in the document "***Functions, Statements and Commands***." Other documents mentioned provide more detailed information,

1. **Graphing**. This is probably QuickCalc's most powerful tool. With just a few simple statements, you can easily produce **publishable-quality graphs** of data you produce in your BASIC program, plus charts, shapes, text, colors, and a whole lot more. Refer to "***Beginning Graphics***", "***Intermediate Graphics***" and "***Advanced Graphics***".

2. **Long (high-precision) Numbers**. How about 1 million digits of precision? All right, let's start with fifty. You can choose how precise you want it to be. And all the functions (trig, powers, roots, logarithms, exponentials, pi, etc.) are there.

Long numbers may be mixed right in with regular numbers whenever you need or want the extra precision. Refer to "***Working With Long Numbers***" for more.

3. **<u>Date/Time Variables</u>**. Dates and times (from microseconds to billions of years) are kept in "DATETIME" variables, which you can create, print out, graph, etc. Leap years and Daylight Saving Time are <u>handled for you,</u> but you can over-ride the time zone and daylight saving time rules, if you want. Refer to "***Working With Dates and Times.***"

4. **<u>User-Defined Functions which Invoke Subroutines</u>**. The original BASIC limited a user-defined function to single statement. QuickCalc expands that to support subroutines, *e.g.*,

   DEF *user-function* (*x, y, z …*) = **GOSUB** *statement-number*.

   The user-defined function invokes the subroutine, which returns a value (with the **RETURN** statement). Refer to the document "***User-Defined Functions***".

5. **<u>Indirect Name Reference</u>**. With this programming tool, you can refer to a variable at execution time with a string (variable or expression) containing the <u>name</u> of the variable. This feature is **<u>described in detail later in this document</u>**.

6. **<u>Print Formatting Enhancements</u>**. QuickCalc provides many new tools to format data for printing. These include:

   - C-type formats (for you C and C++ programmers)
   - "###.cc" takes "integer" cents and formats it as dollars.cents
   - "MAX" which will show the maximum precision in a DOUBLE floating-point number
   - "ENG*xxx*" [Engineering notation] allows you to format numbers as m-amp, K-ohm, etc., with 3 significant figures (decimal points aligned). (See "***Introduction to QuickCalc***")
   - Formatting long (LONGMATH) numbers. (See "***Working With Long Numbers***").
   - Ability to **PRINT** to a string variable, and use **FIELD** statements to format a string variable.
   - You can override the default print format.
   - All format specifications also work with **STR$**.

7. **<u>Statement Numbers are Optional</u>**. You only need them to "label" targets of **GOTO**, **GOSUB**, and **DATA** statements referred to by **RESTORE**. Statement numbers don't have to be in sequence.

8. **<u>INPUTDIALOG and UPDATEDIALOG</u>**. These two (nearly identical) statements/functions make it easy to create a "Windows-style" dialog for entering multiple data values. This powerful tool is **<u>described in detail later in this document.</u>**

9. **MESSAGEBOX**. Another "Windows-style" function that shows you a message and lets you choose what to do next.

10. **SELECT** Dialog. This function allows you to select an item from a string-array. A dialog opens up with a scroll-down list, from which you can make a selection.

11. **SPAWN** statement. Allows you to launch virtually any program, web URL, document, *etc*., from QuickCalc.

12. "**Mouse-Clickable" Shapes.** When drawing shapes on a graph, you can make a shape "Clickable", that is, when the user clicks on it, you can take action depending on which shape was clicked. See the document "***Advanced Graphics***".

13. **BROWSEINPUTFILE$** and **BROWSEOUTPUTFILE$** These functions let you use the Windows file-browsing dialog to search for files to read or create, rather than having to type them in manually.

14. **SERIAL and BINARY Files.** With Serial files you can communicate with micro-controllers like Arduino over USB or Serial ports. With Binary files you can read and write non-text data (See "***Working With Files***")

15. **Degrees** and **Radians**. You can choose to work all trigonometric functions (and graphic shapes) in either degrees or radians, whichever is more comfortable for you. The accuracy is the same.

16. **Append** and **Include**. These statements allow you to piece together code segments or subroutines at run time. Include copies another file into your program at the spot of the Include statement. Append adds the file to the end of the program. They facilitate writing common subroutines and definitions for use in multiple programs.

17. **Help Documents** and **Quick Reference Booklet.** QuickCalc BASIC includes over 200 pages of documentation and references. These are organized into PDF files and accessible from the HELP screen, so you can look up information while you are running QuickCalc. And all the information (or just the portion of interest) may be printed. The Quick Reference booklet may be printed and kept close at hand while you are writing BASIC programs.

18. **The Log File.** Everything that is written to the QuickCalc screen is also written to a log file. This makes it easy to copy and/or print the output of your programs.

