Quattro Pro Formulas, Functions, and Macros

Charles M. Cork, III

March 27, 2018

Contents

Introduction

This online book aims to instruct on the use of Quattro Pro (**QP**) formulas, @functions, and macros, using examples compiled from personal work and online discussions in several venues. Some of these come from discussions on Corel newsgroups or its Office Community forums (**OC**), but most come from WordPerfect Universe forums (**WPU**). Much of the content of this compilation comes from the work of others, and I would call out for particular commendation Kenneth Hobson, Roy "Lemoto" Lewis, "Uli" (full name unknown, but a regular contributor to the Corel QP Newsgroups), Dr. David Seitman, and Jeff Barnes.

This book is for readers who have a basic knowledge of QP procedures and concepts.

The book is typeset using LaTeX. The simulations of QP spreadsheets in this book were originally provided by tables created in WordPerfect (**WP**). More recently, I've used a macro written in QP to generate the coding for those spreadsheets.

Suggestions welcome! This text is an evolving work in progress. Additions and corrections will be made in later versions. Suggestions for improvements should be e-mailed to me. Many users have already made suggestions for changes and corrections. The errors that remain in this text are exclusively my own.

Public Domain. This book is placed in the public domain. It is offered AS IS, without warranties of any sort. The recommendations in this book work for me, but I make no guarantees about whether how they will work for you. Use them at your own risk, and modify them at your own risk. I offer them at no charge, and completely for whatever value you may find in them. If you find this distribution to be of benefit, I simply ask that you remember the poor (Gal. 2:10) and consider giving to appropriate charitable organizations for their relief.

The most current versions of these documents will be found here.

The reason for this book

I wrote this book to compile what I have learned in wrestling with QP over the last several years, first for my own sake, but then for the benefit of others. I am writing it for myself in order to better systematize what I have discovered, and thus to better remember it. I am writing to share it with others just because that's a good thing to do.

Organizing and systematizing are necessary because QP's capabilities and the online explanations of those capabilities are disorganized. In some cases, the explanations are erroneous. Many examples exist of obsolete commands that are either completely dysfunctional or need to be re-purposed, but which still appear in the documentation as serving the same purposes they did when QP was a DOS-based program that displayed everything on a screen containing 24 rows of up to 80 characters. The program started well, but changes in the ownership, programmers, operating systems, the office suite setting, the competition, and the entire computing environment, have left it a patchwork of functions, some of which function better than others.

That said, QP's core functions work well, and in some cases, better than the competition. This book focuses on using those core functions productively.

To the best of my knowledge, there is no useful help manual for using QP's core functions. QP's help file can be searched, and it contains mostly reliable information, but it is disjointed into small articles that are sometimes inaccurate. Those articles use examples that are supposed to represent the grid of a spreadsheet, but don't, making the message at best unclear. Kenneth Hobson has placed online, among other QP related things, a book that extends QP in many interesting directions, but the present text seeks to focus on QP's core functions.

The help files for QP macros are unhelpfully split, so that basic information on macros appears in the main help file under "Automating tasks" (accessible by Help >Help Topics), but details are contained in a separate help file (accessible by Help >Macro Commands). The discussion of "Using macros" in the main help file is not in an order conducive to learning for a beginning user. In fact, I can't deduce what order it is in. The beginner should probably start with the final entry there, "Reference: Using macros" and skip the parts in it dealing with VBA. Then jump back a few entries to "Playing macros" to learn how to start macros running.

The best discussion I've seen for introducing one to @functions and macros actually comes from the help manual for Lotus 1-2-3, on which QP's macros were initially based. That manual is online. The discussion of macros beginning on page 91 is enlightening in many ways for QP macros, though obviously, Lotus and QP do things a bit differently, and things have changed in the last 24 years, so it is not 100% reliable for use in QP.

Conventions

The following conventions are used throughout this book:

- Formulas, functions, and macro commands will be distinguished in the main text by `this font`.
- @Function names are in uppercase letters, but you can use uppercase or lowercase letters. QP renders them in uppercase.
- Macro commands are enclosed in braces, e.g., `{}`. You may use any combination of uppercase or lowercase letters. I combine upper and lower case for readability.
- Arguments for @functions and macros are in italics, mostly lowercase. Example: `@TIME(`*`hour,minutes,seconds`*`)`
- Optional arguments for @functions and macros are in angle brackets <>. Example: `@LASTCELLVALUE(`*`block,<type>`*`)`
- Keyboard keys and on-screen buttons are enclosed in simple brackets. Examples: `[Enter]` and `[OK]`. Control keys to be pushed simultaneously are shown with a plus sign, as in `[Shift+F12]`.
- Menu selections are noted by separating higher-level from lower-level menu items with the right angle bracket. Example: Edit > Clear > Format.

I will attempt terminological consistency with related terms:

- Over time, what we refer to as *text* was first referred to as a *label* and then as *string*. I will use *text* to refer to it generically, and I'll use *string* to refer to specific, ascertainable sets of text characters. My name *Charlie* is text, and the seven letters in it form a particular string of text.
- I will use *block* as the most general term to refer to a set of one or more contiguous cells definable by coordinates such as A1..Z10. I will use *table* to refer to a block that has some unity of purpose, and a *database* as a table with the specific purpose of structuring the storage and retrieval of data.

I will use tables as illustrations of QP screens. In these, I will illustrate the technique of entering a formula in one cell, and then copying it to a block of cells, by highlighting the first cell in cyan and the target block in yellow. Where the target block includes the first cell, it will remain in cyan. It will look like Table 1.

Table 1: Indicating copy and paste by colors

	A	B	C	D	E	F	G	H
1	🟦		🟦	🟨	🟨	🟨	🟨	🟨
2	🟨							
3	🟨	🟦	🟨	🟨				
4	🟨	🟦	🟨	🟨				
5	🟨	🟦	🟨	🟨				
6	🟨				🟦	🟨	🟨	🟨
7	🟨				🟨	🟨	🟨	🟨
8	🟨				🟨	🟨	🟨	🟨

Formula in A1 is copied to A2..A8.
Formula in C1 is copied to D1..H1.
Formulas in B3..B5 are copied to the right, either separately, or collectively by selecting B3..B5, copying (Ctrl+C), and pasting to C3..D3, which fills C3..D5.
Formula in E6 is copied to E6..H8.

Displays of grids will usually show the return values of formulas, rather than the formulas themselves. The formulas/functions themselves will typically appear in the main text or in the block below the grid, but sometimes, the formula/function in one cell will be typed into another cell with arrows (← → ↑ or ↓) indicating the relationship.

I will also use highlighting on complex formulas, either to show how they are progressively constructed or to show different components.

Because some formulas and macros are too long to fit in a cell on the screen, I have broken them onto separate lines. As long as they are in the same cell or indented block, the reader may assume that they are intended to run together in a single cell.

QP quirks, generally

Most QP functions and macros work well and precisely as described in its help materials, but unfortunately, some do not. Particular @functions and macros will be discussed in the chapters on each. Here, I note some problems that exist across the system. These are largely a problem of factors mentioned at the start of this introduction.

Named blocks

Many spreadsheet users name a cell or block of cells using `[Ctrl+F3]` (right-clicking on the cell displays a pop-up menu that also allows you to name the cell), and many guides use such named cells in lieu of referencing them by coordinates. I have been persuaded to avoid using named blocks as much as possible, and therefore the functions and macros in this book will refer to blocks of cells by their addresses, not their block names.

In particular, the names can oddly "drift" so that the name for block A will suddenly *appear* in functions to refer to block B, but the functions generally work correctly. And more seriously, deleting a named block sometimes causes document corruption. The user who is interested in pursuing the quirks of block names in QP should consult these threads: WPU 33831, WPU 26708, WPU 24258, WPU 19272, WPU 16099.

I do not entirely avoid named ranges. They can be useful for expediting the launching of macros, as noted below at page 116.

Many of them can be replaced with `@@` functions. See page 38, below.

QP Dialogs and Form Controls

At some stage, QP developers added the capacity to place various "form controls" (Insert > Form Control) on the sheet. Some of them are fine,[1] but some have been known to cause corruption of the document. Therefore, instead of them, I use the PerfectScript forms. To pursue this problem, see WPU 29987 and WPU 36250.

[1]I use the command button regularly. The listbox and combobox seem to work reliably, if slowly.

Part I

Quattro Pro Formulas

Chapter 1

Basic Formulas

At the most basic level, when you place a mathematical formula in a cell, QP will automatically calculate the numeric result, as shown by formulas in column A in Table 1.1.

Table 1.1: Basic formulas

	A	B
1	12	⇐ What 7+5 in A1 returns
2	-5	⇐ What 7-12 in A2 returns
3	21	⇐ What 3*7 in A3 returns
4	3.1428571	⇐ What 22/7 in A4 returns
5	27	⇐ What 3ˆ3 (3 cubed) in A5 returns
6	9	⇐ What 81ˆ(1/2) (square root of 81) in A6 returns
7	2	⇐ What (3*4)/6 in A7 returns

Table 1.2: Using data in cells as variables

	A	B
1	12	
2	12	⇐ What +A1 in A2 returns
3	21	⇐ What +A1+9 in A3 returns
4	3	⇐ What +A1/4 in A4 returns
5	20	⇐ What (A1/6)*10 in A5 returns
6	144	⇐ What +A1*A2 in A6 returns
7	1	⇐ What +A1= 12 in A7 returns (namely, “true”)
8	1	⇐ What +A1>10 in A8 returns (“true”)
9	0	⇐ What +A1<10 in A9 returns (“false”)

Reference to data in other cells. At the next basic level, illustrated in Table 1.2, QP formulas return the results of such operations based on the content of cells referred to in the formula. The content of the referenced cell is a variable; changing its content changes the results of the formula. Note the + sign in these examples. If you simply enter A1 into A2, QP stores A1 as text. Some sign is necessary to tell QP that you want A1 to refer to the content of a cell rather than to start a text string. Any of these would refer to A1 without altering its meaning: `+A1`, `=A1`, or `(A1)`. More than one cell can be referenced in a formula, as shown by the formula in A6.

When the user simply changes the number in A1, QP recalculates all formulas, as shown in Table 1.3.

Table 1.3: Recalculation of formulas

	A	B
1	30	
2	30	⇐ What +A1 in A2 *now* returns
3	39	⇐ What +A1+9 in A3 returns
4	7.5	⇐ What +A1/4 in A4 returns
5	50	⇐ What (A1/6)*10 in A5 returns
6	900	⇐ What +A1*A2 in A6 returns

One of the most useful features of a spreadsheet is the ability to compose a formula that acts on one cell, and then copy the formula to other cells in a way that adjusts the formula automatically. In Table 1.4, the user enters the number 1 in cell A1, the formula `+A1+1` in cell A2 and the formula `+A1*7` in cell C1. The user then copies cell A2, selects the block A3..A6, and pastes. The numbers highlighted in yellow appear, and the formulas are adjusted as displayed in B3..B6. The same is true in the C column. Copying cell C1, selecting the block C2..C6, and pasting, yields the numbers highlighted in yellow, and the formulas in those cells are adjusted as shown in D2..D6.

Table 1.4: Adjustments to formulas copied to multiple cells

	A	B	C	D
1	1		7	⇐ +A1*7
2	2	⇐ +A1+1	14	D1 pastes in D2 as +A2*7
3	3	A2 pastes in A3 as +A2+1	21	D1 pastes in D3 as +A3*7
4	4	A2 pastes in A4 as +A3+1	28	D1 pastes in D4 as +A4*7
5	5	A2 pastes in A5 as +A4+1	35	D1 pastes in D5 as +A5*7
6	6	A2 pastes in A6 as +A5+1	42	D1 pastes in D6 as +A6*7

References beyond the current sheet. Note that the formulas can refer to cells on a different sheet or an entirely different notebook. To refer to a cell on a different sheet, precede the cell reference with the sheet letter or name, e.g.,

`D:A1` or `Data:A1`

To refer to a cell in a different notebook, precede the sheet with the name of the file in brackets, e.g.,

`[MyNotebook.qpw]Data:A1` .

Relative v. absolute addresses

The effects just noted occur because the references to A1 in both of the initial formulas are **relative references**, the default condition. Thus, for example:

- If a cell containing a formula that references A1 is copied, say, four rows down from the original cell, the reference to A1 will be adjusted to A5.
- If the cell is copied, say, four columns to the right from the original cell, the reference to A1 will be adjusted to E1.

- If the cell is copied both four rows down and four columns to the right from the original cell, the reference to A1 will be adjusted to E5.
- The same relative changes occur when the cell is copied to the same sheet in another notebook.
- The same relative changes occur for each cell if more than one cell is referenced in the initial formula.

Sometimes, however, you do not want the copying and pasting to do this, as where you want to apply the percentage rate in a single cell to a range of numbers. In that case, you want to make an **absolute reference** to that cell. Absolute references are marked by adding the dollar sign to each component of the cell. In Table 1.5, we want to apply the rate in A2 to the numbers in B1..D1. The incorrect relative reference, `+A2*B1`, is created in B2 and then copied to C2..D2. The formula in C2 multiplies C1 not by A2, but by B2. The formula in D2 multiplied D1 not by A2, but by C2. The correct absolute reference, `+$A$2*B1`, is created in B3, and then copied to C3..D3. Formulas are adjusted as desired: B1, C1, and D1 are multiplied solely by A2.

Table 1.5: Incorrect and correct formulas for pasting

	A	B	C	D
1	Rate	100	500	1000
2	0.25	25	12500	12500000
3		25	125	250

Wrong in B2: +A2*B1.
Right in B3: +$A$2*B1

Sometimes, we may want to use **partially absolute references**, as where we want to keep the row or the column constant. In Table 1.6, we want to multiply the numbers at the top in B1..D1 by the rates in the left column at A2..A4, by constructing one formula in B2 and pasting it to the remaining cells. In this case, put the formula `+$A2*B$1` in cell B2, copy it, select B2..D4, and paste, to yield the correct results. The `$A` in the formula assures that the rate will always be found in the A column. The `$1` in the formula assumes that the number to be multiplied will always come from the first row. But the row of the rate and the column of the other number will automatically adjust relative to the initial cell.

Table 1.6: Using a partially absolute formula to paste to entire block

	A	B	C	D
1		100	500	1000
2	0.25	25	125	250
3	0.35	35	175	350
4	0.45	45	225	450

+$A2*B$1 in B2, copied to B2..D4

A shortcut method of making absolute or partially absolute references involves using **the** `[F4]` **key**. While typing a formula referencing cell A2, for instance, while the insertion point is in or just after A2, pressing `[F4]` will convert A2 into `$A:$A$2`. Pressing `[F4]` multiple times will cycle through seven different absolute or partially absolute variations, that are detailed in the QP help file under “Working with formulas and functions.”

Relative ("R1C1") v. normal reference style

This seems to be the appropriate place to introduce, and distinguish, another method of addressing cells that is also "relative" in a different sense. Some of the QP reference materials refer to it as a "relative reference," but it is not like the relative addresses just discussed. Those refer to cells using, and as relative to, the spreadsheet's framework of columns A..end and rows 1..end, which I'll refer to as the "normal reference style". The relative reference style refers to cells by indicating their position relative to the cell that contains the formula, as illustrated in Table 1.7. Contrary to the implication of "R1C1", which is used to describe this reference style in some of the reference materials, the relative reference style begins with `c` (indicating a column offset), a number in parentheses of columns offset, `r` (indicating a row offset), and a number in parentheses of rows offset.

Table 1.7: Relative reference examples

c(0)r(0)	refers to the cell in which the formula is placed
c(-1)r(0)	refers to the cell to the left
c(0)r(-1)	refers to the cell above
c(1)r(0)	refers to the cell to the right
c(0)r(1)	refers to the cell below

When these references are actually placed in a cell, QP converts (translates) them into coordinates on the grid in the normal reference style. They function in a cell just as normal references do, as shown in Table 1.8.

Table 1.8: Relative references in formulas

+c(-1)r(0)*12	multiplies the cell to the left by 12
@SUM(c(0)r(-3)..c(0)r(-1))	sums the three cells above the function
+$c(-1)r(0)*c(0)$r(-1)	multiplies the cell to the left by the cell above. The $ signs in this formula will remain when QP converts it to the normal reference style, and thus will create partially absolute addresses as in the example above.

Since they are more complex than normal references and since QP translates them anyway, this type of reference is not useful for manually entering formulas. However, it is quite convenient when constructing formulas to be put into cells by automated means, particularly with macros. (Note, however, that in macros, additional brackets may be required. See page 106.)

Combining text: Text (string) formulas

In addition to mathematical operations, formulas can return text and can be used to combine different text strings. For instance, the formula `+"January"&" "&"2015"` will return the text "January 2015". The formula begins with + (or =), and each component part is placed in double-quote marks, and they are joined by with ampersand &, *not* a plus sign (which is common in other programming languages).

Note that each component part is expected to be text, and therefore the "2015" in this example has to be text, *not* the number 2015. To combine text and numbers this way, the numbers must first be converted to text by means of a function such as `@STRING`, `@DOLLAR`, `@FIXED`, or `@FRACTION`.

Alternatively, the function `@CONCATENATE` combines text and numbers; it is discussed more fully below at page 40.

Combining text and the content of cells

String formulas become more useful when they combine the contents of various cells. In Table 1.9, *January* is entered in cell A1, *2015* is entered as text in cell B1 but as a number in C1 and D1. The formula +A1 is entered in cell A2. The formula `+A2&" "&B1` is entered in cell B2, and it correctly joins these two items of text with a space between them. Putting the same format, `+A2&" "+C1`, into C2 returns an ERR, however, because one cannot combine text and numbers with this formula. The `@CONCATENATE` formula works in D2 because that function allows the combination of text and numbers.

Table 1.9: Combining Text and Numbers

	A	B	C	D
1	January	2015	2015	2015
2	January	January 2015	ERR	January 2015

+A1 in A2.
+A2&" "&B1 in B2.
+A2&" "&C1 in C2.
@CONCATENATE(A2," ",D1) in D2.

The user should note that all formulas return *the values* of the cells they reference, but they do not automatically include *the formatting*. Thus, if a number in cell A1 is formatted as a date, as currency, or in any other special way, or if cell A1 has a particular font or other attribute, the formula `+A1` in another cell will return the number, but only as the other cell is formatted, not as A1 is formatted. To reproduce the formatting of cell A1 in the other cell, one must take some other step, such as copying and pasting the source cell to the target cell.

How to set up a database showing interest compounding monthly

The task is to demonstrate how interest accumulates on a specific sum ($100) at a specific annual rate (18%) over 48 monthly periods. Table 1.10 shows one way to set it up.

The informational data are set up in A1..A50 and B2..D2, as shown. Put the interest rate (0.18) in cell B1. Put the starting principal (100) in B3. Format B3..D50 as currency. Now for the formulas:

1. Place `+B3*($A:$B$1/12)` in C3. This calculates the monthly interest on the principle by multiplying the current balance by the yearly interest rate, divided by 12 to get the monthly interest rate. The absolute reference to B1 allows the rate in B1 to copy to all the lower cells in the C column.

Table 1.10: Compound interest table

	A	B	C	D
1	Ann. Int	0.18		
2	Month	Beg Balance	Int	End Balance
3	1	$100.00	$1.50	$101.50
4	2	$101.50	$1.52	$103.02
5	3	$103.02	$1.55	$104.57
6	4	$104.57	$1.57	$106.14
7	5	$106.14	$1.59	$107.73
8	6	$107.73	$1.62	$109.34

2. Place `+B3+C3` in D3 to get the balance at the end of the month.
3. Place `+D3` in B4 to get the starting balance of the next month.
4. Copy the formulas in B4, C3, and D3, and paste them into the cells below each one, parallel with the months in the A column.

See discussion in WPU 34782.

How to set monthly installments so that interest payments are equal for each payment

As noted in WPU 37509, most loans are set up in the recognition that the borrower should be paying more interest at the beginning of the loan period, but that the interest should decline as the principal balance declines until the debt is paid. QP has financial functions for these standard calculations, but a QP user wants to structure it differently. The goal will be to set the monthly payment so that it is composed of an equal principal payment and an equal interest payment for the entire period of the loan. Dividing the principal by the number of payments is easy. The user wants to calculate the interest in terms of the average principal balance (half of the principal plus one monthly payment of principal, apparently since the debt is created immediately, but payment is deferred by one month).

Working with these guidelines, I set up the spreadsheet this way. The principal debt is placed in B1, the rate in B2, and the number of monthly payments in B3. Corresponding labels appear in A1..A3. The formulas are:

Monthly principal	`(B1/B3)`
Monthly interest	`((B1+(B1/B3))/2)*B2/12`
Total monthly payment	`(B1/B3)+(((B1+(B1/B3))/2)*B2/12)`
Total payments over the entire period	`((B1/B3)+(((B1+(B1/B3))/2)*B2/12))*B3`

Good luck to the user!

How to type the date of every Friday in a year in a column

1. Type the first Friday date into the topmost cell (say, A1).

   Note that QP will convert a date typed as 1/2/15 into a date number (e.g., 42006), but apply the default date formatting to it, so that the number is displayed according to the chosen default format (in my case, 01/02/15).

2. Type the formula +A1+7 into the next cell below (say, A2).
3. Copy that formula into an adequate number of cells below (say, A3..A52).

   Note that the numbers in cells A2..A52 will not automatically appear as dates. You will need to apply some date formatting to those cells in order to display them as dates.

Chapter 2

Block (Array) Formulas

QP has a possibly unique shorthand way for making a formula that operates on a block (or array) of data. The formula can populate another block of data with calculations using the same formula on each cell in the block, or all of those calculations can be combined using a QP function. QP often automatically wraps the formula in an `@ARRAY` function

Uses of block formulas

Basic block formulas, filling cells with calculations

Table 2.1 illustrates QP's automatic expansion. It contains a simple block of data in cells A1..A10. Different block formulas are entered in cells B1..G1 which act on A1..A10. When entered, QP fills in all the values below.

Table 2.1: Automatic expansion of block formulas

	A	B	C	D	E	F	G
1	1	6	-1	2	0.5	0	0
2	2	7	0	4	1	0	0
3	3	8	1	6	1.5	0	0
4	4	9	2	8	2	0	0
5	5	10	3	10	2.5	1	0
6	6	11	4	12	3	0	1
7	7	12	5	14	3.5	0	1
8	8	13	6	16	4	0	1
9	9	14	7	18	4.5	0	1
10	10	15	8	20	5	0	1

B1 contains +A1..A10+5. C1 contains +A1..A10-2. D1 contains +A1..A10*2.
E1 contains +A1..A10/2. F1 contains +A1..A10=5. G1 contains +A1..A10>5.

The formula in B1 expands into B1..B10, adding 5 to each number in the block A1..A10. The same is true of the mathematical operations in C1, D1, and E1 and the logical operations in F1 and G1.

Condition-testing block formulas, filling cells with 1 or 0

The formulas in F1 and G1 call for special comment. They test for conditions that are either true or false, which return the values 1 or 0, respectively. F1 asks which cells in A1..A10 are equal to 5, and it returns 1 (true) for only one of them, as reflected in cell F5, and 0 (false) in the rest. G1 asks which cells in A1..A10 are greater than 5, and it returns 1 (true) in cells G6..G10, false in the rest. Testing for the truth of conditions will allow the user to extract selected information from the database in one single formula, as developed below.

Because of testing for conditions, block formulas can work on strings of text as well as numbers. Moreover, different tests can be combined, as seen in the following illustration.

Table 2.2 uses a standard way of representing quarterly data for a two-year period in the A and B columns. Random numbers appear in the C column. Block formulas are in D1 and E1. When the formulas in D1 and E1 are entered, QP fills out the rest of the D and E columns.

Table 2.2: Using block formulas to extract specific data

	A	B	C	D	E
1	2014	Q1	113	1	113
2	2014	Q2	134	0	0
3	2014	Q3	160	0	0
4	2014	Q4	156	0	0
5	2015	Q1	163	1	163
6	2015	Q2	157	0	0
7	2015	Q3	170	0	0
8	2015	Q4	173	0	0

D1 contains +B1..B8="Q1". E1 contains (C1..C8)*(B1..B8="Q1").

The formula in D1 tests for a string condition. It returns 1 (true) when the entry in the B column is the string/text "Q1", and 0 (false) otherwise). The text must be in double-quotes.

The formula in E1 shows how to combine block formulas to single out desired data. It returns the value in the C column in cases where the B column contains "Q1", and otherwise it returns 0. It multiplies the values in column C by the 1s and 0s that test whether "Q1" is in the B column.

Note that more than two formulas can be combined by additional multiplication operations. Thus, in this example, to get the results of Q1 in 2015, the formula might be:

```
(C1..C8)*(B1..B8="Q1")*(A1..A8=2015)
```

Applying @-functions to block formulas

Block formulas become even more powerful and useful when combined with @-functions such as `@SUM` and `@MAX`. This will be more properly introduced in the section on @-functions below, but for now, suffice it to note that `@SUM(`*`block`*`)` returns the sum of all numeric values in the block, and `@MAX(`*`block`*`)` returns the greatest value in the block. Such @-functions can perform all of the calculations of a block formula in a single cell, without needing to populate a table of cells.

Hence, in the last example, the function `@SUM(D1..D8)` returns the value of 2, and `@SUM(E1..E8)` returns the value of 276. This should be interpreted as telling us that there are two Q1 entries in this small database, and the sum of values for Q1 in this database is 276. Here‘s where QP gets very convenient, allowing the user to get the same information in one function without setting up D and E columns.

`@SUM(B1..B8="Q1")` returns the value 2 without setting up the D column, telling us that there are two entries in the database representing Q1.

`@SUM((C1..C8)*(B1..B8="Q1"))` returns 276 without setting up D or E columns, telling us that the sum of numbers for Q1 is 276.

`@MAX((C1..C8)*(B1..B8="Q1"))` returns 163 likewise, telling us that the largest number associated with Q1 is 163.

Two cautions are in order, however. First, if all of the numbers that satisfy the tests in a block formula are negative, the formula may return 0 instead of the maximum negative number, because the rows that do not satisfy the test are evaluated as 0. Second, `@MIN` does not behave as a mirror image of `@MAX` for the same reason: rows that do not satisfy the test evaluate as 0, and 0 may be less than the minimum number that satisfies the test. With careful crafting, the block formula can deal with this problem.

Quirk with QP14

As noted in WPU 37034, if a cell is blank in at least one block of a block formula, QP14 may return an ERR. This problem does not seem to appear in other versions of QP.

Concluding comments

In my judgment, block formulas are a more useful way to extract information from a database than `@SUMIF` and `@COUNTIF` functions that yield similar results. A more extensive explanation of block formulas is at WPU 10209.

How to count or sum all of the numbers in a column that are less than a certain number

If one seeks to count all of the entries in, say, A1..A100 that are less than 50, the formula would be `@SUM(A1..A100<50)`. This function totals an array of 100 tests that each yield a value of 1 or 0, depending on whether the item in A1..A100 is less than 50.

If one seeks the sum of all the same entries, we need to multiply that array of 1s and 0s by the values themselves: `@SUM((A1..A100<50)*(A1..A100))`

How to count all entries in a column that equal the data in this/each row

If one seeks to count all of the entries in, say, A1..A100 that equal the data in, say, A10, the formula would be `@SUM(A1..A100=A10))`.

To change this formula so that it could be parallel to the entire column, and thus reflect how many duplicates appear for every item of data in the column, make the block absolute like this, `@SUM($A$1..$A$100=A1))`, place that formula in B1 or some other parallel with A1, and then copy it down in parallel with A1..A100.

How to count or sum all cells that meet two conditions

In this sample (Table 2.3), we will first count the number of rows that are Home and Even, Home and Odd, Away and Even, or Away and Odd.

Table 2.3: Sample data for block formulas

	A	B	C
1	Home	Even	113
2	Home	Odd	134
3	Away	Even	160
4	Home	Odd	156
5	Away	Even	163
6	Away	Odd	157
7	Away	Even	170
8	Home	Odd	173

`@SUM((A1..A8="Home")*(B1..B8="Even"))` .... returns 1 match
`@SUM((A1..A8="Home")*(B1..B8="Odd"))` .... returns 3 matches
`@SUM((A1..A8="Away")*(B1..B8="Even"))` .... returns 3 matches
`@SUM((A1..A8="Away")*(B1..B8="Odd"))` .... returns 1 match

Now we can sum the numbers in the C column for each of these categories:
`@SUM((A1..A8="Home")*(B1..B8="Even")*(C1..C8))` .... returns 113
`@SUM((A1..A8="Home")*(B1..B8="Odd")*(C1..C8))` .... returns 463
`@SUM((A1..A8="Away")*(B1..B8="Even")*(C1..C8))` .... returns 493
`@SUM((A1..A8="Away")*(B1..B8="Odd")*(C1..C8))` .... returns 157

How to add a running total of all cells, or of only cells meeting a condition

Table 2.4 illustrates how to use the data in Table 2.3 to build a formula that keeps running totals of numbers in the C column. In the D column, there will be a running total of all numbers, and in the E column, there will be a running total of all numbers that are paired with “Home” in the A column.

In column D, one could enter `+C1` in D1 and `+D1+C2` in D2, followed by copying D2 into D3..D8. It is, however, better to place this formula in D1 and copy it to D2..D8:

```
@SUM($C$1..C1)
```

This formula always sums the range running from C1, which is set by absolute reference, to the cell in column C next to the copied formula in column D.

In column E, we build a formula that keeps a running total of numbers in the C column that parallel “Home” in the A column. The way to do this is to place the following formula in E1 and copy it into E2..E8:

```
@SUM(($A$1..A1="Home")*($C$1..C1))
```

Table 2.4: Running Totals

	A	B	C	D	E
1	Home	Even	113	113	113
2	Home	Odd	134	247	247
3	Away	Even	160	407	247
4	Home	Odd	156	563	403
5	Away	Even	163	726	403
6	Away	Odd	157	883	403
7	Away	Even	170	1053	403
8	Home	Odd	173	1226	576

D1 = @SUM($C$1..C1)
E1 = @SUM(($A$1..A1="Home")*($C$1..C1))

Another more complex version of this problem appears in WPU 37596. This creates a third column of functions that display subtotals of numbers in the first column, but those functions display the subtotal only if a condition is met in the second column; otherwise it returns a blank.

How to sum the numbers that come within a date range

Using the data in Table 2.5, the task is to sum the numbers in column B corresponding to dates in the ranges from 1/2/12 and 1/8/12, inclusive.

Table 2.5: Sample date-range data

	A	B
1	12/30/11	50
2	12/31/11	25
3	1/1/12	35
4	1/2/12	40
5	1/3/12	31
6	1/4/12	15
7	1/5/12	20
8	1/6/12	16
9	1/7/12	42
10	1/8/12	19
11	1/9/12	17
12	1/10/12	21

Method 1 (Simple Sum)

If the user knows where those dates are already, the simplest formula is `@SUM(B4..B10)`, which returns 183.

Method 2 (Block Formulas)

The method that applies whether the dates in the A column are contiguous or not involves a block formula. Since we happen to know that the start and end dates are in cells A4 and A10, this formula returns the correct sum (183).

```
@SUM((B1..B12)*(A1..A12>=A4)*(A1..A12<=A10))
```

Using the `@DATE` function, this same function could be written without putting the start and end dates into other cells in this way:

```
@SUM((B1..B12)*(A1..A12>=@DATE(2012,1,2))
*(A1..A12<=@DATE(2012,1,8)))
```

Table 2.6 shows a common, useful extension of this technique, in which you put the start and end dates into separate cells (here D1 and D2) and the formula below (D3). Then the user can type different dates into D1 and D2, and the result in D3 will be updated.

Table 2.6: Formula using chosen starting and ending dates

	C	D
1	Start Date	01/02/12
2	End Date	01/08/12
3	Sum:	183

The formula to put into D3 is this:

```
@SUM((B1..B12)*(A1..A12>=D1)*(A1..A12<=D2))
```

A variation of this problem appears in WPU 37627. There, the function will return the sum of entries between a starting date and an ending date if both of those dates are in the date column of a database, but if not, it returns a message calling for action.

A more complex version of this problem appears in WPU 36767. It creates a running total of numbers in a date range that changes from row to row, where one of the dates is variable, determined by conditions in another column, and where the sum is only to be displayed under certain conditions.

How to get the average of numbers in a column that meet a condition

Getting an average of all numbers in a column works the same way as getting a total, by using `@AVG` instead of `@SUM`. `@AVG(C1..C8)` in Table 2.3 returns 153.25.

The same approach does not work, however, for getting an average of numbers that meet some condition. The reason is that `@AVG` does not ignore the numbers that do not meet the condition; it counts them as 0, which skews the averaging of the numbers meeting the condition. Thus, instead, one must get the total meeting the condition and divide by the number of instances meeting the condition. In the same example for rows meeting the "Home" condition, this formula gives the average:

```
@SUM((A1..A8="Home")*(C1..C8))/@SUM(A1..A8="Home")
```

How to get the maximum and minimum number that meets a condition

To use the earlier example in Table 2.3, how do we get the maximum and minimum number in the C column that meets the condition of being "Away" in the A column?

Getting the maximum is straightforward:
`@MAX((A1..A8="Away")*(C1..C8))` .... returns 170 from C7.

Getting the minimum, however, is *not* straightforward, because the mirror image formula

```
@MIN((A1..A8="Away")*(C1..C8))
```

returns 0 instead of the desired 157 from C6. It does so because all of the numbers in C1..C8 that failed the test in A1..A8 were evaluated as 0, and 0 is less than 157. To get the minimum number that meets the test, we need to disqualify the ones that do not meet the test by artificially increasing them. Among many possible ways to do so, creating an `@IF` function that changes their evaluation upward, and prevents them from being the minimum number, will work, as in this case:

```
@MIN(@IF(A1..A8="Away",0,100000)+(C1..C8))
```

which correctly returns 157. If the number in C1..C8 meets the test, it adds nothing (0) to that number, but if it does not meet the test, it adds 100,000, and such numbers will not be the minimum number.

How to get the maximum value in a column on a certain date

In Table 2.7, an example from WPU 35769, we want to determine the largest value in column B that is paired with the date in cell A1. The formula that accomplishes this is:

```
@MAX((A1..A9=A1)*(B1..B9))
```

This formula correctly returns 8, the value of the cell B3. Essentially, it multiplies all the values in column B by whether the corresponding value in column A is equal to the content of cell A1. If it is equal, then the condition is true, which has a value of 1; if not, it is false, which has a value of 0. Column C shows the resulting block of 1s and 0s. Column D shows the result of multiplying Column B by Column C. An `@MAX` formula applied to Column D returns 8. The formula above does not need Columns C or D; it essentially calculates the information that Column D displays and performs the `@MAX` value on it.

Table 2.7: Getting the maximum value on a date

	A	B	C	D
1	11/23/13	6	1	6
2	11/23/13	4	1	4
3	11/23/13	8	1	8
4	11/23/13	2	1	2
5	11/23/13	5	1	5
6	11/28/13	7	0	0
7	11/28/13	6	0	0
8	11/28/13	9	0	0
9	11/28/13	3	0	0

Column C: A1..A9=A1.
Column D column multiplies B1..B9 by C1..C9.

How to get the minimum value in a column on a certain date

One might think that it would be simple to get the smallest value (2, in cell B4) by substituting `@MIN` in the above formula, like this:

`@MIN((A1..A9=A1)*(B1..B9))` *(Wrong!)*

Unfortunately, as the table in the last example shows, the zeros in cells D6..D9 are all less than the 2 in D4, so this function returns a zero.

The trick is to do math that will reverse the order of the numbers that match so that the smallest matching number will become the largest, and then at the end, when the largest is identified, to undo the match that converted the numbers. Instead of multiplying by `B1..B9`, we multiply by `10-(B1..B9)`, which yields different values in column D, as shown in Table 2.8.

Table 2.8: Getting the minimum value on a date, with a trick

	A	B	C	D
1	11/23/13	6	1	4
2	11/23/13	4	1	6
3	11/23/13	8	1	2
4	11/23/13	2	1	8
5	11/23/13	5	1	5
6	11/28/13	7	0	0
7	11/28/13	6	0	0
8	11/28/13	9	0	0
9	11/28/13	3	0	0

Column C: A1..A9=A1;
Column D: (C1..C9)*(10-(B1..B9)).

An `@MAX` formula applied to column D returns the 8 in cell B4, which corresponds to the smallest date in column B for dates equal to A1. The manipulation can now be reversed by subtracting that number from 10, which yields the correct answer. Therefore, we can get the desired answer by:

```
+10-@MAX((A1..A9=A1)*(10-(B1..B9)))
```

Please note that 10 was used in this example because it was known to be greater than any value in column B, so that all values subtracted from it would be positive, and thus greater than the zeros that are returned by non-matches in column D. Instead of 10, a more generic method would be to substitute `@MAX(B1..B9)+1` for 10.

How to mark possible duplicate entries within a time range

In this example (Table 2.9), the task is to mark entries that *may* be duplicates, even if they are entered with dates that differ up to 7 days. (The different dates were due to the date a charge was made and the date that it was collected from the bank.)

Table 2.9: Marking duplicate entries with a block formula

	A	B	C
1	Date	Amount	
2	01/08/14	$245.00	
3	01/17/14	$26.40	
4	01/17/14	$211.25	
5	02/07/14	$245.00	
6	03/04/14	$50.00	
7	03/04/14	$50.00	???
8	03/10/14	$245.00	
9	03/11/14	$245.00	???
10	04/07/14	$245.00	
11	04/08/14	$485.00	
12	04/08/14	$485.00	???
13	04/10/14	$245.00	???

C3=@IF(@SUM((B$2..B2=B3)*(A$2..A2+7>=A3)),"???","")

We look for numbers in the B column that are identical to prior numbers in the B column, but the dates in the A column may be off by 7 days. Thus, we ignore duplicates that are monthly charges. The formula should place *???* in the C column if a potential duplicate appears. The first possible duplicate will be on the third row. This formula works:

```
@IF(@SUM((B$2..B2=B3)*(A$2..A2+7>=A3)),"???","")
```

The `(B$2..B2=B3)` component finds each prior match in the B column, and the inequality `(A$2..A2+7>=A3)` takes corresponding dates, adds 7 days, and then if the resulting date is greater than or equal to the date on this row, a possible duplicate has been identified. The $ anchors allow us to copy the formula down, parallel with the data.

Part II

Quattro Pro Functions

Quattro Pro Functions

@Functions are essentially shorthand versions of more complex mathematical formulas. Most require you to supply one or more “**arguments**,” or data, in the form of numbers, text (enclosed in double quotation marks), coordinates of a cell or block, or a value or condition supplied by an equation or function that returns the value or condition. Conditions can be combined with connectors such as `#AND#` and `#OR#` or `#NOT#`.

Dr. David Seitman has pointed out that one can add in-cell comments to formulas and functions. After completing the formula or function for a cell, add a semicolon and whatever comments you desire. Thus:

```
@TODAY; this will return today’s date
```

and

```
+7+5; Kant’s illustration of synthetic a priori truth
```

return today‘s date and 12, respectively, ignoring the text additions.

This text will not cover most of the functions available in QP. In particular, it will cover none of the extensive engineering, financial, and statistical functions, and it will not cover many specialty functions in other categories. The user should consult the QP help file for information about these.

Chapter 3

Conditional functions

@IF

`@IF(`*`Test,Result1,Result0`*`)` contains three arguments. The `Test` argument tests a proposition that either evaluates as true (1, or apparently any non-zero number) or false (0). If `Test` evaluates as true, the function returns the value specified in `Result1`; otherwise, it returns the value in `Result0`.

The `Test` argument can be very complicated, but it must evaluate as 1 or 0. It usually involves equations and inequalities (such as =, >, <, >=, <=, <>), but it can include functions such as the `@IS***` functions. Both sides of an equation in `Test` may include formulas or functions. Different equations can be combined in the `Test` argument by the connectors `#AND#` and `#OR#`. Unsurprisingly, these mean that both equations joined by `#AND#` must evaluate as 1 for `Test` to evaluate as 1; but if either of the equations joined by `#OR#` evaluates as 1, `Test` evaluates as 1.

`Result1` and `Result0` may include any values. They may include other formulas and functions. In particular, they may include other `@IF` functions, which are then referred to as nested `@IF` functions. Thus, an `@IF` test for whether cell A1 contains a number greater than 0, or less than 0, or equal to 0, might look like this:

```
@IF(A1>0,"A1 is greater than 0",
@IF(A1<0,"A is less than 0","A1 equals 0"))
```

Quirk with array values

David Seitman notes that if one or more of the arguments of an `@IF` function refer to an array, QP will wrap the function with `@ARRAY`. He observes that QP sometimes fails to compute correctly when that happens, and therefore, if the programmer uses `@IF` with arrays, the results should be diligently checked. Instead of using `@IF` with array values, he suggests using appropriate block formulas, covered above beginning at page 9.

@CHOOSE

`@CHOOSE(`*`Number,ListSeparatedByCommas`*`)` returns the item in `ListSeparatedByCommas` indicated by `Number`. The items in the list begin with 0, so if `Number` is 1, the function returns the second item, and so on. The items can be any value (numbers, text, or functions) and they can be cells containing any values.

This function has great utility when the `Number` is supplied by a formula or function that makes it apply in different settings, some of which will appear below.

For the converse function that returns the offset number from the match, see `@MATCH`.

Table 3.1 illustrates the use of `@CHOOSE` to take a date number (42110, which is Thursday, April 16, 2015) in A1..A2 and give the full name of the month and weekday.

Table 3.1: Using @CHOOSE to give information about dates

	A	B	C
1	04/16/15	3	April
2	04/16/15	5	Thursday

B1 contains the formula `@MONTH(A1)-1`, which provides the month number. Because it returns months on a scale of 1 to 12, and `@CHOOSE` uses a scale starting at 0, we subtract 1 from it.

C1 contains this function:

```
@CHOOSE(B1,"January","February","March","April",
"May","June","July","August","September","October",
"November","December")
```

Starting with "January" as item 0, this function returns "April". The formula in B1 could have been substituted into the first argument of this function.

B2 contains the function `@MOD(A2,7)`, takes the number in A2, divides by 7, and if there is a remainder, it returns the remainder. Here, it returns 5.

C2 contains this function:

```
@CHOOSE(B2,"Saturday","Sunday","Monday",
"Tuesday","Wednesday","Thursday","Friday")
```

Starting with "Saturday" as item 0, it takes the number in B2 and returns "Thursday".

Chapter 4

The Properties of a Cell, Page, File, etc.

The basic functions for determining the content or properties of a cell or some other division of QP are `@CELL`, `@CELLPOINTER`; the `@IS` logical functions and a slight variation on them, `@TYPE`; and the duo of `@PROPERTY` and `@COMMAND`.

@CELL, @CELLPOINTER, @CELLINDEX

These functions return information about certain attributes of a cell. The user specifies the attribute in a text argument. `@CELLPOINTER(`*`Attribute`*`)` returns the information about the currently selected cell; `@CELL(`*`Attribute, Cell`*`)` returns it for a cell specified in the function. If more than one cell is specified in the function, almost all of these will return information about the topmost, leftmost cell.

Table 4.1 discusses some useful attributes. For a full list, see the "Attribute Arguments" under `@CELL` in the QP help file.

Table 4.1: Cell Attributes

Attribute	Return Value
address	Returns the absolute address on the sheet, such as $A$1, as text. If the function refers to a cell on a different sheet, it will add the sheet name, such as $B:$A$1. To get only the column and row, use TwoDAddress. To always get the sheet as well, use ThreeDAddress. The help file suggests that one can get the notebook name too by using FullAddress, but that argument seems to do nothing different from ThreeDAddress. It most definitely does not return the file name. (Hat tip to David Seitman for pointing this out.) To get the notebook path and name alone, use NotebookPath.
row	Returns the row number of cell starting from 1.
col	Returns the column number, starting from 1 (column A), 2 (B), etc.
sheet	Returns the sheet number, starting from 1 (sheet A), 2 (sheet B), etc., even if the sheets are given specific names.
contents	Returns the unformatted contents of the cell. For instance, the date formatted as 01/31/15 would be returned as 42035.

type	Returns one of three letters, depending on the nature of the cell: b, if the cell is blank v, if the cell contains a numerical value l, if the cell contains text (a "label")
prefix	Returns the prefix, if any, for the contents of the cell. In earlier versions, cells were aligned by preceding them with a single-quote, for left alignment, a double-quote, for right alignment, or a caret, for centered alignment. A single character (usually - or =) could be repeated across the cell by preceding it with a backslash.
width	Returns the width of the leftmost column.
rwidth	Returns the combined width of the entire block specified.
protect	Returns 1 if the cell is protected, 0 otherwise.
format	Returns a text shorthand for the formatting applied to the cell. For general formatting, it returns G. For typical currency, it returns C2.

Thus, `@CELLPOINTER("type")` will return a letter that says whether the current cell is blank, has a number or text in it. `@CELL("type",A1)` does so for cell A1 on the current sheet (but see the Caution below about use of the function in macros).

Table 4.2: @CELL Return Values

	A	B	C	D	E	F	G	H
1		address	col	row	contents	type	prefix	format
2	Text	$A$2	1	2	Text	l		ERR
3	Text	$A$3	1	3	Text	l	^	ERR
4	42035	$A$4	1	4	42035	v		G
5	01/31/15	$A$5	1	5	42035	v		D4
6	$123.45	$A$6	1	6	123.45	v		C2
7		$A$7	1	7	0	b		G

A2 and A3 contain the word "Text", with and without a centering prefix. If A2 and A3 have already been formatted, either automatically or manually, H2 and H3 would return those formats.
A4 and A5 contain the same number, with or without date formatting.
A6 contains a number formatted as currency.
A7 is blank. Attributes are in B1..H1.
The function @CELL(B$1,$A2) was entered in B2, then copied to B2..H7.

`@CELLINDEX(`*`Block,Col#,Row#`*`)` does what `@CELL` does when the cell is specified by a Block, with column and row offsets, both starting at 0. Thus, `@CellIndex("format",A1..G7,3,5)` returns the format of cell D6 (column D = offset 3; row 6 = offset 5).

Caution: Use in macros

When used in macros, it is important to make the desired cell unambiguous, because if the `Cell` argument *could* refer to a cell on the sheet where the macro command is operating, QP will construe it that way. To make the `Cell` refer to the currently active sheet, precede it with `[]`. Thus, in a macro, `@CELL("contents",A1)` will return the contents of cell A1 on the sheet with that macro command, but `@CELL("contents",[]A1)` will return the contents of cell A1 on the currently active sheet, which may well be a different sheet. (Hat tip to David Seitman for suggesting this caution.) See page 118 below for more information on this problem.

@IS*** logical functions

Table 4.3 shows what the various `@IS` functions returns when applied to the contents of cells B1..J1. F1 is blank. G1 contains a formula that returns a blank, `@TRIM(@CHAR(32))`.

