TriggerCalc

&

TriggerCalc Pro

v.2.2

Another efficient and affordable ACT! Add-On by

http://www.exponenciel.com

Table of contents

INTRODUCTION

Purpose of the add-on

The main objective of TriggerCalc is to allow calculations between fields. The standard version may only perform calculation for the current contact. The Pro version may perform calculations for the current contact, the current lookup or all contacts.

A typical example of use would be that you want the value of FieldB to change when the value of FieldA changes. Using ACT! trigger feature, you could set TriggerCalc to automatically launch when the user exits FieldA, so that FieldB is automatically updated in the event any change was made to FieldA.

How it works

Most of the time, you will want to use TriggerCalc in conjunction with ACT! trigger feature to make automatic calculations when the user enters or exits a field but it may also be used to manually update fields.

ACT! trigger feature allows to launch a program when you either enter or exit a field: in ACT! *Define Field* screen (under the *Edit* menu), each field may be assigned an *Entry* or an *Exit trigger*, ie. the path of a program you want to fire either on entering or on leaving the field.

In this case, the program to be launched will be TriggerCalc. At the end of the TriggerCalc path, you simply add the calculation you want the program to perform, as in:

```
C:\Program Files\Exponenciel\TriggerCalc\TriggerCalc.exe  /[52]= [50]+[51]
|-----------------------------------------------------------------------|-|-------------------|
        TriggerCalc program path                                  +1 space+  calculation
```

Each field is referenced by its internal field ID in-between brackets. Here we are adding the User1 field (referred to as [50]) to the User2 field ([51]). The result is displayed in the User3 field ([52]).

TriggerCalc works with Group fields as well as Contact fields. The method for creating calculations is the same: TriggerCalc will automatically detect the active view.

Version differences

Features	Standard version	Pro version
Ability to run calculations for the current contact	✓	✓
Ability to run calculations for the current lookup or all contacts		✓
Syntax Editor		✓
Trigger Syntax Builder		✓
File referencing in trigger syntax		✓
20 + basic functions	✓	✓
30 + advanced functions		✓

Registering TriggerCalc

You may buy licenses from Exponenciel. Registration is based on ACT! user names, so you will have to supply your ACT! username when registering. Once given a registration code, make sure that ACT! is open then open Windows Explorer and the TriggerCalc folder (by default, c:\Program Files\Exponenciel\TriggerCalc). Double-click the TriggerCalc.exe file.

In the Standard version, dismiss the warning message telling you that no arguments were specified. The About window will show up. Type the registration code in the box below your username and close the window.

In the Pro version, simply go to the *Help* | *About* menu.

To upgrade from TriggerCalc to TriggerCalc Pro, proceed the same way and overwrite the existing registration code with the new one.

Support

For support, please contact support@exponenciel.com.

TRIGGERCALC STANDARD

The information in this section is valid for both the Standard and Pro versions. In some cases, the Pro version has advanced features that make some of this information less useful although still valid.

Setting up a calculation

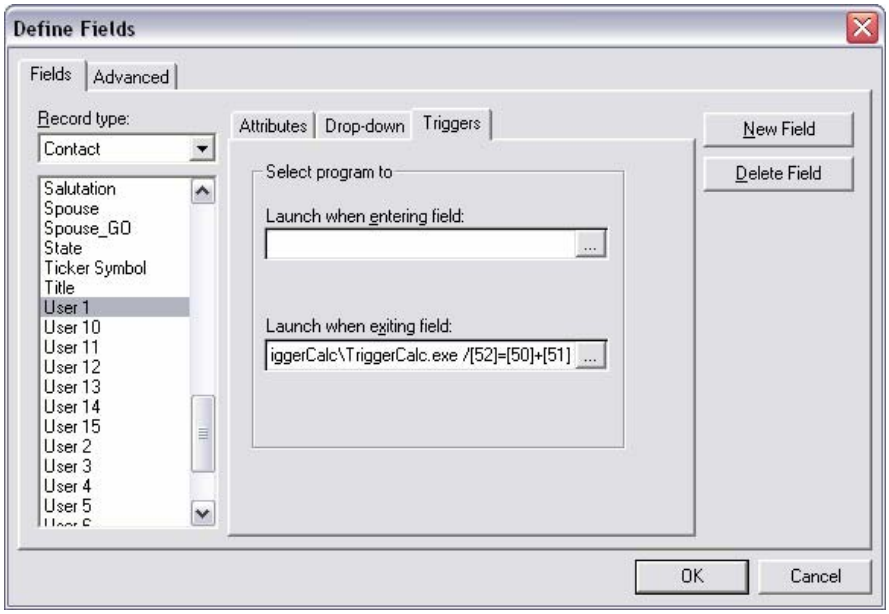
Using the trigger feature for automating calculations

As explained before, a typical use of TriggerCalc is in conjunction with ACT! trigger feature. This ACT! feature allows to launch a program whenever the user enters or exits a field.

To set up a trigger, you need to be an administrator of the database. Go to the *Edit* menu and choose *Define fields*, then select the field you want to set a trigger for and go under the *Triggers* tab. To set up a trigger, click on the "..." button in the trigger field and locate the program you want to be triggered.

In our case, locate TriggerCalc.exe (by default, it would be located in the c:\Program Files\Exponenciel\TriggerCalc folder), then click the *Open* button. The path will now appear in the trigger field. As shown below, simply add your calculation at the end of the line (see *Calculation syntax* below on how to determine the proper syntax for your calculation).

Note: *Make sure you insert a space between the program path and the argument but don't insert any space in the calculation itself or ACT! will declare the path invalid.*



Which field should have a trigger?

In the example shown here, the calculation adds the User1 to the User2 field and writes the total in the User3 field.

You will want this calculation to be performed automatically every time the user exits the User1 or User2 fields, so that you can make sure that whenever the user changes these fields, User3 is re-evaluated.

Therefore you would need to create 2 exit triggers with the same syntax: one for the User1 field and one for the User 2 field.

Calculation syntax

The syntax of your calculation is critical. Make sure you follow these rules:

- It is good practice although not mandatory to start with a forward slash.
- The first item of the calculation is the Target Field ID in-between brackets.
- Then the = sign.
- Then the calculation itself.