19. **Long Hexadecimal Numbers**. You can convert long decimal numbers back and forth to long hexadecimal numbers of virtually any length (the **HEXCONVERT** function), and the hex numbers also may contain a [hexa-]decimal point and data to its right. (See "***Number Representation, Assignment and Conversion***")

20. **Debugging Functions**. No program is perfect, especially the first time you run it. To find out why, QuickCalc provides a lot if different debugging functions (See "***Debugging***"). You can trace or step through your program. In addition, there are over 300 "error messages", designed to show you what went wrong and where. There are also timing functions to find out why your program is running slower than you would like.

21. **Relaxed Number Assignment Rules.** Some BASIC interpreters are very strict about numbers and variables. With QuickCalc, you can use a string variable where a number is required, provided the string represents a number. Number constants and data entered are kept as "number strings" and converted when needed, so that no precision is lost. (See "***Number Representation, Assignment and Conversion***")

22. **BEEP can play any .WAV File**. Usually, the Beep statement does just that, *beep*!. Now you can use it to play music or give you audible cues. How about sound clips from movies ("*Use the Force, Luke*.")

23. **SORTING Arrays.** QuickCalc has a SORT statement to quickly sort arrays and sub-arrays by multiple keys. This feature is **described in detail later in this document.**

24. **"You Can Program Your Own Computer"**. This is the title of a book I wrote for the Kindle, available from Amazon (99 cents). It introduces the non-programmer to BASIC in an easy-to-read book.

25. **Online Library of Sample Programs.** These sample programs, which you may copy and use freely, illustrate the features of QuickCalc BASIC, and demonstrate how they are used. Many are useful applications in their own right.

Indirect Name Reference

(This is an ***advanced function*** of QuickCalc BASIC).

Sometimes the variable or array name you want to reference is not known at the time the program is written, or it may change during program execution, or be different depending on how a subroutine is called.

A variable or array name may be specified *at execution time* as the contents of a string or string expression. Use the function @(*string-expression*). The string expression must contain the name of a variable or array. The resulting variable will be used in the statement.

Example:

```
a = 45
s$ = "a"
b = @(s$)
```

The indirect reference @(*s*$) looks up the current value of *s*$ which is “a”. The variable with the name *a* is then looked up. It contains the value 45, which is then assigned to *b*.

Indirect name references may be used anywhere a name is referenced – in arithmetic expressions, subscripts, function parameters, function names, statement names, dimension statements, etc. They may ***not*** be used in **DATA** statements or any other statement which is processed before the program is run.

The string expression must evaluate to a BASIC variable name. The referenced name does not have to be previously defined. It will simply be substituted into the statement as the statement is executed. The resulting statement, after the substitution, must be valid or errors will occur.

If the string expression contains characters not allowed in variable names, the invalid characters and all following characters will be ignored. For example:

"abc d"	becomes "abc" (space is not allowed)
"a1/2"	becomes "a1" (slash is not allowed)
" abc"	becomes null string, which is invalid and causes an error.

Examples of substitution: (assume *s*$ contains “abc”, *b*$ contains “sin”)

DIM @(*s*$)(10,10)	becomes	**DIM** *abc*(10,10)
@(*s*$) = @(*s*$) + 1	becomes	*abc* = *abc* + 1
d = @(*b*$) (*x*)	becomes	*d* = **sin** (*x*)
e = @(*s*$+*b*$)	becomes	*e* = *abcsin*
f = **SQR** (@(*s*$))	becomes	*f* = **SQR** (*abc*)
g=@(*b*$) (@(*s*$))	becomes	*g*=**SIN** (*abc*)

A good example of indirect name referencing appears when working with complex variables. Since each function must return two variables (real and imaginary), an easy way to accomplish this is to pass the ***name*** of the array containing the complex variable you want returned. This way the subroutine may indirectly place the results in both variables. A simple example:

```
real = 0:  imaginary = 1
DIM     a (2),  b (2),  c (2)  // complex variables
DEF     complex_add (s1$, s2$, s3$) = GOSUB 500
a (real) = 3:  a (imaginary) = 4
b (real) = 6:  b (imaginary) = 2
xx = complex_add ("a", " b", " c")
PRINT c (real), c (imaginary)
END

500  // Subroutine to add two complex numbers
@(s3$)(real)          = @(s1$)(real)          + @(s2$)(real)
@(s3$)(imaginary) = @(s1$)(imaginary) + @(s2$)(imaginary)
RETURN 0  // return code not used in this case.
```

See the document "***User-Defined Functions***" for more examples of complex variable functions.

As another example, you could place the names of **any number of variables** in an appropriate-size [string] array and pass the name of that array to a subroutine. The subroutine could reference the variables indirectly and pass back any number of results. This effectively removes the limits on the number of parameters for user-defined functions.