Table 4.3: Return values of the @IS functions

	A	B	C	D	E	F	G	H	I	J
1		0	1	2	Text			ERR	NA	A1..A20
2	@ISBLANK(B1)	0	0	0	0	1	0	0	0	0
3	@ISBLOCK(B1)	1	1	1	1	1	1	1	1	1
4	@ISERR(B1)	0	0	0	0	0	0	1	0	0
5	@ISEVEN(B1)	1	0	1	ERR	ERR	ERR	ERR	ERR	ERR
6	@ISLOGICAL(B1)	1	1	0	0	0	0	ERR	NA	0
7	@ISNA(B1)	0	0	0	0	0	0	0	1	0
8	@ISNONTEXT(B1)	1	1	1	0	1	0	1	1	0
9	@ISNUMBER(B1)	1	1	1	0	1	0	1	1	0
10	@ISODD(B1)	0	1	0	ERR	ERR	ERR	ERR	ERR	ERR
11	@ISSTRING(B1)	0	0	0	1	0	1	0	0	1

Quirks.

`@ISBLANK`, `@ISEVEN`, `@ISLOGICAL`, `@ISNONTEXT`, and `@ISODD` all wrap themselves in an `@ARRAY` function, needlessly, as far as I can tell. The help files do not reflect that these functions require an `@ARRAY` wrapper; they affirmatively assert that the functions return correct values without any further elaboration.

This is particularly significant in crafting more complex functions and in drafting macros. Because some or all of these `@IS` functions appear to return values as an array, they may be incompatible by themselves with a macro command that expects a non-array value.

A workaround, at least in the case of `@ISBLANK`, is to wrap the `@IS` function inside a function that considers only the first element of the array. Such a function is `@N`, which returns the numeric value of the first element in an array. Hence, @N(@ISBLANK(A1)) will return a 1 if A1 is blank or 0 if it is not, and those values can be used safely in a macro.

`@ISBLOCK` is rendered `@Iblock` (mixed case) on the screen, and it behaves differently from other functions. In the other cases, when the function contains a cell as an argument, it tests the content of that cell. `@ISBLOCK` tests only whether its argument is a valid cell or block. If so, it returns 1; if not, it returns 0. (Hat tip to David Seitman for correcting my initial discussion of `@ISBLOCK`.

See discussion in OC 4993.

@TYPE

A variation on the `@IS` functions is `@TYPE(Value/Cell)`. When applied to a value or cell, it returns 1 if the value/cell is numeric, logical, or empty; 2 if it contains text, 16 if it is ERR, and 64 if it is an array. The help file says that it can be used to test if a function is receiving the type of value that it expects to receive.

@N, @S

`@N(`*`Block`*`)` always returns the numeric value of the topmost, leftmost cell in a block. If the cell contains a string, the function returns the numeric value 0. This can be useful in combination with other functions that might expect a numeric value.

`@S(`*`Block`*`)` always returns the string value of the topmost, leftmost cell in a block. If the cell contains a number, the function returns the null string (""). This can be useful in combination with other functions that might expect a string value.

@PROPERTY

`@PROPERTY("`*`Object.Property`*`")` returns a value about some component of the QP notebook: a cell, a sheet, or the notebook itself. The user must compose text that identifies first the object, then the property, and optionally, sub-properties, all separated by periods. I have placed several appendices showing what properties are available at the end of this section. The function returns values as text, even if they look like numbers.

The `Object` can be:

- a cell (e.g., A1) or block (e.g., A1..A5), or a function that returns a cell *as text*.
- `Active_Block`, which returns values about the currently selected cell or block.
- `Active_Page`, which returns values about the current sheet.
- `Active_Notebook`, which should be self-explanatory.
- `Application` (undocumented).

And though the `Object` can be a specific cell or block in the active notebook other than the selected cell or block, page-level and notebook-level properties can only be read on the active sheet and notebook. Thus, specifying a cell on another page as the `Object` will only allow us to get the properties of that cell, not the page in which it is located.

The QP help file says that the `Property` is a string and provides a link to the main help file, which contains a large array properties grouped into categories and sub-categories. An appendix to this section looks at those properties that apply at four levels: Cell/block; Page (Sheet); Notebook; Application. I have not experimented with the properties that apply to charts, dialogs, menus, and other objects.

Quirks.

Although `@PROPERTY` can use a block of cells as the `Object`, it does not return values in an array, so almost all of the information that it returns is valid only for the first cell in the block.

Attempting to determine whether a row or column is hidden fails. See the workaround in WPU 31127, which relies on a combination of macro commands (`{OnError}` and `{EditGoTo}`) to detect hidden cells.

My sense is that it takes QP a long time to evaluate the argument, and therefore I do not use `@PROPERTY` in macros that rapidly and repetitively use this function.

@COMMAND

`@COMMAND("`*`CommandEquivalent`*`")` performs largely the same functions as `@PROPERTY`. It gets the same settings for pages/sheets and notebooks that `@PROPERTY` gets. It also allows the user to get settings of the QP application itself (but `@PROPERTY` does the same without the help file's documenting that fact). It does not get settings for individual cells, and it cannot be used to get settings for pages and notebooks other than the current one. It returns values as text, even if they look like numbers.

The help file misleadingly suggests that one can get a list of the "command equivalents" that are "acceptable arguments" by pressing `[Shift+F3]`. As it turns out, the format is exactly the same as the format for `@PROPERTY` arguments, except the object is either:

- `Page` (not Active_Page)
- `Notebook` (not Active_Notebook)
- `Application`

Thus, for instance, `@PROPERTY(Active_Page.Zoom_Factor)` yields the same information as `@COMMAND(Page.Zoom_Factor)`.

@FILEEXISTS

`@FILEEXISTS(`*`FileName`*`)` evaluates as 1 if the identified file exists, and 0 if it does not. If the path is not specified for the file, it tests whether the file exists in the current directory.

How (not) to test whether a cell is blank

How to test for whether a cell is blank is complicated by two variables: (a) the cell may contain an entry that gives a false positive for being blank, and (b) the user may or may not want to treat such false positives as if they actually are blank.

Table 4.4 uses seven possible tests to determine whether a cell is blank (A2..A8), as applied to four conditions that may sometimes test as blank: a true blank cell in B1, a zero in C1, a function that returns an empty string in D1, and a single apostrophe in E1. Some of the results are surprising.

All tests applied to the true blank cell in B1 correctly return the value of 1.[1]

Surprisingly, however, a zero in cell C1 tests as if it is blank for the tests in A2, A3, and A3. If the user wants to treat a cell with a zero in it as non-blank, those tests must not be used and must use instead the tests in A4 to A8.

As for functions that return empty strings or cells containing only prefixes (D1 and E1), the user may or may not want to treat them as blank. If they should count as blank, the user may use only the formulas in A2 to A5. If the user wants to treat those cells as non-blank, only the formulas in A6 to A8 give the desired result.

[1]In an earlier version of this test, I considered `@PROPERTY(@CELL("address",B$1)&".Value")=""` as an option, but John Kemmis brought to my attention (OC 15933) some unexpected problems with using it, so it is now withdrawn as an option.

Table 4.4: Seven tests for blank cells, and how they deal with four cases

	A	B	C	D	E
1	Test for blanks:		0		
2	+B$1=""	1	1	1	1
3	@IF(B$1="",1,0)	1	1	1	1
4	@CELL("contents",B$1)=""	1	1	1	1
5	@COUNTBLANK(B$1)	1	0	1	1
6	@COUNT(B$1)=0	1	0	0	0
7	@ARRAY(@ISBLANK(B$1))	1	0	0	0
8	@CELL("type",B$1)="b"	1	0	0	0

B1 is blank C1 contains zero
D1 contains @TRIM(" ") E1 contains '
Formulas listed in column A were entered in B, then copied to C, D and E.

The take-away is this:

- To get all and only the cells that have absolutely nothing in them, only `@CELL("type")`, `@ISBLANK`, and `@COUNT` can be used (see rows 6, 7 and 8). And if the user wants to apply the formula to a block (array) of cells, it is best for the user to use `@ISBLANK`, e.g., `@SUM(@ISBLANK(B1..E1))`. `@CELL` cannot be used because it works only on individual cells, not blocks; if it is used on a block, it returns values based only on the topmost leftmost cell in the block. `@COUNT` can be used only if it is subtracted from the number of columns or rows in the block, e.g., `@COLS(B1..E1)-@COUNT(B1..E1)`.
- If the user wishes to count as blank cells that include a formula that evaluates as an empty string or cells that only include a prefix, but not count the entry of 0 as blank, only `@COUNTBLANK` may be used (see row 5). One might think that `@COUNT` might be used, but as observed at page 80 below, the two functions are not symmetrical on this point.
- If the block to be tested does not contain zeros, text prefixes, or formulas that return empty strings or zeros, then any of these tests will do.

Note the further discussion at OC 4993. Steve Szalai observes that some references to *blank* in these functions should rather be *empty*.

Note also that if the cell coordinates are to be derived via the `@@` function (see the chapter on cell coordinate functions at 34), the text in the function needs to evaluate as a single cell block, e.g., *B2..B2*, rather than a single cell alone, e.g., *B2*. Hence, whether B2 is blank is determined by:

```
@COUNTBLANK(@@(@OFFSET(A1,1,1)))
```

But not by:

```
@COUNTBLANK(@@(@ADDRESS(2,2)))
```

How to identify the last non-blank cell in a column

For a discussion of the differences between types of “blank” cells in identifying the last non-blank cell in a column, see WPU 37622. When the blanks in the L column are empty cells, the last non-empty cell is on a row determined by a function like this:

```
@MAX((@ISBLANK(L1..L1000)=0)*@ROW(L1..L1000))
```

But if blanks in the column are created by formulas or functions, this method will not work because `@ISBLANK` regards a cell containing a formula as non-blank. Instead, `@COUNTBLANK` must be used, but it unfortunately does not work with arrays, so one must create a helper column in parallel with the data in the L column. If the helper column will be M1..M1000, place `@COUNTBLANK(L1)=0` in M1 and copy it to M2..M1000. Then, the last non-blank cell in the L column is on this row:

```
@MAX((M1..M1000=1)*@ROW(M1..M1000))
```

How to enter multiple lines of text in a single cell

Multiple lines of text can be entered into a single cell in at least two ways. The first and easiest is to enter one line, then press `Alt+Enter`, then enter the next line, and so on until finished.

The second is to enter an `@CONCATENATE` function, that alternates between lines of text and `@CHAR(10)`, as in:

```
@CONCATENATE(text1,@char(10),text2)
```

To view the result properly, it will probably be necessary to format the cell(s) to all the display of wrapped text. To do so, right click on the cell(s), choose Selection Properties, click the Alignment tab, and under Cell Options, check the Wrap Text box.

How to determine the width and height of a row in inches

The formula for determining the width of a column in inches is this:

```
@VALUE(@FIELD(@PROPERTY("A1.column_width"),2,
","))/1440
```

(Hat tip to David Seitman for correcting the formula in an earlier version.) The formula for determining the height of a row is this:

```
@VALUE(@FIELD(@PROPERTY("A1.row_height"),2,
","))/1440
```

Hat tip to David Seitman, Roy Lewis, and Martin (?). For a lengthier discussion, see WPU 30996. For Dr. Seitman's impressive development that determines the block of cells that fit in the QP window, see WPU 35788.

Appendix: Cell (Active_Block) Properties

These are values that `@PROPERTY` returns for properties of a particular cell or block or the currently selected cell or block, which is denoted by the variable `Active_Block`. The cell is specified by coordinates, such as "A1" in the following table. That cell can be hard-coded or generated by a function that returns a cell as text. The values in column A were generated by placing `@PROPERTY("A1."& B2)` in A2 and copying it to A3..A31.

In lieu of a static cell, substitute `Active_Block` for the cell's address.

All values returned are text, even the ones that look numeric. The values returned in the A column will change, of course, depending on the cell that is current. It is also advisable to recalculate the spreadsheet first, to force QP to make these references current.

Table 4.5: Active Block Properties

	A	B	C
1	A1.	Properties	Possible Values (all text)
2	General,Bottom,No, Horizontal,0,0,0	Alignment	General | Left | Right | Center | Center Across Block, Top | Center | Bottom, WrapText?(Yes | No), Horizontal | Vertical
3	General	Alignment.Horizontal	General | Left | Right | Center | Center Across Block
4	Horizontal	Alignment.Orientation	Horizontal | Vertical
5	Bottom	Alignment.Vertical	Top | Center | Bottom
6	No	Alignment.WrapText	Yes | No
7	Set Width,2370,1	Column_Width	Operation, WidthInTwips, ColSpacing (the help file separates operations)
8	Protect,General	Constraints	Protect | Unprotect, General | Labels Only | Dates Only
9	General	Constraints.Data_Entry _Input	General | Labels Only | Dates Only
10	Protect	Constraints.Protection	Protect|Unprotect
11	Arial,10,Yes,No,No,No	Font	Typeface, PointSize, Bold, Italic, Underline, Strikeout
12	Arial	Font.Typeface	Typeface
13	10	Font.Point_Size	PointSize
14	Yes	Font.Bold	Yes | No
15	No	Font.Italic	Yes | No
16	No	Font.Underline	Yes | No
17	No	Font.Strikeout	Yes | No
18	NoChange,NoChange, NoChange,NoChange, NoChange,NoChange, 0,0,0,0,0,0	Line_Drawing	Left, Top, Right, Bottom, Vert, Horiz, LeftColor, TopColor, RightColor, BottomColor, VertColor, HorizColor Note: The first six settings can take NoChange | Clear | Thin | Thick | Double
19	A1.	Number_Value	the value in the cell
20	General,2,United States,0	Numeric_Format	Format, Precision | Type
21	NA	Reveal/Hide	Row | Column, Reveal | Hide
22	Set Height,246	Row_Height	Operation, Size (the help file separates operations)
23	Cell:A1..A1	Selection	the coordinates of the cell(s)
24	16,3,Blend1,No	Shading	Color1, Color2, Blend
25	16	Shading.Color_1	0-15
26	3	Shading.Color_2	0-15
27	Blend1	Shading.Blend	Blend1 | Blend2 | Blend3 | Blend4 | Blend5 | Blend6 | Blend7
28	A1.	String_Value	the label in the cell
29	Normal	Style	the named style of the active cells

30	3	Text_Color	0-15
31	A1.	Value	the contents of the cell (as they appear on the input line)

Appendix: Worksheet (Active_Page) Properties

These are values that `@PROPERTY` returns for properties of the active page (the current sheet). All values returned are text, even the ones that look numeric. The values returned in the A column will change, of course, depending on the page that is current. Those values were created by placing `@PROPERTY("Active_Page."& B2)` in A2 and copying it to A3..A24. Note that `@COMMAND("Page."& B2)` would be an equivalent command.

Table 4.6: Active Page Properties

	A	B	C
1	Active_Page.	Properties	Possible Values (all text)
2	No,0,1E+300, 4,3,5,4	Conditional_Color	Enable, SmallVal, GreatVal, BelColor, Normal, AboveCol, ERRCol
3	5	Conditional_Color.Above _Normal_Color	0-15
4	4	Conditional_Color.Below _Normal_Color	0-15
5	No	Conditional_Color.Enable	Yes | No
6	4	Conditional_Color.ERR_Color	0-15
7	1E+300	Conditional_Color .Greatest_Normal_Value	GreatVal
8	3	Conditional_Color.Normal_Color	0-15
9	0	Conditional_Color.Smallest _Normal_Value	SmallVal
10	960	Default_Width	WidthInTwips
11	Yes,Yes,Yes, Yes,Yes	Display	DisplayZeros?(Yes | No), RowBorders?(Yes | No), ColBorders?(Yes | No), HorzGridLines?(Yes | No), VertGridLines?(Yes | No)
12	Yes,Yes	Display.Borders	RowBorders?(Yes | No), ColBorders?(Yes | No)
13	Yes	Display.Borders.Column_Borders	Yes | No
14	Yes	Display.Borders.Row_Borders	Yes | No
15	Yes	Display.Display_Zeros	Yes | No
16	Yes,Yes	Display.Grid_Lines	HorzGridLines?(Yes | No), VertGridLines?(Yes | No)
17	Yes	Display.Grid_Lines.Horiz	Yes | No
18	Yes	Display.Grid_Lines.Vertical	Yes | No
19	ActivePage	Name	Name
20	No,No	Protection	CellLocking?(Yes | No), ObjectLocking?(Yes | No)
21	No	Protection.Cells	Yes | No
22	No	Protection.Objects	Yes | No
23	255,255,255,Yes	Tab_Color	
24	100	Zoom_Factor	10-400

Appendix: Notebook Properties

These are values that `@PROPERTY` returns for properties of the active notebook (the current QPW file). All values returned are text, even the ones that look numeric. The values returned in the A column will change, of course, depending on the notebook that is current. The values in column A were generated by placing `@PROPERTY("Active_Notebook."& B2)` in A2 and copying it to A3..A31. Note that `@COMMAND("Notebook."& B2)` is an equivalent command.

Table 4.7: Notebook Properties

	A	B	C
1	Active_Notebook.	Properties	Possible Values (all text)
2	Yes,Yes,Yes,Show All	Display	VertScroll?(Yes | No), HorzScroll? (Yes | No), Tabs?(Yes | No), Objects (Show All | Show Outline | Hide)
3	Yes	Display.Show_HorizontalScroller	Yes | No
4	Show All	Display.Objects	Show All | Show Outline | Hide
5	Yes	Display.Show_Tabs	Yes | No
6	Yes	Display.Show_VerticalScroller	Yes | No
7	Off	Group_Mode	On enables Group Mode
8	No	Macro_Library	Yes | No
9	(a long series of numbers)	Palette	Color1, Color2, ..., Color16
10	16777215	Palette.Color_1	
11	12632256	Palette.Color_2	
12	8421504	Palette.Color_3	
13	0	Palette.Color_4	
14	255	Palette.Color_5	
15	65280	Palette.Color_6	
16	16711680	Palette.Color_7	
17	16776960	Palette.Color_8	
18	16711935	Palette.Color_9	
19	65535	Palette.Color_10	
20	8388736	Palette.Color_11	
21	32768	Palette.Color_12	
22	32896	Palette.Color_13	
23	8388608	Palette.Color_14	
24	128	Palette.Color_15	
25	8421376	Palette.Color_16	
26	None	Password_Level	None | Low | Medium | High
27	Background, Natural,1,Yes,No	Recalc_Settings	Automatic | Manual | Background, Natural | Column-wise | Row-wise, Iterations
28	(Data on current notebook)	Statistics	Filename, Directory, Created(Date Time), Last_Saved(Date Time), Last_Saved_By, Revision_Number
29	05-Apr-10 05:45 PM	Statistics.Created	
30	(A folder)	Statistics.Directory	
31	@PROPERTY.qpw	Statistics.FileName	
32	21-Feb-15 01:33 PM	Statistics.Last_Saved	
33	Charles Cork	Statistics.Last_Saved_By	
34	52	Statistics.Revision_Number	
35	,,Charles M. Cork III,,	Summary	Title, Subject, Author, Keywords, Comments
36	No	System	Yes | No
37	100	Zoom_Factor	10-400

By the way, the **Palette** is a set of sixteen color selections available by using the notebook properties `[Shift+F12]` and clicking the NBPalette tab. Each possible color is the sum of red, green and blue values, calculated as follows: Red (0-255), Green (0-255)*256, and Blue (0-255)*256*256. The sum of those three settings constitutes a unique color that can be assigned to a Palette option. See the discussion in WPU 35079.

Appendix: Application Properties

The following application properties resulted from placing `@COMMAND(A1)` into B1 and copying it to B2..B69. `@PROPERTY(A1)` would return the same values.

Table 4.8: Application Properties

	A	B
1	Application.Compatibility	18278,256,1000000,No,No,Custom, A:A1..B:B2,Letters,256,<QP8/9 Menu>,QPW
2	Compatibility.AlternateMenuBar	<QP8/9 Menu>
3	Compatibility.AutoArrayWrap	No
4	Compatibility.CompatibilityMode	Custom
5	Compatibility.Def_Columns_Limit	256
6	Compatibility.Def_Rows_Limit	1000000
7	Compatibility.Def_Sheets_Limit	18278
8	Compatibility.File_Extension	QPW
9	Compatibility.Min_Number_Sheets	256
10	Compatibility.Range_Syntax	A:A1..B:B2
11	Compatibility.Sheet_Tab_Label	Letters
12	Country_Settings	$,Prefix,United States
13	Current_File	C:\cmc\Spreadsheets\Samples\Reference\ @COMMAND.qpw
14	Display	None,Yes,Yes,Yes,A..B:A1..B2,Yes,Yes,Yes, Draft,Letters,256,,,Yes,Yes,Yes,100,Yes,Yes,No
15	Display.Clock_Display	None
16	Display.CommentMarkers	Yes
17	Display.Default_View	Draft
18	Display.Default_Zoom	100
19	Display.FormulaMarkers	Yes
20	Display.History_List	Yes
21	Display.Min_Number_Sheets	256
22	Display.Range_Syntax	A..B:A1..B2
23	Display.RealTime_Prev	Yes
24	Display.Sheet_Tab_Label	Letters
25	Display.Shortcut_Keys	Yes
26	Display.Show_GroupBox_As_Line	
27	Display.Show_InputLine	Yes
28	Display.Show_PreSelection	
29	Display.Show_Property_Band	Yes
30	Display.Show_Scroll_Indicator	Yes
31	Display.Show_Tool_Hint	Yes
32	Display.Show_Toolbar	Yes
33	File_Options	C:\cmc\Spreadsheets,,QPW,Yes,3,Yes,No,Yes, C:\Users\Charles Cork\AppData\Roaming\ Corel\PerfectExpert\17\EN\ Custom QP Templates,20,No
34	File_Options.AutoBack_Enabled	Yes

35	File_Options.AutoBack_time	3
36	File_Options.AutoLoad_File	
37	File_Options.AutoRefreshTime	20
38	File_Options.DoRefresh	No
39	File_Options.File_Extension	QPW
40	File_Options.Full_Path_Titles	Yes
41	File_Options.QuickTemplates	Yes
42	File_Options.Startup_Directory	C:\cmc\Spreadsheets
43	File_Options.TempDir	C:\Users\Charles Cork\AppData\Roaming\ Corel\PerfectExpert\17\EN\ Custom QP Templates
44	General	Yes,No,Yes,Yes,5000,No,No,No,Yes,Yes,Down
45	General.Calc-As-You-Go	No
46	General.Cell_Reference_Checker	Yes
47	General.Compatible_Formula_Entry	No
48	General.Compatible_Keys	No
49	General.Delay_Time	5000
50	General.Direction	Down
51	General.Fit-As-You-Go	No
52	General.MoveCellOnEnterKey	Yes
53	General.QuickType	Yes
54	General.Undo	Yes
55	International	$,Windows Default,Prefix,Parens,Windows Default, MM/DD/YY (MM/DD), Windows Default,Suite Default, "English, North American (EN)",No,United States
56	International.Currency	Windows Default
57	International.Currency_Symbol	$
58	International.Date_Format	MM/DD/YY (MM/DD)
59	International.Language	"English, North American (EN)"
60	International.LanguageMode	Suite Default
61	International.LICS_Conversion	No
62	International.Negative	Parens
63	International.Placement	Prefix
64	International.Punctuation	Windows Default
65	International.Time_Format	Windows Default
66	Macro	Both,,Menu Bar,\0,No
67	Macro.Macro_Redraw	Both
68	Macro.Slash_Key	Menu Bar
69	Macro.Startup_Macro	\0
70	Title	Quattro Pro

Chapter 5

The Coordinates of a Cell or Block

Here are the main functions for addressing particular cells or blocks. These will be extended greatly in Chapter 10, which deals with getting data about and from a database, and which begins at page 78.

@@

`@@` is among the more poorly documented of QP's functions. I suspect that its functionality has expanded over the years, but the help file was not updated to reflect it. It is quite useful for different purposes.

Let us begin with an illustration (Table 5.1) of what the formulas return in columns B, D, and F (as printed in columns C, E, and G), using the values in A1..A4.

Table 5.1: Return values of @@ by direct reference

	A	B	C	D	E	F	G
1	7	7	⇐ @@("A1")	ERR	⇐ @@(A1)	ERR	⇐ @SUM(@@(A1))
2	5	5	⇐ @@("A2")	ERR	⇐ @@(A2)	ERR	⇐ @SUM(@@(A2))
3	A1	A1	⇐ @@("A3")	7	⇐ @@(A3)	7	⇐ @SUM(@@(A3))
4	A1..A2	A1..A2	⇐ @@("A4")	ERR	⇐ @@(A4)	12	⇐ @SUM(@@(A4))

From these results, we can derive that `@@` actually does two *quite* different things, either (1) returning the contents of a cell or block/array, or (2) converting text coordinates into non-text coordinates.

Returning the Content of a Cell by Direct or Indirect Reference

`@@(`*`SingleCellText`*`)` returns the contents of the cell identified by the text argument `SingleCellText`. Thus, `@@("A1")` returns the contents of cell A1. See cells B2..B5 in the illustration above. This usage will appear in numerous functions and macros below.

The `SingleCellText` argument can be created in different ways:

- A literal string, like `@@("A1")`.
- A formula, like `@@("A"&"1")`.
- A function, like `@@(@ADDRESS(1,1))`.
- An indirect cell reference without quotes, like `@@(A3)`, where the cell A3 contains the text *A1*. See cell D3 in the illustration above (and F3 simply extends D3).

The `SingleCellText` argument cannot be in the form of a single-cell block, such as A1..A1. Thus, it cannot come from the `@OFFSET` function, which refers to single cells as blocks.

If the cell referred to, directly or indirectly, is blank, the `@@` function returns zero, not a blank.

Returning the Content of an Array of Cells by Indirect Reference

`@@(`*`ReferenceBlock`*`)` returns the content of cells referenced in the `Reference-Block` argument. Thus, in Table 5.2, entering `@@(B1..B3)` into C1 causes QP to produce the array in A1..A3, an array of the cells referenced indirectly in B1..B3. The formula in C1 is wrapped in an `@ARRAY` function.

Table 5.2: Return values of @@ by indirect reference

	A	B	C	D
1	5	A1	5	⇐ @@(B1..B3)
2	10	A3	15	
3	15	A5	25	
4	20			
5	25			

The indirect references (B1..B3) could point to cells anywhere; the cells to which they point need not be contiguous or form a block; and there may be any number of indirect references in the `BlockOfReferences` argument.

Converting Text to Coordinates

`@@(`*`BlockText`*`)` returns nothing that can be used by itself (it returns ERR as a standalone function, as in cell D4 of the first illustration), but converts the representation of a block in the text argument `BlockText` into non-text coordinates of that block that can be used in other functions, such as `@SUM`. This usage allows us to plug variable blocks into functions that expect to receive coordinates without hard-coding the coordinates into the function. For those functions, `@@` converts text-type coordinates into non-text coordinates.

Contrary to the help file, the argument need not be a single cell or a cell at all. The `BlockText` argument can be created in different ways:

- A literal string, like `@@("A1..A2")`.
- A formula compiling literal strings, like `@@("A1"&".."&"A2")`.
- A function, like `@@(@OFFSET(A1,0,0,2,1))`.
- An indirect cell reference without quotes, like `@@(A4)`, where the cell A4 contains the text *A1..A2*.
- Another way of indirectly referencing a block is a function like `@@(A5&".." &A6)`, where A5 and A6 contain the top-left and bottom-right cells of a block.

All of these methods of indirect reference return ERR when used as a standalone functions, but wrapped in `@SUM` or similar functions, they all return 12, as in cell F4.

To use `@@` to get the coordinates of a single cell, the argument must be text like *A1..A1*, a format returned by `@OFFSET`. Any other argument causes `@@` to return ERR. See cells D1, D2, F1, and F2 in the first illustration.

Comments

Contrary to the QP help file, this text argument for `@@` need not be in a different cell. The documentation states that the argument is "a single cell address that contains another cell address or cell name that is written as a label." So, one would not guess that the argument could be a function that creates a text string that represents multiple cells, though it can: the final example in the current help file on `@@` shows its use in this context, but the poor reader trying to find a function that works would be unlikely to have gotten that far. I've only stumbled on this use of `@@` because of trying everything I could imagine to get some functions to work.

Table 5.3 illustrates the significant difference between two apparently slight variations in composing indirect references through a block of cells. The formula in D1 returns the lesser of the values in cell A1 or cell B5; the formula in D2 returns the least value in the block A1..B5.

Table 5.3: The importance of structuring an indirect reference

	A	B	C	D	E
1	50	20	A1	40	⇐ @MIN(@@(C1..C2))
2	40	25	B5	10	⇐ @MIN(@@(C1& ".."& C2))
3	30	30			
4	20	35			
5	10	40			

@ADDRESS

`@ADDRESS(`*`Row#,Col#,<RefType,Format,Page>`*`)` has two mandatory arguments and three optional ones.

`Row` The `Row#` argument is equivalent to the blocks on the left side of the data.

`Col` Column A is `Col#` 1, B is `Col# 2`, and so on.

`RefType` The `RefType` optional argument is a number from 1 to 8 that determines which combination of absolute and relative coordinates (see discussion at 3) will be returned. The default appears to be absolute coordinates for page, column, and cell.

`Format` The `Format` optional argument specifies whether to use the absolute reference system (the default) or the relative system. I do not yet see a reason for using the latter, but perhaps there is one.

`Page` The `Page` optional argument determines whether the reference will be preceded with the name of a page or not. The page can be the default "A", "B", etc., or it can be a named page, such as "Data." If the named page does not exist, the function would return an ERR.

`@ADDRESS` returns the coordinates as text. Some functions and commands can use those text coordinates for further automation, but for others, the text coordinates must be converted to non-text coordinates by wrapping the function in an `@@` function.

For techniques of identifying cells and blocks relative to the currently selected cell, see page 86 below.

@OFFSET

`@OFFSET(StartCell,RowOffset#,ColOffset#,<Height>,<Width>)` has five parameters, the first three of which are mandatory and the latter two are optional.

The first three mandatory parameters set the starting point by identifying the starting cell, a row offset, and a column offset.

The `StartCell` is typically hard-coded as a cell (often A1), but if it is generated by some other function, if must be in the form of a block, such as A1..A1. The block need not refer to one cell; it could refer to an entire database such as Data:A1..Z1000, but offsets will still be measured from the topmost, leftmost cell in the block.

Both offsets start with 0, not 1, so setting those offsets to 0 means that the block begins in the starting cell. If you use only the three mandatory parameters, the `@OFFSET` function identifies only one cell, though it does so as a block like A1..A1.

The two optional parameters identify a block by asking for how many rows and how many columns are considered. The minimum values for each are 1, since a block cannot have zero width or height.

`@OFFSET` also returns the coordinates as text. Some functions and commands can use those text coordinates for further automation, but for others, the text coordinates must be converted to non-text coordinates by wrapping the function in an `@@` function.

For techniques of identifying cells and blocks relative to the currently selected cell, see page 86 below.

Comparing @ADDRESS and @OFFSET

`@ADDRESS` and `@OFFSET` are ways of returning coordinates on the spreadsheet as text. They have a common core function and divergent functions. Where they diverge, the choice between them depends on certain factors:

- `@ADDRESS` returns coordinates with reference to the A1 cell on a spreadsheet; `@OFFSET` returns them with reference to any cell that the user specifies.
- `@ADDRESS` returns the coordinates of a single cell; `@OFFSET` can return the coordinates of a multi-cell block as well as a single cell, and even for a single cell, it returns the coordinates as a block, e.g. A1..A1.
- `@OFFSET` returns only relative coordinates (e.g., A1); `@ADDRESS` can return any combination of relative and absolute coordinates (see discussion at 3), such as $A$1.
- `@OFFSET` uses only the absolute reference style (e.g.,A:A1); `@ADDRESS` can use the "R1C1" or relative reference style (discussed at 5), such as R0C0.

Two additional differences should be noted:

- `@ADDRESS` calls for the row number before the column number; `@OFFSET` calls for the column number first.
- For `@ADDRESS`, column and row numbers start with 1 (e.g., A1 is `@ADDRESS (1,1)`). For `@OFFSET`, the starting cell is at position 0,0 (e.g., A1 is `@OFFSET (A1,0,0)`). A table of functions and macros will show the difference the starting point makes. See Table 18.15.

Both functions return the coordinates as text. Some functions and commands can use those text coordinates for further automation, but for others, the text coordinates must be converted to non-text coordinates by wrapping the function in an `@@` function.

How to replace named blocks with @@

As noted at page iv above, I generally avoid named blocks for a variety of reasons. One way of getting the benefit of named blocks without the baggage is to place the text coordinates of the block that would be named into a cell. Then, an `@@` function that refers to the cell can be used in the place of a named block. The block will not be resized by the insertion or deletion of rows and columns; it can only be resized by changing the coordinates in the cell.

In Table 5.4, the database block is typed in text at F1. `@OFFSET` functions in F2 and F4 use it as the anchor for identifying a single cell and an entire column in the database, respectively. The `@INDEX` function in F3 uses it as the required boundaries of the database for retrieving data within the database. The function in F5 takes the function in F4 and wraps it first in another `@@`, which converts the text coordinates into non-text coordinates, and then wraps that in `@SUM`, which provides the total of all numbers in the column.

Table 5.4: Using @@ in lieu of named data ranges

	A	B	C	D	E	F
1	100	108	116	124	132	A1..E8
2	101	109	117	125	133	C5..C5
3	102	110	118	126	134	120
4	103	111	119	127	135	C1..C8
5	104	112	120	128	136	956
6	105	113	121	129	137	
7	106	114	122	130	138	
8	107	115	123	131	139	

F2 = @OFFSET(@@(F1),4,2) F3 = @INDEX(@@(F1),2,4)
F4 = @OFFSET(@@(F1),0,2,8,1) F5 = @SUM(@@(@OFFSET(@@(F1),0,2,8,1)))

How to get the attributes of a single cell by indirect reference

Say that you want to set up formulas to get cell attributes (in B1..H1 of Table 5.5) for any cell whose address you enter in a particular cell, say A2. In cell A2, it is text, but it needs to be converted to non-text coordinates. As explained above, using `@@(A2)` would not work because `@@` would treat it as seeking the contents of the cell indirectly referred to by A2. `@@("A2")` would not work because that would simply get the content of cell A2 as text. Instead, we need to convert the textual cell-reference in A2 into a textual block-reference, which is fairly easy to do with the formula `A2&".."&A2`.

In this example, we enter G5 into cell A2. Cell G5 contains the word `hello`. Using the grid shown above, we get this:

Table 5.5: Getting cell attributes by indirect reference

	A	B	C	D	E	F	G	H
1		address	col	row	contents	type	prefix	format
2	G5	$G$5	7	5	hello	l		G

B2 = @CELL(B1,@@($A2& ".."& $A2)) copied to C2..H2.
User enters cells to inspect in A2, and the formulas recalculate.

Table 5.6: @BLOCKNAMES illustration

	A	B	C	D	E	F	G
1	🟩	🟩	🟩				
2	🟩	🟩	🟩		🟪	🟪	🟪
3	🟩	🟩	🟩		🟪	🟪	🟪
4	🟩	🟩	🟩				
5	🟩	🟩	🟨	🟨			
6	🟩	🟩	🟩				

A1..C6 are named *Invoice*
E2..G3 are named *Criteria*
C5..D5 are named *Test*

Table 5.7: @BLOCKNAMES Return Values

Function	Block name	Coordinates returned *(with notebook and sheet names)*
@BLOCKNAMES(A1..G6)	Invoice	A1..C6
	Criteria	E2..G3
	Test	C5..D5
@BLOCKNAMES(C1..E2)	Invoice	A1..C6
	Criteria	E2..G3
@BLOCKNAMES(A1..C6)	Invoice	A1..C6
	Test	C5..D5

@BLOCKNAME, @BLOCKNAMES

I generally prefer not to use named blocks for reasons addressed above at iv, but many prefer them. Two functions may be useful.

If a particular block of cells has been named, `@BLOCKNAME(`*`Block`*`)` returns the name that this precise block of cells has been given. Thus, if the user has named cells B2..C7 as *Invoice*, `@BLOCKNAME(B2..C7)` returns the name *Invoice*. A slight difference, like including an additional row with `@BLOCKNAME(B2..C8)` will return ERR unless that block has its own name.

Somewhat more useful is `@BLOCKNAMES(`*`Block`*`)`, which creates a two-column array of every named block that intersects in whole or in part with the `Block` argument. Using the settings in Table 5.6, Table 5.7 shows how to extract block names.

Chapter 6

Functions for Text (Strings)

This is basic information on the most importance text/string @functions. Note that where *`Text`* is used as an argument, the argument can be a cell that contains text, a function that returns text, or any other way that text can be represented.

Creating text: @CHAR, @REPEAT

To create a string that repeats one or more characters a specified number of times, use `@REPEAT(`*`Text,#Repetitions`*`)`.

To create a text character from its ascii number, use `@CHAR(`*`number`*`)`. The `number` argument takes a number from 1 to 255.

This might be useful when you need to add, for instance, the double-quote character to an argument in a function or macro command, but typing it directly would be treated as the start or end of a text string. `@CHAR(34)` generates the double-quote character.

Among other more useful associations, note that 65–90 represent upper-case letters A–Z; 97–122 represent lower-case letters a–z, 163 = £; 167 = §; 176 = °; 182 = ¶; and 191–255 represent a number of non-English letters. For a complete listing, as the help file says, search online.

Deleting non-text: @CLEAN, @TRIM

To remove unwanted nonprinting characters (characters 1–31 of the ASCII code), use `@CLEAN(`*`Text`*`)`. To remove excess spaces (those before and after the rest of the text, and more than one space between words, use `@TRIM(`*`Text`*`)`.

Combining text: @CONCATENATE

To combine text and numbers into text, use `@CONCATENATE(`*`Items, separated by commas`*`)`. See also the discussion of text formulas at page 5 above. If the numbers require special formatting, the list should contain a function that converts the number to suitably formatted text.

`@CONCATENATE(`*`List,separated by commas`*`)` combines into a single string of text a series of items, whether text or numbers, whether hard coded into the list or contained in cells that are referenced in the list. It can use cell addresses to combine the contents of those cells, as shown in Table 6.1.

Table 6.1: Joining text and block functions

	A	B	C	D	E	F	G	H	I	J
1	28	70	50	6	42	53	63	24	83	9
2	Number > 50 = 4									

A2= @CONCATENATE("Number > 50 =",@SUM(A1..J1>50))

An earlier method used a formula like `+"Item1:  "&A1` to combine the text and the contents of cell A1. This still works, but only if all items combined in this way are text. Numeric values have to be converted to text with additional functions.

I like everything about `@CONCATENATE` except for its length. Corel should give us the same function with the name `@JOIN`, or just `@J`.

Dividing text: @LEFT, @RIGHT, @MID, @FIELD

Get as specific number of characters at the beginning or end of an item of text with `@LEFT(`*`Text,Number`*`)` or `@RIGHT(`*`Text,Number`*`)`, respectively.

To get a number of characters somewhere in the text, use `@MID(`*`Text, StartOffset,Number`*`)`, where the `StartOffset` argument represent the character in `Text` from which to begin. The first character in the Text has an offset value of 0, not 1, and therefore `@MID(`*`Text,0,Number`*`)` is equivalent to `@LEFT(`*`Text,Number`*`)`.

`@FIELD(`*`Text,Number,Delimiter`*`)` uses the `Delimiter` to divide up the `Text` argument into segments, numbered from 1 (not 0), and returns the segment identified by `Number`. If there is no such segment, the function returns ERR.

Modifying or deleting text: @SUBSTITUTE, @REPLACE, @SETSTRING

Two similarly named functions modify one text string with another.

`@SUBSTITUTE(`*`Text,PartToReplace,ReplacementText`*`)` returns a text string that starts with the `Text`, but replaces every instance of the string `PartToReplace` with `ReplacementText`. This is the quick way to replace every occurrence of one character(s) with another. By using "" as the `ReplacementText`, the function deletes any occurrences of characters specified in `PartToReplace`.

You can replace or insert text with `@REPLACE(`*`Text,StartOffset, Number,NewText`*`)`. This function places `NewText` at the position in `Text` identified by `StartOffset`, where 0 is the first character, 1 is the second and so on. If `Number` is 0, `NewText` is simply inserted there. If Number is greater than 0, the same number of characters in `Text` beginning at the insertion point will be deleted (thus replaced by `NewText`).

You can add spaces on either or both sides of a text string by using the `@SETSTRING(`*`Text,Length,Alignment`*`)` function, which is useful mainly in connection with fixed-width fonts. It adds spaces to the `Text` so that the resulting string has the total number of characters specified in the Length argument. Spaces are added to the left, or to the right, or on both sides of the `Text` based on the `Alignment` argument.

Modifying the case of text: @UPPER, @LOWER, @PROPER

To set/convert the case of text, use `@LOWER(`*`Text`*`)`, `@UPPER(`*`Text`*`)`, or `@PROPER (`*`Text`*`)`. Most functions handle text in a case-sensitive way, but `@LOWER` and `@UPPER` can be used to neutralize that effect. For instance, if you seek to determine whether the text in A1 is equivalent to the text in A2 without regard to the case, the test `+@UPPER(A1)=@UPPER(A2)` will return 1 (true) even if A1 contains *bill* and A2 contains *BILL*.

Other basic functions: @LENGTH, @FIND

You can get the total number of characters in a string by `@LENGTH(`*`Text`*`)`.

You can determine whether a subtext occurs somewhere in a text, and if so, its starting offset, using `@FIND(`*`Subtext,Text,0`*`)`. The 0 at the end tells QP to search for the `Subtext` from character 0 (the first character) in `Text`; the user could specify some later offset if it proved necessary. If the `Subtext` appears in `Text`, the function returns the offset position where `Subtext` may be first found. Thus, if `Subtext` occurs right at the beginning of `Text`, this function returns 0. If `Subtext` does not appear in `Text`, however, the function returns ERR. Table 6.2 illustrates how to get different return values by using variations on the `@FIND` function.

Table 6.2: Using @FIND to determine whether "s" is in a cell

	A	B	C
1		Not there	it is here
2	@FIND("s",B1,0) ⇒	ERR	4
3	@ISERR(@FIND("s",B1,0)) ⇒	1	0
4	@NOT(@ISERR(@FIND("s",B1,0))) ⇒	0	1

In testing whether a subtext is in a text, you will not always want to return ERR; you may instead want a "Yes" (1) or "No" (0) result. Since `@FIND` answers the question with a number (Yes) or ERR (No), you can use the `@ISERR` function to distinguish. That function, however, returns 1 if the formula causes an ERR, which means that the Subtext is not in the Text, and 0 if it is. Since that is the reverse of what we want to know, it can be wrapped in an `@NOT` function, which returns zero if the argument is a non-zero number, and it returns 1 if the argument equals zero. The resulting formula yields 1 (Yes) if the subtext is in the text, and 0 otherwise. This shows the returned values.

How to count the number of instances of a particular character in a cell

To count the number of occurrences of a particular character (here, +) in a text cell (here, A1), this formula should work:

```
@LENGTH(A1)-@LENGTH(@SUBSTITUTE(A1,"+",""))
```

This formula subtracts from the length of the original string the length of a substitute string where + is replaced by nothing (the empty string). That should equal the number of occurrences of the character in question.

How to count the number of words in a particular cell

Building on the last example, this formula counts the number of words in a text string in cell A1 if the text is well-formed:

```
@LENGTH(A1)-@LENGTH(@SUBSTITUTE(A1," ",""))+1
```

or its equivalent:

```
@LENGTH(A1)-@LENGTH(@SUBSTITUTE(A1,@CHAR(32),""))+1
```

The formula counts the number of spaces in the text, which will normally separate words, and adds 1, since in normal text, there will be no spaces on the outside of text in a typical QP cell.

However, this will not work if there are spaces on the outside of text, or if there are multiple spaces between words. In that case, the original text can be corrected by wrapping it in the function `@TRIM`, or `@TRIM` can be placed inside the function itself, as in:

```
@LENGTH(@TRIM(A1))-@LENGTH(@SUBSTITUTE(@TRIM(A1),
" ",""))+1
```

How to return the last word in a series of words

The answers given in this section assume that text in A1 is well-formed, meaning that there are no spaces at the end of the text and there are only single spaces between words. Text can be rendered this way by applying the `@TRIM` function to it. If the text is not well-formed or it is unknown whether the text is well-formed, either apply `@TRIM` to that text or use `@TRIM(A1)` instead of A1 in the formulas discussed here.

If you know that the text in A1 has a specific number of words, say 3, then the function `@FIELD(A1,3," ")` works easily. In this and the other formulas in this section, `@CHAR(32)` can be substituted for the space inside double quotes.

Unfortunately, not all text comes with a predictable structure, and thus the number of the last word may vary. Unfortunately, `@FIELD` does not allow the user to choose "last". Contrast WordPerfect, which allows sorting by the last word in a line, so depending on what you need to accomplish, it may be the better tool.

But within QP, there are still several ways to do this. There are at least three ways.

Solution 1: Nested if-tests

If the number of fields is always going to be a small number, using nested `@IF` tests may be the conceptually easiest way to go. Since each segment number either returns the field or an ERR, `@IF` functions would first test whether the `@FIELD` function for the largest number returns an ERR; if not, it returns that field; if so, it nests an `@IF` function that does the same for the next largest number, and so on until only one field is left. Table 6.3 is a set of `@IF` functions that test for 2 to 5 words. The additional coloration shows what is added for each additional word. Though conceptually straightforward, and quick and relatively easy for 2–3 words, this method obviously becomes more cumbersome as the number of words increases.

Table 6.3: Nesting if-tests

Words	Formula (broken into parts for clarity)
2	@IF(@ISERR(@FIELD(A1,2," ")),@FIELD(A1,1," "),@FIELD(A1,2," "))
3	@IF(@ISERR(@FIELD(A1,3," ")),
	@IF(@ISERR(@FIELD(A1,2," ")),@FIELD(A1,1," "),@FIELD(A1,2," "))
	,@FIELD(A1,3," "))
4	@IF(@ISERR(@FIELD(A1,4," ")),
	@IF(@ISERR(@FIELD(A1,3," ")),
	@IF(@ISERR(@FIELD(A1,2," ")),@FIELD(A1,1," "),@FIELD(A1,2," "))
	,@FIELD(A1,3," "))
	,@FIELD(A1,4," "))
5	@IF(@ISERR(@FIELD(A1,5," ")),
	@IF(@ISERR(@FIELD(A1,4," ")),
	@IF(@ISERR(@FIELD(A1,3," ")),
	@IF(@ISERR(@FIELD(A1,2," ")),@FIELD(A1,1," "),@FIELD(A1,2," "))
	,@FIELD(A1,3," "))
	,@FIELD(A1,4," "))
	,@FIELD(A1,5," "))

Solution 2: Counting spaces

Here is another way that involves more different types of components, but that is ultimately simpler, and it can handle any number words in a text string. First, we determine the number of words by counting the number of spaces:

```
@LENGTH(A1)-@LENGTH(@SUBSTITUTE(A1," ",""))
```

Now, since the last word will follow the last space, there will be one more word than the number of spaces. Accordingly, the formula to get the last word will be:

```
@FIELD(A1,@LENGTH(A1)-@LENGTH(@SUBSTITUTE(A1,
" ",""))+1," ")
```

This is the easiest and best way.