Determining Field IDs

In a calculation, each field is referred to by its ACT! internal ID. To know what the IDs of your fields are, run the actdiag.exe program located in the ACT! folder by double-clicking it in Windows Explorer, then go to the *Reports* menu and choose *Database Fields Report*. You will be prompted to choose a database and log into it, then asked if you want the report to be displayed in Notepad. Say yes and you'll get a report like this:

```
Database: C:\ My Documents\ACT\Database\ACT6demo.dbf
Record Count: 63
Field Count: 86
Table Name: CONTACT
...
Company         25
Contact         26
Address 1       27
Address 2       28
...
User 1          50
User 2          51
User 3          52
User 4          53
User 5          54
...
```

The field IDs are shown next to the field name.

Note that the field name may be different from the label used in your layout. If you can't find the field you are looking for, go to the *Tools* menu, choose *Design Layouts* and the name that appears in the field is the actual field name.

Ex: You want to write the sum of the User1 and User2 fields in the User3 field as in the example discussed before. You can see that:

User1 field ID is 50.
User2 field ID is 51.
User3 field ID is 52.

Hence the syntax: [52]=[50]+[51]

Operators

Basic Operators:

You may use any of these four basic operators:

+ (plus)
– (minus)
* (multiplied by)
/ (divided by)

You may use parentheses as well.

Examples of calculations:

You may use any combination of the above operators with or without parentheses. You don't have to use field IDs in the calculation if they are not necessary.

For instance:

/[53]=([51]+100)*0.05	→	User4 equals 5% of User2 + 100
/[53]=[50]+[51]-[52]	→	User4 equals User1 + User2 – User3
/[53]=([50]*125) + ([51]/5)	→	etc.
/[53]=[52]	→	Copies User3 to User4
/[53]=0	→	Resets User4 to 0
/[53]=[53]-10	→	Removes 10 from User4
/[53]=[50]+_+[51]	→	Concatenates User1 and User2 w/ a space between

etc.

Operator precedence

Typically the operator precedence (the order in which the operators are evaluated) is */+- which means that:

50*3+1 equals 151 and not 200 (you first multiply then add)

TriggerCalc follows this rule for simple calculations like the one above but for more complicated calculations, **make sure you use parenthesis to help TriggerCalc determine in which order it should proceed with the calculations** (in the example above it would mean writing (50*3)+1 instead of 50*3+1).

Besides the 4 basic operators, you may also use Boolean operators. **Yes** is returned if the comparison is true. **No** if the comparison is false.

- **>, <, >=, <=, <>** allow to compare values.
  Ex: if field 50 = 5 and 51 = 10 and [52]=[50]>[51] then [52] returns No.

- **NOT** is used to perform a logical negation.
  Ex: if field 50 = 5 and 51 = 10 and [52]=NOT([50]>[51]) then [52] returns Yes.

- **AND, OR, XOR** (logical exclusion), **EQV** (logical equivalence), **IMP** (logical implication) may also be used.
  Ex: if field 50=5 and 51=10 and [52]=([50]=5)OR([51]=8) then [52] returns Yes.

Note: *If you would rather get the numeric equivalent of True and False returned instead of* ***Yes*** *and* ***No****, add a tilde sign ~ in front of your calculation.* ***-1*** *will be returned instead of* ***Yes*** *and* ***0*** *instead of* ***No.***

field 50 = 5 and 51 = 10 and [52]=[50]>[51] then [52]="No"
[52]=~[50]>[51] then [52]="0"

Conditional statements

To a certain extent, you may use the ~ option (see the note above) to build conditional statements.

For instance, [50]= ~ - ([51]="blue")* 10 will return 0 if field 51 is not equal to "blue" and 50 if equal to blue (Note the tilde in front and then the minus sign because [51]="blue" will return -1 if true).

For true support of conditional statements, see the If function included in the PRO version.

Functions included in the standard version

TriggerCalc can accept a number of different functions. The function names are not case-sensitive.

Date/Time functions

- **Now()** will return the current date and time.
  Ex: [50]=Now() will insert "1/31/2003 9:10:50 PM" in field 50.

- **Time()** will return the current time.
  Ex: [50]=Time() will insert "9:10:50 PM" in field 50.

- **Day** will extract the day from a date.
  Ex: if field 50 = "2003/01/31" then Day([50]) returns 31.

- **Weekday** will return the day of the week (Sunday=0, Monday=1, etc.)
  Ex: if field 50 = "2003/01/31" then Weekday([50]) returns 6

- **Weekdayname** will return the name of the day.
  Ex: if field 50 = 2003/01/31 then Weekdayname([50]) returns Friday.

- **Month** will extract the month from a date.
  Ex: if field 50 = 2003/01/31 then Month([50]) returns 1.

- **Monthname** will return the name of the month.
  Ex: if field 50 = 2003/01/31 then Monthname([50]) returns January.

- **Year** will extract the year from a date.
  Ex: if field 50 = 2003/01/31 then Year([50]) returns 2003.

- **Hour**, **Minute** and **Second** will extract the hours, minutes and seconds of a date/time.
  Ex: if field 50 = "9:10:50 PM" then Hour([50]) returns 21 (for 9PM).

String functions

Even though not always necessary, it is recommended to always surround strings with quotes.

- **&** is really an operator. It concatenates strings.
  Ex: if field 50= "Mr." and 51= "Huffman" then [50]&[51] returns Mr.Huffman.
  To insert a space use the underscore character as ACT! won't allow a space in the arguments: [50]&"_"&[51] returns "Mr. Huffman".

- **Ucase** will convert to uppercase.
  Ex: if field 50 = "Chicago" then UCase([50]) returns CHICAGO.

- **LCase** will convert to lowercase.
  Ex: if field 50 = "Chicago" then LCase([50]) returns chicago.

- **Fcase** will capitalize every first letter of the words of a string
  Ex: if field 50="A1 HARDWARE" then Fcase[50] returns A1 Harware.

- **Trim** will remove leading and trailing spaces.
  Ex: Trim(" Chicago ") returns Chicago.

- **Ltrim** will remove leading spaces.
  Ex: Ltrim(" Chicago") returns Chicago.

- **Rtrim** will remove trailing spaces.
  Ex: Rtrim("Chicago ") returns Chicago.

- **Len** will return the number of characters in a string.
  Ex: if field 50 = " Chicago" then Len([50]) returns 7.