Dialog Input (INPUTDIALOG and UPDATEDIALOG)

QuickCalc BASIC gives you the ability to **generate input dialog boxes** to facilitate entering your data. These dialogs are under BASIC program control, and do not require any Windows programming knowledge on your part.

These dialog boxes size themselves automatically and give you control over the format of the dialog box. You can

- set the location and size of each field,
- place a prompt above each field,
- Have more than one field on each line (row)
- Generate your own buttons.

There are two new functions to support this: **INPUTDIALOG** and **UPDATEDIALOG**.

> **UPDATEDIALOG** is identical to **INPUTDIALOG**, except that the **edit fields are pre-loaded** with the current values of their associated variables. This allows you to change them without typing them all in again.

Each one has a **statement** form and a **function** form. These are described below.

Format

Statement form:

INPUTDIALOG [*title* **;**] *variable-name-1*, *variable-name-2*, …
UPDATEDIALOG [*title* **;**] *variable-name-1*, *variable-name-2*, …

Note: If used, the *title* must be followed by a semicolon.

Function form:

return-code = **INPUTDIALOG** ([*title* **,**] *string-array*)
return-code = **UPDATEDIALOG** ([*title* **,**] *string-array*)

title is a title (or "caption") for the dialog box. It is optional. It must be a string variable or quoted string constant. If you don't specify it, the title will be "Enter Data:".

string-array Specifies the name of the string array, which is a one- or two-dimensional array containing the data to define the dialog box and the variables you want to enter. This form gives you maximum control over the appearance [layout] and contents of the dialog box, the variables to be entered and prompts for each field.

Note: The string array must have been dimensioned and loaded with the definitions before using it here. Normally, you would use **DIM** *string-arra*y (*n* ,5) where *n* is at least as large as the number of fields you want to define.

variable-name-1, -2, *etc*.

These are used with the statement form. This is the "quick and dirty" way to use **INPUTDIALOG** and **UPDATEDIALOG**, since it can be done with a single program line. You simply list the names of the variables you want to enter (maximum of 6). The variable names should **not** be in quotes and not in parentheses. This is similar to the **INPUT** statement, but looks better and is more user-friendly.

The names may be any valid variables (*i.e*.. string variables, numeric variables, field variables, array entries, *etc*.) The name of each variable will appear in the dialog box as a prompt for each field, so give the variables meaningful names.

Note: The statement form of **INPUTDIALOG** or **UPDATEDIALOG** is really intended for program development and debugging.

If someone else it going to be using the program, it is much more intuitive (and professional-looking) to use the function form with a string-array.

Also, the statement form provides no return code. If the user presses ENTER or the "OK" button, the variables are updated and the program continues. If they press ESC or click the [X] the variables are not updated and the program is terminated.

The INPUTDIALOG Array

The array is a **string array**.
Each row of the array defines one field (or edit box) in the dialog.

The format of *each row* of the array is

variable-name, *prompt*, *row*, *start-column*, *width*

variable-name — is a string **constant** containing the **name of the variable** you want to receive the data you enter into this field. If it is a number (1-99) it means you want a button (see below).

prompt — is a string **constant** containing the **text to display** above the edit box, prompting the user what data is required in that field.

Note: If the prompt string contains commas or colons or significant leading blanks, it must be enclosed in quotes. Do not embed special characters (like line-feed) in the prompt string.

Note: The prompt string is displayed above the edit box and has the same width. Make sure your prompt string is not too long.

Note: If you want to use, as a prompt, the contents of a string variable at run time, assign it to the proper element in the array, *e.g.*, *dialog_array*$ (0, 1) = *s*$, overriding the prompt that you set when you initialized the array.

row is the row in the dialog where you want this field to appear. Rows are numbered **0 through 5**, starting at the top. If two fields are on the same row and they overlap, the second one (which should be to the right of the first) will cause the first one to be shortened.

start-column (Expressed as a **percentage** of the complete row). This defines where the edit field starts in the row. It ranges from **0 to 95**, where 0 is the left side of the dialog. If this field overlaps another, it must start at least 7% higher than the start of the other field.

width (Expressed as a **percentage** of the complete row). This defines the width of the field. It ranges from **5 to 100**, where 100 is the full width of the row. If you specify 0 it assumes you mean 100. If the width extends beyond the right side of the dialog it will be shortened to fit.

Note: You don't need to specify the width unless you want a field which is shorter then the distance to the next field or the right side of the dialog. You can set all the widths to 0 or 100 if you like.

Note: *row*, *start-column* and *width* are numeric values, but must be stored in the string array.

If they are number strings, they can be assigned directly, *e.g., dialog_array*$ (4, 2) = 3,

They can be **read** into the array, *e.g.*, **READ** *dialog_array*$ (4, 2).

If they are in numeric variables, they must be converted to strings, as follows: *dialog_array*$ (4, 2) = **STR$** (**FIX** (row))

Note: Don't confuse the row in the dialog with the row in the array which defines it.