Solution 3: A pseudo-column of data

A still more complex way is to use block functions to determine the offset position of the last space in A1, after which we can determine the number of characters to the right of it by an `@LENGTH` function, after which an `@RIGHT` function will return the text to the right of it. This technique illustrates a trick for creating an array of values as if they were in cells, but without placing them into cells.

Since QP lacks a right-to-left search to find the last space, something similar can be done by (a) converting each character in the text string into a separate entry in a cell in a pseudo-column; (b) testing whether a space in present in each one, yielding a parallel pseudo-column of 1s and 0s; (c) linking the second pseudo-column of 1s and 0s to the Row numbers; (d) using an `@MAX` formula to get the highest row number with a 1 in it, which will also be the offset location of the last space in the original text. The beauty of QP is that all of this can be accomplished in a single function, without placing actual data into actual columns.

The following discussion shows how we can create this function in four or five phases:

(a) Since the `@MID` function can pluck out a single character from a text, a series of `@MID` functions in a column that sequentially plucked out single characters from the text at A1 would create the desired column. It would start with `@MID(A1,0,1)`, then `@MID(A1,1,1)`, and so on. But we need the column to stop at the last character in A1, because otherwise it will yield an ERR. We can determine what the offset number of the last character by `@LENGTH(A1)` and subtracting 1 (since the first character has an offset of 0). If QP syntax allowed it, we could use its Block formula shorthand system and type the invalid formula `@MID(A1,1..@LENGTH(A1),1)` to have QP generate the column, but it does not allow the `@LENGTH(A1)` syntax to specify an array of numbers. However, QP does allow the `@ROW` function to supply the row numbers of a block of cells, and we can create a block of cells large enough to hold precisely each single character in A1. The easiest block of cells would be:

```
@OFFSET(A1,0,0,@LENGTH(A1),1)
```

If the text in A1 had 100 characters, this would return the block A1..A100. (The reader need not worry that this overlaps the source cell and causes the circularity indicator in the application bar at the bottom to display, since this will all be combined in a single function that could be placed anywhere. This function could, however, be written to create the block anywhere.) Since `@OFFSET` returns the block as text, and we'll need to turn it into coordinates, it must be wrapped in an `@@` function, as follows:

```
@@(@OFFSET(A1,0,0,@LENGTH(A1),1))
```

That formula (which can't stand alone—see the section on `@@` at 34) is now ready to wrap in an `@ROW` function:

```
@ROW(@@(@OFFSET(A1,0,0,@LENGTH(A1),1)))
```

This function will create a column of numbers, starting with 1 and going to the number of characters in A1. It is now ready to plug into the @MID function after one change. We need to subtract 1 from each of those row numbers to deal with the offset in `@MID` starts with 0. As such, the final formula that generates a column of single characters from the text in A1 is this:

```
@MID(A1,@ROW(@@(@OFFSET(A1,0,0,@LENGTH(A1),1)))
-1,1)
```

Interestingly, QP automatically wraps this formula in an `@ARRAY` function, so the final product of this stage would look like this if we stopped here:

```
@ARRAY(@MID(A1,@ROW(@@(@OFFSET(A1,0,0,
@LENGTH(A1),1)))-1,1))
```

(b) The rest of the steps are easier to build on, using QPs Block formulas. The last formula can be modified to yield a parallel column of 1s and 0s depending on

whether the column of characters contains a space (1) or not (0), by adding the equation at the end:

```
@ARRAY(@MID(A1,@ROW(@@(@OFFSET(A1,0,0,
@LENGTH(A1),1)))-1,1)=" ")
```

(c) Another column can be created that multiplies the last column of 1s and 0s by the row number, so that we get instead a column of 0s and row numbers.

```
@ARRAY((@MID(A1,@ROW(@@(@OFFSET(A1,0,0,
@LENGTH(A1),1)))-1,1)=" ")
*(@ROW(@@(@OFFSET(A1,0,0,@LENGTH(A1),1)))-1))
```

This formula multiplies the last formula by the formula for row numbers above.

(d) We can therefore get the offset of the last space by applying the `@MAX` function to the last formula. It simply replaces the `@ARRAY` component.

```
@MAX((@MID(A1,@ROW(@@(@OFFSET(A1,0,0,
@LENGTH(A1),1)))-1,1)=" ")
*(@ROW(@@(@OFFSET(A1,0,0,@LENGTH(A1),1)))-1))
```

Since we want the text to the right of that, we want a number of characters equal to the total number of characters, less the number found by the last formula, and less 1 (since otherwise our text would start with that space). So, the final formula works out to:

```
@RIGHT(A1,@LENGTH(A1)-
@MAX((@MID(A1,@ROW(@@(@OFFSET(A1,0,0,
@LENGTH(A1),1)))-1,1)=" ")
*(@ROW(@@(@OFFSET(A1,0,0,@LENGTH(A1),1)))-1))
-1)
```

(e) That is the end if your data will always have a space. However, if the data lacks a space, as in the case of a single word, there would be no maximum row, and the formula would either ignore the first letter in the word or return an `ERR`. To guard against that possibility, add an if-test at the beginning:

```
@IF(@ISERR(@FIND(" ",A1,0)),A1,
@RIGHT(A1,@LENGTH(A1)-
@MAX((@MID(A1,@ROW(@@(@OFFSET(A1,0,0,
@LENGTH(A1),1)))-1,1)=" ")
*(@ROW(@@(@OFFSET(A1,0,0,@LENGTH(A1),1)))-1))
-1))
```

How to return the last name in a string, ignoring suffixes

The preceding article showed several techniques for finding the last word in a string of text. A common use for such techniques is to find the last name in a string containing a person's entire name. The same technique would apply in most cases, but at least in English, the last name is often followed by a suffix such as Sr., Jr., or (in my case) III. How to ignore the suffix in identifying the last name was the subject of WPU 37332. I identified several possible ways to go about the task with macros or using functions to assist manual selections, and these might be preferable in some cases. The user requested a function that would return the name automatically.

I start with the “easiest and best” function, the second solution in the preceding article. It adds +1 to the number of spaces in well-formed text to identify the field-number of the last field (word) of the text. That number needs to be 0 if the last field is one of the suffixes. An `@FIND` function would allow us to search for that last string in a constant text string like "JR.SR.III", and it would return a number if the string is present (and we have a suffix) or it would return ERR if the string is not present (and we have a last name). Wrapping that result in `@ISERR` will return 1 if we have a last name, and 0 if we have a suffix. That’s just what we need. The function that replaces +1 in the second solution above is highlighted in this function:

```
@FIELD(A1,@LENGTH(E1)-@LENGTH(@SUBSTITUTE(A1,
" ",""))+@ISERR(@FIND(@FIELD(A1,@LENGTH(A1)
-@LENGTH(@SUBSTITUTE(A1," ",""))+1," "),
"Jr.Sr.III",0))," ")
```

How to generate a new file name based on the current notebook

It may come in handy, particularly with macros, to create a new file from the current one, and to give it the same name, but a different extension or path.

To give the current file name a new extension involves replacing the last three letters in the name (QPW) with something else, such as TXT or XLS. Several functions can do this. Start by selecting one of the functions that returns the full current path and name of the current QP file. These are:

```
@CELLPOINTER("NotebookPath")
@COMMAND("Application.Current_File")
@PROPERTY("Application.Current_File")
```

The following functions should generate the desired file names, subject to some qualifications:

1. `@SUBSTITUTE(@COMMAND("Application.Current_File"), ".qpw",".txt")`
   This will work nicely if extensions are all lower case. To deal with the possibility that the extension contain upper case characters, you can wrap the `@COMMAND` component in an `@LOWER` function.
2. `@FIELD(@COMMAND("Application.Current_File"),1,".") &".xls"`
   This will work nicely as long as you know that there are no stray periods in the file name or path.
3. `@LEFT(@COMMAND("Application.Current_File"), @LENGTH(@COMMAND("Application.Current_File"))-3) &"txt"`
   This method uses everything except the last three letters and adds the alternative extension, so it does not make any other assumptions about the file name or path.

To combine the file name with a different path, however, a different function must be used to isolate the current file name from its path. The easiest function is `@COMMAND("Notebook.Statistics.FileName")`. Thus to generate a file name in the Temp folder, the previous functions can be revised thus, subject to the same qualifications:

1.
   ```
   +"C:\Temp\"&@SUBSTITUTE(@COMMAND("Notebook.
   Statistics.FileName"),".qpw",".txt")
   ```

2.
   ```
   +"C:\Temp\"&@FIELD(@COMMAND("Notebook.Statistics.
   FileName"),1,".")&".txt"
   ```

3.
   ```
   +"C:\Temp\"&@LEFT(@COMMAND("Notebook.Statistics.
   FileName"),@LENGTH(@COMMAND("Notebook.Statistics.
   FileName"))-3)&"txt"
   ```

Chapter 7

Math Functions

QP has a tremendous number of functions for advanced math, geometry, statistics, engineering, finance, and more, but this text will focus only on more basic functions.

Adding: @SUM, @TOTAL, @SUBTOTAL, @SUMIF

Adding sets of numbers is perhaps the most basic function of a spreadsheet, and `@SUM(`*`Block`*`)` is therefore probably the most basic function in QP.

`@SUMNEGATIVE` and `@SUMPOSITIVE` total only those items in the block that are negative or positive, respectively.

`@TOTAL` appears to be a variety of `@SUM` that totals all numbers in a block *except* those created by an `@SUBTOTAL` function, which appears to differ from `@SUM` only in creating totals that aren't counted by `@TOTAL`.

To sum particular entries in a block that match certain criteria, the user may wish to use block formulas (see page 9) or

`@SUMIF(`*`Block,Criterion,DataColumn`*`)`

`Criterion` is the value *that must appear in the first* column of the data block. `DataColumn` is the column that has the numbers to be summed if the desired value is in the first column. Table 7.1 illustrates this.

Table 7.1: @SUMIF

	A	B	C	D	E
1	2008	3		2008	9
2	2008	6		2009	11
3	2009	7		2010	9
4	2009	4		2011	6
5	2010	9		2012	0
6	2011	6		2013	0

E1 = @SUMIF($A$1..$B$6,D1,$B$1..$B$6)   copied to cells E2..E6
E1 is equivalent to @SUM((B1..B6)*(A1..A6=D1)).

This function works well enough for simple database information retrieval, as illustrated in Table 7.1, but more complex functions can be handled by block formulas (see page 9). A user seeking to do something like the report in D1..E6 should also consider QP's CrossTabs feature under Tools > Data Tools.

Averaging: @AVG, @PUREAVG

A similar function for averaging a set of numbers is `@AVG(`*`block`*`)`. `@AVG` ignores blank cells in the block, but if there are text cells in the block, `@AVG` treats them as zeros, which may cause unexpected errors. If the user wants to average all cells that contain numbers only in a block that contains text, use `@PUREAVG(`*`block`*`)` instead.

Extremes: @MAX, @MIN, @PUREMAX &-MIN, @LARGEST, @SMALLEST

Table 7.2 illustrates QP's functions for determining the maximum, minimum and average values in a block. As in the last case, text in the block is treated as a zero, which may yield unexpected results. If the block will include text, use `@PUREMAX` and `@PUREMIN` instead.

Table 7.2: Comparison of maximum, minimum, and averaging functions

	A	B	C	D	E	F	G	H	I	J
1					Avg	PureAvg	Max	Min	Puremax	Puremin
2		1	2	3	2	2	3	1	3	1
3	Text	1	2	3	1.5	2	3	0	3	1
4		-1	-2	-3	-2	-2	-1	-3	-1	-3
5	Text	-1	-2	-3	-1.5	-2	0	-3	-1	-3

Functions indicated in E1..J1 were typed into E2..J2, with a2..d2 as the argument, then copied, and then pasted below.

`@LARGEST(`*`Block,Rank#`*`)` allows the user to choose the largest, second largest, third largest, and so on, by entering a rank number of 1, 2, 3 and so on. `@SMALLEST(`*`Block,Rank#`*`)` does the same for the smallest numbers, by rank. If some of the values in `Block` are equal to one another, `Rank#` returns the first one, `Rank#+1` returns the second, etc.

Determining relative rank: @RANK

`@Rank(`*`Number,ArrayOfNumbers,Ascending?`*`)` returns the relative rank of a `Number` within an array of numbers (including a block containing numbers), based on either an ascending order or descending order), as illustrated in Table 7.3.

If the `Ascending?` argument is 0, the order is descending, and the greatest numeric value will receive a ranking of 1; the next greatest 2, etc. If the `Ascending?` argument is 1 or some other non-zero number, the order is ascending, and the greatest value will receive a ranking equal to the number of numbers in the array.

If `Number` is in `ArrayOfNumbers`, the function will return a Rank-number between 1 and the number of items in the array. If it is not in the array, the function returns 0.

Table 7.3: @RANK

	A	B	C
1		Ascending	Descending
2		1	0
3	5	1	6
4	10	2	5
5	15	3	4
6	20	4	3
7	25	5	2
8	30	6	1

B3 = @RANK($A3,$A$3..$A$8,B$2) copied to B3..C8

The `ArrayOfNumbers` does not need to be sorted; it can be in an entirely random order. This function will return rankings as if the array were sorted.

Rounding/Trimming: @ROUND, @INT, @TRUNC

QP has functions to modify numbers in useful ways.

`@ROUND(`*`Number,Magnitude`*`)` rounds a number to a given order of magnitude. If the magnitude is expressed in positive numbers, it sets the number of decimal points to which to round. If magnitude is expressed in negative numbers, it rounds to the nearest 10, 100, 1000, etc. Table 7.4 illustrates the levels at which `@ROUND` works.

Table 7.4: Rounding Methods

	A	B	C	D	E	F	G	H
1		3	2	1	0	-1	-2	-3
2	1234.5678	1234.568	1234.57	1234.6	1235	1240	1200	1000

B2 = @ROUND($A$2,B1) copied to C2..H2

In some earlier versions of QP, the following numbers do not equate to the formula `@ROUND(Number,2)`: 0.11, 0.22, 0.44, 0.63, 0.88, 1.01, 1.26, 1.51, 1.76, and so on. This problem appears fixed in QP17, if not earlier versions.

The rounding behavior can be modified by `@ROUNDDOWN` `@ROUNDDOWNXL`, `@ROUND-UP`, and `@ROUNDUPXL`.

`@INT` converts a number to an integer by eliminating digits after the decimal point. A variation on this function is `@TRUNC(`*`Number,Magnitude`*`)`, which allows the user to truncate digits at a specified number of digits.

Table 7.5 shows the comparative operation of `@INT`, and `@ROUND`, and `@TRUNC`.

Table 7.5: Comparison of @INT, @ROUND, and @TRUNC

	A	B	C	D	E	F	G	H	I
1	-1	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1
2	-1	0	0	0	0	0	0	0	1
3	-1	-1	-1	0	0	0	1	1	1
4	-1	-0.7	-0.5	-0.2	0	0.2	0.5	0.7	1

A2 = @INT(A1) A3 = @ROUND(A1,0) A4 = @TRUNC(A1,1)

Remainders and Patterns: @MOD

`@INT` allows the user to isolate in integer portion of a number from the number. To reverse that operation and get only the portion of the number following the decimal, this formula works:

```
+A1-@INT(A1)
```

Another more versatile function can do this as well: `@MOD(`*x*`,`*y*`)`. `@MOD` returns the remainder when `x` is divided by `y`. The decimal portion of numbers can be obtained by:

```
@MOD(A1,1)
```

By changing the `y` variable, `@MOD` can be used as a component of other functions to alter the results on a regular pattern, as will be seen in examples below. Treating every other row, or every seventh day, or every fourth quarter differently from the rest, or displaying a message every 100th cycle of a looping macro, can be accomplished by setting the `y` in an `@MOD` function to 2, 7, 4, or 100, respectively.

Quirk with negative numbers. When the first argument is a negative number, `@MOD` returns negative numbers (when it doesn't return zero). If this is undesirable, one can add to the first argument enough multiples of the second argument to make it a positive number and to yield the expected positive remainder value that `@MOD` returns. For instance, `@MOD(A1+(100*4),4)`, would yield the expected positive number even if the number in A1 could be as low as -400.

How to return the difference between two numbers if positive, or zero if negative

Assume that numbers are in cells A1 and B1, and that you want a function in C1 to show the difference, if A1 is greater than B1, or zero if it is not. These functions would work:

```
@IF(A1>B1,A1-B1,0)

@MAX(A1-B1,0)
```

How to return the difference between two numbers as a positive number

Assume that numbers are in cells A1 and B1, and that you want to return the difference as a positive number, but you don't know which is the greater. These functions would work:

```
@IF(A1>B1,A1-B1,B1-A1)

@MAX(A1-B1,B1-A1)

@ABS(A1-B1)
```

`@ABS` returns the absolute value any number, thus converting negative numbers to positive numbers.

How to mark the largest value in a column

One way to identify the largest value in a column is to use an array formula to wrap an `@IF` test, as in Table 7.6. In this example, the array function in B1 expands to B1..B10, and it applies the if-text to all cells in A1..A10, placing an asterisk in B1..B10 next to the cells in A1..A10 that are equal to the largest cell in A1..A10.

Table 7.6: Marking highest value

	A	B
1	13	
2	19	
3	35	
4	22	
5	54	
6	18	
7	94	*
8	77	
9	89	
10	47	

B1 = @ARRAY(@IF(A1..A10=@MAX(A1..A10),"*",""))

How to compare value changes from last year when this year is not complete

If you have a row of twelve monthly numbers for the preceding year in cells B1..M1 and a row of monthly numbers for the current year in B2..M2, and you want to add a function at N2 that compares this year's numbers with last year's by subtracting last year's numbers from this year's numbers, what should it be? If the years were both complete, the formula would simply be:

```
@SUM(B2..M2)-@SUM(B1..M1)
```

But that would be misleading if the current year is incomplete. So instead, we want a formula in N2 that compares months completed in this year with months completed in the prior year, a formula that will update as we fill results into B2..M2 for the current year.

We first need to identify the months in the prior year that we will need to compare with the current year. This will be equal to the number of months we fill in for the current year. That number can be found with the function `@COUNT(B2..M2)`. The addresses for the corresponding months in the prior year will be:

```
@OFFSET(B1,0,0,1,@COUNT(B2..M2))
```

The sum of those corresponding cells would then be:

```
@SUM(@@(@OFFSET(B1,0,0,1,@COUNT(B2..M2))))
```

The formula for the difference between the rows would then be:

```
@SUM(B2..M2)-@SUM(@@(@OFFSET(B1,0,0,1,
@COUNT(B2..M2))))
```

How to sum every other number in a column

In the sample spreadsheet below (Table 7.7), we want to put a formula in B13 that sums the numbers in the B column that stand opposite months in the A column. Or alternatively, how would one exclude those opposite "Check#" in the A column. What formula should be put into B13? (From WPU 36403; special hat tip to Jeff Barnes).

Table 7.7: Summing alternate numbers

	A	B
1	Jan	123.45
2	Check#	1002
3	Feb	132.54
4	Check#	1004
5	Mar	213.45
6	Chk#	1006
7	Apr	231.54
8	Chk#	1008
9	May	321.45
10	Chk#	1010
11	Jun	312.54
12	Chk#	1012
13		?

Method 1. The least elegant method is to put this formula into B13:

```
+B1+B3+B5+B7+B9+B11
```

For small numbers of entries, this is practical. For larger numbers, this is impractical.

Method 2. Using QP's array formulas and the fact that the numbers to be summed are on odd rows, this formula works:

```
@SUM((B1..B12)*@ISODD(@ROW(B1..B12)))
```

If instead we were summing the even rows, the formula would be:

```
@SUM(B1..B12*@ISEVEN(@ROW(B1..B12)))
```

Method 3. A variation on this approach that does not depend on whether the target data is on odd or even rows is this:

```
@SUM((B1..B12)*(@MOD(@ROW(B1..B12),2)
=@MOD(@ROW(B1),2)))
```

This last uses `@MOD` to set the test for a row to add based on whether the row number, divided by 2, has the same remainder as the first row's number, divided by 2. It thus counts values in every other row, beginning with the first row.

How to average the preceding three numbers in a column, not counting blanks.

Varying a problem raised in WPU 37720, lets assume that you have a column of numbers in column A, say A1..A100, and you want to create a parallel column of functions that average the three preceding numbers on and above the current row. If there were no gaps in column A, you could place in cell B3 the function `@AVG(A1..A3)`, and copy that function in parallel with numbers in column A (here, from B3..B100).

Now, let us add that column A may contain blanks in random cells. If the blanks occurred in a non-random pattern, if might be possible to create a function that could take account of that pattern, using `@IF` and `@PUREAVG`. But there is no predictable pattern of blanks in this example.

I believe that you need to create a helper column. Create that helper column by placing this formula in B1:

```
@ARRAY(((@ISBLANK(A1..A100)=0)*@ROW(A1..A100)))
```

This creates an array in B1..B100 that is populated by zeros or row numbers, depending on whether there is a number in the row in column A.

With this helper column in place, there are two solutions. The first solution involves finding the desired numbers in column A, adding them, and dividing by 3. The second solution involves identifying the block in which those cells occur and using the `@PUREAVG` function.

Solution 1: The first solution is to place this formula in cell C100:

```
(@INDEX(A$1..A100,0,@LARGEST(B$1..B100,1)-1)
+@INDEX(A$1..A100,0,@LARGEST(B$1..B100,2)-1)
+@INDEX(A$1..A100,0,@LARGEST(B$1..B100,3)-1))/3
```

Then copy this function from C100 to cells above it. The copied functions will return ERR when there are no longer 3 values to add, but otherwise, it will average the three. The formula can also be copied down.

The three `@LARGEST` functions will return the first, second and third largest row numbers, which will be the rows of the desired numbers. Each of those is used as the row-offset in an @INDEX function, but since `@ROW` in the helper column starts numbering with 1, but `@INDEX` starts numbering with 0, we have to subtract 1 from the row number. `@INDEX` then returns the numbers from the desired cells, each of which are added, and the sum divided by three.

Solution 2: Place this formula in cell C100:

```
@PUREAVG(@@(@OFFSET($A$1,@LARGEST(B$1..B100,3)-1,0,
(@ROWS(A$1..A100)-@LARGEST(B$1..B100,3)+1),1)))
```

and copy it up to the cells above.

Here, the block in which the desired numbers appear will be calculated by the `@OFFSET` function, converted from text into usable coordinates by the `@@` function, and then averaged with the `@PUREAVG` function. So, the trick will be getting `@OFFSET` to identify the correct block. The `@OFFSET` function has the anchor point (A1), the column offset (0) and the column width (1), so it needs only the number of rows to drop down below the anchor point to find the starting cell, and the number of rows of cells to include. The number of rows to drop down will be the third largest number less one (since `@OFFSET` like `@INDEX` starts numbering at 0), and this number is supplied by `@LARGEST(o$1..o20,3)-1`. The number of rows to be included will be the total number of rows less the starting point, plus one, and this number is supplied by `(@ROWS(n$1..n20)-@LARGEST(o$1..o20,3)+1)`.

Other variations on this problem are addressed on the web page listed at the start of this article.

How to sum the contents of a column if rows or columns are inserted

For a standard sum of numbers in a column, say A1..A10, placing this in A11 works: `@SUM(A1..A10)`. If a user inserts a row between A1 and A10, QP will adjust this formula automatically.

But what if the user is going to insert a row *between the current values in A10 and A11*? The simple `@SUM` function will still relate to A1..A10 and exclude the value in (now-inserted) A11 from the function that is now in A12. However, the following formula in A11 will allow other rows to be inserted above the function, and it will sum the numbers from A1 to the cell above this function.

```
@SUM(@@(@OFFSET(A1,0,0,@ROW-1,1)))
```

But what if a row is inserted *above A1*? Doing that has the effect of changing the A1 in this formula automatically to A2, and because the size of the block created by `@OFFSET` doesn't change, it defines a block that includes this formula. So, instead, we need a function that sets the A1 cell "in concrete," so to speak. Here are two ways of doing so:

```
@SUM(@@(@OFFSET(@@("a1..a1"),0,0,@ROW-1,1)))

@SUM(@@(@CONCATENATE("a1..a",@ROW-1)))
```

Both create the coordinates of a block entered in text format and anchored in cell A1, which are converted to coordinates by the `@@` function before being summed by `@SUM`.

But what if the user might *insert a column* before that column? In that case, the functions that link only to cell A1 would give false answers. Instead, we need a function that links always to the first cell in the column in which the function appears. Unfortunately, `@COLUMN` doesn't work the same way as `@ROW` for reasons that I do not understand, so the cases are not parallel. But `@CELL("col")` gives the same result, so to get the current column for a given function, use:

```
@CELL("col",c(0)r(1))
```

Parenthetically, instead of `c(0)r(1)`, we could type the coordinates of any cell in the same column, but using this method allows us to put the formula into any cell anywhere, and QP will automatically replace it with the cell below the one in which it is pasted. Further parenthetically, I prefer `c(0)r(1)` over `c(0)r(0)`, because its use would cause the circularity indicator to activate.

The last function gives us a column number that will update whenever columns are inserted or deleted. The function `@INDEXTOLETTER` will convert that column number to a letter (see discussion below at 79), but we need to subtract 1 from the column number because `@CELL` returns 1 for column A, but `@INDEXTOLETTER` converts 0 to A, 1 to B, etc.

```
@INDEXTOLETTER(@CELL("col",c(0)r(1))-1)
```

We now have each of the components to create the coordinates of the desired block:

```
@CONCATENATE(@INDEXTOLETTER(@CELL("col",
c(0)r(1))-1),"1..",@INDEXTOLETTER(@CELL("col",
c(0)r(1))-1),@ROW-1)
```

When wrapped with `@SUM` and `@@`, this formula sums the contents of the current column, from the top to the cell above the function, no matter how many rows or columns are inserted or deleted.

```
   @SUM(@@(@CONCATENATE(@INDEXTOLETTER(@CELL("col",
   c(0)r(1))-1),"1..",@INDEXTOLETTER(@CELL("col",
   c(0)r(1))-1),@ROW-1)))
```

Chapter 8

Date and Time Functions

Before getting into the functions, the user should understand how QP understands time.

QP treats each date as a separate integer. The date associated with 0 is (oddly) December 30, 1899. December 31, 1899, is 1; January 1, 1900 is 2, and so forth. January 1, 2015 is 42005. Possible numbers stretch back as far as January 1, 1600 (-109571) and as far forward as December 31, 3199 (474816). Those integers represent midnight at the start of the date in question. In the following, "**Date#**" refers to the integer that QP uses to express a date.

QP treats time as a decimal fraction of the integer. One hour amounts to 1/24th of an integer, or 0.0416667. One minute represents 1/60th of an hour, or 0.0006944. One second represents 1/60th of a minute, or 0.0000115741. Zero is the decimal fraction for midnight, 0.25 for 6:00 A.M., 0.5 for noon, 0.75 for 6:00 P.M., and so forth.

Current date and time: @TODAY, @NOW

`@TODAY` returns the current date as an integer. Yesterday is `@TODAY-1`; tomorrow is `@TODAY+1`.

`@NOW` returns the current date and time as a combination of the date integer and the number of seconds since midnight, expressed as a decimal fraction of an integer. Experimentation with `@NOW` (by entering its value in adjacent cells, getting the differences, and multiplying by 60 x 60 x 24, with the result that the difference are always extremely close to integers) reveals that QP refreshes the `@NOW` function once a second. Greater precision is not available by `@NOW`. Greater precision can be had, however, by a PerfectScript macro (see page 169).

It follows that one can extract the decimal fraction representing only the time by the formula `@NOW-@TODAY`. Another function that does the same thing is `@MOD(@NOW,1)`.

Dates and their components: @DATE, @YEAR, @MONTH, @DAY

`@DATE(`*`Year#,Month#,Day#`*`)` returns the Date# for the date made up of those arguments. The `Month#` argument takes a value from 1 to 12, and `Day#` takes a value from 1 to 31, for obvious reasons, though if 31 is the `Day#` argument with reference to a month that has less than 31 days, the function returns ERR. The `Year#` argument warrants additional comment. I recommend using the 4-digit year number in order to avoid confusion. You may use the 2-digit year number, but be aware that doing so lost its utility at the turn of the last century. QP treats the year 15 as 1915, not 2015. QP will treat 105 as 2015, but the confusion in using that approach more than offsets the time saved in typing the 4-digit number.

Conversely, `@YEAR(`*`Date#`*`)`, `@MONTH(`*`Date#`*`)`, and `@DAY(`*`Date#`*`)` take the date integer and return the year number, the month number, and the day number. Again, `@YEAR` does not return a number equal to the calendar year. It regards 1900 as year 0, and therefore using `@YEAR` on a date integer in 2015 yields the number 105. To convert it to a current date requires the addition of 1900.

To adjust the function to a current calendar year, the user must add 1900, as in the formula `@YEAR(@TODAY)+1900`. Conversely, to adjust a current calendar year to the results of the `@YEAR` function, the user must subtract 1900, as in the formula `@YEAR(@TODAY)-1900`. To use the prior illustration in Table 2.5 on page 13, a way of summing all values in 2012 would be:

```
@SUM((B1..B12)*(@YEAR(A1..A12)=(2012-1900)))
```

or more simply,

```
@SUM((B1..B12)*(@YEAR(A1..A12)=112))
```

Quirk. When using `@YEAR` in a block formula, like those just given, if there is a text entry in the block, the function returns ERR. The same is not true when a block formula simply compares each item with a particular date value.

Information about a date: @DATEINFO

This function returns a lot of useful information about a date number, as Table 8.1 illustrates.

Table 8.1: Values returned by @DATEINFO

	A	B	C
1		2/12/2015	
2	1	Thu	Day of the week (short)
3	2	Thursday	Day of the week (full)
4	3	3	Day of week, 0=Mon to 6=Sunday
5	4	7	Week of the year, from 1 to 53
6	5	Feb	Month (short)
7	6	February	Month (full)
8	7	28	Number of days in this month
9	8	16	Number of days left in the month
10	9	42063	The last day (integer) in the month
11	10	1	Quarter of the year (1 to 4)
12	11	0	Whether a leap year (1=yes; 0=no)
13	12	43	Day of the year (1 to 366)
14	13	322	Days left in the year

B2 = @DATEINFO($B$1,A2)   copied to B3..B14

Quirks. It is unfortunate, then, that `@DATEINFO` does not play well with other functions, such as `@CONCATENATE`. If the output of `@DATEINFO` is put into cells, they can be joined by `@CONCATENATE`. But the function returns nothing inside `@CONCATENATE`. Perhaps that quirk is related to the quirk that QP wraps `@DATEINFO` in an `@ARRAY` function for reasons that are opaque to me. However, in lieu of `@CONCATENATE`, these items can be joined in an `@ARRAY` function using ampersands to combine them.

Thus, in this example,

```
@ARRAY("A "&@DATEINFO(A1,2)&" in "&@DATEINFO(A1,6))
```

returns: *A Thursday in February*.

Days of the week: @WEEKDAY, @WKDAY

On the same subject, `@WEEKDAY(`*`Date#`*`)` returns 1=Sunday, 2=Monday,...7 =Saturday. That is its default setting; other options allow 1–7 or 0–6 for Monday through Sunday. `@WKDAY(`*`Date#`*`)` returns 1–7 for Saturday through Friday.

Comparing dates: @DATEDIF

This isn't an -IF function, but a -DIF function, counting the difference between two dates in one of six optional ways, so the format is `@DATEDIF(`*`StartDate,End-Date,Option`*`)`. QP's help file describes the options as shown in Table 8.2.

Table 8.2: @DATEDIF Options

y	Years
m	Months
d	Days
md	Days, disregarding months and years
ym	Months, disregarding years
yd	Days, disregarding years

The user should be cautious in using it, because it can express the difference between two numbers in unexpectedly negative terms, as Table 8.3's illustration of three days around the 14th anniversary of a start date shows:

Table 8.3: @DATEDIF Return Values

	A	B	C	D	E	F	G	H
1	Start Date	8/8/2000	y	m	d	md	ym	yd
2	End Date #1	8/7/2014	13	168	5112	-1	11	-2
3	End Date #2	8/8/2014	14	168	5113	0	0	-1
4	End Date #3	8/9/2014	14	168	5114	1	0	0

C2= @DATEDIF($B$1,$B2,C$1) copied to C2..H4

See the discussion in WPU 33157.

Other date functions

There are also many other functions handling specific needs, such as business days, holidays, 360-day years, first days, last days, and so on. For instance:

- `@AMNTHS(@TODAY,6)` returns the Date# that is 6 months from today.
- `@EMNTH(`*`Date`*`)` returns the Date# of the last day of the month.
- `@WORKDAY(@TODAY,10)` returns the Date# that is 10 working days later than today (the QP help file appears to misstate the nature of the functions and some optional arguments).
- `@NWKDAY(nth,WkDay#,Mo#,Year#)` returns the Date# of the `nth` (1 to 5) `WkDay#` (1=Saturday to 7=Friday) of `Mo#` in `Year#`. If there is no such date, it returns `ERR`.

The user should consult the help file for more functions.

Time and components: @TIME, @HOUR, @MINUTE, @SECOND

Turning to the more important Time functions, in parallel with the date functions, we can return the Time# with `@TIME(`*`Hourr#,Min#,Sec#`*`)`. `Hour#` runs from 0 to 23; `Min#` and `Sec#` both run from 0 to 59. Likewise in parallel with the date functions, `@HOUR(`*`Time#`*`)`, `@MINUTE(`*`Time#`*`)`, `@SECOND(`*`Time#`*`)` can derive the hour, minute, and second numbers for a give Time#.

How to display a message after a certain time of the day

To have a function display a message at a particular time of the day, you can use the `@TIME` function to set the time, and use `@NOW` and `@TODAY` to check the current time. Thus, to have a cell display at 5:30PM, the baseline time is `@TIME(17,30,0)`. To determine the current time, `@NOW-@TODAY` gives the current time (as does `@MOD(@NOW,1)`). Therefore, a function like this can be put into a prominent cell:

```
@IF((@NOW-@TODAY)>@TIME(17,30,0), "Time to go home!","")
```

Naturally, there are better clocks around, and this one only activates when QP refreshes the screen, so if you're not working on the spreadsheet, the function will not cause the message to appear in the cell. Still, the idea might be useful in other contexts.

How to sum data by year

Assume that you have sales data for several years on a sheet called Sales. Column A contains the date of sales in rows 1 to 1000 and column B contains the amount of sales on those dates. You need to sum the sales for the year 2015. What formula do you need? Try this one:

```
@SUM((Sales:B1..B1000)*(@YEAR(Sales:A1..A1000)
=2015-1900))
```

This is a straightforward application of block formulas and the `@YEAR` function, which regards the year 1900 as year 0, and which therefore requires the user to subtract 1900 from the desired year to get the intended results.

The user may also find it more desirable to type the desired year into a cell and have the formula recalculate, rather than to discover the year coded into the formula. Say that the year will be in cell Sales:C1. This formula will return the total of sales for the year typed into C1:

```
@SUM((Sales:B1..B1000)*(@YEAR(Sales:A1..A1000)
=Sales:C1-1900))
```

How to sum data by month and year

In this example (Table 8.4), the task is to set up formulas that will take the numbers in the B column and use their dates in the A column to sum them by month and year in C1..F13.

Table 8.4: Summing Data by Month and Year

	A	B	C	D	E	F
1	03/27/13	71		2013	2014	2015
2	05/18/13	71	1	0	0	0
3	08/23/13	68	2	0	117	78
4	08/24/13	28	3	71	59	0
5	09/30/13	40	4	0	0	0
6	10/03/13	47	5	71	58	0
7	12/11/13	30	6	0	0	0
8	12/12/13	30	7	0	65	0
9	02/03/14	30	8	96	26	0
10	02/20/14	46	9	40	0	0
11	02/26/14	41	10	47	48	0
12	03/16/14	59	11	0	54	0
13	05/25/14	58	12	60	58	0
14	07/16/14	65				
15	08/23/14	26				
16	10/13/14	48				
17	11/08/14	54				
18	12/08/14	58				
19	02/06/15	35				
20	02/11/15	43				

To do this, we'll need to set up a formula in D2 that sums the B column depending on what `@MONTH` and `@YEAR` return on values in the A column, as compared with the month numbers in C2..C13 and the year numbers in D1..F1.

In D2, we'll use `@MONTH` to compare the dates in A1..A20 with cell C2. When we copy that function to the rest of the grid, we'll want it to vary with the row, but not with the column, so that we're always testing against a value in C2..C13. We do that by specifying the comparison value as $C2.

We'll also use `@YEAR` to compare the dates in A1..A20 with D1. As we copy the formula to the rest of the grid, we'll want it to vary with the column, but not the row. We do that by specifying the comparison value as D$1. But as we've noted, `@YEAR` returns numbers 1900 less than our calendar years, so we'll need to subtract 1900 from the value in D1.

And as we copy the formulas around, we'll need to make sure that they refer to the same data in columns A and B, so those will need to be made absolute. Accordingly, the formula to put into cell D2 is:

```
@SUM(($B$1..$B$20)
*(@MONTH($A$1..$A$20)=$C2)
*(@YEAR($A$1..$A$20)=D$1-1900))
```

Copying that formula to D2..F13 puts the formulas in those cells that correctly return the numbers set out above. A later section (page 152) shows how to use the `{PutBlock}` macro command to fill this table in one step.

Caution. If Columns A and B had text headers, like "Date" and "Number" in A1 and B1, the `@YEAR` function would ERR. In that case, all of the blocks would need to be changed to go from row 2 to row 20.

How to determine the Tuesday after or before a date

The first task is to build a formula that will take a date and determine the date of the Tuesday following it. As constants, we need only the return values of the `@WEEKDAY` function, which returns 1 for Sunday through 7 for Saturday. 3 is the number for Tuesday. Let's assume that the baseline date is in A1, so we want a formula that will yield a positive number between 1 and 7 that, when added to A1, yields a date that is a Tuesday.

A plausible try is the formula

```
+A1+@MOD(7+3-@WEEKDAY(A1),7)
```

which works if we seek the Tuesday "on or after" a certain date, but the task here is to find the Tuesday "after" that date. The core idea of this function is to add to the baseline date in A1 the difference between 3 (Tuesday's number) and the number that `@WEEKDAY` returns for A1. If that return value were 2 or 1, nothing further would be necessary, but because `@WEEKDAY` could return a higher number than 3, and to avoid a negative number here, we add 7 to that difference, but because the resulting number could exceed 7, we use the `@MOD` function with a denominator of 7 to bring the number within the desired range. The problem, though is that if the date in A1 is a Tuesday, in which case the `@MOD` formula returns 0, not 7, and we get only the date in A1, not the *next* Tuesday.

We need a formula that will return a value of 7; subtracting one `@WEEKDAY` value from another will not work, because the maximum range is 1 to 7, so the maximum gap is 6. Instead, we must add 7, and then subtract a number from 0 to 6. That yields this formula:

```
+A1+7-@MOD(@WEEKDAY(A1)-3+7,7)
```

In this case, we add 7 to the baseline in A1, and then subtract the difference between the number returned by `@WEEKDAY` for A1 and Tuesday's number 3, reversing the order of the first try. But because that difference could again yield a negative number, and we need to subtract a positive number, we again add 7, but use `@MOD` to keep the number subtracted from 7 within the range of 0 to 6. Incidentally, if we wanted to get the tenth Tuesday after the date in A1, we would change the +7 to +(10*7).

We turn now to the task of getting the Tuesday before the date in A1. From the preceding discussion, we know in advance that we need to begin by subtracting 7 from that date, and then adding back a number from 0 to 6. Reversing the thinking, we get the following function:

```
+A1-7+@MOD(7+3-@WEEKDAY(A1),7)
```

The number added back after subtracting 7 is the difference between Tuesday's number and the number returned by `@WEEKDAY` for A1, to which we add 7 to ensure that we get a positive number, but use `@MOD` to reduce the number to the range of 0 to 6. This exercise was prompted by the thread in WPU 16834, in which the question was how to determine ten Tuesdays before. To do that, substitute -(10*7) for -7 in the last formula.

How to set up delivery tables

This example combines several features above with some of the unusual date functions that QP provides., namely `@NWKDAY(`*`Week#,DayOfWeek, Month#,Year#`*`)`, which returns the date on which the first, second, etc. specified day of the week falls in a given month and year (e.g., the third Thursday in March of 2015, and `@LWKDAY(`*`DayOfWeek,Month#,Year#`*`)`, which returns the date of the last day in a given month that falls on a specified day of the week.

Varying a question in WPU 36825, a supplier has to make deliveries to one of twenty-four sites (A..X) on a particular schedule and needs to determine how many deliveries will be made in the next seven days in order to make necessary preparations. How do we take a delivery schedule and identify the dates and locations of the deliveries? Table 8.5 shows one way.

The basic data appears in columns B (the name of the delivery site), C (the week of a delivery, from one to four, or 'L' for the last week), and D (the day of delivery, from 1 to 7, with 1=Saturday, 2=Sunday, etc.).

From the data in columns C and D, one needs only the year and month to determine delivery dates with the `@NWKDAY` and `@LWKDAY` functions. The E column should contain dates for the current month using `@MONTH` and `@TODAY`, and the current year can be determined by `@YEAR` and `@TODAY`. But because the 7 day period may stretch into the next month, we'll let column F serve for the next month, and because the next month may also be in the next year, we must take account of that as well. That will be done with formulas in E2..F3.

Finally, we can place a formula in the A column that will display delivery dates that occur in the next 7 days, and that otherwise remains blank. The entries will tell us visually show us when and where to make deliveries in that period.

The cells highlighted in blue contain the functions that, when copied to the yellow cells, operate the sheet. `@YEAR(@TODAY)+1900` is in E2, `@MONTH(@TODAY)` in E3, `@IF(E3=12,E2+1,E2)` in F2, and `@IF(E3=12,1, E3+1)` in F3. E4 contains `@NWKDAY($C4,$D4,E$3,E$2)`, which is then copied to E4..F25. E26 contains `@LWKDAY ($D26,E$3,E$2)`, which is then copied to E26..F27.

A4 contains the formula that yields the desired results:

Table 8.5: Delivery Schedules

	A	B	C	D	E	F
1					Cur. Month	Next Month
2	Deliveries				2015	2015
3	Date	Site	Wk	Day	3	4
4		Site A	1	4	03/03/15	04/07/15
5		Site B	1	5	03/04/15	04/01/15
6		Site C	1	6	03/05/15	04/02/15
7	03/10/15	Site D	2	4	03/10/15	04/14/15
8	03/10/15	Site E	2	4	03/10/15	04/14/15
9	03/11/15	Site F	2	5	03/11/15	04/08/15
10	03/11/15	Site G	2	5	03/11/15	04/08/15
11	03/12/15	Site H	2	6	03/12/15	04/09/15
12	03/12/15	Site I	2	6	03/12/15	04/09/15
13		Site J	3	3	03/16/15	04/20/15
14		Site K	3	4	03/17/15	04/21/15
15		Site L	3	4	03/17/15	04/21/15
16		Site M	3	5	03/18/15	04/15/15
17		Site N	3	6	03/19/15	04/16/15
18		Site O	3	6	03/19/15	04/16/15
19		Site P	3	7	03/20/15	04/17/15
20		Site Q	4	3	03/23/15	04/27/15
21		Site R	4	4	03/24/15	04/28/15
22		Site S	4	5	03/25/15	04/22/15
23		Site T	4	5	03/25/15	04/22/15
24		Site U	4	5	03/25/15	04/22/15
25		Site V	4	6	03/26/15	04/23/15
26		Site W	L	3	03/30/15	04/27/15
27		Site X	L	4	03/31/15	04/28/15

```
@IF(@SUM((E4..F4>@TODAY)*(E4..F4<=@TODAY+7)),
@SUM((E4..F4)*(E4..F4>@TODAY)*(E4..F4<=@TODAY+7)),"")
```

which is then copied to A5..A27. It displays delivery dates in the next seven days, if any, and otherwise displays a blank. If there is a date within the next 7 days in cells E4 or F4, the first `@SUM` function will return 1, otherwise it will return 0. If the latter, the function returns a blank, but if the former, the second `@SUM` function returns that date.

How to determine the number of Wednesdays in a given month

The question was raised about how to determine how many Wednesdays there are in a given month (from WPU 36936). I have three solutions. In each, the year (e.g., 2015) is in cell A1, and the month (e.g., 4) is in cell B1.

Solution 1: @NWKDAY

This solution uses `@NWKDAY` to determine the date of the 5th Wednesday in a given month. If there are 5 Wednesdays, it returns a date; if not, it returns `ERR`. Since no month has less than 28 days nor more than 31, there will either be 4 or 5 Wednesdays in the month, so `@ISERR` will help us decide. This function returns the answer.

```
@IF(@ISERR(@NWKDAY(5,5,B1,A1)),"4 Wednesdays",
"5 Wednesdays")
```

The first `5` seeks to find the 5th instance of the second `5`, which is Wednesday on the scale of Saturday=1 to Friday=7. If there is no 5th Wednesday, ERR results, and this function returns “4 Wednesdays”; otherwise it returns “5 Wednesdays”.

Solutions 2: Pseudo-column of dates

Using QP’s array capabilities, this one uses the trick of creating a pseudo-column of dates corresponding to the days of this month, on which to test whether each is a Wednesday. See the earlier discussion of pseudo-columns at page 44. Here’s how we can do it:

1. We can determine the first day of the month using `@DATE(A1,B1,1)`. This is used three times in the function, so it could be placed in a different cell, and the reference to that cell could be substituted in this function.
2. We determine the last day of the month using the `@EMNTH` function. It requires a date with which to look up the last day of the month, so I use the first day again.

   The difference between these two numbers is the number of days in the month.
3. We now wish to create an array of date-numbers representing each day of this month for testing.
   a) Absent a better way, we first create an imaginary column of cells to house these date numbers. Using the first and last day numbers, this block would house those numbers.

      ```
      @OFFSET(A1,0,0,LastDay-FirstDay+1,1)
      ```

      or

      ```
      @OFFSET(A1,0,0,@EMNTH(@DATE(A1,B1,1))
      -@DATE(A1,B1,1)+1,1)
      ```

      In the case of April, that column would be A1..A30. For 31-day months, it would be A1..A31. These would be text. Converting the text block to non-text coordinates requires wrapping it in `@@`.

      ```
      @@(@OFFSET(A1,0,0,@EMNTH(@DATE(A1,B1,1))
      -@DATE(A1,B1,1)+1,1))
      ```

   b) Wrapping the resulting function inside `@ROW()` yields an array numbering 1..30 in a column of 30 cells. QP automatically wraps that in an `@ARRAY()` function.