- **Space(x)** will insert x spaces.
  Ex: space(10) returns " " (10 spaces).

- **Val** returns a number in a string. It stops at the first character it does not recognize as part of a number.
  Ex: if field 50 = "12C" then Val([50]) returns 12.

Mathematical functions

- **Abs** returns the absolute value of a number.
  Ex: if field 50 = -5.32 then Abs([50]) returns 5.32.

- **Sgn** returns 1 if the number is >0, 0 if the number =0, -1 if the number is <0.
  Ex: if field 50 = -5.32 then Sgn([50]) returns -1.

- **Sqr** returns the square root of a number.
  Ex: if field 50=25 then Sqr([50]) returns 5.

- **Int** returns the integer portion of a number.
  Ex: if field 50=25.52 then Int([50]) returns 25.

Multiple calculations

TriggerCalc allows for multiple calculations. Simply separate your calculations by a backward slash:\. In this case, the forward slash to start any calculation is mandatory.

Example:

[Program_path]\TriggerCalc.exe /[53]=[52]+10\/[52]=[52]-5

In this example, the program sums up User2 + 10 and copies the value to User3, then it removes 5 to User2.

***Note:** the calculations are performed from left to right, which might make a difference if a field is used in a calculation and modified in another one, like in the example above.*

Errors

In case the calculation is invalid (like 1 divided by 0 or 50-"A"), TriggerCalc will return #ERR.

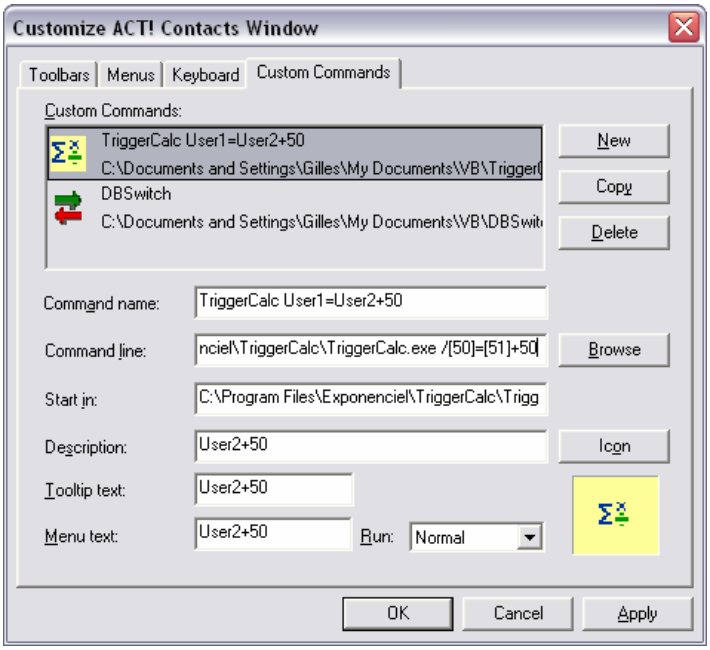
***Note:** numeric or currency fields in ACT! will not allow accept this message, therefore the field will remain blank or will be blanked out if it previously had a value.*
Therefore you may want to run tests before you convert your fields to numeric or currency, so that you can be made aware of any errors. When you are sure the calculations are properly built, you could then convert your fields to numeric or currency.

Creating a button for manual calculations

You don't have to use triggers. You may also add a button to your toolbar instead of using the triggers. That way the calculation is triggered only when the user wants it.

To create a button, follow these steps:

1. Do *Tools | Customize Window...*
2. Click the *Custom Commands* tab
3. Click *New* then *Browse* and locate TriggerCalc
4. Add the proper syntax at the end of the command line (like /[50]=[51]+50 below)
5. Enter a *Command Name* and *Description* Click *Apply* not *OK*



6. Then click the *Toolbars* tab
7. Select *Custom Commands* in the *Categories* dropdown list
8. Select and drag the command and drop it on the toolbar wherever you want.

That's it. When you click the new button, the calculation will be performed.

TRIGGERCALC PRO

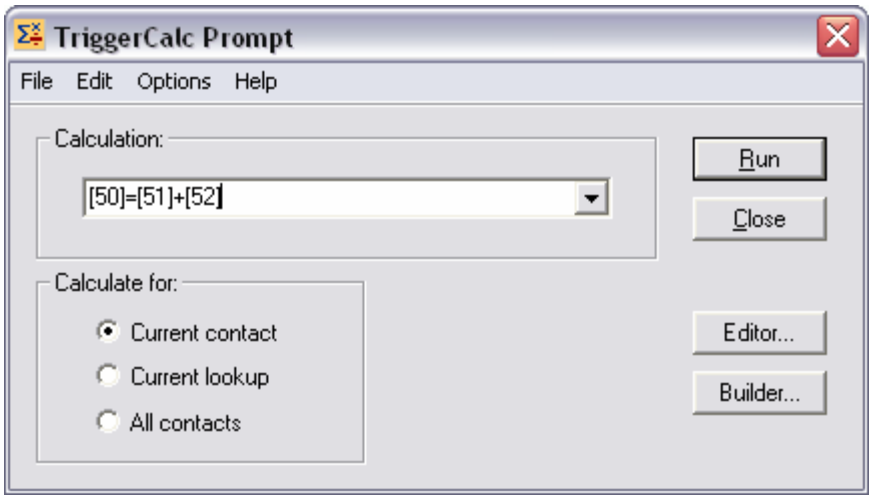
This section details features of the TriggerCalc Pro version only. If you are a registered user of TriggerCalc Standard version, you may upgrade to the Pro version (see Registering TriggerCalc above).

On top of the features of the Standard version, the Pro version features:

- A Prompt screen which allows to run a manual calculation for the current contact, the current lookup or all contacts;
- A Syntax editor with syntax coloring features to help you create and test your calculations;
- The ability to save your calculations in a file and to reference the file in your trigger syntax;
- A Syntax Builder to help you create trigger syntaxes.
- More than 30 additional functions

The prompt screen

In the Pro version, if you run the program without command line argument, you will get the *Prompt* screen.



The *TriggerCalc Prompt* screen allows to run a calculation for the current contact, current lookup or all contacts.