Note: Edit fields should be specified in the following order:

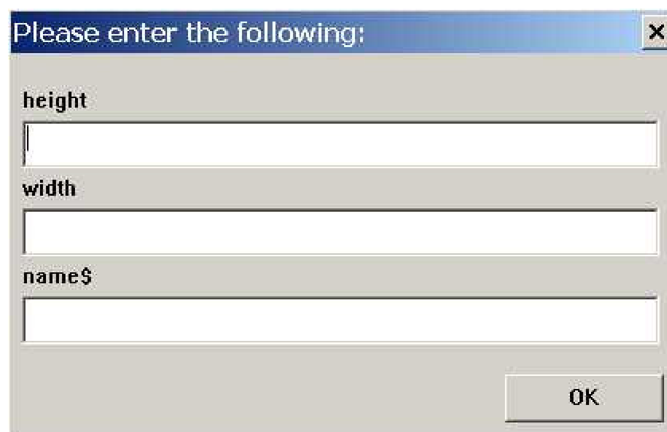
First (leftmost) entry in first row
Next entry in first row

…
Last (rightmost) entry in first row
First (leftmost) entry in second row
(*etc*.)

Example (Statement Form)

INPUTDIALOG "Please enter the following:" **;** *height*, *width*, *name*$

This produces the following dialog:



It can't get much easier than that, and this statement may even be entered from the command line.

However, this form has its limitations:

- You can't set the widths of the fields (each one is the full width).
- The names of the variables are used as the prompts (*e.g.*, name$).
- You are limited to a maximum of six variables.
- You can't create your own buttons.
- There is no return code.

The **function form** is much more flexible (but this one is "quick and dirty", and still looks a lot better then the **INPUT** statement).

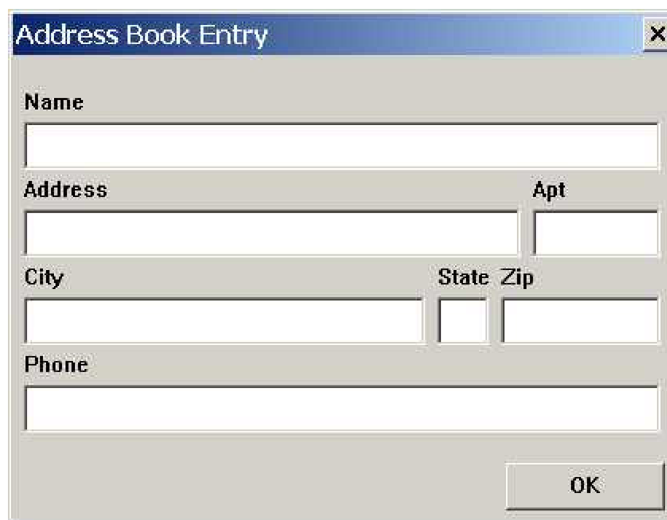
Example (Function Form)

```
DIM     dialog_array$  (7, 5)
DATA    name$,        Name,       0, 0,   100
DATA    address$ ,    Address,    1, 0,   100
DATA    apt$ ,        Apt,        1, 80, 100
DATA    city$,        City,       2, 0,   100
DATA    state$,       State,      2, 65, 100
DATA    zip$,         Zip,        2, 75, 100
DATA    phone$,       Phone,      3, 0,   100

FOR i = 0 to 6
FOR j = 0 to 4
READ dialog_array$ (i, j)         // load up the array
NEXT
NEXT

rc  = INPUTDIALOG  ("Address Book Entry";  dialog_array$)
```

This produces the dialog box shown below:



Type the data into each field, then press "tab" to advance to the next field. After the data is entered, click the "OK" button (or press Enter). The variables referred to in the array will be set to the data that you typed in. The return code will be 0.

If you click on the [X] button, or press ESC, the dialog will be closed. The return code will be -1.

Note: The "OK" button is automatically generated, unless you specify buttons of your own (see below);

Using Different Size Arrays

Normally, the array is 5 wide, as specified above. Narrower arrays may be used, with fewer options, as described below:

If array is 4 wide: (no width specified)
Each field on a line stops 2 % before the specified start of the next field. The last field on a line takes up the remaining width on the line.

If Array is 3 wide: (no start position or width)
All fields start at 0 % and go full width. Make sure not to place 2 fields on the same line. This limits you to a maximum of 6 fields.

If array is 2 wide, (no line number, start position or width)
All fields start at 0% and go full width. Each field is on a separate [consecutive] line, starting with line 0.

If array is 1 wide: (no prompt, line number, start position or width)
The variable name is used as the prompt. Printed as specified (*i.e.*, case).

Note: A 1-wide array means one dimensioned (n, 1) or simply (n). Both are handled the same.