      ```
      @ARRAY(@ROW(@@(@OFFSET(A1,0,0,
      @EMNTH(@DATE(A1,B1,1))-@DATE(A1,B1,1)+1,1))))
      ```

c) Converting that array to the array of date numbers in the current month, we add the FirstDay-1 to each.

```
@ARRAY(@ROW(@@(@OFFSET(A1,0,0,
@EMNTH(@DATE(A1,B1,1))-@DATE(A1,B1,1)+1,1)))
+@DATE(A1,B1,1)-1)
```

That gives us an array of 30 days matching the 30 days in this month.

d) Applying @MOD(#,7) to each number in the array, each becomes a number between 0 and 6 (Saturday through Friday).

```
@ARRAY(@MOD(@ROW(@@(@OFFSET(A1,0,0,
@EMNTH(@DATE(A1,B1,1))-@DATE(A1,B1,1)+1,1)))
+@DATE(A1,B1,1)-1,7))
```

e) Testing whether each is equal to 4 (Wednesdays) yields a series of 0s and 1s.

```
@ARRAY(@MOD(@ROW(@@(@OFFSET(A1,0,0,
@EMNTH(@DATE(A1,B1,1))-@DATE(A1,B1,1)+1,1)))
+@DATE(A1,B1,1)-1,7)=4)
```

4. Replacing the `@ARRAY()` wrapper with `@SUM()` totals the number of 1s, which happens to total 5 for this April of 2015.

```
@SUM(@MOD(@ROW(@@(@OFFSET(A1,0,0,
(@EMNTH(@DATE(A1,B1,1))-@DATE(A1,B1,1))+1,1)))
+@DATE(A1,B1,1)-1,7)=4)
```

Upon changing the year in A1 and month in B1, this function returns the correct number of Wednesdays for the chosen year and month.

Solution 3: Find next weekday, then calculate

This technique uses the method described above (page 63) for finding the next weekday after a date, except that it is modified to find the next weekday on or after a date. The formula for finding the next Wednesday on or after a date, using the scale of Saturday=0 ... Friday=6, is this:

```
+Date#+@MOD(7+Weekday#-@WEEKDAY(Date#),7)
```

so in the case of the first Wednesday in April of 2015:

```
+@DATE(2015,4,1)
+@MOD(7+4-@WEEKDAY(@DATE(2015,4,1)),7)
```

This starts the count at 1, and the question will be how many more exist between the first date and the end date. This turns out to be the integer portion of (the difference between the end date and the start date) divided by 7. For months, it will be safe to use `@INT`, since there are going to be at least some additional Wednesdays, but if we want to use this formula on different dates in which there might not be a Wednesday, `@ROUNDDOWN` will give the correct result.

For shorthand, let us store the Year# (2015) in A1, the Month# (4) in A2, the Weekday# (4) in A3. In A4, we place the first day of the month, using the function `@DATE(A1,A2,1)`. In A5, we place the last day of the month, using the function `@EMNTH(A4)`. This formula returns the number of the Weekday# in that calendar month:

```
+1+@ROUNDDOWN((A5-A4+@MOD(7+A3-@WEEKDAY(A4),7)))/7)
```

Chapter 9

Converting between Text and Numbers

There are many reasons why one might want to convert numbers (including dates, times, and currency) into text, and vice versa. QP supplies a number of functions for doing so.

Converting numbers to generic number-like text: @STRING, @FIXED

Before dealing with functions that convert a number to text, it may be sufficient simply to display the number in a different format. Thus a cell, block, column, or row may be selected, and the user can get a dialog box for the properties of the selected cell(s) by clicking `[F12]`, or right-clicking and choosing "Selection Properties", or clicking Format > Selection Properties on the menu. The second tab provides all of QP's built-in numeric formats, and upon choosing one of them, the selected cells will be rendered with the chosen format. If you want a format other than QP's options, the last option ("Custom") allows you to build your own "Custom" format. Details on doing so are in an appendix on custom numeric formats on page 76.

For converting numbers to generic number-like text, these functions are available:

`@STRING(`*`Number,Decimal`*`)` converts a number into text, using general format, and requiring the user to specify the number of decimal points to which to round the number. The QP help file erroneously states that, apart from the number-like formats available from `@STRING`, conversion to any other requires use of a macro involving the `{Contents}` command, which places the as-formatted number in one cell as text into another cell. Not only do other functions in this chapter convert to other number-like formats, but still other formats can be created using `@CONCATENATE`. Even so, the macro is often a desirable option, as developed below at page 158. I would applaud Corel if it developed a new function that would do precisely what `{Contents}` does.

`@FIXED(`*`Number,Decimal,NoCommas`*`)` converts a number into text, using the currency format, and allowing the user to specify the number of decimal points to which to round the number. If the `Decimal` argument is omitted, QP assumes the number is 2. `@FIXED(1234.567)` and `@FIXED(1234.567,2)` will both yield *1,234.57*. If you do not desire to show the decimal separator, adding the additional `NoCommas` argument 1 will do so. Thus `@FIXED(1234.567,2,1)` will yield *1234.57*. As far as I can tell, you might as well use `@STRING` for this.

`@CONCATENATE(`*`Number`*`)` converts the number in its general format into text. If some other format is desired, a different function must be used.

Table 9.1 illustrates how the functions round the number 1234.567.

Table 9.1: Rounding with @STRING and @FIXED

Command	Result
@STRING(1234.567,2)	1234.57
@FIXED(1234.567)	1,234.57
@FIXED(1234.567,2)	1,234.57
@FIXED(1234.567,2,1)	1234.57

Converting numbers to currency text: @DOLLAR, @DOLLARTEXT

There are two functions for converting a number into currency text.

`@DOLLAR(`*`Number,Decimal`*`)` converts a number into text, using the currency format, and allowing the user to specify the number of decimal points to which to round the number. If the `Decimal` argument is omitted, QP assumes the number is 2. Thus `@DOLLAR(12.34)` and `@DOLLAR(12.34,2)` will both yield *$12.34*. `@DOLLAR(12.5,0)` would round to *$13*. It displays commas as thousands separators.

`@DOLLARTEXT(`*`Number,FormatOption`*`)` spells out the numeric value in words suitable for checks and similar instruments. Format options range from 1 to 5, as shown in Table 9.2.

Table 9.2: @DollarText Options

Command	Result
@DOLLARTEXT(12.34,1)	Twelve
@DOLLARTEXT(12.34,2)	Twelve Dollars
@DOLLARTEXT(12.34,3)	Twelve and 34/100
@DOLLARTEXT(12.34,4)	Twelve and 34/100 Dollars *(the default)*
@DOLLARTEXT(12.34,5)	Twelve Dollars and Thirty-Four Cents

`FormatOption` 4 is the default format if you omit it in writing the function. Interestingly, the `FormatOption` must be hard-coded into the function; a reference to a cell with the argument produces an ERR.

Other conversions: @FRACTION, @CHAR

Other number-to-text conversion functions exist:

`@FRACTION(`*`Number,Denominator`*`)` converts a number into a fraction with a maximum (not minimum) denominator specified in the `Denominator` argument, as shown in Table 9.3.

Table 9.3: @FRACTION Options

Command	Result
@FRACTION(1234.567,2)	1234 1/2
@FRACTION(1234.567,3)	1234 2/3
@FRACTION(1234.567,4)	1234 1/2
@FRACTION(1234.567,5)	1234 3/5
@FRACTION(1234.567,6)	1234 1/2
@FRACTION(1234.567,7)	1234 4/7

`@CHAR(`*`Number`*`)` converts a number from 1 to 255 into the ANSI character associated with that number. Thus, `@CHAR(32)` is a space; `@CHAR(34)` is a double-quote. The converse conversion function is `@CODE(`*`Character`*`)`.

Converting text to numbers: @VALUE, @DATEVALUE, @TIMEVALUE

For converting text to numbers, these functions are available:

`@VALUE(`*`Text`*`)` takes number-like text and converts it into a value. The function returns ERR if the `Text` argument contains non-number-like text. Text can be converted into a value if it contains digits, a single decimal character, thousands separators, the dollar sign at the beginning, and spaces before or after the number. Other characters in `Text` will prevent `@VALUE` from working.

`@DATEVALUE(`*`DateText`*`)` converts text into date numbers, but it is very selective about the format of the date text. The QP help file says that there are only five valid formats, and then it lists seven. Several of them make assumptions about the year (current) or date (first of the month). There are essentially two text formats that allow `@DATEVALUE` to work without assumptions:

- `d-mmm-yy` (or `d-mmm-yyyy`). For example, “4-feb-15 or “4-feb-2015”.
- The long international date format, which can be changed in the Windows operating system, but which is usually `m/d/yy` (or `m/d/yyyy`). For example, “2/4/15” or “2/4/2015”. If that format has been changed in the Windows operating system, this format must be changed too.

If the text is not in one of those formats, it must be put into one of those formats to be useful, either by the source of the text or by an @function that transforms it into one of the acceptable formats. Examples of such @functions are below.

`@TIMEVALUE(`*`TimeText`*`)` converts text into time numbers, but it is also very selective about the format of the time text. There are only four valid formats, but they boil down to:

• `h:m:s am/pm`. Some notes: The hour (`h`) component can be any number from 0 to 12, but 0 and 12 are deemed to refer to the same hour. The hour can also be 13 to 23. The seconds component (`:s`) is optional; in its absence, the number is assumed to be 0. The trailing “`am`” or “`pm`” disambiguates the hour and controls in case of conflict with the hour.

• The long international time format, which is normally `hh:m:s` where hh represents the hour on a scale of 0 to 23 (12AM to 11PM). As above, the `:s` component is optional.

As in the case of `@DATEVALUE`, if the source text does not fit one of these formats, it must be converted by @functions.

How to convert a number to left-padded text

To convert a number, such as 9, to left-padded text, such as “00009”, we will need a formula. Several perform the trick. In this example, assume that the number to be padded is in cell A1 and the number of characters the label is in cell B1. These formulas work:

```
@REPEAT("0",B1-@LENGTH(@STRING(A1,0)))&@STRING(A1,0)

@SUBSTITUTE(@SETSTRING(@STRING(A1,0),B1,2)," ","0")

@RIGHT("0000000000000000000"&@STRING(A1,0),B1)

@RIGHT(@CONCATENATE(@REPEAT("0",B1),A1),B1)
```

And, of course, the user can create a custom numeric format. See the discussion in WPU 30801.

How to convert numeric text formatted as #- into usable numbers

In this problem from WPU 36703, the user gets a file containing negative numbers formatted with the negative sign *after* the number, rather than before. QP would treat numeric-text-with-the-negative-sign-in-front as a number, but it treats numeric-text-with-the-negative-sign-after-the-number as text. So the user needs a function to convert the latter number-like text into numbers on which mathematical operations can occur. Let’s assume that this entry will appear in cell A1. The function that will turn such numeric text into a value is:

```
-@VALUE(@LEFT(A1,@LENGTH(A1)-1))
```

This function isolates the numeric part with the `@LEFT` function, converts it to a number with the `@VALUE` function, and makes it negative by the leading negative sign.

However, since some of the numbers may be properly formatted positive numbers, which QP will place as a number in A1, and the function won’t know in advance whether it has a number or text in A1. Therefore, here is a formula that tests for that issue, and either uses that function or the initial value:

```
@IF(@ISSTRING(A1),-@VALUE(@LEFT(A1,@LENGTH(A1)-1)),
A1)
```

Elaborations can be added if text other than #- might be expected.

How to convert a number to feet and inches

Assume that you have a number like 12.345 in cell A1 and you want to express it in terms of feet and inches. The integer portion is easy to extract as the feet, so the decimal portion needs to be converted into inches. The decimal portion can be stated at `@MOD(A1,1)`, and to convert it from feet to inches, it must be multiplied by 12. Since this type of measurement often uses fractions rather than decimals, we will use a maximum denominator of 16. Finally, we will concatenate all of this with whatever indicators we have for feet and inches. We can use text like “ft” and “in”, or we can use the double and single quotes available with `@CHAR`. Thus:

```
@CONCATENATE(@INT(A1)," ft ",
@FRACTION(@MOD(A1,1)*12,16)," in")
```

returns: 12 ft 4 1/8 in.

```
@CONCATENATE(@INT(A1),@CHAR(39),
@FRACTION(@MOD(A1,1)*12,16),@CHAR(34))
```

returns: 12’ 4 1/8”.

How to convert a date number into the day or month as text

As noted above at page 59, the easiest way to obtain the day of the week from a date number is to use the `@DATEINFO` function. If 2/14/15 (42049) is in A1, `@DATEINFO(A1,1)` returns “Sat” and `@DATEINFO(A1,5)` returns “Feb”.

The problem is that these results cannot be joined as items in an `@CONCATENATE` function. They can be joined as items in an `@ARRAY` function, but each numeric item joined in the function must be separately converted to string, a task that `@CONCATENATE` does automatically.

Another way to do so is to obtain weekday or month numbers and use the `@CHOOSE` function to select text. `@CHOOSE(`*`Number,List-separated-by- commas`*`)` returns a value from a list of values separated by commas, based on the `Number` argument: if `Number` is 0, the first item is returned; if 1, the second item, and so on. Since `@WEEKDAY` returns Sunday to Saturday on a 1 to 7 scale, this formula will return the same as `@DATEINFO(A1,1)`:

```
@CHOOSE(@WEEKDAY(A1)-1,"Sun","Mon","Tue","Wed",
"Thu","Fri","Sat")
```

And this returns what `@DATEINFO(A1,5)` returns:

```
@CHOOSE(@MONTH(A1)-1,"Jan","Feb","Mar","Apr","May",
"Jun","Jul","Aug","Sep","Oct","Nov","Dec")
```

Table 9.4: Converting numeric dates to text

Desired text format	Function to convert date number in A1
Mon 5 Jan 2015	@CONCATENATE(@CHOOSE(@WEEKDAY(A1)-1, "Sun","Mon","Tue","Wed","Thu","Fri","Sat"), @CHAR(32),@DAY(A1),@CHAR(32), @CHOOSE(@MONTH(A1)-1,"Jan","Feb","Mar","Apr", "May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"), @CHAR(32),@YEAR(A1)+1900)
Same	@ARRAY(@DATEINFO(A1,1)&@CHAR(32) &@STRING(@DAY(A1),0)&@CHAR(32) &@DATEINFO(A1,5)& @CHAR(32) &@STRING(@YEAR(A1)+1900,0))
2015-01-05	@CONCATENATE(@YEAR(A1)+1900,"-", @IF(@MONTH(A1)<10,"0",""),@MONTH(A1),"-", @IF(@DAY(A1)<10,"0",""),@DAY(A1))

How to convert numeric dates into text dates

One of the more common problems is to convert a date from a number to text, or vice versa. QP's built-in functions go only so far, and the user must often improvise.

Assume that the date 1/5/15 is in cell A1. Table 9.4 gives sample coding for converting that number into the indicated text format.

How to convert text dates/times into numeric dates/times

If the text to be converted into a numeric date or time does not fit one of the formats approved for `@DATEVALUE` and `@TIMEVALUE`, functions have to be applied to them to extract the needed data and to re-combine it. The main trick is to know how to extract parts of the text with functions like `@FIELD`, `@LEFT`, `@RIGHT`, `@MID`, and either:

- Combine them if necessary into text strings with `@CONCATENATE` to be converted to numbers by `@DATEVALUE` and `@TIMEVALUE`, or
- Convert them into values with `@VALUE`, to be assimilated as arguments into `@DATE` or `@TIME`.

Table 9.5 provides some samples, including some provided by Kenneth Hobson and Roy Lewis.

Table 9.5: Converting text dates to numbers

Text Date in A1	Function to generate a date and/or time number
20091217120000[0:GMT]	@DATEVALUE(@CONCATENATE(@MID(A1,4,2),"/", @MID(A1,6,2),"/",@MID(A1,0,4)))
22/Jun/2015:00:13:32	@DATEVALUE(@SUBSTITUTE(@FIELD(A1,1,":"),"/","-")) +@TIME-VALUE(@RIGHT(A1,8))
10/08/2013 03:29 PM	@DATEVALUE(@FIELD(A1,1," ")) +@TIMEVALUE(@CONCATENATE(@FIELD(A1,2," "),

	@FIELD(A1,3," ")))
July 9, 2010	@DATEVALUE(@CONCATENATE(@SUBSTITUTE (@FIELD(A1,2," "),",",""),"-",@LEFT(@FIELD(A1,1," "),3), "-",@FIELD(A1,3," ")))
Tuesday, May 27, 2008	@DATEVALUE(@CONCATENATE(@FIELD(@FIELD(A1,2,","),3," "), "-",@LEFT(@FIELD(@FIELD(A1,2,","),2," "),3), "-", @RIGHT(@FIELD(A1,3,","),4)))
01-25-2014	@DATE(@VALUE(@FIELD(A1,3,"-")), @VALUE(@FIELD(A1,1,"-")),@VALUE(@FIELD(A1,2,"-")))
01-25-2014	@DATEVALUE(@CONCATENATE(@FIELD(A1,1,"-"),"/", @FIELD(A1,2,"-"),"/",@FIELD(A1,3,"-")))
2010-07-08 00:00:00	@DATEVALUE((@FIELD(@FIELD(A1,1," "),2,"-")&"/" &@FIELD(@FIELD(A1,1," "),3,"-")&"/" &@FIELD(@FIELD(A1,1," "),1,"-")))

Readers are welcome to suggest additions.

How to convert decimal time to hours and minutes

The QP user in WPO 16125 wants to convert a value representing hours and decimal parts of the hour (e.g., 8.2) into text showing hours and minutes (e.g., 8 hours 12 minutes).

If the decimal total is in, say, cell A1, then a function that returns the hours is `@INT(A1)`, a function that returns the minutes is `@INT(@MOD(A1,1)*60)`, and a function that returns the entire number as text in "H hours M minutes" format is `@CONCATENATE(@INT(A1)," hours ",@INT(@MOD(A1,1)*60)," minutes")`.

Custom format bug. I also tried to create a custom format that would display the number with the text attributes, but it does not appear to work. `h hours mmi minutes` was far off. `9 hours mmi minutes` was as close as I could get, but it rounded the hours up and, for some reason, doubled the decimal values when they converted to minutes. This much looks like a fixable bug.

How to convert clock times to seconds

If you want to convert a column of text numbers like "4:30" to a number of seconds, and those text numbers are in the A column starting at A1, you might use the following function in cell B1:

```
(60*@VALUE(@FIELD(A1,1,":")))
+@VALUE(@FIELD(A1,2,":"))
```

How to add one hour to a given time

If you want to create a number representing the time one hour later than a given time in a text field containing a date and time, you will need to convert the date and time to a number, and add 1/24 (one hour, out of a 24-hour day). For example, if *12-21-13 5:15am* is a textual entry in A1, this formula converts it into a usable number:

```
@DATEVALUE(@SUBSTITUTE(@FIELD(A1,1," "),"-","/"))
+@TIMEVALUE(@FIELD(A1,2," "))+(1/24)
```

The resulting number can be formatted to show the date *or* time, using QP's built-in numeric formats.

How to create a chronologically sortable column from text numbers

Say that you have a column of dates in the format: ’01-25-2014, ’03-25-2013, etc. and you want to sort them in chronological order. If nothing changes, the text will be sorted alphabetically, namely by comparing the first letter (whether "0" or "2" in these examples) in both, then the second, then the third, etc. This will not yield chronological order.

Therefore, these text fields will need to be converted in some way to allow the sorting. That can be most easily done by using a “helper column” which will be filled with formulas that massage the original entries into sortable dates, either in a text format or a numeric format. Here, as in other cases, we want to construct a formula that changes the first entry, and then copy it down the helper column in parallel with the original entries.

Method 1 - Text Sorting

If the user desires to sort the dates alphabetically, the dates should be converted to the yyyy-mm-dd format instead of the mm-dd-yyyy. This formula would yield that date format:

```
@CONCATENATE(@FIELD(A1,3,"-"),"-",
@FIELD(A1,1,"-"),"-",
@FIELD(A1,2,"-"))
```

Method 2 - Numeric Sorting

To use them numerically, which is most desirable for most purposes, convert the text into a date value by an @DATE formula like this:

```
@DATE(@VALUE(@FIELD(A1,3,"-")),
@VALUE(@FIELD(A1,1,"-")),
@VALUE(@FIELD(A1,2,"-")))
```

or by an @DATEVALUE formula like this:

```
@DATEVALUE(@CONCATENATE(@FIELD(A1,1,"-"),"/",
@FIELD(A1,2,"-"),"/",@FIELD(A1,3,"-")))
```

Appendix: Custom numeric formats

Creating custom numeric formats

To create a custom numeric format for a spreadsheet, follow these steps.

1. Open the Selection Properties dialog box. That can be done by pressing `[F12]`, using the menu Format > Selection Properties, or right-clicking on a cell and choosing Selection Properties.
2. Choose the "Numeric Format" tab.
3. At the bottom of the list, click the item called "Custom". This will pop up a list of existing "Custom" formats, one of which will be selected (in my case, a format called "CenterBar"), and "Add..." and "Edit..." buttons.
4. Click the "Add..." button to create a new format. This will pop up an "Add Format" dialog box, which is unfortunately not a very intuitive dialog box. It will be also loaded with the settings for the custom format that was pre-selected in step 3, so we'll need to get rid of those first.
   a) Type the name for your new custom format into the top box, and thus remove the name of the pre-selected custom format.
   b) The rules that QP uses to display numbers in a particular format are in the second box, and the rules currently there belong to the pre-selected custom format. I will call this the "**Rules Box**". Select each one and then click the "Delete" button below that box until none are left.
   c) The block in the center under Format Code (which we'll call the "**Format Code Box**") should be cleared of anything it contains. Click on its contents and press Backspace or Delete, until they are gone.

   You are now ready to create your own format.
5. Compose the format.
   a) If you're dealing with a date or time format, you'll probably use the contents of the dropdox bow called "Add Date/Time Code" on the right; otherwise, you'll probably use the contents of the dropdown box "Add Numeric Code" on the left. In either case, as you select items in the dropdown boxes, they will be added/inserted into the Format Code Box. You can add your own elements to that box, which will be added to the formatting codes when the custom format is applied to numbers.

      QP's help file has a list of what each of these options does. Find it at "Editing and formatting spreadsheets" > "Formatting spreadsheets" > "Reference: Formatting spreadsheets".
   b) If you desire, select the text in the Format Code Box and apply any of the attribute codes below that box, such as font, font size, bold, font color, etc.
   c) If you desire, you can make the formatting conditional upon choices made in the "If Condition" section above the "Format Code" section. For instance, if you want to apply the formatting created in a and b above only if the value in the cell is less than 100, then under "First Term" choose "This Cell," under "Operator" choose "<", and under "Second Term" choose "Value" and enter 100 to the right.
   d) This creates a formatting rule, and you are now ready to add the formatting rule to the Rules Box by clicking "Add" under the Rules Box.

   e) You can add more rules (which might be useful if you made the formatting on conditions that might not apply to some cells). First, the last step will cause the dialog to select the newly added Rule, so deselect it to create another rule; otherwise, QP thinks that you are trying to modify the last rule. Then clear out unwanted conditions and format codes, and then repeat steps a to d.

   You are now ready to add this as a custom format.

6. Add this as a Custom Format by clicking `[OK]`. (That was easy.) You're now ready to select some cells and then use steps 1 and 2 to select this format.

Using and re-using custom numeric formats

Now you are also ready to use the custom format. But will you be able to use it tomorrow? The problem is that the custom formats that you create for one session of QP are stored only in memory. They are available for later files in the same session, but after QP is closed, they disappear.

As far as I can tell, there is no place in the system that stores the Custom Numeric Formats that you create (as distinct from those that Corel installs by default). I've tried Format > DefineStyles > Make Defaults | Apply as default for new notebooks, which works for some selections, but not apparently for user-created custom numeric formats. I've tried creating a QuattroPro.qpw file with the format, and saving it in the `ShellNew` subfolder in the program files WPO folder, but that also failed to work (and in one version crashed QP when I attempted to access the numeric format tab of the selection format dialog box). I've looked for other ways too, but have not found one that works between QP sessions. If someone knows how to make something work systematically, please share it with me.

It appears that user-created custom formats get stored only in individual files when they are applied to a cell in the file and the file is saved. They become available for new files by opening files that have used them. As such files are read during a QP session, they accumulate in memory in a list of custom numeric formats.

So, in order to have available an array of custom numeric formats for any session of QP, follow these steps.

1. Create a new file (e.g., CustomFormats.qpw), create the formats as shown above, and apply them to cells in that new QPW file, and save the QPW file.
2. At the beginning of any QP session, or at the time that the formats are needed, open CustomFormats.qpw and close it.

The formats will be available for further work until QP is closed.

Using custom numeric formats to convert Excel data

In WPU 38448, forum members discussed methods of converting date data from Excel in d/m/y format (e.g., 31/7/17) to merge in WordPerfect in standard textual format (e.g., 31 July 2017). One could accomplish the task by creating a QP file that has a custom `d Month yyyy` format and storing it. Then, when one needs to merge such data from Excel, one first opens the file with the custom format, which makes that format available in QP. One then opens the Excel file in QP, and applies the custom format to the data data. After that, the data could be merged into WordPerfect using techniques discussed in Chapter 22.

Chapter 10

Getting information about and from a database

The functions in this section assist with using databases, which I understand as structured sets of data. Though three-dimensional databases can contain data on more than one sheet, I will focus on **two-dimensional databases** that exist only on one sheet. And though this ordering could be reversed, I will use *rows* of data to relate to a particular subject (a person, an event, a thing, a transaction, etc.), and the *columns* to provide particular information about that item. The coordinates of this database will be a single Block.

Some of these functions apply to any database as described above. Other functions depend on the database having a particular structure, and these are called **Indexed Databases**. If the first/leftmost column (the **Index Column**) contains unique identifiers for the subject (Index Numbers), some functions can easily return information about a given subject. These functions have more capabilities if the numbers in the index column are sorted in ascending order (increasing from the top row to the bottom). And if the first/top row of the database contains the names of categories of information, some functions can use those categories to make QP's ability to extract information more powerful (as well as making it more readable for the user).

Paradoxically, I will not be discussing the `@D***` that are described in QP's help file as "database spreadsheet functions," e.g., `@DSUM`, `@DAVG`, `@DCOUNT` and so on. In my opinion, for most practical purposes, all of the useful `@D***` functions can be accomplished at least as well with block formulas (see page 9), and by the other functions described below, but without the extra overhead required by the `@D***` functions. Namely, in addition to the database and the function, the `@D***` functions require the user to have **criteria tables** in certain cells in the spreadsheet. The top row of these criteria tables would have all or some of the column headings in the spreadsheet, and in cells below those headings, one or more values would be entered (the "criteria"), and these would have the effect of narrowing the search through the database to rows that match the criteria. That approach has merit, merit which increases with the complexity of the criteria, but it is useful primarily for a small number of more narrowly focused applications of QP, not for the general purposes that this guide seeks to address. (The approach is a close cousin of the notebook query, which is considerably more useful for ordinary purposes. See page 181 for more.)

The following functions get useful information about and from a database.

Getting the width and height of a block: @COLS, @ROWS, @SHEETS

To get the number of columns or rows in a given block, use `@COLS(`*`Block`*`)` or `@ROWS(`*`Block`*`)`. The minimum value will be one, if the block is valid.

Broken (in earlier versions). `@SHEETS(`*`Block`*`)` returns the number of sheets in `Block` in the most current versions. Until QP17, Service Pack 2, it returned the number of columns in a block. See WPU 21382.

Getting column and row numbers for a block: @COLUMN, @ROW

To get the column or row number of a particular cell, a particular block, or the current cell, use `@COLUMN(`*`Block`*`)` or `@ROW(`*`Block`*`)`.

Both return a number, not text. `@ROW` returns a number equivalent to the headers on the left side of the data; `@COLUMN` returns a number, with 1 representing column A, 2 is column B, etc. If more than one cell is in the block, these functions return an array of numbers. In both cases, QP wraps the function inside `@ARRAY()`.

In both cases, the `Block` argument is optional; if it is omitted, QP assumes that you want the row where this function appears, and it wraps the function in an `@ARRAY` function.

Caution. When used in macros, both functions by default return the row and column of the cell containing the macro command, not the active cell. To get the row and column of the active cell or any other cell, it must be specified in the `Block` argument. And if R1C1 notation is used, it must be prefaced by brackets, e.g., `[]c(0)r(0)`.

Column identifiers: @INDEXTOLETTER, @LETTERTOINDEX

Columns are designated by letter, but some functions require them to be designated by number. It is also useful to designate them by number when devising macros that loop through a series of columns. Two functions are useful for this conversion.

`@LETTERTOINDEX(`*`Letter)`* converts a column letter into a number.

`@INDEXTOLETTER(`*`Number`*`)` converts a number into a column letter.

Both functions operate on the basis that 0 is column A, 1 is column B, etc. Thus, `@LETTERTOINDEX("D")` returns the number `3`. Conversely, `@INDEXTOLETTER(3)` returns the letter `D` as text.

Since this pairing counter-intuitively departs from the pairing used by `@COLUMN` (and the “col” argument for `@CELL` discussed above at page 22), in which column A equates to 1, the user should be careful to subtract or add one from the results of these functions to make them compatible.

Counting (non-)blank cells: @COUNT, @COUNTBLANK

To determine the number of cells in a column of the database that are occupied or blank, use either `@COUNT(`*`block`*`)` or `@COUNTBLANK(`*`block`*`)`.

`@COUNT(`*`Block`*`)` returns the number of non-blank cells in the specified block. The block can be specified by coordinates or by any shorthand that is used to identify a block.

`@COUNTBLANK(`*`Block`*`)` returns the number of blank cells (or cells with formulas that return blanks, but it does not count cells with the numeric value of zero.

Thus `@COUNT(B:A)` returns the number of non-blank cells everywhere in column A on sheet B. That's a useful shorthand if the relevant data starts in the top cell of that column.

To get the number of non-blank cells in a subset of the column that starts below the top, the full coordinates need to be in the parentheses. So, `@COUNT(A4..A1000)` will return the number of non-blank cells in the specified block.

For a discussion of detecting whether cells are blank, given that the user may want to treat cells differently depending on whether they contain the numeric value of zero or a formula that returns a blank, see the discussion of how to detect blanks at page 26.

Caution: The two functions are largely symmetrical but not entirely so. In particular, consider the case of cells that contain a function that returns either blanks or values. If the function returns a value, both `@COUNT` and `@COUNTBLANK` will treat it as non-blank. But if the function returns a blank, `@COUNT` still treats the cell as non-blank, probably because it contains a function, but `@COUNTBLANK` treats it as blank.

Counting conditionally: @COUNTIF

Table 10.1 illustrates how `@COUNTIF` works. It is a simplified version of block formulas (see page 9). It works well enough, but more complex functions should use block formulas.

Table 10.1: @COUNTIF

	A	B	C	D	E	F
1	2008	3	5	⇐@COUNTIF(A1..A6<=2010)	2	⇐@COUNTIF(B1..B6<=5)
2	2008	6	1	⇐@COUNTIF(A1..A6>2010)	4	⇐@COUNTIF(B1..B6>5)
3	2009	7				
4	2009	4				
5	2010	9				
6	2011	6				

Note that the @COUNTIF formula in C1 is equivalent to @SUM(A1..A6<=2010).

Basic functions for extracting numeric information from a database

The basic mathematical functions can be used to extract information from a database: `@SUM`, `@SUMIF`, `@MAX`, `@MIN`, `@PUREAVG`, `@PURECOUNT`, `@PUREMAX`, `@PUREMIN`, `@LARGEST`, `@SMALLEST`.

Retrieval of last entry: @LASTCELLVALUE

`@LASTCELLVALUE(`*`block,<type>`*`)` returns the contents of the last non-blank cell in the block. The function has a required argument, the block in question, and an optional argument, the "type". The "type" can be 1, referring to the last value in a column, or 2, the last value in a row. 1 (column) is the default type. QP searches for the last non-blank cell in the row or column.

Quirks. As far as I can tell, as long as the block defines a single row of data, this function will find the last cell value in the row, even if the type argument is omitted.

Contrary to the help file, if there is no content to return, the function returns a blank, not a zero.

The function returns an ERR when used twice in an `@IF` function. For instance, in B1, type:

```
@IF(@LASTCELLVALUE(A1..A10)="","a",
@LASTCELLVALUE(A1..A10))
```

This will yield an ERR if any values are in A1..A10, but it correctly returns "a" otherwise. One workaround is:

```
@IF(@ROWS(A1..A10)=@COUNTBLANK(A1..A10),"a",
@LASTCELLVALUE(A1..A10))
```

The function also returns an ERR when used as an argument in an `@DATEDIF` function. Thus, comparing a start date in A1 with the last date entered in the column below it, the function

```
@DATEDIF(A1,@LASTCELLVALUE(A2..A10),"y")
```

returns ERR, though

```
@DATEDIF(A1,@MAX(A2..A10),"y")
```

does not. A workaround is to put the `@LASTCELLVALUE` function in another cell and compare the start date with that cell.

`@LASTCELLVALUE` will not accept the shorthand reference to an entire column. The attempt to get the last cell value in column H with `@LASTCELLVALUE(H)` will cause QP to crash.

Retrieval of parallel value: @LOOKUP

`@LOOKUP(`*`ValueinArray1,Array1,Array2`*`)` does not actually require a block, just two columns and a value in the first column. The function returns the parallel value in the second column. If the value in the first column is the 15th item in the column, the function returns the 15th item in the second column.

Actually, the function does not require parallel columns; it could be parallel rows, or any mix of arrays that simply have the same number of items.

QP automatically wraps `@LOOKUP` in an `@ARRAY` function.

Block formulas can do much the same thing. See page 9.

Retrieval by index value: @VLOOKUP, @HLOOKUP, @VHLOOKUP

`@VLOOKUP(`*`IndexValue,Block,Col#,<ExactMatch?>`*`)` is a more structured version of `@LOOKUP`, using an indexed database defined by the coordinates of `Block`. It returns information in desired category (column), which is defined by the `Col#` argument, with 0 representing the index column. It returns the item in the desired column that is parallel with the item in the index column that is found by the `IndexValue` argument.

The final optional argument `ExactMatch?` is either 0, signifying that the IndexValue must have an exact match in the index column, or 1, signifying that it need not have an exact match. (We would normally expect 1 to represent an affirmation and 0 a non-affirmation; here it is the other way around.) If this optional argument is not specified, QP assumes that 1 is the case and no exact match is required. I find that the argument should be 0 if each row in the database reflects unique individual people, events, transactions, etc., but it should be 1 if the rows represent threshold points (such as tax tables or student grade levels) where a range of intermediate numbers are possible. Table 10.2 shows the functional differences when 1 or 0 is the `ExactMatch?` value.

Table 10.2: Return Values of @VLOOKUP, with or without exact matches

	A	B	C	D	E	F
1	10	F			Match	
2	20	F		Grades	1	0
3	30	F		70	C-	C-
4	40	F		82	B-	ERR
5	50	F		90	A-	A-
6	60	D-		92	A-	ERR
7	65	D		97	A	ERR
8	70	C-		100	A+	A+
9	75	C		104	A+	ERR
10	80	B-				
11	85	B				
12	90	A-				
13	95	A				
14	100	A+				

E3 = @VLOOKUP($D3,$A$1..$B$14,1,E$2) copied to E3..F9

If `ExactMatch?` is 0, the function returns ERR only if `IndexValue` is not in the index column. If there are more than item with the same `IndexValue` in the index column, the function returns the item on the first matching row it finds, going from top to bottom in the database. If each entry in the index column is unique, the function returns the desired information about the unique entity that is the subject of the database. The index column need not be sorted in numerical order.

If `ExactMatch?` is 1 (the default), QP expects the index column to be sorted numerically, and when it is, the function selects the first row with an exact match in the index column *if* there is an exact match, but if there is no exact match in the index column, it selects the row with the next lowest value in the index column (which is the row above the point where `IndexValue` would be, if it were in the index column). Thus, if the index column has items 1 and 2, and the `IndexValue` argument is 1.5, QP will use the row with item 1 in the index column. The function returns ERR only if the `IndexValue` is less than the lowest value it finds in the index column, but since it expects the index column to be sorted, it will ignore rows that are out of order. For instance, if the index column is sorted in descending order from 10 at the top to 1 at the bottom, an `IndexValue` of anything less than 10 will return ERR, even if there are exact matches in the index column.

`@HLOOKUP(`*`IndexValue,Block,Row#,0`*`)` is like `@VLOOKUP`, with rows and columns transposed and the same comments apply. Based on the assumptions I make about the structure of the database that are set out at the beginning of this section, however, I will not be giving examples with this function.

`@VHLOOKUP(`*`LeftIndexValue,TopIndexValue,Block`*`)` combines the `@VLOOKUP` and `@HLOOKUP` functions. It returns the item at the intersection of the index column identified by the `LeftIndexValue` argument (usually, the target value in the A column) and the row identified by the `TopIndexValue` argument (usually, the target value in row 1). QP wraps `@VHLOOKUP` in an `@ARRAY` function. This function would be of greater utility in those cases where the index column and index row are both numeric and sorted. Based on the assumptions I make about the structure of the database that are set out at the beginning of this section, however, I will not be giving examples with this function.

Retrieval by column and row: @INDEX, @XINDEX

`@INDEX(`*`Block,Col#,Row#`*`)` returns the content of the cell in the Block that is at the intersection of the `Column#` and `Row#`. `Column#` and `Row#` start at 0, not 1. Thus, to return the content of cell A1 in Block A1..Z10, the correct formula is `@INDEX(A1..Z10,0,0)`. This is a good function for use with macros that run sequentially through the entire database. I will be using it a lot below.

See also `@CELLINDEX`, above at page 22; its "contents" argument performs a similar function.

`@XINDEX(`*`Block,ColHead,RowHead`*`)` does the same as `@INDEX`, except that instead of column and row offsets, it uses the row and column headers themselves. That makes it less useful for macro programming, though perhaps useful for other applications. QP wraps `@XINDEX` in an `@ARRAY` function.

Quirks. However, note that if the cell containing the information that the function calls for is blank, this function returns zero (0), not a blank. This result can be a nuisance. If the cell is either going to be text or blank, a workaround is to ensure that QP treats the source cell as text by adding the empty string to it, e.g., `@INDEX(A1..Z10,0,0)&""`. If the source cell is blank, this formula returns an empty string. This workaround will yield an ERR, however, if the source cell contains a number.

Also, in some earlier versions of QP, the `@INDEX` function has failed if negative numbers were in the Block, as noted in WPU 33863.

Comparing the preceding functions

Table 10.3 shows how the preceding functions can find and obtain specific data from a database. The object here will be to return the value in cell D5 from the database in A1..D7.

Table 10.3: Comparing functions for retrieving data

	A	B	C	D	E
1	Month	Date	Name	Amount	@LOOKUP("Doris",C1..C7,D1..D7) *
2	Jan	01/01/15	Alan	$70.58	@VLOOKUP("Apr",A1..D7,3,0)
3	Feb	01/08/15	Betty	$557.78	@HLOOKUP("Amount",A1..D7,4,0)
4	Mar	01/15/15	Charlie	$935.92	@VHLOOKUP("Apr","Amount",A1..D7) *
5	Apr	01/22/15	Doris	$798.28	@INDEX(A1..D7,3,4)
6	May	01/29/15	Ed	$342.29	@XINDEX(A1..D7,"Amount","Apr") *
7	Jun	02/05/15	Fran	$780.72	@CELLINDEX("contents",A1..D7,3,4)

* QP wraps these functions in @ARRAY

An astute reader will wonder what is the difference between `@VHLOOKUP` and `@XINDEX`. Apart from the order of the arguments, I see no difference.

Retrieval by text coordinates: @@

The chapter on @Functions that deal with cell coordinates (see page 34) supplies another means of obtaining data from a database. If the text coordinates of a cell are known or can be constructed, the `@@` function can return its contents.

Hence, in the last example, the name in cell A:C2 would be returned by, among others:

```
@@("C2")
@@("C"&"2")
@@("A:"&@ADDRESS(2,3))
@@(@ADDRESS(2,3))
@@(@ADDRESS(2,3,1,1,"A"))
```

The ability of `@ADDRESS` to generate an address from row and column numbers will allow for automating the method of retrieving data through the use of `{For}` loops.

Determining column/row offset from matches: @MATCH

The converse of obtaining an item by knowing the column and row is to obtain the column or row from knowing the item. `@MATCH(`*`ItemToMatch, Array<,ExactMatch>`*`)` does that, as Table 10.4 illustrates.

The `Array` is typically a block that may only be a single column *or* a single row. If the `Array` has more than one column *and* more than one row, the function returns ERR.

Table 10.4: Return Values of @MATCH, with or without exact matches

	A	B	C	D	E
1	10	F		Match	
2	20	F	Grades	1	0
3	30	F	70	7	7
4	40	F	82	9	ERR
5	50	F	90	11	11
6	60	D-	92	11	ERR
7	65	D	97	12	ERR
8	70	C-	100	13	13
9	75	C	104	13	ERR
10	80	B-			
11	85	B			
12	90	A-			
13	95	A			
14	100	A+			

E3 = @MATCH($D3,$A$1..$A$14,E$2)   copied to E3..F9

The `ItemToMatch` is a value to be located in the `Array`. If there is an exact match, the function returns the offset number from the first item in the `Array`, which is numbered 0.

The `ExactMatch?` argument is optional. It can have the value of 1, 0, or -1, and if that argument is left out, the value is deemed to be 1. The meanings are like the `ExactMatch?` argument used by `@VLOOKUP`, which can be 1 or 0. Here, the value of -1 is the opposite of 1.

• If `ExactMatch?` is 0, the function returns the offset of the first item in `Array` that is an exact match, and if there is no exact match, the function returns ERR. The items in `Array` need not be sorted in any order.

• If `ExactMatch?` is 1, the function expects the items to be sorted in ascending order (lowest value in the first position, increasing to the last position). It returns the offset of the first item in `Array` *if* there is an exact match, but if there is no exact match, it selects the offset with the next lowest value in `Array`. The function returns ERR only if the `IndexValue` is less than the first value in `Array`.

• If `ExactMatch?` is -1, the function expects the items to be sorted in descending order. It returns the offset of the first item in `Array` *if* there is an exact match, but if there is no exact match, it selects the offset with the next highest value in `Array`. The function returns ERR only if the `IndexValue` is greater than the first value in `Array`.

Incidentally, the `Array` need not refer to cells; it can refer to an array of values, and in this respect, it is the converse of the `@CHOOSE` function. Thus:

```
@MATCH("Tue",{"Sat","Sun","Mon","Tue","Wed",
"Thu","Fri"},0)
```

returns the offset value of 3.

Getting coordinates of extreme values: @MAXLOOKUP, @MINLOOOKUP

`@MAXLOOKUP(Block)` returns the address of the cell with the maximum value in the block in the same format that `@OFFSET` does.

`@MINLOOKUP(Block)` unsurprisingly returns the address of the cell with the minimum value in the block, again in the same format that `@OFFSET` does.

How (not) to derive coordinates relative to the current cell

The task in the following exercises is to come up with functions that will always return desired coordinates relative to the current cell, even if rows or columns are inserted or deleted around the current cell.

Cautions

Using these functions in macros. As noted below (page 106), when used in macros, R1C1 notation requires brackets in order to make references relative to the active cell rather than to the cell containing the current macro command. Therefore, for purposes of these exercises, use `[]c(0)r(0)`, not `c(0)r(0)`. And as noted above (page 79), when used in macros, `@ROW` and `@COLUMN` return row and column numbers for the cell in which the macro command occurs, not for the presently active cell. To deal with this problem in macros, specify the block to which the function relates: e.g., `@ROW([]c(0)r(0))`.

Circularity. Using these functions in macros (as opposed to using a macro to place one of these functions in a cell), does not cause the circularity indicator to come on. But if you place a function in a cell that refers back to the same cell, as some of these functions do, the circularity indicator comes on. As far as I can tell, there are no adverse consequences in doing so for purposes of identifying a block. Because circularity is a source of problems in at least some settings, though, the programmer should make a decision about which type of function will be best. The *basic block* method avoids circularity in functions placed in the cells.

The basic block

This method of avoiding circularity involves creating a formula that returns the block of cells running from A1 to the current cell, which I call **the basic block**. Here is the formula:

```
@OFFSET(@@("A1..A1"),0,0,@ROW,@COLUMN)
```

If this function is entered into, say, cell D2, it returns A1..D2 (as text).

The first argument, `@@("A1..A1")`, is necessary to anchor the block at the A1 cell. If, instead, the first argument were `$A$1`, the block would shift incorrectly if columns were later inserted to the left of the A column, or if rows were inserted above row 1.

Same Row, Column A

One might imagine that one could use the `@ROW` or `@COLUMN` functions for the first two offsets, to determine, for instance, the coordinates of the cell on the same row in the A column with

`@OFFSET(@@("A1..A1"),@ROW,0)` (Wrong!)

or that one might derive it by

`@ADDRESS(@ROW,1)` (Wrong again!)

QP regards `@ROW` as an "invalid argument type" in these instances. This is particularly odd in the case of `@OFFSET` because it is valid for the `HeighInRows` (fourth) argument, but not the `RowOffset#` (second) argument. Go figure.

There are other tries, such as:

`@CONCATENATE("A",@ROW)` (Not good)

This would seem simplest, in that if the cell containing it is D7, it returns "A7" as text, but it suffers from a couple of flaws. *First*, the return value differs from those of `@OFFSET`, which would return a single cell as the block (e.g., A7..A7), and which can be used for further functions. In particular, "A7" as text, even when wrapped in `@@`, cannot be used as the `StartCellBlock` (first) argument in a wrapping `@OFFSET` function; it would have to be converted to "A7..A7" and wrapped in `@@` to be used as the base for `@OFFSET`. *Second*, after it is initially entered, it does not update if rows or columns are inserted, and thus it cannot be relied upon if insertions could happen.

We can, however, finesse it by using the `@ROWS` function to convert the basic block into a number of rows that can be used for the second argument in an `@OFFSET` function.

```
@ROWS(@@(@OFFSET(@@("A1..A1"),0,0,@ROW,@COLUMN)))
```

returns the number of rows in the basic block. Because `@ROWS` starts with 1, rather than 0, we will need to subtract 1 from its result to get the correct offset:

```
@ROWS(@@(@OFFSET(@@("A1..A1"),0,0,@ROW,
@COLUMN)))-1
```

Based on this background, the following function will always give us the coordinates of the cell in the A column in the same row as this function:

```
@OFFSET(@@("A1..A1"),
@ROWS(@@(@OFFSET(@@("A1..A1"),0,0,
@ROW,@COLUMN)))-1,0)
```

This is the best way I know. By increasing the final 0 to 1, 2, etc., it would return the parallel values in columns B, C, etc.

Same Row, Column A, with circularity

For sake of completeness, however, I note that another method is a bit simpler, but it causes the circularity indicator to light up, because the `@CELL` function in it requires inserting the address of its cell in the function. Thus, if the formula is entered in cell D6, it would have D6 in it:

```
@OFFSET(@@("A1..A1"),@CELL("row",D6)-1,0)
```

or

```
@ADDRESS(@CELL("row",D6),1)
```

Without knowing in advance what the cell's address is into which the function will be put, this will place that circular function correctly:

```
@OFFSET(@@("A1..A1"),@CELL("row",c(0)r(0))-1,0)
```

or

```
@ADDRESS(@CELL("row",c(0)r(0)),1)
```

Remember to preface any R1C1 addresses used in macros with brackets, e.g., `[]c(0)r(0)`.