A typical use of the prompt screen would be the following: you already have created a trigger to automatically run a calculation when future changes happen to your contact fields and want to run the same calculation for all contacts of your database first. All you have to do is enter the calculation part of your trigger syntax in the *Calculation* field, select *All contact* and press *Run*.

You may also use the *Prompt* screen to regularly update your database by running a calculation the same way without even setting up a trigger.

If you created a file through the *Syntax Editor* (see below) for your trigger, you may load it through the *File* | *Open* menu and run it.

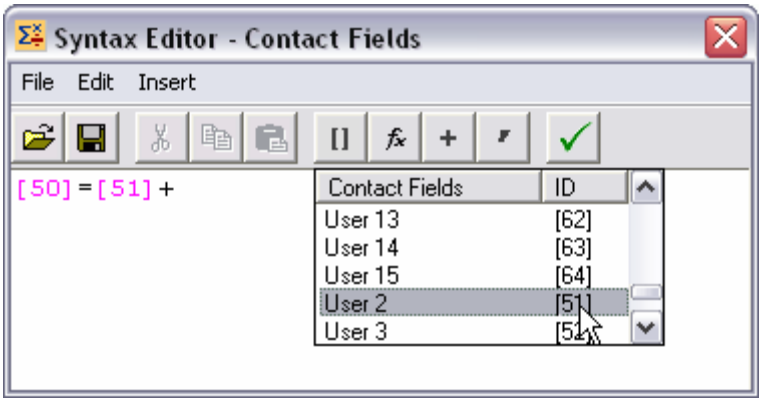
The Syntax editor

The Syntax Editor is a full featured syntax editor with automatic syntax coloring. To open the editor, click the *Editor…* button in the *Prompt* screen.

You may type directly in the window or use the different buttons set up to help you create your calculations.

Inserting fields

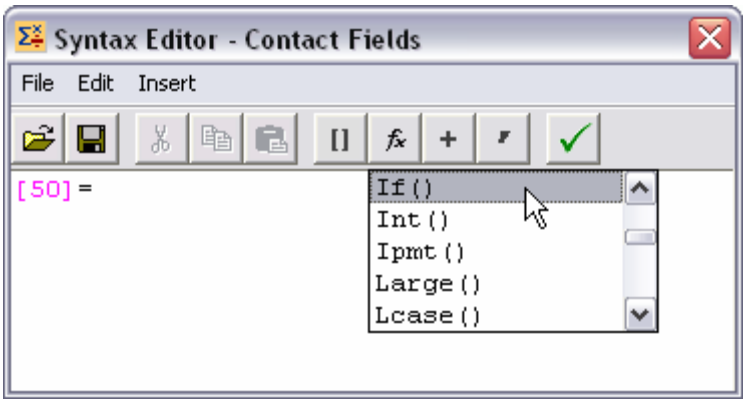
With the Syntax Editor, no need to run the actdiag.exe file to find out what your field IDs are. The *[ ]* button displays a list of all fields in your database and the field ID surrounded with brackets is automatically inserted when you click a field.



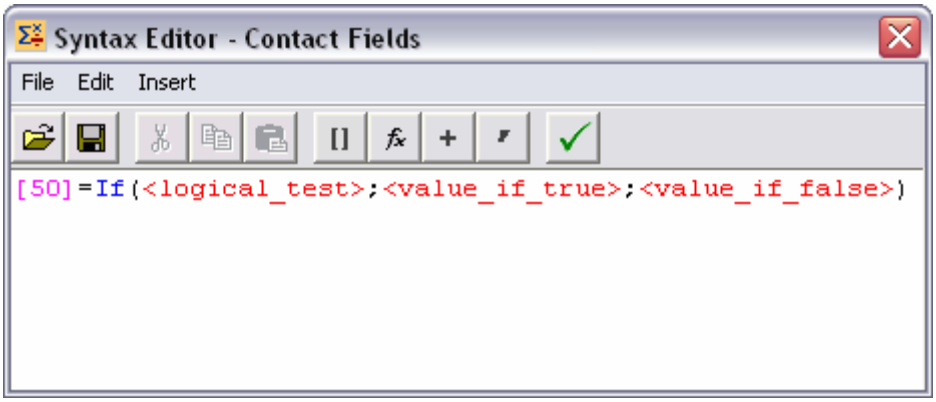
Note: if you are in the *Contact* screen when you launch TriggerCalc, the contact fields will appear in the field list. If you are in the *Group* screen, the group fields will appear.

Inserting functions

The *fx* button lists all the available functions so that you may quickly insert them.



Functions are inserted with arguments placeholders shown in red. Simply clicking once on a placeholder allows to select it, so that you may type the real argument (or insert a field, or another function).



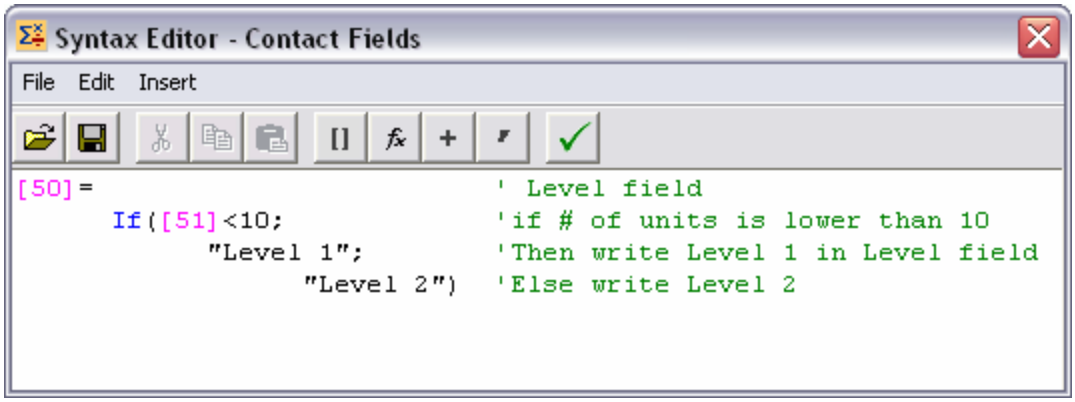
Inserting an operator

The + buttons lists all operators for easy insertion in your calculation.