Overlapping Fields

If fields are specified (on the same line) which overlap, the program will attempt to adjust the sizes so the overlap is corrected.

Each field on a given line takes precedence over the one before it.

So when they overlap, the first one is shortened, so that it ends at 2% before that start of the next one.

If the width of the first field is now less than 5% it will cause an error.

If a field extends beyond the right side of the window, it will be shortened to fit.
If the result is less than 5%, it will cause an error.

Order of the Fields

Fields should be specified in the array in **increasing row order**, the first row being 0.

Within a row, fields should be specified in **increasing column** [percentage] value, [left-to-right].

Columns specified in the wrong order may give undesirable results.

Tab stops are set in the order the fields appear in the array. Adding a field out of order will cause the tab key to jump to the wrong field.

It is OK to skip a row if you want to leave some empty space. The row skipped still counts as one of the six.

Generating Your Own Buttons (Function mode)

The program normally places an "OK" button on the line below the last row of the dialog. You can override that and generate your own buttons.

Simply replace the *variable-name* field with a number (from 1 to 99). Each button must have a different number.

When you specify your own button[s], the default "OK" button is not generated.

The string you specified in the prompt for that field will appear **on** the button. Make sure the prompt is short enough to fit on the button.

The same rules for row, start column, width and overlapping apply for buttons. You may place buttons anywhere on the dialog, **including row 6**.

Clicking on any button will close the dialog and return to the program (after checking numeric fields). Action is the same no matter which button is clicked.

The **return code** will be the **number** (which you specified as the *variable-name*) **of the button** you clicked. You can test the return code and perform the appropriate action.

If you clicked on [X] or pressed ESC, the return code will be -1.
If you pressed ENTER, the return code will be 0.

A Simple Example of Buttons:

```
DIM    dialog_array$  (4, 5)
DATA   name$,         Name,        0, 0,   100
DATA   address$ ,     Address,     1, 0,   100
DATA   1 ,            Delete,      2, 0,   100
DATA   2 ,            Save,        2, 52,  100

FOR i = 0 to 3
FOR j = 0 to 4
```

```
READ dialog_array$ (i, j)                    // load up the array
NEXT
NEXT

rc  = INPUTDIALOG  ("Address Book Entry",  dialog_array$)
```

This produces the following dialog:

The return code will be 1 if you clicked “Delete” and 2 if you clicked “Save”

Data Entry Rules:

Data entered into the fields must match the format of the target variable. No editing is done on the data *as it is entered*.

When the dialog is closed (by pressing ENTER or a button, **numeric fields** (DOUBLE and LONGMATH) are **checked for validity**. If they are not valid numeric data, a message box pops up letting you know, and prompting you to re-enter the data. You may then re-enter (or edit) the number and try again. You can't leave the dialog until all fields are OK, unless you cancel out of it (ESC or [X]).

Note: If you cancel out of a dialog after re-editing a numeric value, the variables higher up in the dialog array will have already been stored.

Input variables may be strings, **FIELD** variables, numeric (DOUBLE or LONGMATH) variables, string array elements, or numeric (DOUBLE or LONGMATH) array elements.

For **numeric variables**,

What you type in must be convertible into a number. If not, it will generate an error message when it tries to assign the value to the numeric variable. It will then allow you to re-type it.
If you leave the field blank for a numeric variable, the variable will be set to zero.
If you enter a quoted string (*not recommended*), it must contain at least one digit and no invalid characters (including blanks).

Note: Some precision ***may*** be lost when you update a number, even if you don't change it. DOUBLE variables are converted into a string format for display/edit, then converted back to DOUBLE. LONGMATH numbers will be shortened to about 244 digits (+ sign + decimal-pt + "E+nnnnnnn" = 255) for editing. Even if you don't change it, the result will be the new length. If the number is < 244 digits, there will be no loss of precision.

Note: You can, of course, take numeric data in as string data and convert it later. If you enter a number into a string variable, it will be saved as a string which you may convert later, assign it to a numeric variable, or do further tests on it.

For **string variables**:

String data entered is subject to same rules as for the **INPUT** statement.

Strings containing new-line or carriage-return characters **must be enclosed in quotes** and use \n and \r in place of those characters. If the string is in quotes, then you must also substitute \" for a quote, \\ for a single backslash, \+ and \- for superscript and subscript codes (see the **TEXT** statement).

> **Exception** [different than **INPUT**]: strings with embedded blanks or starting with numbers, but not containing the above characters, do not need quotes.

Leading and trailing blanks are stripped off, unless the string is in quotes. For **FIELD** variables, trailing blanks are always removed.

If the string you enter starts with a quote, it must also end with one.

If a string begins with a quote and that quote is considered to be part of the string, then the preceding rule applies, *i.e.*, enclose string in quotes and use \" for the leading quote (*e.g.*, for "abc enter "\"abc"). A quote or backslash in the *middle* of an *unquoted* string is OK.