The Block from Column A

To get the block from the A column to the current cell requires adding only two coordinates to the `@OFFSET` formula:

```
@OFFSET(@@("A1..A1"),
@ROWS(@@(@OFFSET(@@("A1..A1"),0,0,
@ROW,@COLUMN)))-1,0,1,@COLUMN)
```

Circular versions con be constructed as well:

```
@ADDRESS(@CELL("row",c(0)r(0)),1)&".."&
@CELL("address",c(0)r(0))
```

or

```
@OFFSET(@@("A1..A1"),@CELL("row",[]c(0)r(0))
-1,0,1,@COLUMN([]c(0)r(0)))
```

Same Column, Row 1

Based on the above discussion, the best way to construct the coordinates of the cell in row 1 of the column in which the function is entered would be this:

```
@OFFSET(@@("A1..A1"),
0,@COLS(@@(@OFFSET(@@("A1..A1"),0,0,
@ROW,@COLUMN)))-1)
```

Similar circular shortcuts are available.

The Block from Row 1

To get the block of the column from row 1 to the current cell requires the addition of two arguments to the last formula:

```
@OFFSET(@@("A1..A1"),
0,@COLS(@@(@OFFSET(@@("A1..A1"),0,0,
@ROW,@COLUMN)))-1,@ROW,1)
```

Circular shortcuts can be created.

The Cell to the left

Using the technique I recommend to refer to the cell to the left of the current cell requires finessing the `@OFFSET` function for both the `RowOffset#` and `ColumnOffset#` arguments. This takes the method for finding the cell in the A column on the same row and substituting for the final zero the function that returns the offset for the column to the left of the current column.

```
@OFFSET(@@("A1..A1"),
@ROWS(@@(@OFFSET(@@("A1..A1"),0,0,@ROW,@COLUMN)))-1,
@COLS(@@(@OFFSET(@@("A1..A1"),0,0,@ROW,@COLUMN)))-2)
```

The beauty of this function is that it always refers to the cell to the left of the function, even if rows are inserted anywhere between the left edge and the column with the function in it. By modifying the offsets in this formula (the -1 in the `RowOffset#` (second) argument and the -2 in the `ColumnOffset#` (third) argument, the function can refer to the address of any cell that has a defined position relative to the cell containing the formula.

The Cell to the left, with circularity

I hasten to add that there is a simpler, but circular, function that will accomplish the same thing. If the function is in cell D6, this works:

```
@OFFSET(@@("A1..A1"),@CELL("row",D6)-1,
@CELL("col",D6)-2)
```

And likewise, if we do not know the host cell in advance, QP will adjust if from this:

```
@OFFSET(@@("A1..A1"),@CELL("row",c(0)r(0))-1,
@CELL("col",c(0)r(0))-2)
```

A version using `@ADDRESS` is also available.

```
@ADDRESS(@CELL("row",c(0)r(0)),
@CELL("col",c(0)r(0)))
```

How to retrieve data from cells relative to the current cell

Getting the contents of cells relative to the current cell is not as difficult as getting their coordinates, because `@INDEX` accepts `@ROW` as an argument at all places. Because `@INDEX` needs a `Block`, we can supply it automatically with the “basic block” described in the last application. Since it is text, it will need to be wrapped in an `@@` function to use as coordinates for `@INDEX`:

```
@@(@OFFSET(@@("A1..A1"),0,0,@ROW,@COLUMN))
```

Same Row, Column A

To get the contents of the cell in the A column on the same row, use those coordinates of the basic block inside an `@INDEX` function as follows:

```
@INDEX(@@(@OFFSET(@@("A1..A1"),0,0,@ROW,@COLUMN)),
0,@ROW-1)
```

We have to subtract 1 from the last `@ROW` because it returns values on a scale starting with 1, not 0. QP wraps this function in `@ARRAY`.

There is, again, a simpler but circular way. If we put the function in D6, this works:

```
@@(@ADDRESS(@CELL("row",D6),1))
```

That function can be created without knowing the cell, again, by this:

```
@@(@ADDRESS(@CELL("row",c(0)r(0)),1))
```

Same Column, Row 1

To get the contents of the cell in the first row of the same column:

```
@INDEX(@@(@OFFSET(@@("A1..A1"),0,0,@ROW,@COLUMN)),
@COLUMN-1,0)
```

QP wraps this function in `@ARRAY` as well.

The Cell to the Left

Using this technique to refer to the cell to the left of the function's host cell, this works.

```
@INDEX(@@(@OFFSET(@@("A1..A1"),0,0,@ROW,@COLUMN)),
@COLUMN-2,@ROW-1)
```

Note that if you want to get data from cells below and/or to the right of the host cell, they must be found within the block defined by the `@OFFSET` function, wo it may be necessary to add numbers to its fourth and fifth arguments.

The circular method is also available with this function (if entered in cell D6):

```
@@(@ADDRESS(@CELL("row",D6),@CELL("col",D6)-1))
```

and this function works when pasted into any cell:

```
@@(@ADDRESS(@CELL("row",c(0)r(0)),
@CELL("col",c(0)r(0))-1))
```

How to identify the first blank row and cell in a column of data

The complexity of the formula for determining the first blank cell depends on how well-formed the database is. In the best-formed database that begins with entries in the top cell of a column and that continues without gap until the last entry in the column, the formula is at its simplest. If the column is A on sheet A, the formula

```
@COUNT(A:A)
```

returns the number of non-blank cells in the column, which is also the offset number for the first blank cell. That is, if there were no entries at all in the column, the number would be 0, and to find the first blank cell, `@OFFSET(A1,0,0)` would return the cell A1. If there were only a header in the column, the number would be 1, and `@OFFSET(A1,1,0)` would return the cell A2. If there are items (header and data) in the column, `@OFFSET(A1,100,0)` would return the cell A101. We can deduce that the formula for the first blank cell in a well-formed column is:

```
@OFFSET(A1,@COUNT(A:A),0)
```

If one feels squeamish about the `A:A` notation, `A:A1..A10000` will work if you will always have fewer than 10,000 entries:

```
@OFFSET(A1,@COUNT(A:A1..A10000),0)
```

One must use such notation if one starts below the first row. Thus, if the database begins at row 5, the formula would be:

```
@OFFSET(A5,@COUNT(A:A5..A10000),0)
```

Since the parallel `@ADDRESS` function starts counting rows and columns from 1 rather than 0, the comparable functions would be:

```
@ADDRESS(@COUNT(A:A)+1,1)
@ADDRESS(@COUNT(A:A1..A10000)+1,1)
@ADDRESS(4+@COUNT(A:A5..A10000)+1,1)
```

The last function requires adding 4 at the start to account for the four row gap before the database starts on row 5.

All of that assumes that there are no gaps in the data, and that there are no stray entries below the data. `@COUNT` is a blunt instrument, though, and if there are gaps or strays, it will easily yield a number other than the first blank row below the data. A more complex formula is necessary to find the first blank row after the last entry. We will need to identify the last entry and determine its row, which will be the offset value of the first blank row after the data. An initial try would be this block formula:

```
@MAX((A1..A10000<>"")*@ROW(A1..A10000))
```

`@ROW` returns row numbers, and the formula seeks to determine which cells in A1..A10000 are not blank (`<>""`). But as the review of tests for blank cells at page 26 shows, this would treat as blank cells that contain 0 and functions that return an empty string, which would not typically be the goal in looking for the first blank cell after all data. That study showed that the only function that treats those as non-blanks and that works on arrays is `@ISBLANK`, so the function needs to incorporate it. We're looking for non-blank cells, so we need the opposite of `@ISBLANK`. Wrapping `@ISBLANK` in `@NOT` does not produce an array, but an equation does. The formula, then, for the first blank row after the last entry is:

```
@MAX((@ISBLANK(A1..A10000)=0)*@ROW(A1..A10000))
```

The formula that gets the coordinates for that cell, therefore, is:

```
@OFFSET(A1,@MAX((@ISBLANK(A1..A10000)=0)
@ROW(A1..A10000)),0)
```

The parallel `@ADDRESS` function would be:

```
@ADDRESS(@MAX((@ISBLANK(A1..A10000)=0)
@ROW(A1..A10000))+1,1)
```

How to get the coordinates of a continuous column of entries

This is a slight variation of the last application, but the same logic applies. By using the final two optional arguments of `@OFFSET`, and starting from the 0,0 offset, we can define the entire block. In a well-formed database, the formula for the entire block would be:

```
@OFFSET(A1,0,0,@COUNT(A:A),1)
```

or, as long as there are fewer than 10,000 entries in the column,

```
@OFFSET(A1,0,0,@COUNT(A:A1..A10000),1)
```

If, for example, there are five entries in A1..A5, these formulas return as text some variation of “A1..A5”. A similar approach would derive the coordinates of the entire table:

```
@OFFSET(A1,0,0,@COUNT(A:A1..A10000),
@COUNT(A:A1..HH1))
```

For a range that starts at A5, the latter sort of formula is needed. This should work, if the number of items in the column will be less than 10,000.

```
@OFFSET(A:A5,0,0,@COUNT(A:A5..A10000),1)
```

And if we want to take account of the possibility that there may be gaps in the entries, the final formula in the last application can be used as follows:

```
@OFFSET(A1,0,0,@MAX((@ISBLANK(A1..A10000)=0)
@ROW(A1..A10000)),1)
```

Remember that these `@OFFSET` functions return the coordinates *as text*. If a different function requires those coordinates *as non-text coordinates*, the functions should be wrapped in an `@@` function.

How to count the non-blank cells in the index column of a given block

Assume that we’re given the coordinates of a block in cell A1, a block which could be almost anywhere in the spreadsheet. The leftmost column of the block is an index column. We want to know how many items are in the block. For such a block, this formula should do:

```
@COUNT(@@(@OFFSET(@@(A1),0,0,10000,1)))
```

This formula will not work if it appears in one of the last 10,000 rows of the spreadsheet, which by default has one million rows. If it is needed that far down, an adjustment of the `10000` argument in the formula should work.

How to get the coordinates of a named block

Using the `@BLOCKNAMES` function, which returns two-columns of block names and addresses, we can use the array it creates with an `@INDEX` function to select the coordinates in typical cases. For example, if we want to find the coordinates for a block named *Invoice*, this function should ordinarily work:

```
@INDEX(@BLOCKNAMES(Invoice),1,0)
```

However, given the possibility that there may be overlapping blocks, this will not necessarily return the desired information; it may return the coordinates for one of the other blocks that intersect with the coordinates of *Invoice*. If overlapping blocks are a possibility, then instead we can use the slightly more complex `@VLOOKUP` function, as follows:

```
@VLOOKUP("Invoice",@BLOCKNAMES(Invoice),1,0)
```

A simpler method exists using `@PROPERTY`:

```
@PROPERTY("Invoice.Selection")
```

returns the coordinates of a block called *Invoice*.

How to get the value in column that is parallel to the largest value in another column.

A user wanted to know how to find the value in a column (here, A2..A30) that is parallel to the greatest value in another column (here, B2..B30). Here are two ways that work:

```
@INDEX(A2..A30,0,@MATCH(@MAX(B2..B30),B2..B30,0))
```

And:

```
@ARRAY(@LOOKUP(@MAX(B2..B30),B2..B30,A2..A30))
```

How to get every 10th value in a column

Assume that you want to place the contents of A1, A11, A21, etc. into B1, B2, B3, etc. Put this formula into cell B1 and copy it into lower cells:

```
@INDEX($A$1..$A$10000,0,(@ROW(A1)-1)*10)
```

QP will wrap this in `@ARRAY`. If the list in A extends beyond 10,000, the number should be increased. Essentially, this function returns the value in the block a1..a10000 in the row calculated by `(@ROW(A1)-1)*10`, which is copied as `@ROW(A2)` in B2, `@ROW(A3)` in B3, and so on. That parameter returns a value of 0 in A1 because we subtracted 1 from it, thus, 0 rows below A1), 10 in B2 (thus, 10 rows below A1), and so on.

How to get/sum the last five values in a column to which data is added

In Table 10.5, the task is to come up with formulas for cells B1..B5 that will always reflect the last five values in the A column, to which data will be added. (This arose in the context of setting up a graph that would show only the last five values.)

The formulas in B1..B5 are shown in C1..C5, and they will recalculate as entries are added to the bottom of column A. `@INDEX` returns the value from the A column, with the row offset determined by subtracting from the total number of entries (determined by `@COUNT`) the numbers 5 through 1. Adding another number in A10 would have the result of moving the results in B1..B5 up one cell, and the content in A10 would then appear in B5.

Another user wanted to know how to sum the last five values in a column to which numbers would be added. The address for the last five rows in the column would be this:

```
@OFFSET(A1,@COUNT(A1..A1000)-5,0,1)
```

Table 10.5: Getting the last five values in a column

	A	B	C
1	0.226	0.656	⇐ @INDEX($A$1..$A$1000,0,@COUNT($A$1..$A$1000)-5)
2	0.417	0.073	⇐ @INDEX($A$1..$A$1000,0,@COUNT($A$1..$A$1000)-4)
3	06.79	0.767	⇐ @INDEX($A$1..$A$1000,0,@COUNT($A$1..$A$1000)-3)
4	0.133	0.052	⇐ @INDEX($A$1..$A$1000,0,@COUNT($A$1..$A$1000)-2)
5	0.656	0.502	⇐ @INDEX($A$1..$A$1000,0,@COUNT($A$1..$A$1000)-1)
6	0.073		
7	0.767		
8	0.052		
9	0.502		
10			

Formulas for returning the last five entries in the A column, as entries are added.

and thus the function that sums the last five entries in the column is this:

```
@SUM(@@(@OFFSET(A1,@COUNT(A1..A1000)-5,0,5,1)))
```

How to average the last three numbers, skipping blanks

In WPU 37720, a user asked how to average the three latest numbers in a column (N) that included numbers or blanks, and place the calculations in the M column. I offer two solutions, both of which require a helper column.

I placed random numbers in cells in N1..N20, but left six cells blank. I populated the helper column by placing `@ARRAY(((@ISBLANK(N1..N20)=0)*@ROW(N1..N20)))` in O1. This creates an array in o1..o20 that is populated by zeros or row numbers, depending on whether there is a number in the row in column N.

Solution 1: Finding the three numbers

On my assumptions, the first solution is to place this formula in cell M20:

```
(@INDEX(N$1..N20,0,@LARGEST(o$1..o20,1)-1)
+@INDEX(N$1..N20,0,@LARGEST(o$1..o20,2)-1)
+@INDEX(N$1..N20,0,@LARGEST(o$1..o20,3)-1))/3
```

The three `@LARGEST` functions will return the first, second and third largest row numbers, which will be the rows of the desired numbers. Each of those is used as the row-offset in an `@INDEX` function, but since `@ROW` in the helper column starts numbering with 1, but `@INDEX` starts numbering with 0, we have to subtract 1 from the row number. `@INDEX` then returns the numbers from the desired cells, each of which are added, and the sum divided by three.

Copy this function from M20 to cells above it. The copied functions will return `ERR` when there are no longer 3 values to add, but otherwise, it will average the three. The formula can also be copied down.

Solution 2: Applying @PUREAVG to a block

Place this formula in cell M20 and copy it to the cells above:

```
@PUREAVG(@@(@OFFSET($N$1,@LARGEST(o$1..o20,3)-1,0,
(@ROWS(n$1..n20)-@LARGEST(o$1..o20,3)+1),1)))
```

Here, the block in which the desired numbers appear will be calculated by the `@OFFSET` function, converted from text into usable coordinates by the `@@` function, and then averaged with the `@PUREAVG` function. So, the trick will be getting @OFFSET to identify the correct block. The `@OFFSET` function has the anchor point (N1), the column offset (0) and the column width (1), so it needs only the number of rows to drop down below the anchor point to find the starting cell, and the number of rows of cells to include. The number of rows to drop down will be the third largest number less one (since `@OFFSET`, like `@INDEX`, starts numbering at 0), which is supplied by `@LARGEST(o$1..o20,3)-1`. The number of rows to be included will be the total number of rows less the starting point, plus one, which is supplied by `(@ROWS(n$1..n20)-@LARGEST(o$1..o20,3)+1)`.

How to determine if the cellpointer is in a particular block

The task is to determine whether the Cellpointer (active cell) is in a particular block. This addresses two-dimensional blocks; a three-dimensional block is not exactly parallel.

For a block on a single sheet, a cell must pass three tests to be within the block; if it fails any of the tests, it is outside the block. (1) It must be on the same sheet. (2) Its column number must be within the range of the columns defining the block. (3) Its row number must be within the range of rows defining the block. `@CELLPOINTER` gives us its locations with its `sheet`, `col`, and `row` arguments.

If we know the block in advance, we can see if the three numbers that `@CELLPOINTER` gives equal the sheet number, are greater than or equal to the left-most column number, less than or equal to the right column number, and likewise for the row numbers. Here, lets say that the block is A:A1..C4. The Sheet is 1, columns are 1, 2, and 3; rows are 1, 2, 3, 4. This formula will return 1 if the cellpointer is in the block, 0 if it is not:

```
@CELLPOINTER("sheet")=1
#AND#@CELLPOINTER("col")>=1
#AND#@CELLPOINTER("col")<=3
#AND#@CELLPOINTER("row")>=1
#AND#@CELLPOINTER("row")<=4
```

which can be wrapped with an `@IF` test to put the result into English:

```
@IF(@CELLPOINTER("sheet")=1
#AND#@CELLPOINTER("col")>=1
#AND#@CELLPOINTER("col")<=3
#AND#@CELLPOINTER("row")>=1
#AND#@CELLPOINTER("row")<=4,"in block",
"not in block")
```

But what if we want to be able to code a function and put a block into it without first calculating all of the sheet, column, and row numbers? We can use `@CELL` to get the sheet, column and row numbers for the first cell in the block, and we can use `@ROWS` and `@COLS` to get the number of rows and columns in the block so as to calculate the last row and column numbers, and then substitute those into the tests above. But that is cumbersome.

There is, however a more elegant way to determine whether the cellpointer is in the range of columns and rows of a block, using `@MATCH` with `@ROW` and `@COLUMN`. Specifically, the cellpointer will be in the same columns as the block if this function returns a number:

```
@MATCH(@CELLPOINTER("col"),@COLUMN(Block),0)
```

If it returns ERR, the cellpointer is not in the block. Likewise, the cellpointer is in the same rows as the block if this function returns a number:

```
@MATCH(@CELLPOINTER("row"),@ROW(Block),0)
```

In order to turn these results so that the return 1 if the cellpointer is in the range and 0 otherwise, they need to be wrapped in `@NOT(@ISERR())`.

Pulling all three tests together, this formula works:

```
@CELL("sheet",Block)=@CELLPOINTER("sheet")
#AND#@NOT(@ISERR(@MATCH(@CELLPOINTER("col"),
@COLUMN(Block),0)+@MATCH(@CELLPOINTER("row"),
@ROW(Block),0)))
```

And to wrap it in an `@IF` formula:

```
@IF(@CELL("sheet",Block)=@CELLPOINTER("sheet")
#AND#@NOT(@ISERR(@MATCH(@CELLPOINTER("col"),
@COLUMN(Block),0)+@MATCH(@CELLPOINTER("row"),
@ROW(Block),0))),"not in block","in block")
```

How to determine if a value is in a column or row

The `@MATCH` function determines whether a value is in a column or row of values (see page 84), and if it is, it returns the offset number, but if not, it returns ERR. This can have unfortunate consequences for other folders, so we often need to modify it to return 1 if the value is in the block, and 0 if it is not. To test for whether those contents are in the block, yes or no, use `@ISERR`. This formula returns 1 if the contents *are not* in the block, and 0 if they are in the block. To reverse this, so that the formula returns 1 if the contents *are* in the block, and 0 otherwise, use `@NOT`.

```
@NOT(@ISERR(@MATCH(Value,Block,0)))
```

The `Value` argument can be hard-coded, a cell reference, or anything else that returns a value. The `Block` must be a one-dimensional (single row or single column) block. This variation of the `@MATCH` function will return 1 if `Value` is in `Block`, and 0 if it is not.

How to mark duplicates in a column

Assume that you want to determine if there are any duplicate entries in the block A1..A100. An easy way to do this would be to put this formula into B2 (or into the second cell of any column that is empty from row 1 to row 100, but not the first cell), and then copy the formula into the cells parallel with A1..A100 below:

```
@IF(@NOT(@ISERR(@MATCH(A2,$A$1..A1,0))),
"Match!!","")
```

This tests each cell from A2 down to see if it matches an earlier one, and if it does, it shows Match!! in the cell.

How to return the (first) matching row of data in a database

A common task is for functions to find an item in a database and return associated information on the same row. The item can be hard-coded into the function, but it is often simply typed into a cell. When a different item is typed into the cell, the function recalculates and returns information from the appropriate row.

`@MATCH` is used to get the row offset in the database, and `@INDEX` can then easily return data from that row by column. As long as the item is unique, this method is straightforward. If the item is not unique, however, this method returns only the first match. That may suffice, but if not, the complications are explored in the next applications.

Table 10.6 contains a database of random data in A1..C7. We want to type a name in E2, and have QP show the row in the database in E5..G5.

Table 10.6: Getting information from a database with @MATCH

	A	B	C	D	E	F	G
1	Name	Date	Score		Search For		
2	Eddie	06/18/14	71		Dotty		
3	Betty	02/10/10	72				
4	Dotty	11/02/09	73		Name	Date	Score
5	Eddie	07/03/11	73		Dotty	11/02/09	73
6	Fannie	01/12/12	73				
7	Ginny	10/30/10	81				

We first get the row offset with the formula `@MATCH(E2,A2..A7,0)`, which returns an offset value of 2 in this example. That will be inserted into an `@INDEX` function placed in E5..G5 to return data from that row:

E5 = `@INDEX(A2..C7,0,@MATCH(E2,A2..A7,0))`
F5 = `@INDEX(A2..C7,1,@MATCH(E2,A2..A7,0))`
G5 = `@INDEX(A2..C7,2,@MATCH(E2,A2..A7,0))`

Typing into E2 any name that appears in A2..A7 will cause the functions in E5..G5 to return the first matching row from the database.

Typing anything else into E2 will cause the functions to return ERR, because that is what `@MATCH` returns if there is no match.

But note that typing *Eddie* into E2 will return only the values from row 1, not the values from row 5, because `@MATCH` finds only the first match.

How to find and return matches after the first one

As just noted, `@MATCH` will find the first matching item in a column/array, but it is not set up to match later ones. (It would be great if Corel could devise a function that returned later matches with an additional parameter, as exists in functions like `@LARGEST`.) What if we want to return every match, not just the first one?

Notebook queries can be set up to get all matches fairly painlessly, but they do require their own set-up and manual operation, even if that is made easier by macros. The QuickFilter can also be used to get useful information. Each requires manual operation.

In this section, we want to do it with @Functions, which can be used, but it requires some careful work.

We will use random data (4 people, 3 instances each, randomly arranged) as shown in A1..B13 of Table 10.7. There are at least three strategies for getting matches beyond the first one.

- One method is to restrict the block that `@MATCH` searches in each row, so as to exclude prior matches.
- Another is to create a parallel array of values in which each row has a unique value, but that the values for rows that should match will have a particular quality that allows them, and only them, to be found by @functions.
- A third is to create a parallel array of values in which the values increment (from 0 to 1, then 2, etc.) only when a match occurs, in which case, `@MATCH` for successive numbers starting with 1 will return the desired rows.

All of these methods require the use of **helper columns**. These are columns in which some intermediate calculations are done, before the functions return the desired data. Aesthetically, they may not be very pleasing, but if so, they can be **hidden**. To hide columns, select the column headers (like C, D, etc.) to be hidden and right click, and in the popup menu, select "Hide." (To reverse the operation, select the columns on both sides of the hidden columns, right click, and select "Reveal.")

You will also need to decide how many rows of functions that you want to set in your field for results. We don't know in advance whether any or all of the items in the column to be searched will match. The only way to be sure to get all matches is to have a number of rows of functions equal to the rows of the data itself. For most purposes, you will be able to make do with less.

Method 1: Changing the block

In Table 10.7, we type the name to be returned in C1. The helper columns D and E will successively set the block to be searched (column E) and the row number (not row offset) on which a match is found (D). They start with the initial search block in E1. The `@MATCH` function in D2 looks in the range in E1, and either finds a match or returns ERR; if it finds a match, it calculates the row offset number on which the match can be found. The `@CONCATENATE` function in E2 calculates the range to search on the next row (using the function in D3 here). The `@@` function in F2 returns the name on the row specified in D2, but if D2 is ERR, and `@IF` function causes it to be blank. The same is true for the function in G2, which returns the score on the row specified in D2. We can then copy the functions in D2..G2 down as long as we want. Here, they are copied down to row 5.

Here is how to set up the table:

```
@MATCH($C$1,@@(E1),0)+D1+1
```

is in D2. `@MATCH($C$1),@@(E1),0)` looks for an exact match of C1 in the block specified in E1. We add D1 (`+D1`) to the resulting offset number simply to anticipate

Table 10.7: Finding matches by successively changing the block to be searched

	A	B	C	D	E	F	G
1	Name	Score	Andy		a1..a1000	Name	Score
2	Dotty	95	1	5	a6..a1000	Andy	89
3	Dotty	73	2	7	a8..a1000	Andy	71
4	Charlie	73	3	10	a11..a1000	Andy	99
5	Andy	89	4	ERR	ERR		
6	Dotty	95					
7	Andy	71					
8	Betty	82					
9	Charlie	88					
10	Andy	99					
11	Betty	72					
12	Betty	73					
13	Charlie	91					

copying this formula down the column, because each number in this column must be cumulative of prior rows. We add 1 (+1) because the search block started on row 1. If it started on row 3 (e.g., if E1 contained `a3..a1000`), we would need to add +3 instead of +1.

```
@CONCATENATE("a",D2+1,"..a1000")
```

is in E2. This creates the next range in which to search.

```
@IF(@ISERR(D2),"",@@(@CONCATENATE("a",D2)))
```

is in F2. `@CONCATENATE("a",D2)` returns "a5" as text, and when wrapped in the `@@` function, it returns the value in A5. This is wrapped in an `@IF` test to determine if D2 found a true match. This cell does not need that, but when it is copied down to F5, this cell would return ERR but for this `@IF` test.

Both this and the function in G2 assume that the search starts on row 1. If not, the difference between the row on which it starts and row 1 must be added to the second `D2`. Thus, if E1 contained `a3..a1000`, this function would be `@IF(@ISERR(D2),"",@@(@CONCATENATE("a",D2+2)))`.

```
@IF(@ISERR(D2),"",@@(@CONCATENATE("b",D2)))
```

is in G2. It returns the value in the B column on the row identified in cell D2.

This is complicated, but it works. Change the name in C1 to any of the other three names, and the data in F2..G5 change for that person.

Method 2: Array of Unique Values

In this approach, shown in Table 10.8, we create a helper column in C that will assign each row a unique value by adding two numbers: *(a)* 1 or 0, depending on whether the row matches the name in D1, and *(b)* a decimal calculated as 1 divided by the row number. The result is that all matching rows retain values greater than 1, but they get progressively smaller as we go down the column, and each row will therefore have a unique value. With unique values, we can calculate the largest in order in E2..E5 with the `@LARGEST` function. We determine the offset rows in F2..F5 that use `@MATCH` to find the four largest values in the helper C column. With those offsets, it is easy to generate the names and scores in the G and H columns.

Table 10.8: Assigning distinctly high values to matches

	A	B	C	D	E	F	G	H
1	Name	Score		Andy			Name	Score
2	Dotty	95	0.500	1	1.200	4	Andy	89
3	Dotty	73	0.333	2	1.143	6	Andy	71
4	Charlie	73	0.250	3	1.100	9	Andy	99
5	Andy	89	1.200	4	0.500	1		
6	Dotty	95	0.167					
7	Andy	71	1.143					
8	Betty	82	0.125					
9	Charlie	88	0.111					
10	Andy	99	1.100					
11	Betty	72	0.091					
12	Betty	73	0.083					
13	Charlie	91	0.077					

We set up the table this way:

```
(A2..A13=D1)+(1/@ROW(A2..A13))
```

is in C2. QP will wrap this in `@ARRAY` and automatically fill in cells C3..C13.

```
@LARGEST($C$1..$C$1000,D2)
```

is in E2. D2 supplies the ranking (1), so this formula returns the largest number in the helper C column. When copied to lower cells, it uses the changing rank in the D column to tell QP what to look for in the F column.

```
@MATCH(E2,$C$1..$C$1000,0)
```

is in F2. This returns the row offset in the C column of the unique value in E2.

```
@IF(E2>1,@INDEX($A$1..$B$1000,0,F2),"")
```

is in G2. With the row offset in F2, we can find the name in the A column. The `@IF` test eliminates non-matches, because any match must have a value greater than 1. If the number in the E column is not greater than 1, this function returns a blank.

```
@IF(E2>1,@INDEX($A$1..$B$1000,1,F2),"")
```

is in H2. This returns the score, using rhe row offset in F2. E2..H2 can be copied down as many rows as desired, as long as numbers in the D column continue as indicated.

Method 3: Array of Values that Increment with Matches

In this approach (see Table 10.9), which I believe is simplest, we create a helper column in C that will assign each row a value that is determined by whether it matches the name in D1. If it matches, 1 will be added to a running total. If not, the running total remains the same. Since `@MATCH` will return the offset of the first value matching the numbers in D2..D5, `@MATCH` generates offsets in E2..E5 that allow us to get the names and scores in columns F and G.

Table 10.9: Incrementing a counter value with each match

	A	B	C	D	E	F	G
1	Name	Score		Andy		Name	Score
2	Dotty	95	0	1	4	Andy	89
3	Dotty	73	0	2	6	Andy	71
4	Charlie	73	0	3	9	Andy	99
5	Andy	89	1	4			
6	Dotty	95	1				
7	Andy	71	2				
8	Betty	82	2				
9	Charlie	88	2				
10	Andy	99	3				
11	Betty	72	3				
12	Betty	73	3				
13	Charlie	91	3				

This table is set up as follows:

```
+C1+(A2=$D$1)
```

is in C2. This gets the starting value of 1 or 0. It is copied to C3..C13. This gives a running total of matches.

```
@IF(D2<=@MAX($C$1..$C$1000),
@MATCH(D2,$C$1..$C$1000,0),"")
```

is in E2. The `@MATCH` component looks for the first place in the helper C column in which the number in the D column appears. It is wrapped in an `@IF` test that will return a blank if the number in the D column exceeds the maximum number in the C column (which is also the maximum number of matches in the A column). This is copied to the cells below.

```
@IF(D2<=@MAX($C$1..$C$1000),
@INDEX($A$1..$B$1000,0,E2),"")
```

is in F2. The `@INDEX` component of this function finds names in the A column at the offsets determined in the E column. The `@IF` test will return a blank if the number in the D column exceeds the maximum number in the C column. It is copied to rows below.

```
@IF(D2<=@MAX($C$1..$C$1000),
@INDEX($A$1..$B$1000,1,E2),"")
```

is in G2. This returns the score in parallel with names returned in the F column.

How to find the top three scores in a database

Taking the last application and going one step further, we now use functions to return the top 3 values in a database with random scores. There are again at least two ways to do it, depending on whether we use the technique of restricting the range (method 1) or assigning a unique value (method 2).

Method 1: Changing the block

Starting with the database of people and scores in A1..B10, we want to show the top three scores, and in case of ties, we may show more than three names, though in this case we will max out at six. In Table 10.10, we use three helper columns (C..E) and display our results in F..H.

Table 10.10: Ranking, using different blocks for equal scores

	A	B	C	D	E	F	G	H
1	Andy	87		TargetScore	Offset	Place	Name	Score
2	Betty	96	1	98	7	1	Henry	98
3	Charlie	96	2	97	6	2	Ginny	97
4	Dotty	80	3	96	1	3	Betty	96
5	Eddie	78	4	96	2	3	Charlie	96
6	Fannie	95	5	95	5			
7	Ginny	97	6	87	0			
8	Henry	98						
9	Izzy	87						
10	Johnny	78						

Column C simply contains consecutive numbers for the maximum of six slots that we will show. Column D uses `@LARGEST` to get the values that `@MATCH` will find. Column E calculates the offset of the row where the match will be found, using `@IF` to choose between two `@MATCH` formulas, depending on whether we're looking for someone whose score tied the person in the earlier row. Column F uses `@RANK` to return the ranking. Columns G and H use `@INDEX` to get the data from A and B based on the offset calculated in E. Specifically:

```
@LARGEST($B$1..$B$1000,C2)
```

is in D2. This function returns the largest values in the B column, based on the number in the C column. It treats duplicates separately, as in cells D4..D5, which return the same value because there are two occurrences of the score *96*.

```
@IF(D2<>D1,@MATCH(D2,$B$1..$B$1000,0),
@MATCH(D2,@@(@OFFSET($B$1, E1+1,0,1000)),0)+E1+1)
```

is in E2. Let's break E2's function down into its three parts.

1. With `@IF(D2<>D1`, the function first tests whether the cell in the D column equals the cell above it, which tells the function whether we've already found a person with that score, which in turns tells us whether we look for the first occurrence of that score with `@MATCH` in the entire block, or we restrict the block.
2. Thus, if we are looking for the first occurrence, we use the simpler search function, `@MATCH(D2,$B$1..$B$1000,0)` to get the correct block.
3. If we are looking for the next occurrence, we use the more complex search, `@MATCH(D2,@@(@OFFSET($B$1,E1+1,0,1000)),0)+E1+1)`. This formula creates a smaller block in which to search, `@OFFSET ($B$1,E1+1,0,1000)`, which starts the block from the cell below the offset calculated in the row above (`E1+1`), which is turned into coordinates by the `@@` function. But since this block start lower than the top row, to create an offset from the top cell that

`@INDEX` can use in columns G and H, we must add the offset again at the end (`+E1+1`).

```
@IF(@RANK(D2,$B$1..$B$1000,0)<=3,
@RANK(D2,$B$1..$B$1000,0),"")
```

is in F2. The `@RANK` function tells us what relative rank the value in D2 has in the numbers in the B column, and the trailing *0* makes it descending order, so that the top score has rank 1. This is wrapped in an `@IF` function that tests whether the rank is below 3 or not; if it is, the function returns a blank (since we only want to show the top 3 places); otherwise it returns the rank. The formulas here show a tie for third place.

```
@IF(F2>0,@INDEX($A$1..$B$1000,0,E2),"")
```

is in G2. This uses the offset in the E column to get the name in A, but if this is not one of the top three scores, the `@IF` function leaves it blank.

```
@IF(F2>0,@INDEX($A$1..$B$1000,1,E2),"")
```

is in H2. This does the same as the last function in getting the score from column B.

D2..H2 are then copied into the rows below.

Method 2: Array of Unique Values

The method in Table 10.11 uses a fourth helper column (C..F), but it is easier to comprehend.

Table 10.11: Ranking by assigning unique values

	A	B	C	D	E	F	G	H	I
1	Andy	87	1.000		TargetScore	Offset	Place	Name	Score
2	Betty	96	1.5000	1	100.125	7	1	Henry	98
3	Charlie	96	1.333	2	10.143	6	2	Ginny	97
4	Dotty	80	0.250	3	1.500	1	3	Betty	96
5	Eddie	78	0.200	4	1.333	2	3	Charlie	96
6	Fannie	95	0.167	5	1.000	0			
7	Ginny	97	10.143	6	0.250	3			
8	Henry	98	100.125						
9	Izzy	87	0.111						
10	Johnny	78	0.100						

This is how to set up the table:

Column C creates a unique score for each number in B in this way. If the score matches what would be returned as the largest value in column B, using `@LARGEST`, the result (1) is multiplied by 100; if it matches the second largest, the result (1) is multiplied by 10. Any match equal to the third largest will be 1, and non-matches will start as 0. To each, we add a decimal calculated as 1 divided by the row number. The result is that the ranking will keep all ties for the same spot in numeric order, but within each tie, the values get progressively smaller as we go down the column, and each row will therefore have a unique value that can be found by `@MATCH`. Thus:

```
((B1..B10=@LARGEST(B1..B10,1))*100)
+((B1..B10=@LARGEST(B1..B10,2))*10)
+((B1..B10=@LARGEST(B1..B10,3)))
+(1/@ROW(B1..B10))
```

is in C1. QP wraps this in `@ARRAY` and fills column C.

The D2..D6 block contains consecutive numbers for the maximum number of rows we will display.

```
@LARGEST($C$1..$C$10,D2)
```

is in E2. It finds the largest unique value in the C column, as called for by D2, which contains 1, the value in D2.

```
@MATCH(E2,$C$1..$C$10,0)
```

is in F2. It locates in the C column the offset row of the unique value in E2.

```
@IF(E2>100,1,@IF(E2>10,2,@IF(E2>1,3,"")))
```

is in G2, which calculates the rank. We can't use `@RANK` because we created unique values in column C, so no ties would be possible. This formula does the same thing by treating anything with a value over 100 as tied for rank 1, anything with a value over 10 as tied for rank 2, and anything with a value over 1 as tied for rank 3, but anything equal to or less than 1 as not a top-3 score.

```
@IF(E2>1,@INDEX($A$1..$B$10,0,F2),"")
```

is in H2. It takes the offset value in column F and returns the name from A, as long as the cell in G is not blank.

```
@IF(E2>1,@INDEX($A$1..$B$10,1,F2),"")
```

is in I2. It takes the offset value in F and returns the score from B, as long as the cell in G is not blank.

E2..I2 are then copied into the rows below.

And now, we press on to QP's macros, which will enable the user to automate much more.

Part III

Quattro Pro Macros

Chapter 11

Introduction to Quattro Pro Macros

A macro is a set of commands that cause the program to perform certain tasks automatically. The primary subject of this section is QP's own historic "native" macro language. A more or less parallel version of that language exists in Corel's PerfectScript, and some of the ways it extends QP's native macros will be discussed. Peculiarities of QP programming in PerfectScript will be addressed in Chapter 24, beginning at page 235.

Unfortunately, QP's built-in help files are unhelpfully split into two parts, so that basic information about macro programming is contained in the main help file under "Automating tasks" (accessible by Help > Help Topics), while details about macro commands appear in a separate help file (accessible by Help > Macro Commands). Furthermore, the discussion of "Using macros" in the main help file is not in an order conducive to learning for a beginning reader. In fact, I can't deduce what order it is in. I would recommend that the beginning reader start with the final entry there, "Reference: Using macros" and skip the parts in it dealing with VBA. Then jump back a few entries to "Playing macros" to get the mechanics of running macros.

Macro commands and text

QP's native macros are text written directly into the cells of a QP notebook. They may be located on a completely different sheet of the notebook. They may even be located in a different notebook that is open, or a hidden "system" notebook.

Macro commands are contained in braces like these: `{}`. Like functions, the braces contain either:

- A simple one-word command, or
- A command and one or more arguments.

  To QP, an argument that refers to a cell in R1C1 reference style is ambiguous: it could refer to a cell relative to the current selected cell, or it could refer to a cell relative to the current macro command. To distinguish them, brackets in front of the address will render it relative to the cursor; without brackets, the command executes relative to the macro command. E.g.,

  `{Let C(0)R(5),"From macro"}` puts "From macro" five cells below that command.

`{Let []C(0)R(5),"From cursor"}` puts “From cursor” five cells below the cursor.

See page 118 below for more information on this problem.

Macros may also contain text. The textual content may be:

- Ordinary text. Essentially, text in a macro script will be handled as if the user typed the same keystrokes while in Edit mode, which is not the normal mode of data entry. Normally one enters text by selecting a cell and typing. To change what was already written, one can enter Edit mode by pressing `[F2]`. One can enter text for the first time in the same mode, as well, but in either case, one must leave Edit mode to continue, usually by pressing `[Enter]` to keep the changes or `[Esc]` to abort the changes.

  So, be mindful of stray keystrokes in your macro script, particularly spaces at the end or beginning of a cell. QP will attempt to type spaces when it reaches that point.

  Such ordinary text can become a number, a formula, or a function. The script will handle them just as if the user had personally typed them.
- `~` (the tilde) is a special case. It usually functions like typing the `[Enter]` key, as the help file says, but (undocumented) in the context of entering text into a cell, it simply ends the process of text entry. Thus, if the default action of the cursor is to drop down a line after pressing the enter key, one tilde in the macro at the end of text entry will not cause the cursor to move, but two tildes would.
- Comments. These are macro commands that begin with a semicolon, and can be used to document the macro or for any purpose other than causing QP to execute commands. Example: `{; my non-executing comment}`

Dynamic Modification

Macros may change dynamically. They can consist of text formulas (page 5) that change, depending on the changing content of other cells. Doing so can provide a workaround where QP requires the macro to be hard-coded.

For instance, one has to hard-code offsets when using R1C1 style in macros, which means that if one wishes to put the contents of cell A1 into, say, a specific cell to the right of the cursor without moving the cursor, one must hard-code numbers into the parentheses. `{Let []C(1)R(0),A1}` will work because a number is coded into the parentheses. Referring to the number in cell A2, as in `{Let []C(A2)R(0),A1}`, would not work. However, using a text formula or function to construct the macro command surmounts this difficulty:

```
+"{Let []C("&@STRING(A2,0)&")R(0),A1}"
```

Such functionality may be extended quite far. One of David Seitman’s macros examines screen conditions and then writes many lines of macro code, which it then executes. A macro of Jack Kearns sets up a master sheet of keystroke macros that allow the same keystroke macro to perform different functions, depending on which sheet of the QP notebook is the current one.

Running Macros

When the macro commands are written in cells in the spreadsheet, the macro can be run by either:

- Pressing `[Alt+F2]`, selecting the starting cell, and clicking `[OK]`.
- Associating that cell with a command button (see page 118), and clicking the button.
- Naming the cell in a way that allows a keystroke combination to run it.

Normal sequence of commands

The typical macro executes commands starting from one cell, and proceeding to commands in the next cell below, from top to bottom, until the last command is executed and QP reaches a blank cell. There may be more than one command in a cell, and in that case, QP *typically* executes commands from left to right before going to the next row.[1]

Table 11.1 shows a simple macro script in cells B1..B3. When the user runs the macro by pressing `[Alt+F2]`, selecting cell B1, and clicking `[OK]` (or `[Enter]`), QP runs through the script from top to bottom, and within cells from left to right, as described in cells C1..C4.

Table 11.1: Sequence of macro commands

	A	B	C
1		{Home}	⇐ Cursor/cellpointer goes to A1
2		{D}@TODAY~	⇐ Goes down one cell; Enters @TODAY function
3		{U}Today~	⇐ Goes up; Enters the word "Today"
4			⇐ Script stops at the blank cell

Making commands conditional: {If}

The `{If *Test*}` command is just a gateway to further commands in the same cell. If the `Test` argument tests true, then later commands in the same cell execute. If it tests false, then the macro skips to the next line down. Modifying the last illustration, we add `{If}` tests in B2 and B3 in Table 11.2.

Table 11.2: Conditional commands with {If}

	A	B
1		{Home}
2		{D}{If A2=""}@TODAY~
3		{U}{If A1=""}Today~

The net result is that the script will put the `@TODAY` function only if the formula `A2=""` tests true. Otherwise, the macro will do nothing in cell A2. It will run a similar test in A1, and if `A1=""` tests as true it will type *Today*, but if something is there, the macro will do nothing more.

The `Test` argument tests a proposition that either evaluates as true (1, or apparently any non-zero number) or false (0). The `Test` argument usually involves equations and inequalities (such as =, >, <, >=, <=, <>), but it can include functions

[1]The text file commands discussed on page 204 in Chapter 21 appear to be an exception: after executing one, further commands on the same line are not executed.

such as the `@IS***` functions. Both sides of an equation in `Test` may include formulas or functions. Different equations can be combined in the `Test` argument by the connectors `#AND#` and `#OR#`.

Detouring or Breaking: {Cell} and {Branch Cell}, {Quit}, {Return}

The normal sequence of commands to be executed may be altered commands to jump to commands stored in a different (or "detour") cell. If you want the macro commands to return to the original sequence, you create a *subroutine* by placing the detour cell in braces. If you do not want the program to return to the original sequence, use a `{Branch}` command.

Subroutines. The command `{Z1}` would cause the macro to detour to cell Z1 and start executing commands there. When those commands concluded, the macro would return to the point in the original sequence after the `{Z1}` command, and it would then proceed to execute other commands.[2]

Branching. The command `{Branch Z1}` would cause the macro to detour to cell Z1 and start executing commands there, but it would not return to the original sequence.

The macro in Table 11.3, which fills A1..A4 with 1,2,3,4, illustrates the difference between subroutines and branches.

Table 11.3: Branching and subroutines

	A	B	C	D
1		{Let A1,1}		
2		{C2}	{Let A2,2}	
3		{Branch D4}	{Let A3,3}	
4		{Let A4,0}		{Let A4,4}

Subroutine in B2; Branch in B3

After executing the command in B1, the `{C2}` command in B2 passes control to the subroutine at C2. After executing commands in C2 and C3, the macro automatically returns to the B column, and then executes the `{Branch}` to D4, which executes the command in D4. Then the macro stops because it reaches a blank cell; it does not return to the B column, and thus the command in B4 does not execute.

Either subroutines or branches can be conditional. There can be multiple levels of branching and subroutines. That is, the macro in the A column can call a macro in the B column, which can call a macro in the C column, and so forth. There are limits to the number of nested subroutines, but I do not know the limit, and it is good programming not to test the limits.

The advantages of using subroutines and branches are several. What would otherwise be a very long series of commands is broken up into more readable segments. Each segment needs to be written one-time only, instead of being incorporated into each macro that would use it. Changes to the segment may thus be made in one place, rather than in every macro that my use it.

[2]Note that subroutines do not work reliably when working with external text files. See the discussion at page 205 below.

To pass parameters to a subroutine, begin the subroutine with the `{Define}` command. I see this command as a way to make macro commands more readable. Since that approach emphasizes the use of named cells and I don't, I simply refer the interested reader to it.

To loop back to the same line, use `{Branch c(0)r(0)}`. To branch to other cells in the same macro, which makes it more portable, change the value within `r(0)`.

Breaking. To stop all macros running before the normal ending point, use the command `{Quit}`. To end a subroutine before its normal ending point, use `{Return}`. This command will return control of the commands to the macro that called this subroutine. If no macro called this subroutine, `{Return}` stops all macro processing, just like `{Quit}`.

Error handling: {OnError}

`{OnError` *`BranchCell<,MessageCell,ErrorLocationCell>`*`}` tells the macro what to do when it encounters and Error condition. The `BranchCell` argument diverts the flow of the macro to commands appearing in a cell identified by the argument. The programmer may optionally store the error message itself in the cell provided in the `MessageCell` argument. The cell containing the macro command that produced the Error condition can be set with the `ErrorLocationCell` argument.

`{OnError D1}` thus branches to cell D1.