Inserting comments

You may add comments using the quote (') button. Comments appear in green and start with the quote sign. Any character right of a quote sign and up to the end of a line (rather up to the next carriage return since a line may wrap) is considered a comment and ignored by TriggerCalc.

Because spaces and tabs are ignored and with the ability to add comments, you can make your calculations easily readible by anyone, as in the following example.



Saving your syntax in a file

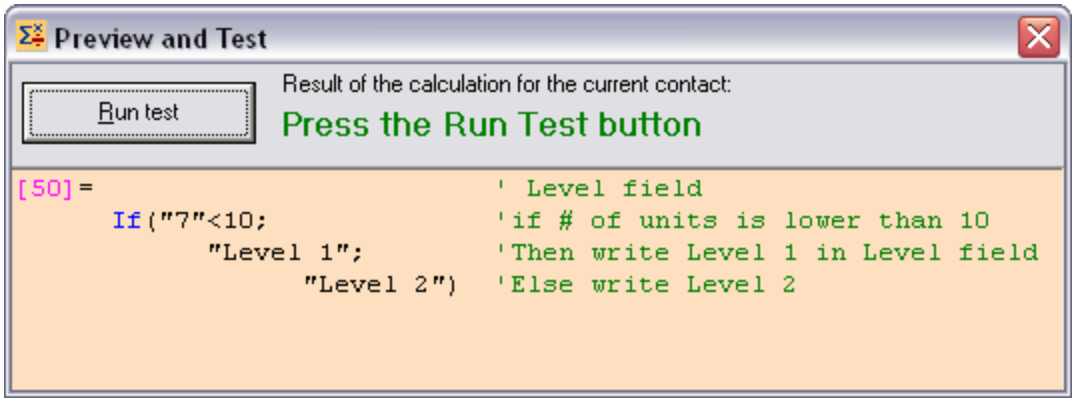
You may save your syntax in a .tcs (TriggerCalc Contact Syntax) or .tgs (TriggerCalc Group Syntax) file using the *File* | *Save* (or *Save As…)* menu. These files may be directly referenced in a trigger as explained in *Referencing a file in your trigger syntax* below.

Loading a file

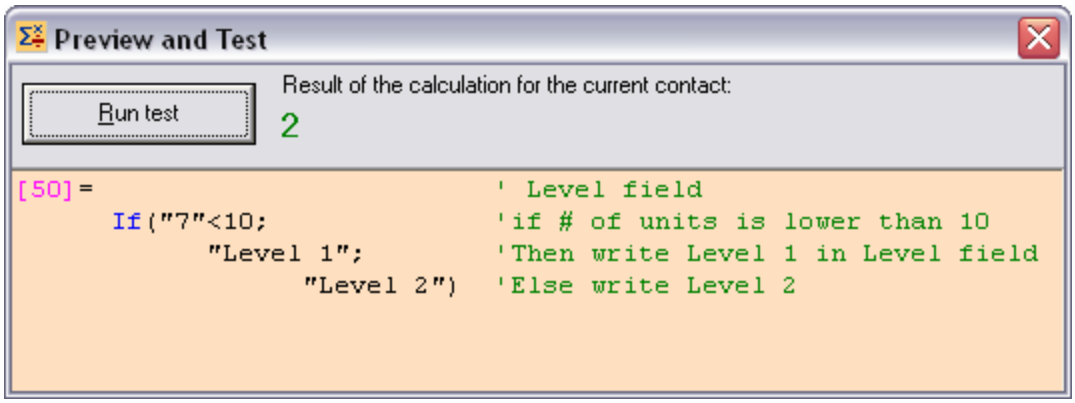
You may load a previously saved file using the *File* | *Open*… command.

Previewing and testing your syntax

Clicking the Preview and Test button (✓) opens the *Preview and Test* window.



In this window, the field IDs are automatically replaced with the values of the corresponding fields in the current contact. If you click the *Run test* button, the result of the calculation is displayed in green next to the button.



Note: to run further tests, you may change the values contained in the fields of the current contact, save the contact (if you want the new values to show in the window) and press the *Run test* button again to see the impact of your changes.

When you are done, close the *Preview and test* window by pressing the Windows close button (X) and you will be taken back to the *Syntax Editor* window.

Running multiple calculations

As explained in the TriggerCalc standard version, you may run multiple calculations at once. Calculations need to be separated with a backward slash and a forward slash (\ /). This applies to the *Syntax Editor* as well.

Closing the syntax editor

When you close the *Syntax Editor* window though the *File* | *Close* menu or the regular Windows close button, you will be prompted to save your work. Whether you save it or not, you will be taken back to the *Prompt* screen and the *Calculation* field will be automatically filled so that you may run your calculation by clicking the *Run* button.

If you saved your syntax, the field will show the proper syntax referencing the file you just saved (see *Referencing a file in your trigger syntax* below). If you did not save it, the field will show the first line of your syntax properly formatted to be run (ie. with spaces and comments removed).

Referencing a file in your trigger syntax

One of the most interesting festures of the Pro version is the ability to reference a file, in which your calculation syntax is stored, in the trigger syntax.

The proper way to reference a file is the following:
C:\Program Files\Exponenciel\TriggerCalc\TriggerCalc.exe /File: File_Path

In which *File_Path* is the path to the file.

Suppose that you saved your calculation in a file named *calc.tcs.* Remember that ACT! does not tolerate spaces in the arguments so make sure you replace any space with the ? character. In the case mentioned above the file path would therefore be:

/File:C:\Program?Files\Exponenciel\TriggerCalc\Calc.txt

If the argument file is in the same folder as the TriggerCalc executable, you may use the **AppPath** keyword, as in **AppPath\Calc.txt** so the actual whole command line would be:

C:\Program Files\Exponenciel\TriggerCalc\TriggerCalc.exe /File: AppPath\Calc.txt

NOTE: Referencing a text file may result in higher response time particularly if the file is not stored on your own machine.

Using a function inside another function

It is possible to use a function in one or more arguments of another function. But you then need to "escape" its semi-colons with a backslash.

For instance, you may create a function like this one:

[50] = If ([51] = "True" ; Avg ([51] \; [52] \; [53]) ; "n.a.")
The Avg function is an argument of the IF function. Note the presence of the backslashes which "escape" the second level of semi-colons.