Quotes will be stripped off and substitutions made when the strings are stored.

The maximum length string you may enter is 255 **minus** the added characters from the substitutions and the leading and trailing quotes. This shouldn't be a problem in normal use.

You may enter very long strings into the edit fields [they will scroll horizontally], however they will be truncated to 255 if assigned to a string, and to the field length if assigned to a **FIELD** variable.

Using FIELD variables in INPUTDIALOG and UPDATEDIALOG.

Since **FIELD** variables can be used as variables in **INPUTDIALOG** and **UPDATEDIALOG**, this provides an **easy way** to update RANDOM files. Define a dialog referencing the fields in your RANDOM file record, **GET** the record, and do the **UPDATEDIALOG**. When the dialog is exited, all the updated fields are already in place in the file buffer, ready for you to do a **PUT**. You can, of course, do further checks on the data before writing it, and do another **UPDATEDIALOG** with the same or different fields. Some fields may be from one **FIELD** statement and some from another, or a different source. There is a lot of flexibility here.

(See the sample program "Address Book" for an example of using **INPUTDIALOG**, **UPDATEDIALOG**, Buttons, **FIELD** statements, and RANDOM files.)

The SORT Function and Sorting Arrays

New to version 2.4.1 is the **SORT** statement. This gives you a very quick and simple way to sort the rows of an array by different keys (columns in that row).

Features

- We can sort arrays which are DOUBLE, String, or LONGMATH.
- Arrays may be of any legal order and size (subject to memory limitations).
- Sort can be Ascending or Descending.
- We can ignore case [default] or consider it when comparing strings.
- We can sort a sub-array (defined by a subset of the indices).
- We can sort by something other than the first entry in the column.
- Sorts are, by default, stable.

The SORT Statement:

SORT *array-name* [(*index-1*, *index-2*, …)]
[, count=*number-of-rows-to-sort*]
[, keycol1=*primary-sort-column*]
[, keycol2=*seconday-sort-column*]
[, keycol3=*tertiary-sort-column*]
[, DESCENDING]
[, CASE]
[, UNSTABLE]

array-name is the array you want to sort. Don't specify indices (or subscripts) unless you want to sort on a sub-array (see discussion below).

(***index-1***, ***index-2***, …)

If you want to treat a lower-order portion of an array as a "sub-array" and sort on that, specify the higher-order indices that define that sub-array.

For example, if the array is dimensioned (4,2,10,40) and you specify *array-name* (2, 1), you will sort the sub-array beginning at *array-name* (2,1), which is an array or order 2 (10 rows by 40 columns). The rest of *array-name* is left alone.

count specifies the number of rows to sort. It must be an integer. If *count* is 0, (or not specified) the number or rows sorted will be the highest dimension [*i.e.* the number of rows] of the array (or sub-array). If *count* is greater than the highest dimension, it will be set to the highest dimension.

Note: If the array is not filled, the empty elements will be zero or null strings, which will usually come out first in the sort. Count allows you to sort on just the entries which have been used.

keycol1, ***keycol2***, ***keycol3***

represent the **columns** in the row which are used as the **sort keys**. Each may be from 0 to 1 less than the second-highest dimension ("column"), *e.g.*, a valid index.

If the *keycol* parameters are not specified, the sort will be on column 0, or the first entry in the row. If there are no columns (order=1), the sort will be on the only entry in the row, and these parameters will be ignored.

Note: You may specify none, 1, 2 or 3 key columns. The second and third are used to "break a tie" when the first keys compare equal.

Note: If you specify *keycol2*, you must also specify *keycol1*. If you specify *keycol3*, you must also specify *keycol2*.

DESCENDING sorts in reverse order – highest to lowest. (Default is ascending).

Note: You can't sort by one *keycol* ascending and another descending in the same sort.

CASE causes upper and lower case letters to be sorted differently. The default is to ignore case differences. This parameter is only used for string arrays.

Note: When CASE is specified, all capital letters will sort before any lower-case letters, *i.e.*, "Z" sorts before "a" (ASCII order).

UNSTABLE causes an "**unstable sort**" to be performed.

By default, the sort is "**stable**". This means that the original order is preserved in rows which compare equal. A stable sort takes about 10% more time than an unstable sort, however, the difference is typically less than 1 millisecond for 1000 rows. If you don't care about preserving order for equal keys and want that extra millisecond back, specify "UNSTABLE".

For an example of a BASIC program using SORT, see the sample program "**sort demo.txt**".

Notes on Rows and Columns:

The whole concept of rows and column breaks down when the order of the array is greater then 3. It is difficult to even imagine a 4- or 5-dimensional array. In

this discussion, "**row**" will be used for the highest order index or the array (or sub-array), and "**column**" for the 2nd highest index.