`{OnError c(0)r(1)}` branches to the cell below the one that caused the error, essentially telling the program to skip that step.

`{OnError c(0)r(0)}` branches back to the same cell that caused the error, essentially telling it to try one more time.

The command applies only to a subset of error conditions. According to the online help file, it applies when a command seeks to operate on a file that does not exist, but though it appears to be true of opening QPW files, it does not appear to be true of operations on other sorts of text files. I use it in connection with searches and notebook queries, in case the search fails to find a match.

Pausing: {?} and {Wait}

The `{?}` command suspends the macro until the user presses `[Enter]`. The user can do almost anything at that point. When the user presses `[Enter]`, that simply returns control to the macro. If another `[Enter]` is necessary to complete an operation, the macro should include the tilde `~`.

`{Wait` *`Expiration`*`}` suspends the macro until a particular date and time established by the `Expiration` argument. The date and time is a numeric value like that returned by `@NOW`. For instance, in order to suspend the macro until 6:00PM on March 18, 2015, the `Expiration` argument would need to be 42081.75. Since we humans do not usually find that method of keeping time helpful, and since QP would simply hang until that time, the `Expiration` argument is usually constructed with @functions like this, which pauses execution for three seconds:

```
{Wait @NOW+@TIME(0,0,3)}
```

Repetition (Looping) by branching back

By using the `{Branch}` command and branching to a position earlier in the same macro, the macro will essentially loop around. Unless some term in the macro script breaks the cycle, the macro will continue in the same infinite loop; it will continue doing the same things until the user presses `[Ctrl+Break]` or some other Error condition occurs. Since it is not desirable to have a macro run infinitely, the user should recognize the possibility of infinite looping and set commands in the macro that will stop it.

Infinite Loops. The macros in A1 and B1 of Table 11.4 are equivalent, infinite loops. They both cause the cellpointer to go down (`{D}`) and up (`{U}`), and then continuously repeat that sequence. This will continue until the user stops the repetition by pressing `[Ctrl+Break]`.

Table 11.4: Infinite loop

	A	B
1	{D}{U}	{D}{U}{Branch c(0)r(0)}
2	{Branch A1}	

Ending Infinite Loops with Conditions for Stopping. Infinite loops are not desirable, so the programmer should add conditions to terminate them. Table 11.5 is an example.

Table 11.5: Conditions for stopping an infinite loop

	A	B
1		{Let A1,A1+1}
2		{If A1>10}{Quit}
3		{Let A2,A1*10}
4		{Branch B1}

In this example, an otherwise infinite loop stops when the value in A1 is greater than 10. The `{Let}` command in B1 adds 1 to whatever number is current in A1 (initially 0). (We'll come back to `{Let}` later.) In B2, the macro tests whether A1 has increased beyond 10, and if so, the macro quits. Otherwise, the command in B3 places a number 10 times greater than A1 into cell A2. Then the command in B4 reverts to B1. At the end, A1 will have 11 in it, but A2 will have 100.

It is also common to loop down a column of data and to stop the loop when the data ends and blank cells begin. See How to test for blanks at 26. Another technique is to have the macro enter a value at the end of a column that will tell it when to stop (e.g., *9999* or *STOP*); loop down the column, testing the values in it (via `@CELLPOINTER` or `@INDEX` or similar commands) until stop-value is reached, and then optionally delete the stop-value.

Repetition for a specific number of loops: {For}

The user can define the number of times that macro commands can execute with the `{For}` command. This command has five required arguments, and it is therefore more complicated than most other commands, but once mastered, it is quite useful. The form is:

`{For` *`CounterCell,Start#,Stop#,Step#,MacroCell`*`}`

It repeatedly runs the macro starting at `MacroCell`, as long as the number stored in the `CounterCell` has not *passed* the `Step#`. The number stored in the `CounterCell` is initially the `Start#`, and on each time the macro in `MacroCell` is run, the number in `CounterCell` is increased by the number in `Step#`.

`CounterCell` is the cell where the number is stored that determines whether to continue running the macro. A blank cell is usually chosen. In addition to determining whether to run the macro at `MacroCell`, it can be used for other purposes. For instance, it can provide an argument for a function to process data on one row of a database after another, or one column in a database after another.

`Start#` is the initial value that the `{For}` command puts into the `CounterCell`. You can use any number, but often it is best to pick a number that can serve multiple purposes, such as the row number of a database.

`Stop#` stops the `{For}` command from executing the macro at `MacroCell`, if the value in `CounterCell` has *passed* the `Stop#`. Thus, if the `CounterCell` merely equals the `Stop#`, the macro at `MacroCell` will execute.

`Step#` is the number that the `{For}` command adds to the `CounterCell` after each time it executes the macro at `MacroCell`. Most of the time, this number will be 1, but it need not be. It could be 2, if you want the macro to work on every other item of data. It could be 7 if you want the macro to work on a particular day of the week. It could be a negative number, such as -1, if you want to work backwards from a higher `Start#` to a lower `Stop#`, such as searching a database for matches from the bottom of it rather than the top.

`MacroCell` is the cell in which a subroutine starts that the `{For}` command will play.

The `{For}` loop in Table 11.6 does the same thing that the last branching loop did: it runs the command at B3 10 times. The result will be that A1 contains 11, but A2 contains 100. This is because the `{For}` command will execute the B3 command even when A1 reaches 10, but when it reaches 11, it will stop before executing the B3 command.

Table 11.6: A simple {For} loop

	A	B
1		{For A1,1,10,1,B3}
2		
3		{Let A2,A1*10}

Stopping the loop with {Return} or {ForBreak}

Since the `{For}` command amounts to calling a subroutine, the *current* subroutine can be halted by a `{Return}` command. When that happens, the `{For}` command behaves as it does when the subroutine has finished normally: it adds the `Step#` to the `CounterCell` and determines whether to run the subroutine again.

In addition, *all* repetitions of the `{For}` command can be halted by the `{Forbreak}` command. The macro will proceed with commands after the `{For}` command. If you use `{ForBreak}` with nested `{For}` loops, be aware that the command will stop the `{For}` loop that is currently executing. If that loop is the nested (second) loop, it should stop the nested loop, but the original (first) loop should continue.

Structuring the {For} loop

Putting these concepts together, I find it useful to structure the subroutine like this:

1. Use an `{If}` test to see if we should terminate the `{For}` loop. If so, use `{Forbreak}`. This is particularly useful if we have passed the end of a database and there is no need to process further rows. In this connection, I typically use an if-test to determine if the macro has reached a blank cell. For constructing such tests, see the discussion above at page 26.
2. Use an `{If}` test to see if this instance need not be processed, but allow for further instances to be processed. If so, use `{Return}`.
3. Process this instance with one or more appropriate commands.
4. Use an `{If}` test to see if we should terminate the `{For}` loop. If so, use `{Forbreak}`.

Speed #1 – Display Options: {WindowsOff} and Redraw

QP gives you the option to increase the speed of macro execution by limiting the screen changes it makes and by limiting the recalculation of formulas, each of which take up some fractional parts of seconds. If you only use simple macros, speed will not typically be an issue, but if your macros require a lot of processing of a lot of data, you may wish to consider ways to speed up the process.

Windows On/Off

One way to prevent the QP window from redrawing is to use the `{WindowsOff}` command. At the point that it occurs, the window will not redraw as the macro executes until the macro ends or a `{WindowsOn}` command occurs.

For example, if at the start of a macro, the user is on sheet A, `{WindowsOff}` is executed, and then the macro selects a cell on sheet B for user input with `{?}`, the user will not see sheet B, only sheet A. The only indication that the user is on a different sheet will be the Cell indicator above the Row headers and, perhaps, the Cell Editing box above the Column headers. If the user needs to see the context in sheet B, place a `{WindowsOn}` command before the `{?}` command.

Panel On/Off?

QP's help file also offers the `{PanelOn}` and `{PanelOff}` commands, which appear to be parallel to `{WindowsOn}` and `{WindowsOff}`, but which are in fact obsolete. In the DOS days, the program had a "panel" that included a set of menu items on the top row of the screen and prompts that explained menu items on the next row. Those would change as the macro executed keystrokes to cycle through the menu system. `{PanelOff}` would prevent those menus and prompts from updating, which in turn would speed up macro execution. `{PanelOn}` would return the display to change during macro execution. As far as I can tell, there is no longer any such thing as a "panel" in QP, and therefore, this command does nothing.

Redraw Mode

In QP's settings (via Tools > Settings), under the *Macro* node, there appear to be four options under the topic *Suppress-Redraw*: namely, "None", "Window", "Panel", "Both".

It is not clear from this presentation whether "None" or "Both" applies to the verb "Suppress" or to the verb "Redraw", and the system is ambiguous:

- The help file would suggest that selecting "Both" would suppress the redrawing of both the Window and the Panel. (There hasn't been a "Panel" since the DOS days, I suspect.)
- The language of the macro that automates this choice leaves out "suppress," and therefore suggests that selecting "Both" causes everything to redraw during macro execution.

My investigation of all permutations suggests that the latter, the macro command language, is correct. The help file is in error, and the dialog box for QP's settings is misleading. Thus:

```
{Application.Macro.Macro_Redraw "Both"}
```

allows the QP window to redraw during the macro;

```
{Application.Macro.Macro_Redraw "None"}
```

suppresses the redrawing.

Usage Comments

How do these two sets of commands work together? My testing of permutations indicates that, unless a different command forces the window to redraw, either set of options can be used to suppress the redrawing, and the only way to see the window redraw during the macro using these commands is to set the Macro_Redraw option to "Both" (either by default or by using the command `{Application.Macro.Macro_Redraw "Both"}`) *and* to use `{WindowsOn}`.

As we will see (page 136), one way to force the window to redraw, that overrides any of these combinations, is to use the command `{Wait @NOW}`. There is no logical reason why that command should have that undocumented function, or for the help file's calling it "obsolete."

Recommendation. In view of these findings, my recommendation is that the programmer leave the Macro_Redraw option set at "Both" and use only `{WindowsOff}` to restrict the macro from redrawing the QP window. The starting assumption should be that the user should see how the macro progresses, and the programmer should restrict that vision only for a good reason, such as macro performance.

Speed #2 – Recalculation Options: Manual, {Calc} and {Recalc}

Another way to speed execution of a macro is to prevent it from recalculating formulas and functions. Each time data is entered, all formulas and functions in the entire spreadsheet recalculate by default. If you have few formulas or functions in the spreadsheet, this is acceptable, but if your macro causes numerous changes to the data, which trigger recalculations of a spreadsheet with lots of formulas and functions, the extra and unnecessary time spent in doing so may not be acceptable. In those cases, the user could consider changing the notebook's recalculation settings. Manually, one does this through the menu by Format > Notebook Properties and selecting the *Recalc Settings* tab, where the *Mode* section give the options of "Automatic", "Background" or "Manual".

Automatic forces all formulas and functions to recalculate immediately and stops everything until it finishes. The macro command is:

```
{Notebook.Recalc_Settings "Automatic;
Natural;1;Yes;No"}
```

Background recalculates between keystrokes. I'm not sure that I've ever noticed any difference between *Automatic* and *Background*. The command is:

```
{Notebook.Recalc_Settings "Background;
Natural;1;Yes;No"}
```

Manual recalculates nothing except the function that the user is working on, and by extension, any function that the macro is operating on. The command is:

```
{Notebook.Recalc_Settings "Manual;
Natural;1;Yes;No"}
```

Caution. If you use *Manual*, you must be careful about whether you are counting on the correct result of some formula or function that may not be updated. If you are, you can recalculate the entire spreadsheet when needed by adding the `{Calc}` command to your macro or have the best of both worlds by using the `{Recalc` *`CellOrBlock`*`}` command. This command will recalculate only the cells identified in the `CellOrBlock` argument on a row-by-row basis. If you need the recalculation on a column-by-column basis (as where you need to recalculate all formulas in the column to the left of a formula that depends on the column to the left), use `{RecalcCol` *`CellOrBlock`*`}` instead.

Recommendations. I recommend leaving *Automatic* or *Background* as your default setting. If you use *Manual*, I recommend adding a command at the end of the macro to reset it to your default.

In larger spreadsheets with multiple macros, I place commands that speed the program into one cell by turning off screen redrawing and by setting recalculation to manual; place those that restore normal operations in another; begin most macros with a call to the first cell; and end those macros with a call to the second cell. This system is illustrated in Table 11.7.

Table 11.7: Combined techniques for speed

	A	B
1	{B1}	{Notebook.Recalc_Settings "Manual;Natural;1;Yes;No"}{WindowsOff}
2	{; commands}	
3	{; commands}	{Notebook.Recalc_Settings "Background;Natural;1;Yes;No"}{WindowsOn}
4	{B3}	

Speed #3 – Close other notebooks and minimize use of functions

Except as limited by the techniques above, QP recalculates all or most functions on all sheets of all open notebooks under various conditions. The more QP must recalculate those functions, the longer it takes for a macro to finish. Therefore, closing other notebooks before running a macro will remove the need to make other calculations. Designing the notebook to minimize the number of functions on its sheets will also speed the performance of the sheet.

Automating startup and shutdown macros

When you open or close a QP notebook, you can have QP automatically run a macro by naming the cells in which the macro begins.

Name (by `[Ctrl+F3]`) the cell `_NBStartMacro` to have QP play the macro automatically, at the time the file is opened. Another way to do this is to name the cell with the name shown in the "Start-up macro" on the dialog box that pops up on clicking Tools > Settings > Macro. By default, that name is `\0` (which is a zero, not an upper case o).

Name the cell `_NBExitMacro` to have QP play the macro automatically when you close the file.

Caution: This macro replaces the normal, built-in procedure for closing the file, including the saving of changes, so if you wish to save changes, you will need to add a command such as `{FileSave}` at the end of the macro.

Naming cells to expedite launching macros

Though I normally avoid using named cells for reasons mentioned above (page iv), naming them can make it easier to launch them.

Naming a cell with a backlash and a single letter allows the user to cause the macro to run by pressing `[Ctrl+Shift+`*`letter`*`]`. Thus, naming a cell `\c` allows the user to invoke it by pressing `[Ctrl+Shift+c]`. This is the main purpose for which I named cells.

Naming a cell with any name (not just a backslash and letter) causes it to appear in the dropdown box at the bottom the the *Play Macro* dialog box that comes up when the user presses `[Alt+F2]`. Figure 11.1 is an example.

Thus, the user can invoke this dialog and select (or scroll down to) the desired macro, and then click `OK`.

Please note that QP has preemptively taken some letters for keystroke combinations, namely:

`f` (speed format)
`l` (quick fill)
`n` (new from project)
`s` (styles)
`z` (redo)

To use those letters with the Ctrl+Shift combination to run a macro, you must apparently disable shortcut key combinations using the dialog box for Commands under Tools > Customization.

See WPU 38479 for a development of this idea, in response to a request for keystrokes that would automatically insert user-defined text. The QPW file there gives the user access to 21 backslash macros, enabling the user to assign up to 21 standard items of text.

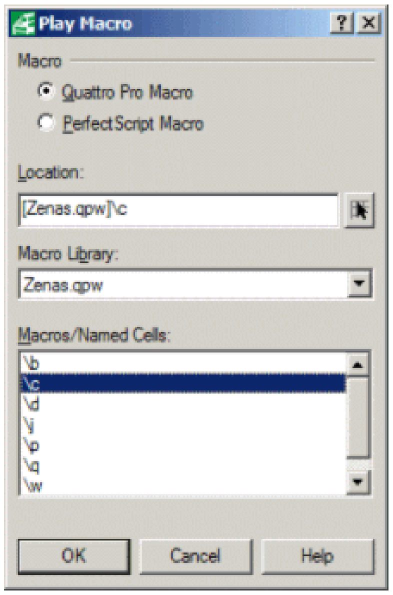


Figure 11.1: Macro Launcher

This idea can be pushed further, to allow each of the keystroke macros to behave differently, depending on the sheet that is current when the macro is invoked. Jack Kearns set up a table with sheet names on the top row and a series of named cells containing keystroke macros from A to Z in the left column. Each cell in the grid could contain a command that applied only to the currently active page. A simplified, reduced-scale version of the idea appears in Table 11.8.

Table 11.8: Sheet-specific keystroke macros

M:	A	B	C	D
1	Keystroke	A	{Let M:B2,@PageName(@Cellpointer("sheet")-1)}	
2	Sheetname	Jones	{Putblock @VHLookup(M:B1,M:B2,M:B5..D7,0),M:C3}	
3			{Putcell Jones}	
4				
5			Jones	Smith
6	{Let M:B1,"A"}{Branch M:C1}	A	{Putcell Jones}	{Putcell Smith}
7	{Let M:B1,"B"}{Branch M:C1}	B	{Branch M:C50}	{Branch M:D50}

In this truncated sample, cells A6..A7 are named cells: `Ctrl+Shift+A` invokes the macro in A6, `Ctrl+Shift+B` invokes the macro in A7, and the sequence could be continued. Assuming that the user starts from a page called "Jones" and clicks `Ctrl+Shift+A`, the macro would place `A` in cell B1 and branch to cell C1. The command in C1 would place the current page name, here `Jones`, into cell B2. The command in C2 would find the command in the grid B5..D7 at the intersection of the "Jones" column and row "A," and place that command in cell C3, which would then be executed.

Creating toolbar icons and keystroke shortcuts to launch macros

Macros can also be stored in QP itself, not just the current notebook, and launched by toolbar icons or shortcut keys. This text discusses how to create such macros in the context of a concrete example (inserting dates with particular formats) below at page 166.

Adding command buttons to launch macros

Launching macros manually is not as easy as pushing a button. You can add a macro button easily by clicking Insert > Form Control > Push Button. The mouse will take on a peculiar shape, and you drag it where you want the button to appear.

Buttons have a few variables, but the most significant are the text appearing on the button and the macro. Right-click the button and choose Selection Properties (`[F12]`).

In the tab for *Macro*, enter the macro commands (if simple enough) or the location where the macro can be found. Thus, if you want to run commands starting at Macros:Z15, you would enter `{Macros:Z15}`. (Don't forget the braces.)

In the tab for `Label Text`, enter the words that you wish to appear on the button.

The `Object Name` tab allows you to change the name of the button. This may make it easier to program a macro that changes the words on the button and its function.

Troubleshooting ambiguous cell references in macros

Macros can fail to yield the desired results because cell references in macro commands are ambiguous at the time the macro executes. The following types of ambiguity should be noted:

- References to a cell such as `A1` within a macro command are ambiguous, because each sheet has a cell designated `A1`, and each open notebook has a series of sheets, each with a cell designated `A1`. The reference may be ambiguous when the macro is written. It may become ambiguous when the macro is moved from one sheet to another, or when it is executed from different sheets. Such ambiguities are corrected as shown at page 3 above.
- In addition, when a macro is running, two cells are active at any time: the currently selected cell, and the cell containing the macro command that is being executed. This is particularly problematic where the macro is on one sheet, and the selected cell is on another sheet or in a different notebook. If QP can construe the cell reference as pointing to the sheet on which the macro executes, it will do so. To avoid that outcome, precede the cell reference with brackets ([]).

  The following macro illustrates the differences: In a new spreadsheet, put "Sheet A" into cell A:A1, and set up sheet B as shown in Table 11.9. Then return to cell A1 of sheet A, and from there, run the macro in B:B1. The presence of brackets will control (a) which sheet provides the source data in its cell A1, and (b) to which sheet that data will be written, yielding the results described in C1..C4.

Table 11.9: Results when B:B1 macro is run from sheet A

B:	A	B	C
1	Sheet B	{Let A2,@cell("contents",A1)}	Puts B:A1 into B:A2
2		{Let []A3,@cell("contents",A1)}	Puts B:A1 into A:A3
3		{Let A4,@cell("contents",[]A1)}	Puts A:A1 into B:A4
4		{Let []A5,@cell("contents",[]A1)}	Puts A:A1 into A:A5

- The situation is one degree more problematic when using R1C1 notation. For instance, modifying the last example by using R1C1 notation for the source data (see Table 11.10), the first two lines of the macro place those commands themselves into the target cells.

Table 11.10: Same, using R1C1 notation

	A	B	C
1	Sheet B	{Let A2,c(0)r(0)}	Puts B:B1 into B:A2
2		{Let []A3,c(0)r(0)}	Puts B:B2 into A:A3
3		{Let A4,[]c(0)r(0)}	Puts A:A1 into B:A4
4		{Let []A5,[]c(0)r(0)}	Puts A:A1 into A:A5

- When running QP macros via PerfectScript, some commands work on the sheet containing the macro regardless of whether another sheet is active. See page 236 below for more information on this quirk. The gist of the solution is to use commands that don't have that effect.
- The same or similar problem occurs when running QP macros using the built-in debugger (Tools > Macro > Debugger or `Shift+F2`). See the discussion at WPU 38484. The solution again is to use the subset of commands that work.

Chapter 12

Basic File System Macros

Opening and using notebooks: {FileOpen} and {Activate}

`{FileOpen` *`FileName<,HandleLinks?,AsTemplate?>`*`}` is the command for opening other notebooks. The only mandatory argument is the *FileName* of the notebook to open. Thus, if you simply wish to open another QP notebook, it is usually adequate to write something like this:

```
{FileOpen "C:\Spreadsheets\CheckBook.qpw"}
```

If the notebook is in the default QP spreadsheet folder, simply supplying the name of the file (e.g., “CheckBook.qpw” or simply “CheckBook”) should work, but I prefer to specify the full path to avoid possible ambiguities.

HandleLinks? Some notebooks have links to others, where a formula or function in one of its cells refers to cells in a third notebook. The *HandleLinks?* argument will specify whether QP should open up the third notebook, simply update the cells that link to the third notebook, or do nothing. If argument is not specified in the command, QP will ask the user when the other file is opened, but only if the other file has links to a third notebook. If it has no links, this argument can be ignored.

AsTemplate? By default, `{FileOpen}` will open the other file just like any other notebook, on which you can make changes and save them. If you do not want any changes to be saved in the other notebook — that is, you want to use the other notebook as a template (copy of itself) only — then type 1 as the third argument, with or without a second argument. E.g.,

```
{FileOpen "C:\Spreadsheets\MyTemplate.qpw",,1}
```

Using `{FileOpen}` opens the notebook and makes it the active notebook. If that notebook is already open, `{FileOpen}` does not open a separate copy, but simply makes it active again.

In order to move back and forth between open notebooks, use the `{Activate` *`FileName`*`}` command and supply the name of the file, which can be the full path or abbreviated as indicated with the `{FileOpen}` command.

The programmer can cause a macro to run whenever a file is opened by naming the cell where the startup commands begin as `_NBStartMacro`. See the discussion at page 116. Various examples are in the following pages.

Cautions. When working with two or more open files, be careful to disambiguate cell references. Each of the files will have cell A:A1, and you will need to be sure that you are acting on the intended cell. See page 3 for the correct way to reference other sheets and page 23 for using functions in macro commands.

You should also look closely at the results of running your macro in order to ensure that it is acting correctly. Most macro commands will work on an open notebook other than the currently active one, but it is possible that not all will.

Finally, there is a particular quirk when using the PerfectScript version of these commands. See page 236 for the discussion.

Saving a QP Notebook: {FileSave}, {FileSaveAll}, and {FileSaveAs}

The command `{FileSave}` saves the currently active notebook in its current status, overwriting the prior file. `{FileSaveAll}` does the same for all currently open notebooks. The help file claims that the user can add one of three parameters, `Replace`, `Backup`, or `Confirm` to both commands, but they make absolutely no difference: the current notebook or notebooks are saved. If the user wants to have the option whether to save the file, see the `{Alert}` command, discussed below at page 132.

The `{FileSaveAs}` command allows the programmer to save the notebook with another name and in a file format other than Quattro Pro. The latter capacity will be explored at page 201 below. To save the file with a new name, use a command like this:

```
{FileSaveAs "C:\Spreadsheets\NewName.qpw"}
```

Closing a QP notebook: {FileClose} and {FileCloseAll}

In order to close the current QP notebook, use the command `{FileClose}`. It follows that if an open QP notebook is not currently active, the macro must activate it first. If you desire to close all QP notebooks, use `{FileCloseAll}`.

By default, these commands display a Save-File dialog if any notebook is currently unsaved. This is equivalent to using the command with the parameter `1`, as in `{FileClose` *1*`}`. To avoid the display, the programmer can use the parameter `0`, as in `{FileClose` *0*`}`. A potentially large problem with doing so is that this will simply close the notebooks without saving them. If saving them is required, the command should be preceded by one of the appropriate `{FileSave}` commands just discussed.

The programmer can cause a macro to run whenever a file is closed by naming the cell where those commands begin as `_NBExitMacro`. See the discussion at page 116. Various examples are in the following pages.

Chapter 13

Navigation and Selection Macros

Here we review commands that cause the cursor (cellpointer or selector) to move from one cell to another.

Cursor Keys: {U}, {D}, {L}, {R}, {Home}, {End}, {PgDn}, {PgUp}

Most of the commands do simply what pressing the associated keys on the keyboard would do when manually pressed. Unsurprisingly, `{Home}` does what the `[Home]` key does, and `{End}` what the `[End]` does; `{Left}` or `{L}` what the `[ ← ]` left arrow does; `{Right}` or `{R}` what the `[ → ]` right arrow does; `{Up}` or `{U}` what the `[ ↑ ]` up arrow does; `{Down}` or `{D}` what the `[ ↓ ]` down arrow does; `{PgUp}` what the `[PgUp]` does; and `{PgDn}` what the `[PgDn]` key does.

Note that if you combine arrow key commands with other keys, such as Ctrl or Alt or Shift, QP requires the arrow key to be fully spelled out. Thus, to extend the current selection to the right, type `{Shift+Right}`, not `{Shift+R}`.

The macros that parallel the arrow keys and page keys allow for an optional `Repetition` argument. Thus, `{R 6}` moves the cursor 6 columns to the right; `{D 2}` moves it two rows down.

In Windows versions, pressing `[Tab]` or `[Shift+Tab]` would simply move the cursor one cell to the right or left, respectively. In earlier DOS versions, those keys would move the cursor an entire screen to the right or left. That is, if the user currently saw all of columns A..J, pressing `[Tab]` would move the cursor to the next column that was not fully visible (column K). QP may be made to behave that same way by clicking Tools > Settings, selecting the *General* node, and checking the *Compatibility Keys (QP-Dos)* box. Then `{Tab}` and `{BigRight}` will go to the next screen of columns to the right, as in earlier DOS versions, and `{Backtab}` or `{BigLeft}` will go to the screen of columns to the left. Without checking the DOS compatibility options, those commands simply behave like the right and left arrow keys.

But without fooling with those subtleties, the keystroke `[Ctrl+→]` moves right to the next screen of data, as does the macro command `{Ctrl+Right}`. `[Ctrl+←]` moves leftward one screen of data, as does `{Ctrl+Left}`. Both macro commands can include an optional `Repetition` argument.

Likewise, `{Ctrl+PgDn}` and `{Ctrl+PgUp}` do what the corresponding keystrokes do, go to the next sheet in the notebook or the previous sheet, respectively.

Selecting Cells/Blocks: {Shift}, {Goto}, {SelectBlock}, {EditGoto}, etc.

Emulating keystrokes

As noted in the discussion of the cursor keys, above, the programmer can select a block by selecting a cell and then combining movement keys with `SHIFT`. For instance, `{Shift+Right 5}` will select the starting cell and five cells to its right.

This example selects A1..C3 by emulating the keystrokes of a user who would first select A1 and then drag the mouse to C3 or hold the `[Shift]` and use cursor keys to extend the selection to C3:

Table 13.1: Selection by simulating keystrokes

	A	B	C
1		{SelectBlock A1}	
2		{Shift+Down 2}	
3		{Shift+Right 2}	

Goto

The original navigation command-set was `{GoTo}`*`Cell`*`~`, in which `{GoTo}` started the command, a `Cell` was specified, and the `~` executed the movement. It still works splendidly, but it selects only a cell, not a multi-cell block.

QP has two other useful commands that both select a cell or block, and for that basic function, there is no reason to prefer one over the other. They can perform separate and unique functions, however.

SelectBlock

`{SelectBlock` *`CellorBlock<,ActiveCell>`*`}` selects the Cell or Block specified as the first argument. If you select more than one cell in the `CellorBlock` argument, the active cell will be the topmost, leftmost cell in the Block. For example, if the command `{SelectBlock E1..H2}` is followed by the command `{D 1}`, the active cell will be E2.

A unique feature of `{SelectBlock}` is that it allows you to select which cell will be active. If you want to select some other cell within the block, add the optional `ActiveCell` argument. Thus, if the command `{SelectBlock E1..H2,G2}` is followed by the command `{D 1}`, the active cell will be G3. `ActiveCell` should refer to a cell within `CellorBlock`; if not, QP may crash.

Another unique feature of `{SelectBlock}` as that it allows you to select discontinuous blocks. If so, the `CellorBlock` argument must be enclosed in parentheses and contain the blocks, separated by commas or semicolons.

EditGoto

`{EditGoto` *`CellorBlock<,ExtendSelection>`*`}` selects the Cell or Block specified as the first argument. If you select more than one cell in the `CellorBlock` argument, the active cell will be the topmost, leftmost cell in the Block.

The unique feature of `{EditGoTo}` is that it can extend the selection from the current selection to the `CellorBlock` specified if the user adds the optional argument *1*, which indicates that the current selection is extended. If the argument is omitted or *0*, `{EditGoto}` simply goes to the cell or block and selects it.

To illustrate, the macro in Table 13.2 would first select cells A1..A2 with `{SelectBlock}`. The `{EditGoto}` command would extend that selection to the entire block of A1..C3.

Table 13.2: Selection by SelectBlock and EditGoTo

	A	B	C
1		{SelectBlock A1..A2}	
2		{EditGoTo C3,1}	

Other methods

`{QGoTo}` summons a dialog box for the user to select the destination. Unfortunately, as designed, it does not stop macro execution, so later macro commands will execute often before the user has the opportunity to deal with the dialog box, very much contrary to user expectations. It appears that the `{PauseMacro}` command was designed to deal with that by stopping further execution, but unfortunately, at least in the case of `{QGoTo}`, commands after `{PauseMacro}` do not execute. If `{QGoTo}` is useful at all, it appears to work as expected only at the end of a series of macro commands.

The `{Navigate}` command is apparently for selecting the entire table in which the cursor is located when the command is executed, or for jumping to one of the sides or corners of the table. The most useful option would be `{Navigate.SelectTable}`, which easily identifies the bounds of the table. `@PROPERTY("Active_Block.Selection")` will return the coordinates of the table thus selected. For other uses of `{Navigate}`, consult the help file.

Hyperlinks can also be set up using the {Comment.EditURL} command. This is discussed below, beginning at page 228.

The visual effect of selecting a cell

If the destination of navigation (the “target cell”) is on the current screen, no change of row or column indicators occurs. The focus simply shifts to the target cell.

If the target cell is not on the screen, QP places the target cell in the top left corner of the window and displays the cells below and right of that cell; it does not display cells above and to the left. This may disorient the user. If so, consider comments on how to position the display at the end of the macro at page 140.

How to move to the leftmost cell on the row

There are several possible ways to move to the leftmost cell in the same row.

```
{Left @CELLPOINTER("col")-1}
{SelectBlock @OFFSET(A1,@CELLPOINTER("row")-1,0)}
{Left 1000}
```

The last one works if you will never have more than 1000 columns to the right; the default maximum is 256).

How to move to the first blank cell at the bottom of a column of data

Here are some ways to move to the first blank cell at the bottom of data in that A column.

First, using the ways given in the discussion of determining the first blank row (see page 90), a `{SelectBlock}` using those functions as an argument should suffice. These include:

```
{SelectBlock @OFFSET(A1,@COUNT(A:A),0)}
{SelectBlock @OFFSET(A1,@COUNT(A:A1..A10000),0)}
```

An earlier version using cursor keys would look like this:

```
{Goto}A1~{End}{D}{D}
```

If the destination is in the same window, no change of the column and row headers occurs. But if the destination is not, the destination cell will be on the bottom row of the screen. If that might disorient the user, consider techniques to reposition the display at page 140.

How to move to the first blank cell at the right of a row of data

Parallel to the last example, here are some ways to move to the first blank cell on row 1.

```
{SelectBlock @OFFSET(A1,0,@COUNT(A:1))}
{SelectBlock @OFFSET(A1,0,@COUNT(A:A1..IV1))}
{Goto}A1~{End}{R}{R}
```

How to move to the same cell on the next page

If you want to move from the currently selected cell, say A:C5, to the same cell on the next page (here, B:C5), Table 13.3 contains a macro that automates the process. (See WPU 36198.) Running the macro in B1 stores the current address (C5) in cell C:A1, goes to the next page by doing what happens when you manually press `[Ctrl+PgDn]`, and then selects the cell C5 on that page. The address can be stored in any cell, so long as the address is the same in the first and last commands.

Table 13.4 shows another way to do it, with a single, more complex macro command using the `@IndexToLetter` function (see page 79 above). This command se-

Table 13.3: Movement to same cell on next page

	A	B
1		{Let C:A1,@CELLPOINTER("TwoDAddress")}
2		{Ctrl+PgDn}
3		{SelectBlock +C:A1}

Table 13.4: Moving to the same cell on the next page with a single command

	A
1	{SelectBlock @CONCATENATE(@INDEXTOLETTER(@CELLPOINTER("sheet")),":", @INDEXTOLETTER(@CELLPOINTER("col")-1),@CELLPOINTER("row"))}

lects the cell that is pieced together by the `@Concatenate` function. In the example where the cursor starts in A:C5, we want a function that returns the address B:C5. We use the `@CELLPOINTER` function to get the starting information, which we must then massage. The `@CELLPOINTER` functions for the cell A:C5 return these values:

`@CELLPOINTER("sheet")`	returns 1
`@CELLPOINTER("col")`	returns 3
`@CELLPOINTER("row")`	returns 5

The row is correct as is, but the sheet and column have to be converted into letters, as illustrated in Table 13.5. The `@INDEXTOLETTER` function does that, but there is a complication. For functions like `@CELL`and `@CELLPOINTER`, the first sheet, column and row are always 1, not 0. For `@INDEXTOLETTER`, the first letter (A) is always correlated with 0, not 1. To use `@INDEXTOLETTER` with `@CELLPOINTER` to get the desired letter, we must subtract 1 from the number that `@CELLPOINTER` returns. This explains the use of `-1` in the function. Since we don't want to return to the same sheet, but want to go to the next one, we do not subtract 1 from `@CELLPOINTER(“sheet”)`. The `@CONCATENATE` function then returns B:C5, and the selectblock macro command moves the cursor to that cell.

Table 13.5: How different functions treat columns

	A	B	C	D
1	1	2	3	⇐ @CELL/@CELLPOINTER values for columns A B and C
2	B	C	D	⇐ How @INDEXTOLETTER transforms the numbers in row 1

How to move to the same cell on the prior page

To move to the same cell on the previous page, substitute `{Ctrl+PgUp}` in Cell B2 in Table 13.3, or alternatively, substitute this function in Cell A1 in Table 13.4:

```
{SelectBlock @CONCATENATE(@INDEXTOLETTER
(@CELLPOINTER("sheet")-2),":",
@INDEXTOLETTER(@CELLPOINTER("col")-1),
@CELLPOINTER("row"))}
```

How to move to each page successively

The macro in Table 13.6 shows how to execute some commands on a series of sheets before returning to the starting sheet. The command in B1 sets the starting cell, and the command in B3 will return to that cell at the end of the macro. The `{For}` command in B2 runs the commands starting at B3 10 times. Broken down, the five elements of the `{For}` command do this: *(1)* A1 stores the counter, letting us know when it reaches the end; *(2)* 1 is the starting number; *(3)* 10 is the ending number; *(4)* 1 is the amount of the increase with each repetition from the starting number to the ending number, and the combination of these three elements direct that the macro will run 10 times; *(5)* B5 is the starting cell for the macro to run 10 times.

Table 13.6: Repeating actions on different pages

	A	B
1		{Let F:A1,@CELLPOINTER("ThreeDAddress")}
2		{For F:A2,1,10,1,F:B5}
3		{SelectBlock +F:A1}
4		
5		{; insert one or more rows of macro commands here}
6		{Ctrl+PgDn}

How to round-trip, to move from one cell to another, and then return

Table 13.7 shows a macro that will take whatever the current cell's content is and paste it to a particular cell elsewhere (e.g., A5), and then return. The command in B1 stores the address to return to in A1. B2 copies the content of the current cell. B3 navigates to the cell where that content is to be pasted. B4 pastes the content there. B5 then returns to the address that was stored in A1. If the macro pastes to another sheet, sheet letters or names should be added.

Table 13.7: Round-tripping after copy/paste

	A	B
1		{Let A1,@CELLPOINTER(“address”)}
2		{EditCopy}
3		{SelectBlock A5}
4		{EditPaste}
5		{SelectBlock +A1}

How to select the current and adjoining cells

As noted above at page 123, the programmer can select a block starting from the current cell by using `{Shift}` in connection with cursor keys. To select the current cell and three cells to the right, use:

```
{Shift+Right 3}
```

To do the same with `@OFFSET`, the programmer can navigate to the topmost, leftmost cell in the block and use an R1C1 reference to anchor the block. For example, to select the 1x4 block consisting of the current cell and three cells to the right, use:

```
{SelectBlock @OFFSET(c(0)r(0),0,0,1,4)}
```

How to select the same cells on restarting, after closing the file

The macro in Table 13.8 illustrates how you can save the selected cell or cells at the time of closing a file and have QP select the same cells when you re-open the file. This is true even if you have selected non-contiguous blocks of cells. Cell B2 is named _NBStartMacro, and B5 is named _NBExitMacro. Naming the cells that way causes QP to run B2 upon opening the file, thus selecting whatever address is stored in A1. It causes QP to run B5 on closing the file, thus saving the address of any and all blocks selected in A1 and saving the file with that information.

Table 13.8: Selecting the same cells on restarting

	A	B
1		_NBStartMacro
2		{SelectBlock +H:A1}
3		
4		_NBExitMacro
5		{Let H:A1,@PROPERTY("Active_Block.Selection")}
6		{FileSave}

How to select cells in a different notebook

The methods for selecting cells in other notebooks are basically the same as for selecting cells in the same notebook, but there are some differences that warrant attention.

First, if the target notebook is not already open, the macro must open it. If the target notebook is open but not active, the macro must use `{Activate}` to make it the active notebook.

Most, but not all, methods for selecting a cell will then work. If you know in advance the cell, it can be hard-coded in with something like `{SelectBlock D1}` or with a hard-coded formula like `{SelectBlock @Offset(A1,0,3)}`, which would both select the cell D1 on the active page of the active document.

However, it is tricky to use variables in the original notebook to determine the cell to select in the target notebook. For example, if the address "D1" were stored in cell A15, one could select cell D1 by using the command `{SelectBlock +A15}`, but if one did so after activating the target notebook, an error message would be extremely likely to occur. The reason is that, after activating the target notebook, when QP attempts to locate the cell A15, it is looking it the then-current notebook, not the original one. Unless the cell A15 in the target notebook contains an address, an error message will occur.

One method of dealing with this problem would be to make a reference back to the original notebook, as with `{SelectBlock +[Original]A15}`. Another way to deal with the problem is to use dynamic modification (see page 107) to construct a line of macro code that will be read as if it had been hard coded. For instance:

```
+"{SelectBlock "&A15&"}"
```

To ensure that the macro has correctly updated this command, I would precede it with a {Recalc} command.

How to select a cell in a given column based on a row number stored in a particular cell

The task here is to compile a `{SelectBlock}` command so that it selects a cell in a known column, at the row designated by the number in a particular cell. The desired column is known to be `AC`, and the row number that is stored in cell `U1` will be variable. Any of these should work:

```
{SelectBlock @CONCATENATE("AC",U1)}

{SelectBlock +"AC"&@STRING(U1,0)}

{SelectBlock (AC)&@STRING(U1,0)}

{SelectBlock @OFFSET(AC1,U1-1,0)}
```

How to move to a given value in the index column of a database

Assume that you have a well-formed database in Data:A1..Z1000, and that column A (the index column) contains unique index numbers for each item (row) of data. From some other part of the notebook, you want to move to the row of data relating to a particular value in the index column. How do you program that?

First, you will need to give the macro the index number to which you want to move. That can be done by several means: using the current cell's value (with or without error checking to ensure that it is a number), a `{GetNumber}` command, or otherwise.

Then you need to determine whether that number is in the index column. If not, use an `{Alert}` command, or some other method of giving the message, that the number is not in the index. If so, calculate its offset from the top cell by using `@Match` and its address by `@Offset`, and wrap that function in `{SelectBlock}`.

The following macro (Table 13.9) illustrates these points. B1..B3 get a number from the user, and provide error checking that will be more fully explained below (page 134). If no number was entered, the macro branches to the error message in B7 and stops.

B4 tests whether the number is in the index column of the database. If not, the macro branches to the error message in B9 and stops. If the number is there, the argument in B5 calculates the location of the value in the index column, and the command takes the user there.

Table 13.9: Jumping to index value in a database

	A	B
1		{Let A1,@ERR}
2		{GetNumber "Enter index No.",A1}
3		{If @IsErr(A1)}{Branch B7}
4		{If @IsErr(@Match(A1,Data:A1..A1000,0))}{Branch B9}
5		{SelectBlock @Offset(Data:A1,@Match(A1,Data:A1..A1000,0),0)}
6		
7		{Alert "Error","No number entered",A7}
8		
9		{Alert "Error","Number not in database",A9}

How to test whether the active cell is in a particular sheet

To test if the cellpointer (active cell) is in particular sheet, identified by number, where the first sheet in a notebook is 1, the second is 2, and so on, use a command like this.

```
{If @CELLPOINTER("sheet")=2}
```

And to test whether it is *not* in a particular sheet, use something like this:

```
{If @CELLPOINTER("sheet")<>2}
```

To test whether the active sheet is in a sheet identified by its name (in this example, the name “Status”), use something like this:

```
{If @PROPERTY("Active_Page.Name")="Status"}
```

Chapter 14

Interface Macros

Here we look at the ways one can give information to the user and get a response. Note that some ways in the help file, such as `{Indicate}`, are obsolete and simply do not work.

Sound: {Beep}

QP gives some basic sound options through the `{Beep` *`Pitch#`*`}` command. According to the help file, the `Pitch#` argument is optional, but if so, `{Beep}` alone makes no sound on my laptop. The `Pitch#` argument is a number from 1 to 10, going from low to high.

An external sound can be played using QP's file launching commands, such as `{EXEC}`, or through PerfectScript macros. See Chapter 23 here and WPU 29939.

Giving information: {Message}

The `{Message}` command displays a message on the screen, even if it is outside the QP window. The screen position must be set in advance, and the coordinates are relative to the screen, not the QP window. Unless the user specifies when the message disappears, the message remains until the user presses a key. The macro is suspended until the message disappears. This is the format:

```
{Message MsgBlock,ScreenX,ScreenY,<Expiration>}
```

`MsgBlock` is the cell block containing the message. It may be more than one cell. The message will appear just as it appears in the block, with all highlighting, fonts, size, alignment, etc. If the block is too narrow to hold the message, the message is truncated.

`ScreenX` is the horizontal position for the message to appear, starting from 0 on the left to the last position on the right of the screen. On my screen resolution with a horizontal width of 1366, the rightmost `ScreenX` value that would display a character was 194. Your monitor and resolution may differ from this.

`ScreenY` is the vertical position for the message to appear, starting from 0 on the top to the last position on the bottom. On my screen resolution with a vertical width of 768, the bottommost `ScreenY` value that would display a character was 45. Again, your monitor and resolution may differ from this.

`Expiration` This argument is optional. If it is omitted, the message remains until the user presses a key. The `Expiration` argument should contain a number representing the date and time of day (as QP recognizes it) when the message disappears. For instance, a message that should expire at 6:00PM on March 18, 2015 would need to include a value of 42081.75. Since we humans do not usually find that method of keeping time helpful, and since QP would simply hang until that time, the `Expiration` argument is usually either left blank or constructed with @functions like those below.

Thus, to display the message in A1 at screen position 0,0 for 3 seconds:

```
{Message A1,0,0,@NOW+@TIME(0,0,3)}
```

This sets the duration of the message by taking the current time (`@NOW`) and adds 3 seconds to it (`@TIME(0,0,3)`) to construct the expiration time that is three seconds from now. I believe that decimals in the third argument of the `@TIME` function are rounded to the nearest integer.

How to see where the message will appear

In addition to trial and error, the little macro in Table 14.1 displays a short message about the screen coordinates at the screen position of those coordinates. Enter preferred column and row coordinates in B1 and B2. The function in B3 will compile a message. Run the macro in C1 to have that message display and the coordinates typed in B1 and B2.

Table 14.1: Where {Message} displays

	A	B	C
1	Col	100	{Message B3,B1,B2}
2	Row	25	
3	Msg	Col 100, Row 25	⇐ @CONCATENATE("Col ",B1,", Row ",B2)
4			

Making basic choices: {Alert}

The `{Alert}` macro displays a simple dialog box that gives the user basic information, and it shows buttons that call for a basic choice, such as *OK*, *Yes*, *No*, *Cancel*, etc. The macro stores that choice as a number in the `ChoiceCell` for further processing. The macro suspends until the user clicks one of the buttons. The format is:

```
{Alert Title,Message,ChoiceCell<,ButtonType,Icon,
DefaultButton>}
```

`Title` is the short descriptive text at the top of the dialog box. It can be hard-coded or stored in a cell.

`Message` is the text of the information given to the user. It can be hard-coded, stored in a cell, or composed by a formula.

`ChoiceCell` is the cell (e.g., A1) where the user's choice will be stored as a number. The choices that will be stored there are:

`[OK]` is stored in `ChoiceCell` as 1
`[Cancel]` is stored in `ChoiceCell` as 2
`[Abort]` is stored in `ChoiceCell` as 3
`[Retry]` is stored in `ChoiceCell` as 4
`[Ignore]` is stored in `ChoiceCell` as 5
`[Yes]` is stored in `ChoiceCell` as 6
`[No]` is stored in `ChoiceCell` as 7

`ButtonType` is an optional argument. If nothing is specified, the dialog box displays only an `[OK]` button. These values will display these buttons:
0 displays `[OK]` (the default)
1 displays `[OK]` and `[Cancel]`
2 displays `[Abort]`, `[Retry]`, and `[Ignore]`
3 displays `[Yes]`, `[No]`, and `[Cancel]`
4 displays `[Yes]` and `[No]`
5 displays `[Retry]` and `[Cancel]`

`Icon` is optional. If nothing is specified, the dialog uses 0 as the default. The options are:
0 displays an *X* icon (Error)
1 displays a *?* icon (Question)
2 displays an *!* icon (Warning)
3 displays an *i* icon (Information)

`DefaultButton` is optional. It determines which button will be the default. If it is omitted or set to 0, the first button is the default. If it is set at 1, the second button is the default. If it is set as 2, the third button is the default.