You could have more than 1 argument using a function, as in:
[50] = If ([51] = "True" ; Max ([51] \; [52] \; [53]) ; Min ([51] \; [52] \; [53]))

If you want to add more levels of functions, you may do so by adding a backslash for each additional level, as in:

[50] = Min (if ([51]>=1000\; 1000\; if([51]>=500 \\; 500 \\; 0)) ; [52])

This example of multi-level calculation could be explained like this:

[50] will be the lowest value of field52 and a value defined like this:

- if field51 is higher than 1000 then the value is 1000;
- If field51 is higher than 500 but lower than 1000 then the value is 500;
- If field51 is lower than 500, the value is 0

Important note concerning the syntax editor: the syntax editor will not detect if you insert a function inside another function. So you have to escape the semi-colons with a backslash manually.

Using a formula to define the target field

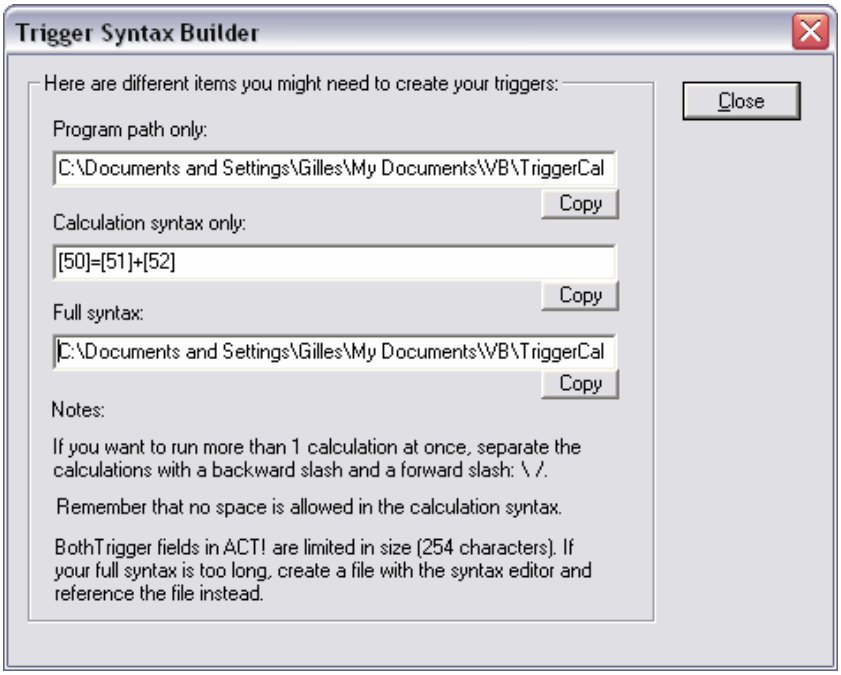
Starting with version 2.2, you may define the target field by a calculation. In this case though, make sure you use a double = sign to separate the calculation from the target field calculation.

Ex: imagine that field [50] may have 2 values. When it contains ValueA, you want today's date to be entered in field [51]. When it contains ValueB, you want today's date to be entered in field [52]:

```
[51+If([50]="ItemA";0;1)]==Now()
```

The Trigger Syntax builder

As seen in the TriggerCalc Standard version section of this manual and in *Referencing a file in your trigger syntax* above, there are several things not to forget when writing a trigger syntax, like inserting a space between the path and the calculation and avoiding spaces in the calculation or the file reference.



The *Trigger Syntax Builder* which you may display by clicking the *Builder…* button of the *Prompt* screen writes your syntax for you, based on the content of the *Calculation* field, so that you may simply copy and paste the syntax in ACT! trigger field.

Additional functions included in the Pro version

The Pro version of TriggerCalc also includes the following functions. Note the use of semi-colons to separate items when multiple arguments are needed.

If function

The syntax of the if function is the following:

- **If (logical_test;value_if_true;value_if_false)**
  Ex: if field [51]=1 then If([51]=1;"Yes";"No") returns "Yes"

The value_if_true and value_if_false arguments are optional.
Omit them if you don't want the value of the field to be changed.
Ex: if field [51]=0 then If([51]=1;"Yes";) will not modify the target field.
Ex: if field [51]=1 then If([51]=1;;"No") will not modify the target field.

If, on the contrary, you wish to blank out the target field, then use double-quotes.
Ex: if field [51]=0 then If([51]=1;"Yes";"") will blank out the target field.

Statistical functions

- **Min(value1;value2;...)** returns the smallest value (numeric values only: non-numeric values are ignored)
  Ex: if [51]=10 and [52]=20 then Min([51];[52])=10

- **MinC(value1;value2;...)** returns the smallest value (characters are accepted, the sort is based on the characters, so in this case 10 is higher than 20)
  Ex: if [51]="ProductA" and [52]="ProductB" then MinC([51];[52])="ProductA"

- **Max(value1;value2;...)** returns the highest value (numeric values only: non-numeric values are ignored)
  Ex: if [51]=10 and [52]=20 then Max ([51];[52])=20

- **MaxC(value1;value2;...)** returns the highest value (characters are accepted, the sort is based on the characters, so in this case 10 is higher than 20)
  Ex: if [51]="ProductA" and [52]="ProductB" then MaxC([51];[52])="ProductB"

- **Count(value1;value2;...)** counts the number of values that are numeric
  Ex: if [51]=1, [52]="X" and [53]=2 then Count([51];[52];[53])=2

- **CountC(value1;value2;...)** counts the number of values that are not blank
  Ex: if [51]=1, [52] is empty(blank) and [53]=2 then CountC([51];[52];[53])=2

- **CountBlank(value1;value2;...)** counts the number of values that are blank
  Ex: if [51]=1, [52] is empty(blank) and [53]=2 then CountBlank([51];[52];[53])=1

- **CountIf(logical_test; value1;value2;...)** counts the number of values that meet the given condition
  Ex: if [51]="Yes",[52]="Yes",[53]="No" then CountIf("="Yes";[51];[52];[53])=2

- **Avg(value1;value2;...)** calculates the arithmetic mean of the values (numeric values only. Blank and characters are ignored)
  Ex: if [51]=10, [52] is empty(blank) and [53]=20 then Avg([51];[52];[53])=15