Notes on sub-array sorting:

A **one-dimensional** array has an **order of 1**. It has a single index to specify the entry number [row].

One-Dimensional Array

Rows

(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)

A one-dimensional array is just a list. It has rows, but not columns – it cannot have a sub-array. Each row contains one entry. The sort is done on that one entry.

A **two-dimensional** array has an **order of 2**. It has two indices to specify row and column.

Think of it as a table, with rows and columns. The highest-order index specifies the row.

Two-Dimensional Array

Columns

Rows				
(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)
(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)
(5, 0)	(5, 1)	(5, 2)	(5, 3)	(5, 4)
(6, 0)	(6, 1)	(6, 2)	(6, 3)	(6, 4)
(7, 0)	(7, 1)	(7, 2)	(7, 3)	(7, 4)
(8, 0)	(8, 1)	(8, 2)	(8, 3)	(8, 4)
(9, 0)	(9, 1)	(9, 2)	(9, 3)	(9, 4)

Each row can have several columns, and each column has one entry. Therefore, each row can be thought of as a sub-array with an order of 1.

If the entire array is being sorted (no indices provided) then the sort key for each row defaults to the entry in the first column (column 0) for that row. If a key column is provided, the sort key[s] is/are taken from column[s] *keycol*n (relative to 0) for that row.

Example: If the columns represent last name, first name, …, then

SORT *array-name*, keycol1=1

will sort on column 1 (relative to 0), *i.e.*, first name.

If we want to sort a sub-array, i.e., a single row in this case, we specify

SORT *array-name* (*row-number*).

This means that only that row will be sorted. In this case, all the columns of that row will be sorted. No other rows in the array will be changed.

Note: Since the array order is 2, the sub-array has an order of 1, which means that a ***keycol* cannot be used here**, as each column (of the sub-array) contains only one entry.

A **three-dimensional** array has an **order of 3**. It has three indices to specify row, column and "plane" (for want of a better word).

Three-Dimensional Array

(0, 0, 0)	(0, 1, 0)	(0, 2, 0)	(0, 3, 0)	(0, 4, 0)
(1, 0, 0)	(1, 1, 0)	(1, 2, 0)	(1, 3, 0)	(1, 4, 0)
(2, 0, 0)	(2, 1, 0)	(2, 2, 0)	(2, 3, 0)	(2, 4, 0)
(3, 0, 0)	(3, 1, 0)	(3, 2, 0)	(3, 3, 0)	(3, 4, 0)
(4, 0, 0)	(4, 1, 0)	(4, 2, 0)	(4, 3, 0)	(4, 4, 0)
(5, 0, 0)	(5, 1, 0)	(5, 2, 0)	(5, 3, 0)	(5, 4, 0)
(6, 0, 0)	(6, 1, 0)	(6, 2, 0)	(6, 3, 0)	(6, 4, 0)
(7, 0, 0)	(7, 1, 0)	(7, 2, 0)	(7, 3, 0)	(7, 4, 0)
(8, 0, 0)	(8, 1, 0)	(8, 2, 0)	(8, 3, 0)	(8, 4, 0)
(9, 0, 0)	(9, 1, 0)	(9, 2, 0)	(9, 3, 0)	(9, 4, 0)

(0, 0, 1) (0, 1, 1) (0, 2, 1) (0, 3, 1) (0, 4, 1)
(1, 4, 1) (2, 4, 1) (3, 4, 1) (4, 4, 1) (5, 4, 1) (6, 4, 1) (7, 4, 1) (8, 4, 1) (9, 4, 1)

(0, 0, 2) (0, 1, 2) (0, 2, 2) (0, 3, 2) (0, 4, 2)
(1, 4, 2) (2, 4, 2) (3, 4, 2) (4, 4, 2) (5, 4, 2) (6, 4, 2) (7, 4, 2) (8, 4, 2) (9, 4, 2)

(0, 0, 3) (0, 1, 3) (0, 2, 3) (0, 3, 3) (0, 4, 3)
(1, 4, 3) (2, 4, 3) (3, 4, 3) (4, 4, 3) (5, 4, 3) (6, 4, 3) (7, 4, 3) (8, 4, 3) (9, 4, 3)