The simplest use of an `{Alert}` is to give information. This command validly displays a message to be cautious with an `[OK]` button and without a title:

```
{Alert "","Be Cautious!",A1}
```

Even so, a cell for placing the result of clicking `[OK]` is mandatory.

`{If}` tests can be used to take the chosen result of an `{Alert}` command for further processing, as show in Table 14.2. In this simple macro, the `{Alert}` command in cell B1 first asks the user to choose *Yes* or *No*; `ButtonType` 4 give the user a yes/no choice, and the command stores the choice in cell A1. If the user chose *Yes* (which would put *6* into cell A1), the commands in cell B2 will show another informational `{Alert}`, confirming the *Yes* choice. If the user chose *No* (which would put *7* into cell A1), the commands in cell B3 would confirm the *No* choice with another informational `{Alert}`.

Table 14.2: Running different commands, depending on Yes/No choice

	A	B
1		{Alert "Choice","Choose yes or no",A1,4,1,0}
2		{If A1=6}{alert "Info","You chose YES",A2,0,3,0}
3		{If A1=7}{alert "Info","You chose NO",A2,0,3,0}
4		

Quirk. If the text for the title or message is stored in another cell, that cell must be coded in the `{Alert}` command as a formula rather than as pure text. That is, if it is stored in cell A1, the command should identify cell A1 as *+A1* or *=A1* or *(A1)*, but not simply as A1. `{Alert}` treats a reference like *A1* in the `Title` or `Message` arguments like text, even though it is not wrapped in double-quotes, and thus it would display *A1* as the title or message. `{Alert}` treats A1 in the `ChoiceCell` argument as a cell, which needs nothing further to identify it as a cell.

Getting input: {Get}, {GetNumber}, {GetLabel}

`{Get` *`KeystrokeCell`*`}` stops processing until the user presses a single key. It stores that single key in `KeystrokeCell` and then proceeds with later commands. It does not give a prompt, so the user must be given reminders of what to do by other means.

`{GetNumber` *`Prompt,NumberCell`*`}` displays a dialog box that gives a message up to 70 characters in length defined by the `Prompt` argument and a box where the user may type a number up to 160 characters long (some number!). On pressing `[Enter]` or `[OK]`, the macro stores the number in `NumberCell`. If the user enters text for `{GetNumber}`, the macro places ERR in the target cell.

`{GetLabel` *`Prompt,TextCell`*`}` displays a dialog box that gives a message up to 70 characters long defined by the `Prompt` argument and a box where the user may type up to 160 characters of text (which was called a *label* in earlier days). On pressing `[Enter]` or `[OK]`, the macro stores the text in `TextCell`. If the user enters a number for `{GetLabel}`, the macro places the number-like text in the target cell.

Quirk. What if the user presses `[Cancel]`? The `[Cancel]` button on both `{GetNumber}` and `{GetLabel}` dialog boxes appears ineffective, and whatever was in the target cell remains there. So, in order to test for pressing the cancel button, the only way to do so it to pre-set the target cell in such a way that it can be tested for non-change after the macro. One effective way to do it is to use `@ERR` and `@ISERR`:

```
{Let A1,@ERR}{GetNumber "number",A1}
{If @ISERR(A1)}{Quit}
```

Inhibiting input, avoiding crashes

I have occasionally experienced a bug where, while a macro is running, I key in information before the macro finishes. On those occasions that the bug bites, QP goes into an infinite loop typing the same key into a cell, and I must use the Windows Task Manager to kill the program.

In the earlier DOS days, commands like {Look} and {Clear} could manage the “typeahead buffer,” but those commands seem ineffectual now. If anything, the former is more likely to cause this crash. There unfortunately appears to be no equivalent of WordPerfect’s `InhibitInput` command.

The fixes for this problem are therefore behavioral: Don’t type until the macro finishes. The macro can give a warning at the outset, using `{Message}` or `{Alert}` to advise the user not to type until the macro concludes, but I find it easy to ignore the messages while doing repetitive data entry. My best recommendation is to put `{Beep 1}` at or near the beginning of the commands, and `{Beep 7}` at the end.

Pop-up menus: {Menubranch}

Two remnants of the DOS days are {MenuBranch} and {MenuCall}. In those days, the commands created an alternative menu on the top line of the screen, with explanations of the items in the second line. That function is obsolete, but the commands have been re-purposed in QP for windows as creating a pop-up menu. The obsolete functions explain the odd structure for this macro. As far as I can tell, they function identically in windows, so I will refer only to {MenuBranch}.

Figure 14.1 shows what the pop-up menu might look like, and Table 14.3 shows how to set this up. This macro displays a pop-up menu on the right side of the screen like the one shown in Figure 14.1 containing the menu-items listed in B4..E4. When you select one, it launches the macro commands in B6..E6, respectively. B4..E4 would have been the menu displayed on the top row of a DOS screen. B5..E5 would have been explanations for each of the menu items, but are completely meaningless in a pop-up menu, so those are left blank. But for backward compatibility, I suppose, the structure still starts the macro commands on the next row, B6..E6, here. The macros in the B..D columns place a new number in cells A1..A3, but any macro commands are available. The {Branch} command at the end of those columns simply re-starts the process with the pop-up menus. The *Exit* item in E4 provides a graceful way out of the otherwise infinite loop.

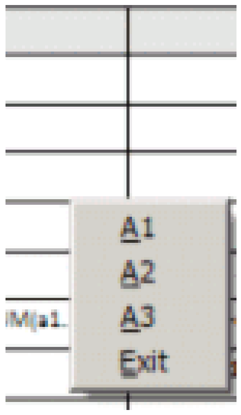


Figure 14.1: MenuBranch popup

Table 14.3: MenuBranch

	A	B	C	D
1		{MenuBranch A4..D4}		
2				
3				
4	A1	A2	A3	Exit
5				
6	{Let A1,1+@Sum(a1..a3)}	{Let A2,1+@Sum(a1..a3)}	{Let A3,1+@Sum(a1..a3)}	{Quit}
7	{Branch B1}	{Branch B1}	{Branch B1}	
8				

Displaying information in cells

Giving the user information while the macro progresses is tricky. In the DOS days, the {Indicate} command placed a short message in the application bar at the bottom right, but that has long since been obsolete and/or broken, though the programmers and help-file writers seem not to have noticed.

How (not) to display a “Waiting . . . ” message

To give a user a message that the program is pausing for three seconds, for instance, one might think that the macro in Column B of Table 14.4 would work, because it *should* work. For some reason, though, it doesn’t, as explored in WPU 33199. Hat tip to Jeff Barnes for experimenting and finding an intermediate command in C2 that would not seem to change the outcome, but that does in fact do so.

Table 14.4: 3-second “Waiting” message

	A	B	C
1		{Let A1,"Waiting..."}	{Let A1,"Waiting..."}
2		{Wait @NOW+@TIME(0,0,3)}	{Wait @NOW}
3		{Let A1,"The wait is over"}	{Wait @NOW+@TIME(0,0,3)}
4			{Let A1,"The wait is over"}

I have tried all 12 permutations of window redrawing commands (see Speed and Display Options at page 113), and the macro in Column B does not work with any of them, but the macro in Column C works with all of them. Redrawing the screen by the `{Wait @NOW}` may be surprising, but as developed below, it is quite useful.

How to display progressing values

It is often useful to give the user a sense that the macro is progressing as planned, even if the QP window display is not changing. The same basic display as in the last example could show that the program is working.

Table 14.5: Displaying a message while macro is working

	A	B
1		{Let A1,"Working..."}{Wait @NOW}
2		{; whatever commands}
3		{Let A1,"Done!"}

The macro in Table 14.5 would place “Working...” into A1 and immediately display it, then execute any other commands before concluding by putting “Done!” into A1. If the processing takes enough time to be apparent, the user will see “Working...” in A1.

That, however, would not indicate a progression. Another way to indicate a progression in, for example, a `{For}` loop would be to use `{Wait @NOW}` with each repetition.

The macro in Table 14.6 would place “Working ...” in A1, and the `{For}` command would put the counter into cell A3 for 1000 repetitions of the macro at B5. Each time it runs the macro at B5, the `{Wait}` command would cause this QP window to redraw, and the changing numbers in A3 would be apparent, thus indicating a progression, until the counter passes the number stored in cell A4.

Table 14.6: Displaying numbers from 1 to 1000 in A3 as macro loops 1000 times

	A	B
1		{Let A1,"Working..."}
2		{For A3,1,A4,1,B5}
3		{Let A1,"Done!"}
4	1000	
5		{Wait @NOW}
6		{; whatever commands}

Caution #1. Since redrawing the window slows performance down, you may wish to redraw the window only every 10th time, or 50th time, or 100th time. If so, use the `@MOD` function in an `{If}` command like this:

```
{If @MOD(A3,100)=0}{Wait @NOW}
```

This line, in lieu of B5 in the above example, would execute the `{Wait}` command each time the counter in A3 was evenly divisible by 100. The user would see the number in A3 progressing from 100 to 200, etc.

Caution #2. The programmer also doesn't want to create redundant window-redrawing. For instance, if QP is set to redraw the window and the programmer has not written a `{WindowsOff}` command (see Speed and Display Options, at page 113), anything that causes the window to redraw will cause a slowdown. It appears that, in addition to `{Wait @NOW}`, commands that would cause QP to change the headers (Row numbers on the left and column numbers at the top) or the sheet tab cause such redrawing, e.g., `{PgUp}`, `{PgDn}`, `{Ctrl+PgUp}`, and `{Ctrl+PgDn}`, as well as navigation to any place off the initial window. In addition, scrollbar operations using `{VLine}` and `{HLine}` would cause a redrawing. In those cases, with redrawing enabled, adding a `{Wait @NOW}` command could easily cause needless slowdowns.

How to display a progress bar

Based on the foregoing, we are now able to make two slight adjustments to the last macro in order to show a progress bar in cell A2.

First, we insert the command into cell B5 that essentially places the percentage of progress into cell A2, using the formula of 100 x the counter in A3 divided by the end number in A4.

Second, we apply a custom numeric format to cell A2. Select A2, press F12, and select the Numeric Format tab. In the listbox, click "Custom" at the bottom. A set of selections will appear on the right. Click "Left Bar" and OK.

Now, when this macro is run, a bar will appear on the left inside portion of cell A2, and it will grow larger toward the right as the macro progresses to its end. Table 14.7 shows what it would look like at the half-way point in the loop.

Remember that if this slows the macro excessively, the `{Wait}` command can be prefaced by an `{If}` command with an `@MOD` function. I often precede the `{For}` loop with `{WindowsOn}` and setting recalculation settings to manual. See page 113 for details.

Table 14.7: Displaying a progress bar

	A	B
1	Working...	{Let A1,"Working..."}
2	🟥	{For A3,1,A4,1,B5}
3	500	{Let A1,"Done!"}
4	1000	
5		{Let A2,100*(A3/A4)}
6		{Wait @NOW}
7		{; whatever commands}

How to display a floating (flickering) progress bar

By modifying the last example slightly in Table 14.8, we can show the same progress bar in a floating `{Message}` display. Say that we want to display it in the top left corner of the screen (position 0,0). We simply modify the command in B6 as shown below.

Table 14.8: Displaying flickering progress bar in top left of screen

	A	B
1		{Let A1,"Working..."}
2		{For A3,1,A4,1,B5}
3		{Let A1,"Done!"}
4	1000	
5		{Let A2,100*(A3/A4)}
6		{Message A2,0,0,@NOW}
7		{; whatever commands}

If, instead of cell A2, everything relevant to it were in some other place in the spreadsheet out of sight, the user would see only the message in the top left of the screen. However, the message would likely flicker a lot. It displays only until QP updates its timer, which it does once per second. It does not display while every other command is executed. If many other commands are executed, there may be lengthy periods in which it is not displayed. If non-flickering is desirable, use the last application to show a progress bar somewhere in the file, preferably on a sheet on which the data will not be changing.

And as in prior cases, if this slows the program down excessively, the `{Message}` command can be prefaced by an `{If}` command with an `@MOD` function. In this format, the user will see the progress bar both in cell A2 and in the top left corner of the screen.

To give some idea of how much this slows processing down, a macro that imported 1,857 rows of text and distributed it into eight cells per row took 21 seconds to accomplish without this progress bar. With the progress bar set to update on every 20th row, the entire process took 41 seconds, almost doubling the time. With the bar set to every 100th row, the process took 28 seconds.

The programmer could well question whether the benefit is worth the performance hit.

How to put highlighted text into a cell for user input

Many dialog boxes that seek the user's input place text into the edit box and select or highlight it. It may be a default response or a hint on how to respond. The user can leave it in the box by pressing `[Enter]` or begin typing, which immediately deletes the message.

In QP, this is fairly easy to do as well. It simply requires placing the message in a cell, moving the cursor/cellpointer to that cell, place the cell in Edit mode, and select the entire text. The simple macro in Table 14.9 takes the user to cell A1 and asks for the user's name.

Table 14.9: Placing an input message in the cell itself

	A	B
1	Enter your name	{Let A1,"Enter your name"}
2		{SelectBlock A1}
3		{Edit}{Ctrl+Shift+Home}{?}
4		

If the macro is not to continue with other commands, the `{?}` pause command is superfluous, but if the macro will continue with other commands, pausing until after the user responds in cell A1 will usually be necessary.

Table 14.10 shows a variation on the same macro:

Table 14.10: Another way to put an input message in the cell itself

	A	B
1	Enter your name	{SelectBlock A1}
2		Enter your name
3		{Ctrl+Shift+Home}{?}
4		

A further development of this idea that I find useful is to place all messages in a single block of cells that can be edited in that block, including multi-line messages, but used in macros like the one in Table 14.11. The block of messages is in the D column, and the macro in the B column would put the multiline instruction from D4 into cell A1.

Table 14.11: Storing input messages for easy revision

	A	B	C	D
1		{SelectBlock A1}		Enter your name
2		+D4		Enter your address
3		{Ctrl+Shift+Home}{?}		Enter your phone number
4				Compose your code this way: 1 First three letters of a color 2 Last three digits of phone number

Table 14.12 extends the concept by adding tests to check whether the input-type was correct. In this example, the entry of t branches one way, the entry of n branches

another, and if neither was entered, the macro loops back to request re-entry of a t or an n.

Table 14.12: Adding error checking for in-cell input

	A	B	C
1	Type t to add text or n to add a number, and Enter		
2		{SelectBlock A1}	
3		Type t to add text or n to add a number, and Enter	
4		{Ctrl+Shift+Home}{?}	
5		{If @LOWER(A1)="t"}{Branch B8}	
6		{If @LOWER(A1)="n"}{Branch C8}	
7		{Branch B2}	
8		{GetLabel "What Text?",A2}	{GetNumber "What Number?",A3}
9			

The macro beginning in B3 places the question in A1. It asks for the user to enter *t* or *n* in cell A1, and if the user does so, it executes commands at B8 or C8, respectively, but if not, the command in B7 causes the macro to loop back and put the same question to the user again.

How to position the display at the end of the macro sequence

In moving the current cell to a different location at the end of the macro, one problem to deal with is to ensure a desired view, and this problem can sometimes be sufficient to warrant reprogramming so as to move data without moving the cursor. Short of that, I have dealt with the problem in one of three ways. (See WPU 36246.)

1. In one, I add codes at the end of the macro to move the cursor back-and-forth, or up-and-down, as needed. So, to move the cell more to the center, I might add `{L 4}{R 4}` to the end of the macro, which sends the cursor four columns to the left, and then back, so that the four columns to the left show up. It often takes some experimentation to get this right.
2. In another, I use the `{VLine}` and `{HLine}` commands. `{HLine -4}` will probably do what I suggested in the last paragraph. However, this also takes some experimentation. In some cases, depending on where the cursor goes before this command, the effects of these commands can cumulatively push the target cell off the other side of the screen.
3. In a third, I precede the final `{SelectBlock}` command with a similar command to go to a cell from which the final `{SelectBlock}` command gives a consistent appearance. That also takes some experimentation.

How to select blocks by keys other than arrow keys

The typical manual way to select a block is by dragging with the mouse or holding down the `[Shift]` key and using arrow keys. One user wants a macro to allow him to do so using only the keys u, d, l and r, instead of the arrow keys. Table 14.13 contains a macro that does it.

Table 14.13: Extending a selection by arrow keys

	A	B
1		{Get A1}
2		{If A1="l"}{Shift+Left}{Branch B1}
3		{If A1="r"}{Shift+Right}{Branch B1}
4		{If A1="u"}{Shift+Up}{Branch B1}
5		{If A1="d"}{Shift+Down}{Branch B1}
6		{If A1="x"}{Blank A1}{Return}
7		{Beep 4}{Branch B1}

The `{Get}` command in B1 places a keystroke into A1, and then if-tests determine what to do with it. If the keystroke is l, r, u, or d, the macro extends the selection in the appropriate direction, and then returns back to B1. If the keystroke is x, this macro terminates with a `{Return}`. If some other keystroke occurs, the macro beeps and returns to B1.

How to change buttons that run macros

If you have placed a command button on a QP screen (see page 118), a macro can easily change the text appearing on the button and change its function by using the `{SetObjectProperty}` command, which is discussed more fully below at page 142. The command button is known by the sheet on which it appears and the name appearing in its *Object Name* tab. Thus, if the button is on sheet A and is named *Toggle*, its name is `A:Toggle`. The text appearing on the button is `A:Toggle.Label_Text` and the macro command is `A:Toggle.Macro`. So, if the text appearing initially on the button is “Pro” and it points to the macro beginning at A1, if you want to change it to “Con” and point to the macro beginning at B1, these commands would work:

```
{SetObjectProperty "A:Toggle.Label_Text","Con"}

{SetObjectProperty "A:Toggle.Macro","{B1}"}
```

Setting up an If-test to determine what is on the text of the button, the user could toggle back and forth between these two captions and macros.

Chapter 15

Cell Property Macros

{GetObjectProperty}, {GetProperty}

These commands allow the programmer to get a property from some part of the QP window and put it into a cell. `{GetProperty` *`TargetCell,Property`*`}` stores the specified `Property` of whatever object is selected (cell, block, whatever) in `TargetCell`. `{GetObjectProperty` *`TargetCell,Object.Property`*`}` stores the specified `Property` of a specifically named `Object` in `TargetCell`. See the earlier appendix on object properties (page 29). (Kenneth Hobson has put together a spreadsheet that illustrates the use of `{GetObjectProperty}` which is part of his online book and which appears in WPU 32606 and elsewhere.)

The following macros at A:B1 and A:B3 in Table 15.1 should store the same information about cell A:A1 into cells A:A2 and A:A4.

Table 15.1: GetProperty and GetObjectProperty

	A	B
1		{GetObjectProperty A:A2,"A:A1.Font.Typeface"}
2		
3		{SelectBlock A:A1}{GetProperty A:A4,"Font.Typeface"}
4		

Quirk. Not all of the strings return the value expected. Attempting to determine whether a row or column is hidden fails. See the workaround using `{OnError}` in WPU 31127.

{SetProperty}, {SetObjectProperty}

These commands allow the macro-writer to change the property of some part of the QP window. `{SetProperty` *`Property,Option`*`}` sets the specified `Property` of whatever object is selected (cell, block, whatever) as `Option`. `{SetObjectProperty` *`Object.Property,Option`*`}` sets the specified `Property` of a specifically named `Object` as `Option`. See the earlier appendix on object properties (page 29).

The following macros at A:B1 and A:B3 in Table 15.2 should both set the font for cell A:A1 as Courier New.

Quirk. These commands do not always work as one might expect. For instance, at least some cell properties can be set after selecting them and us-

Table 15.2: SetProperty and SetObjectProperty

	A	B
1		{SetObjectProperty "A:A1.Font.Typeface","Courier New"}
2		
3		{SelectBlock A:A1}{SetProperty "Font.Typeface","Courier New"}

ing {SetProperty}, but they cannot be set without selecting the cells and using {SetObjectProperty}, as noted in WPU 30532.

Samples. To apply default white background color to a range of cells, here A1..F100, use a command like this:

```
{SetObjectProperty "A:A1..F100.Fill/Pattern.Fill_Color";"0;0;Solid"}
```

If the target block and property were already saved as text in a cell, say G1, the same command could be written more easily:

```
{SetObjectProperty (G1);"0;0;Solid"}
```

If only the target block were saved as text in a cell, say G1, the same command could be written more easily:

```
{SetObjectProperty (G1)&".Fill/Pattern.Fill_Color";"0;0;Solid"}
```

To add or remove bold face to a block of cells, use a command like this, with `Yes` or `No` to add or remove the attribute:

```
{SetObjectProperty "A1..D100.Font.Bold";"No"}
```

To apply automatic row height to a block of cells, use a command like this:

```
{SetObjectProperty "C1..C100.Row_Height";"Auto Height;;1"}
```

To specify a particular height for a row containing a particular cell, use a command like this (which will set the row height for row 10) at 300 (which is divided by 20 to appear in the format dialog as a height of 15.00):

```
{SetObjectproperty "A10.Row_Height";"Set Height;300"}
```

To hide a particular row, use a command like this with a cell on the row, which will hide row 10:

```
{SetObjectproperty "A10.Reveal/Hide";"Rows;Hide"}
```

How to copy (all) cell formats from one cell to another

A way to copy all formatting properties of one cell to another is to select the source cell, copy it, go to the target cell, and use Paste Special to paste only properties. Table 15.3 shows how to do it.

But that method copies *all* properties. If you want to copy fewer than all properties, use `{SetProperty}` or `{SetObjectProperty}`. Thus, for example, if you want to copy the same alignment from cell A1 to A2, use `{SetObjectProperty "A2.alignment","A1.alignment"}`.

Table 15.3: Copying Properties

	A
1	{SelectBlock *SourceCell*}
2	{EditCopy}
3	{SelectBlock *TargetCell*}
4	{PasteSpecial Properties}

How to copy selected cell properties from one cell to another

One can transfer selected cell properties from a source cell to a target cell by using either of the `{SetProperty}` and `{SetObjectProperty}` commands for each property to be transferred. But note that in one respect, they are not symmetrical.

In this example, we want to transfer the content (`Value`), the text color (`Text_Color`), and background color (`Fill/Pattern.Fill_Color`) from cell C1 to C5. Table 15.4 shows how to do this with `{SetObjectProperty}`.

Table 15.4: Transferring select properties with {SetObjectProperty}

	A
1	{SetObjectProperty "c5.Value";"c1.Value"}
2	{SetObjectProperty "c5.Text_Color","c1.Text_Color"}
3	{SetObjectProperty "c5.Fill/Pattern.Fill_Color","c1.Fill/Pattern.Fill_Color"}

One would imagine that the same thing could be done by selecting cell C5 and then using the same pattern with the `{SetProperty}` command, but running the macro in Table 15.5 yields an error message.

Table 15.5: Wrong way to transfer properties with {SetProperty}

	A
1	{SelectBlock C5}
2	{SetProperty "Value";"c1.Value"}
3	{SetProperty "Text_Color","c1.Text_Color"}
4	{SetProperty "Fill/Pattern.Fill_Color","c1.Fill/Pattern.Fill_Color"}

When using this method, instead, the programmer must wrap the second argument in an `@Property` function. Thus Table 15.6 shows the way that works.

Table 15.6: Right way to transfer properties with {SetProperty}

	A
1	{SelectBlock C5}
2	{SetProperty "Value";@Property("c1.Value")}
3	{SetProperty "Text_Color",@Property("c1.Text_Color")}
4	{SetProperty "Fill/Pattern.Fill_Color",@Property("c1.Fill/Pattern.Fill_Color")}

How to view the 256 cell colors available by default

One sets the color of a cell by setting its “Fill/Pattern” property with three variables: a foreground color, a background color, and a pattern. There is no easy way to see what any combination of these variables looks like. However, my testing shows that there are only 256 built-in foreground colors and background colors.[1] I therefore created a macro that would show each variation. The full macro is on the "Colors" tab of the spreadsheet for this chapter, but it begins as shown in Table 15.7. The entries in column B, which show available solid cell colors, were created by placing this entry in B3 and copying it down parallel with the numbers in column A from 0 to 255:

```
@Concatenate("SetProperty ",@Char(34),"Fill/Pattern",@Char(34),
";",@Char(34),A3,";",$B$2,";Solid",@Char(34),"d")
```

A similar function in the C column showed how a single foreground color interacted with background colors 0-255, using the “Pepita” shading pattern.

I commend this table as a reference point for programming the coloration of cells.

Table 15.7: Displaying available cell colors and mixed shading patterns

	A	B	C
1		Background color	Foreground color
2		0	4
3	0	{SetProperty "Fill/Pattern";"0;0;Solid"}{d}	{SetProperty "Fill/Pattern";"4;0;Pepita"}{d}
4	1	{SetProperty "Fill/Pattern";"1;0;Solid"}{d}	{SetProperty "Fill/Pattern";"4;1;Pepita"}{d}
5	2	{SetProperty "Fill/Pattern";"2;0;Solid"}{d}	{SetProperty "Fill/Pattern";"4;2;Pepita"}{d}
6	3	{SetProperty "Fill/Pattern";"3;0;Solid"}{d}	{SetProperty "Fill/Pattern";"4;3;Pepita"}{d}
7	4	{SetProperty "Fill/Pattern";"4;0;Solid"}{d}	{SetProperty "Fill/Pattern";"4;4;Pepita"}{d}
8	5	...	...

How to color/format cells conditionally

QP has the ability to apply “conditional” colors to numbers in cells based on sheet-level properties. See Format > Sheet Properties (or `[Ctrl+F12]`) and the conditional color. QP also allows you to create a custom numeric format (page 76) that can do this and more to blocks of cells throughout the same notebook. Both of these allow you to specify that the numbers in the cell will be colored differently depending on whether they are, for instance, greater or less than certain values.

However, QP does not have the ability automatically to shade the cell itself (as opposed to the content of it), based on whether the number in it is greater or less than such values or based on the content of another cell. For things like that, you need a macro.

In the example in Table 15.8, we need to change the color and shading of numbers in the A column. If the number is less than 80, it should be colored red. If the

[1]Other colors can be constructed and stored in the notebook’s “Palette.” See the discussion at page 32 and the discussion linked there.

number is less than 65, the cell itself should be shaded a lighter shade of red. The following macro does it. Place the cursor in A1 and run the macro in B1.

Table 15.8: Conditional Formatting

	A	B
1	83	{If @CELLPOINTER("contents")=""}{Up}{End}{Up}{Quit}
2	66	{If @CELLPOINTER("contents")<80}{SetProperty "Text_Color","4"}
3	55	{If @CELLPOINTER("contents")<65}{SetProperty "Fill/Pattern";"33;0;Solid"}
4	90	{Down}{Branch B1}
5	62	
6	72	
7	74	
8	96	
9	58	
10	92	

C1 tells the macro to cease modifications when it comes to a blank, go back up to the top of the column, and then to stop. C2..C3 test whether the current cell contains a number below 80 and 65, respectively, and if so, it applies the red text color and an reddish shading. The macro then drops down and line and loops back to the top. The results are as depicted in the A column.

The variation in Table 15.9 highlights dates that are on a weekend.

Table 15.9: Highlighting weekend dates

	A	B
1	03/01/15	{If @CELLPOINTER("contents")=""}{Up}{End}{Up}{Quit}
2	03/02/15	{If @WKDAY(@CELLPOINTER("contents"))<3} {SetProperty "Fill/Pattern";"33;0;Solid"}
3	03/03/15	{Down}{Branch c(0)r(-2)}
4	03/04/15	
5	03/05/15	
6	03/06/15	
7	03/07/15	
8	03/08/15	
9	03/09/15	

How to color/format cells based on conditions in other cells

In Table 15.10, the task is to color numbers in columns A and B based on whether the number in column C is 1, 2, or 3. This solution takes one developed by Roy Lewis (OC 4061) and generalizes it a bit. Place the cursor on C1 and run the macro at E1. E1 tells the macro to cease highlighting when it comes to a blank, go back up to the top of the column, and then to stop. E2..E4 test whether the current cell contains a 1, 2, or 3, and apply particular colors (4, 5, and 24, respectively) to the two cells to the left of the cursor, which are identified by R1C1 numbers as c(-2)r(0)..c(-1)r(0).

The macro then drops down and line and loops back to the top. The results are as depicted in the A and B columns.

Table 15.10: Coloring text based on conditions in other cells

	A	B	C	D	E
1	a	b	1		{If @CELLPOINTER("contents")=""}{Up}{End}{Up}{Quit}

2	a	b	2		{If @CELLPOINTER("contents")=1}
					{SetObjectProperty "c(-2)r(0)..c(-1)r(0).Text_Color","4"}

3	a	b	3		{If @CELLPOINTER("contents")=2}
					{SetObjectProperty "c(-2)r(0)..c(-1)r(0).Text_Color","5"}

4	a	b	1		{If @CELLPOINTER("contents")=3}
					{SetObjectProperty "c(-2)r(0)..c(-1)r(0).Text_Color","24"}

5	a	b	2		{Down}{Branch E1}
6	a	b	3		
7	a	b	1		
8	a	b	2		
9	a	b	3		

The same is easy to set up for shading the cells as shown in Table 15.11. Setting the cursor in C1 and running the macro in D1 produces the results shown here.

Table 15.11: Shading cells based on conditions in other cells

	A	B	C	D
1	a	b	1	{If @CELLPOINTER("contents")=""}{Up}{End}{Up}{Quit}

2	a	b	2	{If @CELLPOINTER("contents")=1}
				{SetObjectProperty "c(-2)r(0)..c(-1)r(0).Fill/Pattern","33;0;Solid"}

3	a	b	3	{If @CELLPOINTER("contents")=2}
				{SetObjectProperty "c(-2)r(0)..c(-1)r(0).Fill/Pattern","29;0;Solid"}

4	a	b	1	{If @CELLPOINTER("contents")=3}
				{SetObjectProperty "c(-2)r(0)..c(-1)r(0).Fill/Pattern","24;0;Solid"}

5	a	b	2	{Down}{Branch D1}
6	a	b	3	
7	a	b	1	
8	a	b	2	
9	a	b	3	

For similar applications, see Ron Hirsch's macro that tests whether a date in a date column is within one of three ranges, and if so, it applies blue or red color to 13 cells on the same row (WPU 35861).

How to shade all cells in a block that contain a particular word

In the example in Table 15.12, we want to apply yellow highlighting to any cell containing a particular word. The cells to search are in the A column, and the search term is in cell C1. Since the word we're looking for may not be the only word in the cell, we'll need to use the `@FIND` function. This macro works by placing the cursor in A1 and running the macro in C3. The results are as indicated below.

Table 15.12: Highlighting cells that contain a particular word

	A	B
1	Insurance	Appeal
2	Domestic	
3	Business	{If @CELLPOINTER("contents")=""}{Up}{End}{Up}{Quit}
4	Business Appeal	{If @ISERR(@FIND(B1,@CELLPOINTER("contents"),0))}{Down}{Branch B3}
5	Corporate	{SetProperty "Fill/Pattern";"9;0;Solid"}
6	Insurance Appeal	{Down}{Branch B3}
7	Contract	
8	Domestic Appeal	
9	Fraud	
10	Contract Appeal	

C3 tells the macro to cease modifications when it comes to a blank, go back up to the top of the column, and then to stop. C4 tests for whether the text in C1 appears anywhere in the text of the cell in the A column. If not, an ERR conditions results, and the `@ISERR` function in C4 tells the macro to move to the next cell down and loop again. If no ERR results, we've found the word in the A column, and we then apply the shading in C5, followed by dropping down to the next cell and restarting the loop.

Note that this looks for an exact match. To ignore differences in upper and lower case spellings, use a canonizing function like `@UPPER` or `@LOWER`. For instance, this would serve the purpose in C4:

```
{If @ISERR(@FIND(@UPPER(C1),
@UPPER(@CELLPOINTER("contents"),0)))}
```

A modified version of this macro appears at WPU 38078.

How to zebra-stripe a block

It is often desirable to alternate the background colors of rows of data, a feature sometimes called zebra-striping. Table 15.13 shows one way to do it. Place the block to be zebra-striped in cell A1. Choose a desirable color for the stripes and get its associated number.[2] Place that number in the macro command in B4. In this example, the number 27 is by default a pale blue (cyan). Then run the macro.

B1 clears any prior shading in the block given in A1. B2 runs a `{For}` loop for the rows of data, but instead of using the usual fourth argument of 1, which would act on each row, the fourth argument here is 2, which means that it skips a row. As a result, the macro in B4 applies the zebra-shading to every other line, as intended.

[2]See the discussion at page 145 of the 255 colors available and how to view them. Alternatively, get the number by recording a macro choosing the color and applying it to a cell. If there is an easier way to get the colors, I'm not aware of it.

Table 15.13: Zebra-striping a block

	A	B
1	E1..H10	{SetObjectProperty (A1)&".Fill/Pattern";"0;0;Solid"}
2		{For A2,0,@ROWS(@@(A1))-1,2,B4}
3		
4		{SetObjectProperty @OFFSET(@@(A1),A2,0,1,@COLS(@@(A1)))& ".Fill/Pattern";"27;0;Solid"}

Chapter 16

Cell Content Macros

There are a number of commands that place data into cells. Which one is appropriate and best for a particular purpose depends on the particular behaviors of the command:

- Does it get data only from the current cell, or can it get data without navigating?
- Does it place the data only into the current cell, or can it place data without navigating?
- Can it get data from only one cell at a time, or can it take data from a block of cells?
- Can it put data into only one cell at a time, or can it place data into a block?
- Does it overwrite the formatting of the target cell with that of the source cell, or does it leave the format of the target cell unchanged? (Unless the cell is protected, they all overwrite the prior *content* of the target cell.)
- How does it handle formulas to be placed in the target cell? Are they converted to values, or do they remain formulas? Can the default handling of formulas be changed?
- How does it handle data from source cells that are blank? Does it place a blank in the target cell, or does it place a zero?
- Does it place text in the target cell with an apostrophe as a prefix (the old style) or does it omit the apostrophe?

A summary of findings appears at page 158 below.

Placing data anywhere: {Let}

`{Let` *`TargetCell,Data<:string>`*`}` places the `Data` argument into the `TargetCell`.

`TargetCell` is a single cell that can be anywhere, and thus need not be the current cell. `TargetCell` can be specified directly in the command (e.g., `A1`), or as the result of a function (e.g., `@OFFSET(A1,0,0)` or `@ADDRESS(1,1)`), or a string formula (e.g., `("A"&"1")`), all of which point to the same cell.

`Data` can be specified directly in the command (e.g., `12`), it can come indirectly from another cell, whether containing a value or a formula or a function (e.g., `A2`), and it can come from a formula or function written in the command (e.g., `+7+5` or `@NOW`). If the `Data` is from another cell that is blank, `{Let}` places a zero in `TargetCell`.

By default, any formula or function in `Data` is converted into a number or text when it is placed into `TargetCell`. If the command places text, the text is preceded by the apostrophe prefix. The command does not leave a formula or function as a formula or function.

`Data:string` places any `Data` as text into `TargetCell`. Any formula or function, including formulas and functions that return only text, will appear as text with an apostrophe prefix.

`{Let}` does not change the formatting of `TargetCell`.

Placing data into the current cell: {PutCell}, {PutCell2}

`{PutCell` *`Data<,DateOption>`*`}` and `{PutCell2` *`Data`*`}` put the `Data` argument into the current cell.

The only official difference between the two commands consists in how they treat a date-like string, such as *1/5/2015*. `{PutCell "1/5/2015"}` treats that number as a mathematical formula dividing 1 by 2 and then by 2015, rather than as a date, unless the date option 1 is added (e.g., `{PutCell "1/5/2015",1}`), but `{PutCell2 "1/5/2015"}` treats it as a date. Unofficially, `{PutCell2}` may fix a few quirks in `{PutCell}`. For most purposes, `{PutCell2}` should be preferred over `{PutCell}`.

The `Data` **argument** can be specified directly in the command (e.g., `12`), it can come indirectly from another cell, whether containing a value or a formula or a function (e.g., `A2`), and it can come from a formula or function written in the command (e.g., `+7+5` or `@NOW`). If `Data` comes from a blank cell `{PutCell}` places a zero in the current cell.

If the `Data` argument is a text string beginning with # or ., `{Putcell}` mishandles it, but `{Putcell2}` handles it correctly.

`Data:string`. By default, any formula or function in `Data` is converted into a number or text when it is placed into the current cell. However, if the formula or function is wrapped by quotation marks, or marked by an added `:string` argument, it places the formula itself into the current cell. Thus:

- `{Putcell2 @SUM(A1..A10)}` places the sum of all numbers in A1..A10 into the active cell.
- `{Putcell2 "@SUM(A1..A10)"}` places the function `@SUM(A1..A10)` into the active cell. `{Putcell2 @SUM(A1..A10):string}` does the same.

Some have reported in earlier versions of QP that when `Data` contains a formula or function that itself contains a double quotation mark, the command does not return the expected result. In QP17, that problem does not appear, but if it appears, there are workarounds such as `{Let @CELLPOINTER("address"),Data}` or `{Let []c(0)r(0),Data}`.

However, if you wish to place a function within double quotation marks, and the function contains double quotation marks within it, neither `{Putcell}` command places the function as desired. So, instead, some workaround should be used. For instance, if the null string ("") is desired, `@TRIM(@CHAR(32))` works. `@CHAR(34)` replaces a double quotation mark in string formulas.

If the commands place text, the text is not preceded by the apostrophe prefix.

The commands do not overwrite the existing format of the current cell, except where they place dates. In those cases, they apply impose a suitable date format on the cell.

`{PutCell}` seems to be equivalent to a macro command that simply enters the contents, followed by the tilde `~`. Thus, `{PUTCELL "Today's Date"}` works like `Today's Date~`.

`{PutCell}` requires navigation to the cell where the data will be placed.

Placing data or formulas into cells or blocks: {PutBlock}, {PutBlock2}

`{PutBlock}` and `{PutBlock2}` differ from `{PutCell}` and `{PutCell2}` only in allowing an optional `TargetBlock` argument following the `Data` argument, which allows the programmer to put the `Data` into each cell(s) in `TargetBlock`.

As with the other commands, the `TargetBlock` argument can be specified by a text formula or function. Thus, `{PutBlock}` does *not* require navigation to the cell where the data will be placed *if* you use the `TargetBlock` optional argument. For instance, `{SelectBlock A1}{PutCell +7+5}` can be replaced with `{PutBlock +7+5,A1}`.

The ability to put the same `Data` into a block of more than one cell may not be terribly useful in typical cases, but it can be quite useful when the `Data` is a formula that gives different results in each cell of the block. The formula must be wrapped in double quotation marks. Using the example in Table 8.4, in which a range of data (columns A and B) is to be analyzed by year and month, a single `{PutBlock}` command populates the entire block.

```
{PutBlock "@SUM(($B$1..$B$20)
*(@MONTH($A$1..$A$20)=$C2)
*(@YEAR($A$1..$A$20)=D$1-1900))",D2..F13}
```

In the earlier example, the formula wrapped in double-quotation marks was placed into D2 and copied to D2..F13. This `{PutBlock}` command does that in one step.

Incidentally, the `TargetBlock` argument can be specified in other ways, as by `@OFFSET` or `@ADDRESS` or string formulas. Thus, in the last example, this command would do the same thing:

```
{PutBlock "@SUM(($B$1..$B$20)
*(@MONTH($A$1..$A$20)=$C2)
*(@YEAR($A$1..$A$20)=D$1-1900))",
@OFFSET(D2,0,0,12,3)}
```

Compared with `{Let}` and `{PutCell}`, `{PutBlock}` seems to have the best of both worlds.

Putting data from one block into another: {Put}

`{Put` *`Block,Col#,Row#,Data`*`}` is the reverse of the `@INDEX` function, which retrieves information from a `Block` based on the `Col#` and `Row#` within the `Block`. `{Put}` places a `Data` into a `Block` at the intersection of the column and row specified in `Col#` and `Row#`, which are both offsets from the topmost leftmost cell in the `Block`, where `Col#` and `Row#` are 0.

`Data` can be specified directly in the command (e.g., `12`), it can come indirectly from another cell, whether containing a value or a formula or a function (e.g., `A2`), and it can come from a formula or function written in the command (e.g., `+7+5` or `@NOW`). If `Data` comes from a blank cell, `{Put}` places a zero in the target cell.

By default, any formula or function in `Data` is converted into a number or text when it is placed into the appropriate cell of the `Block`. If the command places text, the text is preceded by the apostrophe prefix. The command does not leave a formula or function as a formula or function.

`Data:string` places any `Data` as text into the designated cell withing `Block`. Any formula or function, including formulas and functions that return only text, will appear as text with an apostrophe prefix.

`{Put}` does not change the formatting of cells in the `Block`.

Since `{Put}` does not place formulas, and since its arguments imply transferring data into a table-like structure, `{Put}` is an excellent way to transfer values from one block of data into another, using `{For}` loops. See the chapter on retrieving information from databases on page 170.

Recreating a block of formatted data: {BlockValues}

`{BlockValues` *`SourceBlock,TargetCell`*`}` takes data in the `SourceBlock` and places it in a same-sized block that starts in (has its topmost leftmost cell as) `TargetCell`. Both arguments can be specified in string formulas or functions. In copying the data, the `{BlockValues}` command:

- Converts any formulas in the data into values.
- Copies the formats as well, thereby overwriting both data and formats.
- Copies blank cells as blanks, not as zeros.
- Copies text without adding a prefix.

This is a very quick and easy way to replicate particular blocks of data in a new location. However, it has a few possible drawbacks:

- It copies all formatting from the source cells, and you may only want numeric formatting.
- I have also occasionally sensed that it is slower than other methods, perhaps since `{BlockValues}` gets all (or almost all) thirty properties from every source cell (page 29) and sets the parallel cell in the target block with each property. That takes time. For a small number of cells, the time is negligible and the advantage of preserving formats can be great, but for numerous cells, the slowdown can be annoying, at least in some contexts. On the other hand, there are techniques for speeding performance (page 113).

Recreating a block of data, with options: {BlockCopy}

`{BlockCopy` *`SourceBlock,TargetCell<,ModelCopy?,Options>`*`}` takes data in `SourceBlock` and locates it in a same-sized block that starts in (has its topmost leftmost cell as) `TargetCell`. Both locations may be specified by text formulas or functions. Blank cells in the `SourceBlock` are rendered as blanks, not zeros, in the Target.

According to the help file, `{BlockCopy}` copies everything from the cells of the `SourceBlock` to the cells in the Target Block, beginning with the `TargetCell`: it copies formulas, values, properties, objects, row and column sizes, text (labels), and numbers. The help file suggests that to copy less than everything from the `SourceBlock`, the first option (`ModelCopy?`) must be changed from its default value of 0 to 1. As the discussion below indicates, the help file is partially right and partially wrong in discussing the options.

The `Formulas?` option. `{BlockCopy}` copies formulas in `SourceBlock` to the Target block by default. When it does so, absolute references remain fixed to their original targets, but relative references do not. For example, if a cell in A2 contains the formula `+A1`, `{BlockCopy A2,A10}` would place the formula `+A9` into cell A10. However, if A2 contained the formula `+$A$1`, `{BlockCopy A2,A10}` would place `+$A$1` into A10.

Hence, if you are going to copy formulas with relative references, it makes the most sense to copy them into a framework that is similar to the framework from which they were copied, or at least into some pre-planned framework. Otherwise, it makes no sense to copy formulas with relative references.

`{BlockCopy}` does not contain an option to convert a formula or function to a value. If you need to convert those cells to values, `{BlockCopy}` is not a good choice.

The only option regarding formulas is to omit them. To do so, `ModelCopy?` must be set to 1, and the `Formula?` option set to 0: `{BlockCopy Source, Target,1,0}`. The target cell would be left unchanged.

The `Values?` option. `{BlockCopy}` by default copies values to the target (of course). To omit values, if there were ever a reason to do so, the `Values?` option must be set to 0: `{BlockCopy Source,Target,1,?,0}`. If values are omitted, the target cell would remain unchanged.

In these and the following examples, a question mark is left in the formula where the user can choose 1 or 0 for other options. Some choice of earlier options must be made in order to effectively choose a later option.

The `Formulas?` and `Values?` options provide an indirect way to test if a formula is in a `Source` cell which is otherwise lacking in QP. After clearing a `Target` cell, setting `Formulas?` to 1 and `Values?` to 0 will change the `Target` cell only if the `Source` cell contains a formula or function.

The `Properties?` Option. By default, `{BlockCopy}` copies at least some formatting from the Source cells to the parallel Target cells. To omit the copying that formatting, the `Properties?` option must be set to 0: `{BlockCopy Source,Target,1,?,?,0}`. Otherwise, it applies at least the most obvious cell formatting to the target cell (including row height).

The `Objects?` option. Unlike those cases, `{BlockCopy}` does *not* by default copy objects (in my case mostly command buttons) adhering to source cells into target cells. In order to copy objects, `ModelCopy?` must be affirmatively set to 1. Then objects will be copied unless the `Objects?` option is set to 0.

The `Row/Col_Sizes?` option. This option would suggest that both row height and column width are controlled by the same means, but that does not appear to be true. It appears that row size is changed by default (see the `Properties?` option, above), unless that option is deselected. However, unlike the row height and the other cases above, `{BlockCopy}` does not appear to copy the column width of the source cells unless *both* `ModelCopy?` is set to 1 and

`Row/Col_Sizes?` is set to 1. Hence, to copy the row and column sizes to the target, use `{BlockCopy Source,Target,1,?,?,?,?,1}`. To copy neither, use `{BlockCopy Source,Target,1,?,?,?,?,0}`.

Other options. The final two options, `Labels?` and `Numbers?`, are obscure. The help file would suggest that one could omit copying text (labels) or numbers by setting those options to 0, but my testing shows that such settings have no effect whatsoever. The help file states that the `Numbers?` option is “reserved for Cell Comments,” whatever that means.

Pasting from the clipboard: {EditCopy} and {EditPaste} or {PasteSpecial} or {PasteFormat}

A macro can put a block of cells into the clipboard and paste it to another location. The macro navigates to the source cells and selects them. That can be done with `{SelectBlock}` or `{EditGoto}`. `{EditCopy}` then puts those cells into the Windows clipboard. The macro then navigates to the place where the block will be pasted. Blank cells are pasted as blanks, not zeros.

EditPaste

`{EditPaste}` will place the data with all formats (except column width) into the target location. Formulas are not converted to values. As with `{BlockCopy}`, formulas with absolute references to other cells will continue to refer to the same cells, but formulas with relative references will adjust relative to the new location. See comments on `{BlockCopy}` for a discussion of that issue.

This is the command equivalent of clicking Edit > Paste or `[Ctrl+V]`.

PasteSpecial

`{PasteSpecial` *`Options`*`}` allows the programmer to pick and choose certain features of the data in the clipboard to paste, and unless listed, other features are not pasted. It appears to be the equivalent of clicking Edit > Paste Special when the clipboard contains data copied *from QP*, which opens the dialog box in Figure 16.1.