- **Large(k,value1;value2;...)** returns the k-th largest value (numeric values only. Blank and characters are ignored)
  Ex: if [51]=10, [52]=12 and [53]=20 then Large(2;[51];[52];[53])=12

- **Small(k,value1;value2;...)** returns the k-th smallest value (numeric values only. Blank and characters are ignored)
  Ex: if [51]=10, [52]=12 and [53]=20 then Small(3;[51];[52];[53])=10

Mathematical functions

- The **cos, sin, tan, exp, atn** and **log** functions are supported.
- **Round(value;#_of_decimals**) returns a number rounded to a specified number of decimal places.
  Ex: if [50]="123.4875" then Round([50];3)="123.488"

String functions

- **Left(value;#_of_chars)** returns the first #_of_chars characters from the value, starting from the left
  Ex: if [50]="414-555-1212" then Left([50];3)="414"

- **Right(value;#_of_chars)** returns the first #_of_chars characters from the value, starting from the right
  Ex: if [50]="414-555-1212" then Left([50];8)= "555-1212"

- **Mid(value;start_pos;#_of_chars)** returns #_of_chars from the value, starting from the start_posth character. If #_of_chars is omitted, it returns all characters right of the pos.
  Ex: Ex: if [50]="414-555-1212" then mid([50];5;3)= "555"

Date functions

- **Age(date)** returns the age of a date (new in v.2.2.4)

- **DateDiff(interval;date1;date2;firstdayofweek;firstweekofyear)** returns the number of time intervals between two specified dates
  Ex: if [51]="12/12/2003" and [52]="12/15/2003" then datediff("d";[51];[52])=3

  You can use the **DateDiff** function to determine how many specified time intervals exist between two dates. For example, you might use **DateDiff** to calculate the number of days between two dates, or the number of weeks between today and the end of the year.

  To calculate the number of days between **date1** and **date2**, you can use either Day of year ("y") or Day ("d"). When **interval** is Weekday ("w"), **DateDiff** returns the number of weeks between the two dates. If **date1** falls on a Monday, **DateDiff** counts the number of Mondays until **date2**. It counts **date2** but not **date1**. If **interval** is Week ("ww"), however, the **DateDiff** function returns the number of calendar weeks between the two dates. It counts the number of Sundays between **date1** and **date2**. **DateDiff** counts **date2** if it falls on a Sunday; but it doesn't count **date1**, even if it does fall on a Sunday.

  If **date1** refers to a later point in time than **date2**, the **DateDiff** function returns a negative number.

Interval		**Firstdayofweek**		**firstweekofyear**	
Setting	**Descr.**	**Value**	**Descr.**	**Value**	**Descr.**
yyyy	Year	1	Sunday	1	Start with week in which Jan1 occurs
q	Quarter	2	Monday		
m	Month	3	Tuesday		
y	Day of year	4	Wednesday	2	Start with the 1st week that has at least 4 days in the new year
d	Day	5	Thursday		
w	Weekday	6	Friday		
ww	Week	7	Saturday	3	Start with first full week of the year
h	Hour				
n	Minute				
s	Second				

Firstdayofweek and **firstweekofyear** are optional. If omitted, a value of 1 is assumed.

- **DateAdd(interval;number;date)** returns a date to which a specified time interval has been added.
  Ex: if [51]="12/12/2003" then DateAdd("d";3;[51])="12/15/2003"

  You can use the **DateAdd** function to add or subtract a specified time interval from a date. For example, you can use **DateAdd** to calculate a date 30 days from today or a time 45 minutes from now.

  To add days to ***date***, you can use Day of Year ("y"), Day ("d"), or Weekday ("w").

The **DateAdd** function won't return an invalid date. The following example adds one month to January 31:

```
DateAdd("m", 1, "31-Jan-95")
```

In this case, **DateAdd** returns 28-Feb-95, not 31-Feb-95. If ***date*** is 31-Jan-96, it returns 29-Feb-96 because 1996 is a leap year.

Interval	
Setting	**Descr.**
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

- **DatePart(interval;date;firstdayofweek;firstweekofyear)** returns the specified part of a given date.
  Ex: if [51]="12/12/2003" then DatePart("y";[51])=346 (346th day of the year)

  You can use the **DatePart** function to evaluate a date and return a specific interval of time. For example, you might use **DatePart** to calculate the day of the week or the current hour.

  The ***firstdayofweek*** argument affects calculations that use the "w" and "ww" interval symbols.

Interval		**Firstdayofweek**		**firstweekofyear**	
Setting	**Descr.**	**Value**	**Descr.**	**Value**	**Descr.**
yyyy	Year	1	Sunday	1	Start with week in which Jan1 occurs
q	Quarter	2	Monday		
m	Month	3	Tuesday		
y	Day of year	4	Wednesday	2	Start with the 1st week that has at least 4 days in the new year
d	Day	5	Thursday		
w	Weekday	6	Friday		
ww	Week	7	Saturday	3	Start with first full week of the year
h	Hour				
n	Minute				
s	Second				

Firstdayofweek and **firstweekofyear** are optional. If omitted, a value of 1 is assumed.

Format function

The syntax of the format function is the following:

- **Format (expression;format)**
  Ex: if field [51]=2343.3 then Format([51];"###0.00") returns 2343.30

The format argument is a string of characters with one or more of the following characters:

String formatting:

@	A character should appear at this position. If there is no character for this position, a space is inserted. If there are more than one @, they are applied from right to left.
&	Same as @, except that no space is inserted if there is no character at this position.
!	Reverses the order in which @ and & are applied (ie. becomes from left to right).
<	Converts all characters to lowercase.
>	Converts all characters to uppercase.

Ex: Format("AbcD"; ">")= ABCD (similar to the UCase function)
Format("4142345678";"(@@@) @@@-@@@@")=(414) 234-5678
Format("2345678";"(@@@) @@@-@@@@")=( ) 234-5678
Format("2345678";"!(@@@) @@@-@@@@")=(234) 567-8

Number formatting:

	(blank)The digit is returned without formatting
0	A digit is supposed to appear at this position. If there is none, then 0 is displayed. If the format string contains more 0 than the number to be formatted, extra 0s are added in front or at the end.
#	Similar to 0 except that nothing is inserted if no digit is to appear
.	Combined with 0 or #, specifies the number of digits which should appear on each side of the decimal character.
%	Multiplies the number by 100 and adds a % sign.
,	Inserted in series of 0 or #, indicates the thousand separator. A double comma indicates that the number should be divided by 1000.
E-, E+	If the format string contains at least a 0 or #, converts the number to scientific notation.