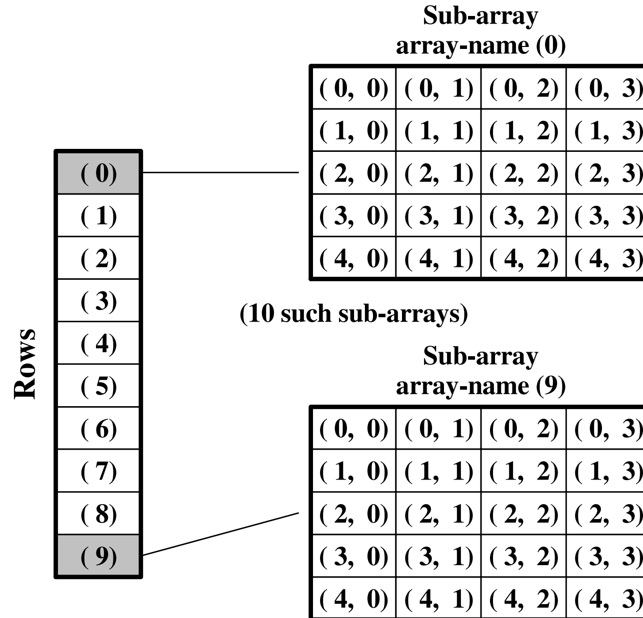
Rows
Columns
Planes

Each row of the array contains one or more columns and each column contains one or more planes. Specifying (row, column, plane) completely identifies exactly one element of the array.

If you specify a row, then that row contains all the elements beneath it. For example, if the array is dimensioned (10, 5, 4) then each row contains 5 columns of 4 elements each, or 20 elements). When you sort by rows, all 20 elements get "moved" along with the row.

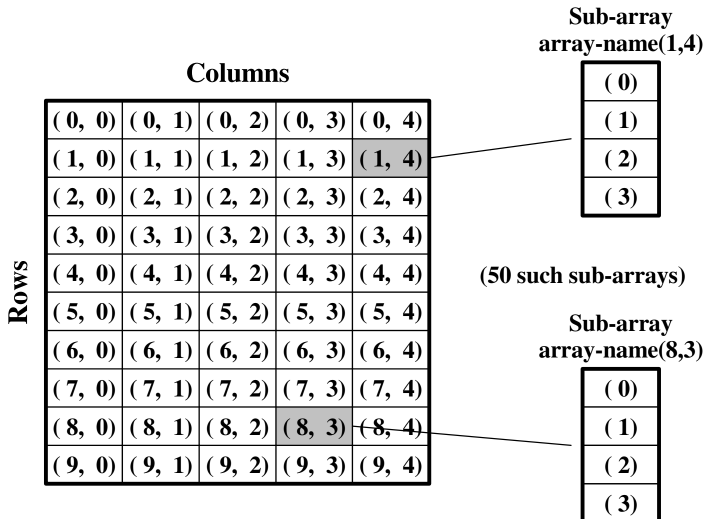
<u>A 3-dimensional array can be made into sub-arrays</u> in two different ways:

> (1) You can specify a single row, i.e., *array-name* (*row-number*). In this case, the specified row becomes a two-dimensional sub-array.



You can sort the rows in the sub-array by the first entry in each column [default] or specify a *keycol* to specify which columns contain the desired sort key[s].

(2) You can specify a single row and column, i.e., *array-name* (*row*, *column*). Now the plane specified by (*row*, *column*) becomes a one-dimensional sub-array whose entries can be sorted.



Note: Since the array order is 3, the sub-array has an order of 1, which means that the ***keycol* cannot be used here**, as each plane contains only one dimension.

Four-and Five-dimensional arrays are just extensions of the three-dimensional concept.

Summary (for arrays and sub-arrays)

If the order is 1, you can't make a sub-array.
If the order is 1, you can't use a *keycol*.
If the *keycol*s are omitted or zero, the sort key is
array-name (*row*, 0,0,…)
If a *keycol* is not zero, the sort key is
array-name (*row*, *keycol*, 0,0…), etc.
If a sub-array is desired, the number of indices supplied must be less than the order.
If a sub-array is desired, the order of the sub-array is (order - *#-of-indices)*.
Example: if order = 5 and you specify *array-name* (*n*, *m*) [two indices], then the sub-array has an order of 5-2 = 3.
All rows, columns, entry-numbers, etc., must be valid according to the DIM for the array.

Stable *vs* Unstable Sorts

Qsort (the sort routine provided in the Microsoft "C" library) is unstable, *i,e*, duplicate items don't necessarily retain their original sequence. Using secondary and tertiary sort keys can help, *e.g*., by numbering the rows in a particular column.

This implementation **is stable**. How this is accomplished will be discussed in a technical paper. In order to make it stable, we must allocate a 1-dimensional array which uses 4 bytes per entry (for String and LONGMATH) or 8 (for DOUBLE). At most, this would require 8K for 1000 entries.

> **Note**: It is theoretically possible for the sort to fail if it couldn't allocate the memory. If that happened [extremely unlikely], you could try using an unstable sort.

Stable sorts typically take from 5 to 15% longer. This usually not noticeable, since the extra overhead for a 1000 entry sort is typically about 1 millisecond (out of a total of 3 milliseconds). Perhaps that is too much of a difference for you, or you are sorting some extremely large arrays, in which case, you could specify "UNSTABLE".