Ex: Format(2343.3;"0000.00") = 02343.20
Format(2343.3;"$###,###.00") = $2,343.00
Format(45, "+###") = +45

Date and time formatting

c	Displays the date (as ddddd), the time (as ttttt) if one or both are specified.
d	Displays the day (1 to 31).
dd	Displays the day using 2 digits (01 to 31).
ddd	Displays the day using 3 characters (Sun to Sat).
dddd	Displays the full day (Sunday to Saturday).

ddddd	Displays the date in Short date format (your Windows settings).
dddddd	Displays the date in long time format (your Windows settings).
w	Displays the weekday (Sunday is 1, etc.).
ww	Displays the week #.
m	Displays the month (1 to 12).
mm	Displays the month using 2 digits (01 to 12)
mmm	Displays the month using 3 characters (Jan to Dec)
mmmm	Displays the full month (January to December)
q	Displays the quarter
y	Displays the day (1 to 366)
yy	Displays the year using 2 digits
yyyy	Displays the full year (1000 to 9999)
h	Displays the hours
n	Displays the minutes
s	Displays the seconds
ttttt	Displays the time in Time format (your Windows settings)

Ex: Format(01/15/2004;"dddd")=Thursday
Format(Now();"ttttt")="10:07:19 AM"

Financial functions

- **FV(rate;nper;pmt;pv;type)** returns the future value of an annuity based on periodic, fixed payments and a fixed interest rate.

  The **rate** and **nper** must be calculated using payment periods expressed in the same units. For example, if **rate** is calculated using months, **nper** must also be calculated using months.
  For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

  rate: interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
  nper: total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
  pmt: payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
  pv: (optional) present value (or lump sum) of a series of future payments. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make. If omitted, 0 is assumed.
  type: (optional) when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

- **IPmt(rate, per, nper, pv, fv, type)** returns the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

  The **rate** and **nper** arguments must be calculated using payment periods expressed in the same units. For example, if **rate** is calculated using months, **nper** must also be calculated using months.
  For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

  rate: interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
  per: payment period in the range 1 through **nper.**
  nper: total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
  pv: present value, or value today, of a series of future payments or receipts. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
  fv: (optional) future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
  type: (optional) when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

- **NPer(rate, pmt, pv, fv, type)** returns the number of periods for an annuity based on periodic, fixed payments and a fixed interest rate.

  For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

  rate: interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
  pmt: payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
  pv: present value, or value today, of a series of future payments or receipts. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
  fv: (optional) future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
  type: (optional) when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

- **Pmt(rate, nper, pv, fv, type)** returns the payment for an annuity based on periodic, fixed payments and a fixed interest rate.

The ***rate*** and ***nper*** arguments must be calculated using payment periods expressed in the same units. For example, if ***rate*** is calculated using months, ***nper*** must also be calculated using months.
For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

rate: interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
nper: total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
pv: present value, or value today, of a series of future payments or receipts. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv: (optional) future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
type: (optional) when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

- **PV(rate, nper, pmt, fv, type)** returns the present value of an annuity based on periodic, fixed payments to be paid in the future and a fixed interest rate.

  The **rate** and **nper** arguments must be calculated using payment periods expressed in the same units. For example, if **rate** is calculated using months, **nper** must also be calculated using months.
  For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

  rate: interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
  nper: total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
  pmt: payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
  fv: (optional) future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
  type: (optional) when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.

- **Rate(nper, pmt, pv, fv, type, guess)** returns the interest rate per period for an annuity.

  For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers. **Rate** is calculated by iteration. Starting with the value of ***guess***, **Rate** cycles through the calculation until the result is accurate to within 0.00001 percent. If **Rate** can't find a result after 20 tries, it fails. If your guess is 10 percent and **Rate** fails, try a different value for ***guess***.

  nper: total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
  pmt: payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
  pv: present value, or value today, of a series of future payments or receipts. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
  fv: (optional) future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.
  type: (optional) when payments are due. Use 0 if payments are due at the end of the payment period, or use 1 if payments are due at the beginning of the period. If omitted, 0 is assumed.
  guess: value you estimate will be returned by **Rate**. If omitted, ***guess*** is 0.1 (10 percent).

EXAMPLES OF CALCULATIONS

Calculating the age of a contact (Pro version)

If [50] is the date of birth field:

```
DateDiff("yyyy";[50];Now())
```

Converting a date into a string (Pro version)

If [50] is the date to convert:

```
Weekdayname(Weekday([50]))&",_"&Monthname(Month[50])&"_"&Day([50])&",_"&
Year([50])
```

Ex: 01/11/2004 is converted into Sunday, January 11, 2004

Capitalizing the first character of a field (Both versions)

If [50] is a string:

```
UCase(left([50];1))&Lcase(Mid([50];2))
```

Ex: "Send an Invoice" is converted to "Send an invoice"

Note: to capitalize the first letter of each word, see the Fcase function in the String functions of the standard version.

Simulating a checkbox (Pro version)

To simulate a checkbox, write a calculation that inserts an X if the field is blank and blanks the field if the field already has an X.

If [50] is the field you want to use for a checkbox: use this trigger syntax as an Entry trigger for field [50]:

```
[50]=if([50]<>"";"";"X")
```

Note: You probably would want to remove the checkbox field from the tab order (*Tools* | *Design layouts*) to prevent automatic triggering of the calculation when tabbing through fields.

Simulating radio buttons (Pro version)

Simulating radio buttons uses the same approach as simulating a checkbox. If [50] and [51] are the 2 fields that you want to work together, you want when you insert an X in [50] to blank out [51] and vice-versa.
For the trigger of field [50] and [51], use:

```
[50]=if([50]="";"X";"")/\[51]=if([50]="";"X";"")
```

Note: You probably would want to remove the 2 fields from the tab order (*Tools* | *Design layouts*) to prevent automatic triggering of the calculation when tabbing through fields.

Copying the content of a field to another field (Both versions)

This is really simple. If [50] is the field to copy to [51]:

```
[51]=[50]
```