The help file suggests that the user needs to enclose the parameters in double-quotes and place them in a particular order, with excluded parameters marked by an empty string of two double-quotes (""), but experimentation shows that to be inaccurate. Instead, programmers can list the desired features to be pasted in *any* order, without double-quotes, and ignoring unwanted parameters. In the order I find most useful, the parameters are:

`Label cells` allows the pasting of text (not formulas that create text).

`Number cells` allows the pasting of numbers (not formulas that return numbers).

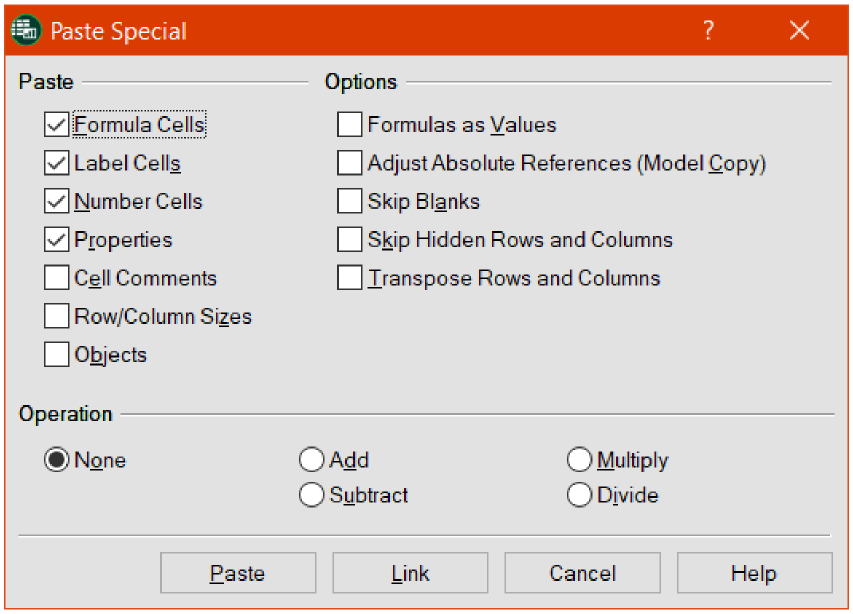
`Formula cells` allows the pasting of formulas that generate text or numbers.

`Formula Values` converts formulas to the values they would have in the source block before pasting the values in the target.

`Properties` pastes all of the formatting that would be pasted by `{EditPaste}`.

`Transpose` pivots columns and rows.

`Other`. There are also `NoBlanks` and `Cell_Comments` options that are obscure to me.


Paste Special

Paste
- [✓] Formula Cells
- [✓] Label Cells
- [✓] Number Cells
- [✓] Properties
- [ ] Cell Comments
- [ ] Row/Column Sizes
- [ ] Objects

Options
- [ ] Formulas as Values
- [ ] Adjust Absolute References (Model Copy)
- [ ] Skip Blanks
- [ ] Skip Hidden Rows and Columns
- [ ] Transpose Rows and Columns

Operation
[● None] ○ Add ○ Subtract ○ Multiply ○ Divide

Figure 16.1: Paste Special from QP to QP

Some examples:

- To paste formats ("properties") only, use:
  `{PasteSpecial Properties}`
- To paste a series of formulas as their values, use:
  `{PasteSpecial Formula cells,Formula Values}`
- To paste all data with their values, use:
  `{PasteSpecial Label Cells,Number cells,Formula cells,`
  `Formula Values}`
- To add formats but without converting formulas to values (and thus to emulate `{EditPaste}`), use:
  `{PasteSpecial Label Cells,Number cells,Formula cells,`
  `Properties}`
- To do the same but convert formulas to values, use:
  `{PasteSpecial Label Cells,Number cells,Formula cells,`
  `Formula Values,Properties}`

PasteFormat

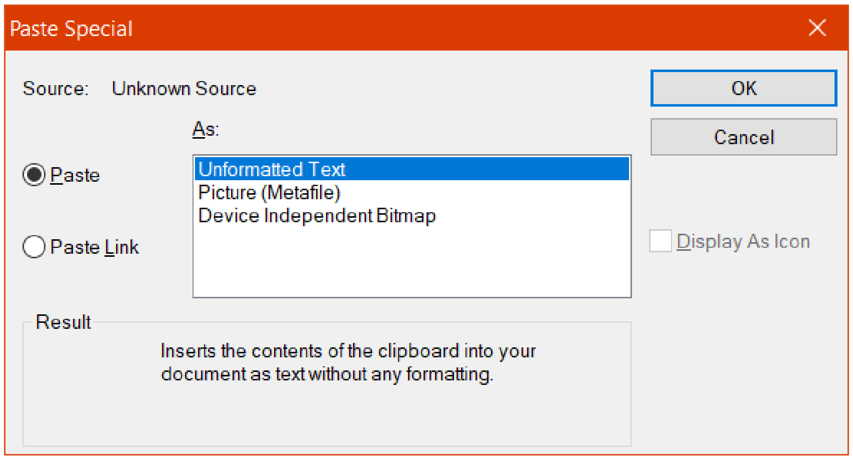
`{PasteFormat` *`FormatType`*`}` is poorly documented, but it appears to allow the programmer to paste in one of the *Windows* format options that appear when one uses the Edit > Paste Special menu selection when the clipboard contains data copied *from sources other than QP*. Doing so opens the dialog box in Figure 16.2.

The `FormatType` options are not specified in the help file, apparently because they vary depending on what is currently in the Windows clipboard.

Assuming that text data in a cell has been copied into the clipboard, the easiest way to paste that text without the cell's formats is:

```
{PasteFormat Text}
```

This has been so useful in my work that I've added a button to a toolbar containing this command.



Inserts the contents of the clipboard into your document as text without any formatting.

Figure 16.2: Paste Special to QP from Non-QP Sources

Placing data and selected properties with {SetObjectProperty}

One overlooked method of placing data is by setting the `.Value` property of a cell with a given value, using `{SetObjectProperty}`. And like the methods of copying values from one cell to another, we can use parallel construction. Thus, to set the value of cell A1 on the Report sheet as the same as the value of cell A1 on the Data sheet, this command works.

```
{SetObjectProperty "Report:A1.Value",
"Data:A1.Value"}
```

This command converts formulas to values, and blank cells are placed as blanks, not zeros. One particular advantage of this approach is that the same sort of formula can be used to transfer selected format properties at the same time; all other commands transfer either all or no properties. For example, the macro can transfer the numeric format this way:

```
{SetObjectProperty "Report:A1.Numeric_Format",
"Data:A1.Numeric_Format"}
```

Those commands are not very useful in that format, but they can be constructed by formulas, and the cells can be varied by appropriate `{For}` loops so as to allow us to transfer all the values in one database to another location. For example, if the Row counter for one `{For}` loop is in A1 and the Column counter for another is in A2, this formula would generate the command to transfer values from the Data sheet to the Report sheet in a way that is correct for every cell:

```
{SetObjectProperty ("Report:"&@ADDRESS(A1,A2)&
".Value")("Data:"&@ADDRESS(A1,A2)&".Value")}
```

Techniques for coding such loops are given in the chapter on retrieving data from a database (page 170).

Comparison of Data Transfer Commands

Comparing the preceding discussion with the questions raised at the beginning of this section, Table 16.1 shows the major features of each of these methods of placing data into cells.

Table 16.1: Comparison of Data Transfer Commands

	Let	Putcell2	PutBlock2	Put	BlockValues	BlockCopy	EditPaste	PasteSpecial	SetObjectProperty
Can source be remote?	Y	Y	Y	Y	Y	Y	N	N	Y
Can target be remote?	Y	N	Y	Y	Y	Y	N	N	Y
Can source be a block?	N	N	N	N	Y	Y	Y	Y	N
Can target be a block?	N	N	Y	N	Y	Y	Y	Y	N
Change format of target?	N	N	N	N	Y	Opt	Y	Opt	Opt
Convert formulas to values?	Y*	Opt	Opt	Y*	Y	N	N	Opt	Y
Places blanks as zero?	Y	Y	Y	Y	N	N	N	N	N
Add ' prefix to text?	Y	N	N	Y	N	N	N	N	N

* Macro can optionally convert formulas to text.

Converting numbers to formatted numberlike text: {Contents}

@Functions provide a number of ways of converting formatted numbers to formatted text (see page 68). They are not, however, comprehensive, as a look at date and time numbers will attest. To generate text that always looks like a formatted number, use the macro command `{Contents}`.

`{Contents` *`TargetCell,SourceCell,<Width,NumericFormat#>`*`}` takes a number as it appears formatted in the `SourceCell` and places it as text in the `TargetCell`. This is an excellent method of converting numeric formats, and it is unfortunate that there is no @Function that does the same thing. In particular, there is no good single @Function for converting a numeric date into a text. `{Contents}` is a decent second option.

If `SourceCell` is too narrow, QP displays asterisks through the cell (e.g., *****), and the optional `Width` argument circumvents the potential problem by (the help file says) making QP pretend that the column housing `SourceCell` is of a larger width (though it isn't). Placing 1000 in that argument is pretty safe. However, at least in QP17, this does not actually appear to be a real problem: `{Contents}` does not send asterisk to the `TargetCell` even if they are displaying in the `SourceCell`.

The optional `NumericFormat#` says that a format number from 1 to 127 will impose a particular format on a number. In many cases it does, but the help file is inaccurate.[1] I used a little macro to generate a list what is returned when each format number is used for a value of 42071.567. Table 16.2 shows the results.

Table 16.2: Numeric options for {Contents}

Code	Help file's description	What it actually does for 42071.567
0-15	Fixed (0-15 decimals)	Nothing
16-31	Scientific (0-15 decimals)	42072; 42071.6; 42071.57; 42071.567; 42071.5670; 42071.56700; etc.
32-47	Currency (0-15 decimals)	Same
48-63	% (percent, 0-15 decimals)	Same
64-79	, (comma, 0-15 decimals)	Same
80-95	not mentioned	42,072; 42,071.6; 42,071.57; 42,071.567; 42,071.5670; 42,071.56700; etc. This is comma format
96-111	not mentioned	42071.567
112	+/- (bar chart)	********************************
113	General	42071.567
117	Text	
125-126	not mentioned	
127	Default	
114	Date [1] (DD-MMM-YYYY)	08-Mar-15
115	Date [2] (DD-MMM)	08-Mar
116	Date [3] (MMM-YYYY)	Mar-15
121	Date [4] (Long International)	03/08/15
122	Date [5] (Short International)	03/08
119	Time [1] (HH:MM:SS AM/PM)	01:36:29 PM
120	Time [2] (HH:MM AM/PM)	01:36 PM
123	Time [3] (Long International)	1:36:29 PM
124	Time [4] (Short International)	1:36

Removing data and formats: {Blank}, {EditClear}, {ClearFormats}, {ClearContents}

Several commands are relevant to remove data and/or formats.

To remove the data in a Block, but leave formats intact, use `{Blank *Block*}` or select the Block first and use `{ClearContents 0}`.

To remove only the formats but leave the data intact, select the Block first and use `{ClearFormats 0}`. The formats removed are those that apply to the cell; in-cell formatting will not be removed unless the data is.

To remove the data and the formats in a Block, navigate to the Block and use `{EditClear}`.

[1]The help file's examples might have been valid when QP was a DOS program, but they are completely erroneous now.

Using search and replace in a macro: {Search}

QP allows the user to search and replace text with `[Ctrl+F]` or Edit > Find and Replace. These can be automated by using the `{Search}` macro command,[2] but for reasons noted in WPU 36272, the help file is mistaken and misleading in some respects. The ensuing discussion resulted in a macro like Table 16.3.

Table 16.3: Search and replace by macro

	A	B	C
1	Smith, John	🟨	{Search.Reset}
2	Smith, Jane	🟨	{Search.Block A1..A100}
3	123 Main St		{Search.Direction Row}
4	Anywhere, GA		{Search.Find @concatenate(B1)}
5	(478) 555-1212		{Search.Look_In Value}
6			{Search.ReplaceBy @concatenate(B2)}
7			{; Search.Match Whole}
8			{Search.ReplaceAll}

Place the term to be replaced in B1 and the term to substitute in B2.

C1 is necessary to clear out any pre-existing settings that might interfere with the intended search and replace.

C2 sets the block within which to search; modify it appropriately.

C3 may be unnecessary for this macro; it determines whether the program looks first on rows or columns before going to the next column or row, respectively.

C4 locates the term to be found in B1. The discussion in WPU 36272 looked at several ways of referencing the content of B1 and B2, some of which allowed a replacement of text but not numbers, others that allowed replacement of numbers but not text. The `@CONCATENATE` function worked in both cases.

C5 states whether you are looking for a term to replace in the value in the cell or in a formula. The default is to search in formulas; this macro changes that.

C6 locates the term to substitute in B2.

C7 is an optional command requiring the term in B1 to match the entirely of a cell in the block specified in C2. The default is that it will find and replace any matches within a cell, so to have it match the entirety of a cell, remove the leading semicolon.

C8 is the command that makes the replacement.

How to deal with blanks in the source cells

As noted above, many of the functions and macros that transfer data from one cell to another will treat blanks as zeros: `{Let}`, `{Put}`, `{PutCell}`, `{PutBlock}`, and functions like `@@`, `@INDEX`, and `@VLOOKUP`. The user may ignore them, but if the user would prefer not to see zeros where there should be blanks, the user has several options.

- At the sheet level, one can choose not to display zeros. Press `[Ctrl+F12]`, select the *Display* tab and choose the *No* radio button under *Display Zeros*. Of

[2]For a macro-driven alternative to the `{Search}` command that lists matching cells in a block, displays their content, and allows the user to jump to chosen matches, see page 231.

course, one may wish to display some zeros or may simply want blank cells to transfer as blank.

- In the source data, one can fill the blank cells with a prefix like the apostrophe. These methods will transfer apostrophes to the target cell, but zeros will not be there.
- Before the command that would place a zero, use an `{If}` test to determine whether the source cell contains a blank. If it does, use a `{Return}` to stop before placing a zero in the target cell.
- Have the command place an `@IF` function instead of the source cell. A function of the format `@IF(SourceCell="","",SourceCell)` will be placed in the target cell as a blank cell with an apostrophe prefix.
- Use a different method of transferring data that does not have the same effect, such as `{BlockValues}`, `{BlockCopy}`, or `{SetObjectProperty}`.

How to transfer data with no formatting except numeric formatting

Except for `{SetObjectProperty}`, all of the commands above that transfer data from one cell to another either transfer no format properties or (nearly) all of them. That means that they either *(a)* transfer numbers like $123.50 as 123.5, and dates like 04/06/15 as 42100, or *(b)* transfer numbers with their justification, alignments, font attributes, line-heights, highlighting, etc. I am probably not alone in wishing that numeric data could transfer with the numeric formatting, but without the other cell and display formats. There are several strategies for accomplishing the objective:

- One may preset the target with the desired numeric format. Then, commands that transfer numbers without any formatting will yield the correct result.
- The macro can apply desired formatting after the data transfer.
- If it is not important that the numbers remain numbers, or conversely, if it is adequate that the numbers be saved as text (as where they may be imported as text into WP), the macro `{Contents}` can convert the number to a text representation of the way it appears formatted onscreen. The formatted text can be stored in the destination cell.
- One may accomplish the transfer by two iterations of `{SetObjectProperty}` per cell, one that transfers the `.Value` to the target cell from the source cell, and then another that transfers the `.Numeric_Value`.
- A variation on the last strategy would be to transfer the data without formats and then to test for particular numeric formatting in the source cell. If `@CELL("format",SourceCell)` returns some value other than "G", the macro could then use `{SetObjectProperty}` to apply a particular `.Numeric_Value` to the target cell.
- A macro may transfer the cells with all formats, and then apply (and create if necessary) a special style that would strip out all formatting except numeric formats. Under Format > Define Styles, one can create a style that limits the properties that may be varied to those involving numeric format.

  To create the style manually, get the Styles dialog box by clicking Format > Define Styles. In the style name box, type a suitable name; I use *JustNumbers*. There are seven check boxes. Make sure that each one of them is checked except *2 Format*, which allows the numeric format to vary, but it fixes the others (Alignment, Protection, Line Drawing, Shading, Font, Text Color) on a standard, which should also be your normal standard. The Style will appear as an option in a dropdown box on the property bar, and it may then be applied to a block of cells manually.

The programmer can use the {NamedStyle} command to create the style just described, as shown in Table 16.4.

Table 16.4: A macro to create a format-style

	A
1	{NamedStyle.Protection "Arial;8;No;No;No;No"}
2	{NamedStyle.Line_Drawing "Clear;Clear;Clear;Clear;Clear;Clear;Clear;Clear; 0;0;0;0;0;0;3;3"}
3	{NamedStyle.Fill/Pattern "0;3;Solid"}
4	{NamedStyle.Text_Color "3"}
5	{NamedStyle.Define JustNumbers;0;0;1;1;1;0;1}

After creating the style, apply it by selecting the relevant block and this command: `{SetProperty Style;JustNumbers}`.

How to enter in the current cell the number above it plus one

A user asks "How do I create a Quattro Pro macro that adds 1 to the cell above? My problem is that I don't want to reference a specific cell such as A:A1, but rather the cell above the one in which my cursor is currently located."

Several commands should work. In WPO 16115, I recommended:

```
{Putcell c(0)r(-1)+1}
```

`c(0)r(-1)` references the cell in the same column (zero offset) and the row above (-1 offset).

How to edit every cell in a column/block

There are many reasons to perform some operation on every non-empty cell in a column. A standard method for doing this is to begin with a test about whether the current cell is empty. If so, the macro ends. If not, it performs certain actions, drops down to the next cell in the column, and then branches back to the beginning test. This can often be written in a single cell, as in Table 16.5. The if test determines where the current cell has any content. If not, the `type` argument evaluates as *b* and the macro stops. Otherwise, it performs the variable functions that the programmer fills in. The `{D}` command moves the cursor down to the next cell, which may not be necessary if the commands the user fills in would do that anyway (the ~ or `{CR}` commands may be set to do that). The `{Branch}` command at the end branches back to the same if-test.

Of course, the macro need not be compressed into a single cell. More complex editing cannot be compressed into a single cell.

Table 16.5: Single-cell repeating macro

	A
1	{If @CELLPOINTER("type")<>"b"}Fill in Command(s){D}{Branch c(0)r(0)}
2	

If there are gaps in the column, macros like these may stop before you want it to stop. In such cases, a `{For}` loop might be better, or a loop that begins by asking you how many times to repeat it. In Table 16.6, the user would supply the block to be edited in cell E1 and run the macro in F1. The `{For}` loop in F1 would cycle through the rows of the block, and call the `{For}` command in F3, which would in turn cycle through the cells in the block on that row, column-by-column, and run the macro commands starting in F5, which place the cursor on each cell in the row. Commands in F6 and below would perform some operation on that cell.

Table 16.6: Basic {For} loop to modify every cell in a block

	E	F
1	A1..C10	{For E2,0,@ROWS(@@(E1))-1,1,F3}
2	Row	
3	Col	{For E3,0,@COLS(@@(E1))-1,1,F5}
4		
5		{SelectBlock @OFFSET(@@(E1),E2,E3)}
6		[Fill in command(s)]

Applying Constraints. So, for instance, where a column of entries includes text and numbers, but the user then imposes a Labels-only "constraint" (`[F12]`, *Constraints* Tab) on the column so that all entries will be treated as "Labels" (text), this does not automatically change prior numbers into number-like text. However, a macro that presses `{Edit}` twice would effectively reenter each cell, and the constraint would then convert the number into number-like text:

```
{If @CELLPOINTER("type")<>"b"}{Edit}{Edit}{D}
{Branch c(0)r(0)}
```

Converting number-like text to numbers. If the column contains number-like text, this macro will replace it with the actual numbers:

```
{If @CELLPOINTER("type")<>"b"}
{Putcell @VALUE(@CELLPOINTER("contents"))}{D}
{Branch c(0)r(0)}
```

Apply proper case. If the column consists of text written in inappropriate upper and lower case letters, these can be converted to proper case by `@PROPER` and a macro.

```
{If @CELLPOINTER("type")<>"b"}
{Putcell @PROPER(@CELLPOINTER("contents"))}{D}
{Branch c(0)r(0)}
```

Adding Prefixes. If the column contains numbers or text without a prefix, a macro is an easy way to add them. This macro would add the right-aligning double-quote prefix:

```
{If @CELLPOINTER("type")<>"b"}
{Edit}{Home}"{D}
{Branch c(0)r(0)}
```

Removing Prefixes. If the column contains text beginning with a prefix, a macro can easily remove the prefix:

```
{If @CELLPOINTER("type")<>"b"}
{Edit}{Home}{Del}{D}
{Branch c(0)r(0)}
```

Incrementing cells. If you need to add one to each cell in the column, use this:

```
{If @CELLPOINTER("type")<>"b"}
{Putcell c(0)r(0)+1}
{Branch c(0)r(0)}
```

Reversing signs. If you need to change the signs of every number in a column, this macro will do so by multiplying the contents by -1:

```
{If @CELLPOINTER("type")<>"b"}
{Putcell c(0)r(0)*-1}{D}
{Branch c(0)r(0)}
```

Converting relative references to absolute references. If a column contains formulas such as `+A1`, these can be converted to the form `+$A:$A$1` by a macro:

```
{If @CELLPOINTER("type")<>"b"}{Edit}{Abs}{D}
{Branch c(0)r(0)}
```

Kenneth Hobson has a version of this macro that extends its functionality at WPU 26908.

How to determine if a cell contains a formula

There appears to be no direct way to determine whether a cell contains a formula using an @function. Several ways of doing so by macro are suggested in WPU 32603 and WPU 36660.

The PerfectScript macro command `GetCellFormula` returns the formula within a cell, so if it differs from the value returned by `GetCellValue`, there is a formula in the cell.

A method of doing so in QP uses the ability of `{BlockCopy}` to copy formula cells without copying value cells to a particular destination. The macro in Table 16.7 uses that command to copy the current cell to A1 (which can be changed), and then tests whether A1 is blank (meaning that a formula is not in the current cell) or not (a formula is present). It puts a message to that effect in A1.

Table 16.7: A test for whether a cell contains a formula

	A	B
1		{Blank A1}
2		{BlockCopy @cellpointer("address"),A1,1,1,0}
3		{if @cell("type",A1)="b"}{Let A1,"No formula"}{quit}
4		{let A1,"This contains a formula"}

How to override in-cell formatting

QP displays the properties of the selected cell on its property bar, but the user can select all or part of the contents of the cell and apply different formatting. This can cause confusion when the apparent format of the cell differs from the format reflected by the property bar.

The question arises: How can one change that formatting? Changing the properties of the cell will not change in-cell formatting. Selecting the cell and using `{ClearFormats 0}` will not change the in-cell formatting, but it will remove the cell-level formatting.

The only easy way to change the in-cell formatting is simply to replace the cell's contents with a new copy. The easiest way is to use {Let} to replace the cell with its contents:

```
{Let []c(0)r(0),[]c(0)r(0)}
```

This would remove in-cell formatting, but the data would conform to the cell-level format. Using techniques for editing every cell in a block (page 162), this set of commands removes in-cell formatting for an entire block. See the discussion at WPU 35608, which includes Kenneth Hobson's PerfectScript solution.

Chapter 17

Date and Time Macros

Entering dates with Ctrl+D or a {Putcell} Macro

Until QP17 Service Pack 2, `[Ctrl+D]` was hard-coded to enter a date in the "short" version of whatever date format is selected in Tools > Options > Application > International > Date Format. Now, entering `[Ctrl+D]` places the date in whichever numeric-date format has been applied to the cell, as the user is likely to expect, but before this revision, one had to choose the default format in QP's settings. The first option in the settings defaulted to the Windows System's default, which can be set in the Windows Control Panel, under Region and Language. The remaining options display a long form and the short form in parenthesis. `[Ctrl+D]` enters the short form.

Caution 1. The Windows System default option allows a greater variety of useful date formats, but its "short" version is comparable to QP's long version, and its "long" version is considerably longer. That is, if you have already formatted dates using QP'a long version and switch to the Windows System default, your columns may need to be widened to display the dates properly.

Caution 2. Another problem with using the first "Operating System Default" option is the loss of a useful date-entry function. Using QP's options, the user can enter a date by typing "12/3/14," and QP would interpret that as a date. Using the Operating System Default option, typing those same numbers simply yields the formula of 12, divided by 3, divided by 14.

A discussion of these problems appears at WPU 36591.

Using the {Putcell} Macro Alternative

For the above reasons, `[Ctrl+D]` is not always a desirable method. A workaround is to create a macro that inserts the value `@TODAY` into the current cell. The macro can be made permanent in several ways.

System notebook. QP allows the creation of a macro library in a "system notebook," but I don't use that method.

Toolbar macro. I prefer to create a macro as part of a toolbar, and either add a button to the toolbar that plays the macro, or a shortcut keystroke to play the macro, or both. Here are the steps:

1. Click Tools > Customize > Customization > Commands
2. In the dropdown box under "Commands," click Macros. A visually confusing layout appears, but this is where you add macros. The list box on

the left contains all macros *previously* added, and we will want to add a macro to that list.

3. To add a new macro to the listing, click “Add.” A new item will appear in the list box to the left. Despite any appearance to the contrary, this item is not yet linked to any macro.
4. Enter the macro commands in the “Enter Macro” field. Here, I enter `{Putcell @TODAY}`. If you want to be sure to apply a particular date format to the cell, add something like

   ```
   {Setproperty Numeric_Format;"Long Date Intl.;2; United
   States;0"}
   ```

   Note that, in lieu of a native QP macro, you could specify that the item to should be linked to a PerfectScript macro at this stage.
5. Click “Apply” to link those macro commands (or PerfectScript macro) to that item.
6. Now the item can be drug to a toolbar.

   Changing the toolbar icon. If you prefer some other image for a toolbar icon, versions of QP before QP17 allowed the user to change a grid on the Appearance tab of the dialog box. Starting with QP17, like WP17, the user must now import an image from an external file. Image files can be created with Corel’s Presentations or other software. In QP17, an `[Import ...]` button on the General tab allows the selection of the image file.

Keystroke shortcut macro. You can also assign the macro to a keystroke by following the preceding steps that link a macro to an item in the Macros list box.

1. At this point click the “Shortcut Keys” tab.
2. In that tab, click in the edit box under “New Shortcut Key:”
3. Click your preferred key combination. In this example, I would suggest `[Ctrl+Shift+D]`. Those characters will appear in the edit box.
4. Then click the “Assign” button.

After closing out of the customization dialog box, either the toolbar icon or the keystroke shortcut should be usable.

How to enter the current time in a cell

There are several options for placing the current time in a cell. These examples assume that the user wants HH:MM AM/PM format, but other time formats can be substituted.

To put the time into the current cell, use:

```
{Putcell @Now}
{Setproperty Numeric_Format;"HH:MM AM/PM;2;United States;0"}
```

To put the time into a particular cell, such as A1 here, use:

```
{Let A1,@Now} {SetObjectproperty A1.Numeric_Format;"HH:MM
AM/PM;2;United States;0"}
```

The number in both can be converted to text by using `{Contents}`.

How to display the date and time in a cell

If you need to insert the date *and time* into a single cell, this will require creating a custom numeric format (see page 76) that displays both, and then formatting the cell with that custom format. A macro can put a date into one cell and a time into another, but there is no built-in format for doing so in one cell. Here is a way to do it.

1. Create a special format that displays such numbers in the way you want. Mine is called "mm/dd/yyyy time" and looks like mmo/dd/yyyy hh:mmi: ssam in the special format dialog box.
2. The macro would be

```
{Putcell @NOW}{Setproperty Numeric_Format;
"mm/dd/yyyy time;2;United States;0"}
```

A different macro could be written if you want to display this date and time as string/text rather than a number. If it is a number, the column in which it is put needs to be wide enough to display it in full. If it is a string, it can overlap the next column.

How to save the closing date and time with the file

Give one cell (here, A1) your preferred numeric format for dates, and give another cell (here, A2) your preferred numeric date for time. Then, in the macro initiated in the cell named `_NBExitMacro`, place these commands.

```
{Let A1,@NOW}{Let A2,@NOW}
```

You may add other commands to the macro, but the macro should include `{FileSave}` in order to save the date and time.

How to run commands only the first time a notebook is opened each day.

You may want a notebook to run commands once, but only once, a day, usually at the time the notebook is first opened in a day. Table 17.1 shows one way to do it. First, decide on a cell to store the date that the notebook is opened. Here, we'll use cell A1. Then in a cell that you name `_NBStartMacro` (here B2), test for whether the date in A1 is today's date. If not, the macro branches to the commands (here at C1), to be run once a day. Those commands put today's date into A1 and save the notebook. If the notebook is closed and reopened on the same day, the test in B2 will not cause the macro to run the commands in C1.

Table 17.1: Running commands automatically, but only once a day

	A	B	C
1		_NBStartMacro	{;Once a day commands}
2		{If A1<>@Today}{Branch C1}	{Let A1,@Today}
3			{FileSave}

Numerous variations on this theme are possible.

How to get time precision

As noted in the discussion of the `@NOW` function (page 58), QP cannot give more precision than 1 second intervals. PerfectScript, however, allows for greater precision.[1] The macro in Table 17.2 (which can be run by `{PlayPerfectScript}`) will place the current time into the current cell with far greater precision.

Table 17.2: Time precision with PerfectScript

```
Application (QP; "QuattroPro"; Default!)
vTime :=TimePart()
nTime :=NumStr(vTime)
qp.PutCell2(nTime)
```

[1] See the fuller discussion of PerfectScript in Chapter 24, beginning at page 235.

Chapter 18

Databases: Retrieving Information

QP has many ways to retrieve information in databases. We'll look at some here, but pass on others like CrossTabs and Forms under Tools > Data Tools. In the following examples, we'll work with the following assumptions.

- Sheet A will be the user's interface with the data.
- Sheet B will store reports generated from the data.
- Sheet C will contain the macros.
- Sheet D will contain the data itself.

I recommend that you keep such functions on separate sheets.

Also, to keep addresses straight, the illustrations of spreadsheets will show the page letter in the actually blank top left header cell.

The data on Sheet D will be stored in a well-structured database. In these examples, we'll assume that the database has unique index values (invoice numbers) in column A, dates in B, customers' names in C, quantities in D, and prices in E. It will look like Table 18.1.

Table 18.1: Sample well-structured database on Data sheet

D:	A	B	C	D	E
1	Invoice	Date	Customer	Quantity	Price
2	100	03/04/15	Smith, John	2.39	$295.05
3	101	03/05/15	Jones, Mary	2.20	$271.59
4	...	...	...	...	...

We'll use D:A1..E1000 as the block of data, but you can easily modify that in the macros below.

In fact, instead of hard-coding this block into each macro command, you can place that block as text in some cell, and refer to it in macro commands by using the `@@` function. Thus, for instance, if you store *D:A1..E1000* in cell C:A100, then @@(C:A100) in each macro command would refer to it, and if you later need to expand the block, you simply change the contents of C:A100.

Many would name the block something like "Data" for the same purpose, but given QP's problems with block names noted in the introduction at page iv, I either hard-code the block or use the `@@` function.

Simple @VLOOKUP

With the sort of structure in this sample, it is easy to set up the interface sheet (A) so that when the user enters a number in a cell (A2), QP displays relevant information. Hence, in Table 18.2, typing 101 into cell A2 causes the functions in B2..E2 to display relevant information:

Table 18.2: Retrieving by index with @VLookup

A:	A	B	C	D	E
1	Invoice	Date	Customer	Quantity	Price
2	101	3/5/15	Jones, Mary	2.25	$234.56

B2= @VLOOKUP(A2,D:A1..E1000,1,0)
C2= @VLOOKUP(A2,D:A1..E1000,2,0)
D2= @VLOOKUP(A2,D:A1..E1000,3,0)
E2= @VLOOKUP(A2,D:A1..E1000,4,0)

Instead of `@VLOOKUP` functions in B2..E2, a macro could get the invoice number and place the data in the same cells,. The macro in Table 18.3 does that:

Table 18.3: Retrieving by index with a macro

C:	A	B
1		{GetNumber "Enter invoice number",A:A2}
2		{Let A:B2,@VLOOKUP(A:A2,D:A1..E1000,1,0)}
3		{Let A:C2,@VLOOKUP(A:A2,D:A1..E1000,2,0)}
4		{Let A:D2,@VLOOKUP(A:A2,D:A1..E1000,3,0)}
5		{Let A:E2,@VLOOKUP(A:A2,D:A1..E1000,4,0)}

Reporting the entire database with {BlockValues}

If all data in the database needs to be displayed in a report, the `{BlockValues}` command allows the programmer to do so in a single command. In our sample database, the function `@OFFSET(D:A1,0,0,@COUNT(D:A1..A1000),4)` defines the entire non-blank database. Therefore, this command transfers that block as a whole to the report page:

```
{BlockValues @OFFSET(D:A1,0,0,@COUNT(D:A1..A1000),
4),B:A1}
```

This method will not be appropriate if you need to be selective about the data to be reported.

Building reports with {For} loops

To include some rows of data into the report and exclude others, the macro will need to examine rows of data individually. This section looks at how to do so. It starts with macros for transferring all data in the database. It then shows how to blend in tests for either including a row in the report or excluding it. It concludes with a summary of the structural features of report-building macros.

Transfer of all data by row using a single {For} loop.

In the simplest example, let's say that we want to transfer all of the data from sheet D to make a report on sheet B. We'll say that sheet B is blank. (If not, the first command might be `{Blank B:A1..E1000}`.) The `{For}` loop in Table 18.4 does the job, but with a less than optimal result.

Table 18.4: Transferring data by row with a single {For} loop, starting offset of 0

C:	A	B
1	D-Row	{For C:A1,0,@ROWS(D:A1..E1000)-1,1,C:B3}
2		
3		{Put B:A1..E1000,0,C:A1,@INDEX(D:A1..E1000,0,C:A1)}
4		{Put B:A1..E1000,1,C:A1,@INDEX(D:A1..E1000,1,C:A1)}
5		{Put B:A1..E1000,2,C:A1,@INDEX(D:A1..E1000,2,C:A1)}
6		{Put B:A1..E1000,3,C:A1,@INDEX(D:A1..E1000,3,C:A1)}
7		{Put B:A1..E1000,4,C:A1,@INDEX(D:A1..E1000,4,C:A1)}

Note that B3..B7 could be replaced by a single command the transfers the entire block:
{BlockValues @OFFSET(D:A1,C:A1,0,1,5),@OFFSET(B:A1,C:A1,0)}

In the `{For}` command in B1, note that the initial counter is 0, because the rows recognized by both `{Put}` and `@INDEX` start with an offset of 0. That counter is stored in cell A1. Note that the final counter is the number of rows in the database less 1. That is because the numbering of rows starts with 1, not 0.

If, however, you prefer to have the row counter in A1 match the rows on the screen, for aesthetic or other reasons, it is easy to modify the macro accordingly, as in Table 18.5.

Table 18.5: Same, with starting row of 1

C:	A	B
1	D-Row	{For C:A1,1,@ROWS(D:A1..E1000),1,C:B3}
2		
3		{Put B:A1..E1000,0,C:A1-1,@INDEX(D:A1..E1000,0,C:A1-1)}
4		{Put B:A1..E1000,1,C:A1-1,@INDEX(D:A1..E1000,1,C:A1-1)}
5		{Put B:A1..E1000,2,C:A1-1,@INDEX(D:A1..E1000,2,C:A1-1)}
6		{Put B:A1..E1000,3,C:A1-1,@INDEX(D:A1..E1000,3,C:A1-1)}
7		{Put B:A1..E1000,4,C:A1-1,@INDEX(D:A1..E1000,4,C:A1-1)}

Again, B3..B7 could be replaced by:
{BlockValues @OFFSET(D:A1,C:A1-1,0,1,5),@OFFSET(B:A1,C:A1-1,0)}

In writing the macro, the programmer must be alert at all times to the distinction between numbering systems that start with 0 and those that start with 1.

Finally, you could replace *all* of the commands with those that use only the screen coordinates beginning at 1. To do so here,

- Replace `@INDEX` with the combination of `@@` and `@ADDRESS`, which uses the row and column headers that start with 1. Thus the `@INDEX` function in B3 can be replaced by `@@("D"&@ADDRESS(C:A1,C:A2))`, which returns the same content.
- Replace `{Put}` with any of several cell content commands (page 150) that use screen coordinates, such as `{Let}`.

If you use this approach, B3 would contain this command:

```
{Let ("B:"&@ADDRESS(A1,1)),
@@("D:"&@ADDRESS(A1,1))}
```

In my testing, using the 0 offset with `{Put}` and `@INDEX` is a hair faster than methods that use the screen coordinates. And once you get the knack of it, easier to program. But there is something to be said for consistently using only the screen grid's coordinate system. And although the choice between these two macros may be purely aesthetic, one could be preferable based on whether all or most of the other commands use an offset of 0 or use column and row numbers with an offset of 1.

Note that there is a gap between B1 and B3. If the commands in B3 started in B2, they would execute again after the `{For}` loop ended, which is probably not intended.

Transfer of all data by row and cell, using a double {For} loop.

A variation on this same macro reduces the number of lines of code by creating a double `{For}` loop. As shown in Table 18.6, the first in B1 cycles through the rows, as in the last example, but it then calls the second `{For}` loop at C2, which cycles by column through each cell on the row and stores the column in cell A2. It then calls a single `{Put}` command which takes the row and column coordinates from A1 and A2 to place data on the B sheet in parallel with the D sheet.

Table 18.6: Transferring data by row and cell with double for loop

C:	A	B	C
1	D-Row	{For C:A1,0,@ROWS(D:A1..E1000)-1,1,C:C2}	
2	D-Col		{For C:A2,0,@COLS(D:A1..E1000)-1,1,C:B3}
3		{Put B:A1..E1000,C:A2,C:A1,@INDEX(D:A1..E1000,C:A2,C:A1)}	

Note: C2 and B3 could be replaced in C2 by:
{BlockValues @OFFSET(D:A1,C:A1,0,1,5),@OFFSET(B:A1,C:A1,0)}

And as before, if you prefer to keep row and column counters in A1..A2 equal to the rows and columns of the affected cells, the changes in Table 18.7 would occur.

Table 18.7: Same, with a starting row of 1

C:	A	B	C
1	D-Row	{For C:A1,1,@ROWS(D:A1..E1000),1,C:C2}	
2	D-Col		{For C:A2,1,@COLS(D:A1..E1000),1,C:B3}
3		{Put B:A1..E1000,C:A2-1,C:A1-1,@INDEX(D:A1..E1000,C:A2-1,C:A1-1)}	

Note: C2 and B3 could be replaced in C2 by:
{BlockValues @OFFSET(D:A1,C:A1-1,0,1,5),@OFFSET(B:A1,C:A1-1,0)}
Or in B3: `{Let ("B:"&@ADDRESS(C:A1,C:A2)),`
`@@("D:"&@ADDRESS(C:A1,C:A2))}`

Terminating the Loop

One major problem with the macros so far has to do with blank rows. The macros above process all 1000 rows even if there are only twenty rows of records in the database. This problem is exacerbated by the fact that `@INDEX` returns 0 for every blank cell in each of those blank rows.

One way to deal with the problem of needless rows of data is to substitute a function that counts the number of rows filled in the A column, namely `@COUNT(D:A1..A1000)`, for the function that counts the rows, as shown in Table 18.8. This works as long as the database is well-formed, with no gaps in the index column and no stray data below.

Table 18.8: Setting the maximum row with @COUNT

C:	A	B	C
1	D-Row	{For C:A1,1,@COUNT(D:A1..A1000),1,C:C2}	
2	D-Col		{For C:A2,1,@COLS(D:A1..E1000),1,C:B3}
3		{Put B:A1..E1000,C:A2-1,C:A1-1,@INDEX(D:A1..E1000,C:A2-1,C:A1-1)}	

Note: C2 and B3 could be replaced in C2 by:
{BlockValues @OFFSET(D:A1,C:A1-1,0,1,5),@OFFSET(B:A1,C:A1-1,0)}

Another way to deal with the problem of needless rows is to add a command that tests for whether the first cell in the row of data is blank or not, and if it is blank, to end the loop with a `{Forbreak}` command, `{Branch}` to a terminal series of commands elsewhere, otherwise terminate the loop, or simply to ignore that line with a `{Return}` command. Thus, the command inserted into C2 in Table 18.9 would stop the `{For}` loop, as long as no zeros are in the index column on sheet D.

Table 18.9: Setting the limit by an {If} test and {Forbreak}

C:	A	B	C
1	D-Row	{For C:A1,1,@ROWS(D:A1..E1000),1,C:C2}	
2	D-Col		{If @INDEX(D:A1..E1000,0,C:A1-1)=""}{Forbreak}
3			{For C:A2,1,@COLS(D:A1..E1000),1,C:B4}
4		{Put B:A1..E1000,C:A2-1,C:A1-1,@INDEX(D:A1..E1000,C:A2-1,C:A1-1)}	

Note: C3 and B4 could be replaced in C2 by:
{BlockValues @OFFSET(D:A1,C:A1-1,0,1,5),@OFFSET(B:A1,C:A1-1,0)}

However, as noted in the discussion about testing for whether a cell is blank (page 26), the `{If}` test in C2 would be evaluated as true if a zero is in the cell tested. If zeros could be in the index column, a more accurate test would be:

```
{If @COUNTBLANK(@@(@OFFSET(D:A1,C:A1-1,0)))}
{Forbreak}
```

The `@OFFSET` function returns the address of the cell to be tested as text, which the `@@` function converts to coordinates, which the `@COUNTBLANK` function tests, and it returns 1 if the cell is blank or 0 otherwise. If the function returns 1, the `{If}` test evaluates as true, and the macro executes the `{Forbreak}` command.

Transferring rows by exclusion or inclusion

Most reports are intended to display only a subset of the entire database. That means that as the macro cycles through each row, it tests whether some condition is true or not. The structure of the macro determines whether the default rule is to include the row unless the condition excludes it, or to exclude the row unless the condition includes it.

To set a default rule of **inclusion** unless the `{If}` test excludes the row, follow the `{If}` test with the `{Return}` command on the same line. If the row satisfies the test for exclusion, the `{Return}` command causes the `{For}` command to go on to the next row. If it does not satisfy the test for exclusion, commands under the `{If}` test execute, and in Table 18.10, the row is transferred to the report sheet.

Table 18.10: Excluding rows with invoice number less than 200, with {If} and {Return}

C:	A	B	C
1	D-Row	{For C:A1,1,@ROWS(D:A1..E1000),1,C:C2}	
2	D-Col		{If @INDEX(D:A1..E1000,0,C:A1-1)<120}{Return}
3			{For C:A2,1,@COLS(D:A1..E1000),1,C:B4}
4		{Put B:A1..E1000,C:A2-1,C:A1-1,@INDEX(D:A1..E1000,C:A2-1,C:A1-1)}	

Note: C3 and B4 could be replaced in C2 by:
{BlockValues @OFFSET(D:A1,C:A1-1,0,1,5),@OFFSET(B:A1,C:A1-1,0)}

If you impose multiple tests for each row, it would make logical sense to start with the one most likely to exclude the row.

To set a default rule of **exclusion** unless the `{If}` test includes the row, follow the `{If}` test with a `{Branch}` command that points to commands at some other location, and no commands are under the `{If}` test. If the row satisfies the test for inclusion, the macro executes command at the other location, and in Table 18.11, the row is transferred to the report sheet. If the row does not satisfy that test, there are no other commands to execute, and the `{For}` command goes to the next row.

Table 18.11: Including rows with invoice number greater than 199, with {If} and {Branch}

C:	A	B	C	D
1	D-Row	{For C:A1,1,@ROWS(D:A1..E1000),1,C:C2}		
2	D-Col		{If @INDEX(D:A1..E1000,0,C:A1)>119}{Branch D3}	
3				{For C:A2,1,@COLS(D:A1..E1000),1,C:B4}
4		{Put B:A1..E1000,C:A2-1,C:A1-1,@INDEX(D:A1..E1000,C:A2-1,C:A1-1)}		

Note: D3 and B4 could be replaced in C2 by:
{BlockValues @OFFSET(D:A1,C:A1-1,0,1,5),@OFFSET(B:A1,C:A1-1,0)}

It should be understood that, instead of writing numbers like 200 or 199 into the macro, it will usually be more useful to place those numbers as variables into another cell, say A3, and write the macro to compare this row of data with that cell.

Orderly writing to the Report Page

Transferring fewer than all rows introduces a complication. If we made no other changes, the report would have rows that include some data, but would have gaps where rows were excluded. We don't want that, so we need commands that will put each non-excluded row of data onto the next blank row of the report. We can do this by using the `@COUNT` function on the report, which we will do immediately after the row has passed all of the tests for excluding the row. It would look something like Table 18.12.

Table 18.12: Excluding rows, but setting the correct row on report

C:	A	B	C
1	D-Row	{For C:A1,1,@ROWS(D:A1..E1000),1,C:C2}	
2	D-Col		{If @INDEX(D:A1..E1000,0,C:A1)<200}{Return}
3	B-Row		{Let C:A3,@COUNT(B:A1..A1000)}
4			{For C:A2,1,@COLS(D:A1..E1000),1,C:B5}
5		{Put B:A1..E1000,C:A2-1,C:A3,@INDEX(D:A1..E1000,C:A2-1,C:A1-1)}	

Note: C4 and B5 could be replaced in C2 by:
{BlockValues @OFFSET(D:A1,C:A1-1,0,1,5),@OFFSET(B:A1,C:A3-1,0)}

Table 18.13: Including rows, but setting the correct row on the report

C:	A	B	C	D
1	D-Row	{For C:A1,1,@ROWS(D:A1..E1000),1,C:C2}		
2	D-Col		{If @INDEX(D:A1..E1000,0,C:A1)>199}{Branch D3}	
3	B-Row			{Let C:A3,@COUNT(B:A1..A1000)}
4				{For C:A2,1,@COLS(D:A1..E1000),1,C:B5}
5		{Put B:A1..E1000,C:A2-1,C:A3,@INDEX(D:A1..E1000,C:A2-1,C:A1-1)}		

Note: D4 and B5 could be replaced in C2 by:
{BlockValues @OFFSET(D:A1,C:A1-1,0,1,5),@OFFSET(B:A1,C:A3-1,0)}

For macros that include rows, it would look something like Table 18.13.

A further complication is that the header row of the data is likely to be excluded by most tests, so if we want it in the report, we ought to be sure it is there in advance (or as the first commands of the macro).

Summary

The net result is that the macro needs to be structured in this sequence:

1. Preset the Report sheet to include appropriate headers;
2. Create a `{For}` loop to cycle through the rows on the Data sheet;
3. Devise `{If}` tests to terminate the loop with `{Forbreak}` (if not already done in setting the parameters of the `{For}` command);
4. Devise `{If}` tests to exclude rows on the Data sheet with `{Return}`, or to include them with a `{Branch}` command;
5. Determine the next blank row on the Report sheet;
6. (Optional:) Create a `{For}` loop to cycle through the cells on the current row of the Data sheet by column;
7. Transfer data from the row on the Data sheet to the blank row on the Report sheet by one of the commands for placing content in cells (see the chapter starting at page 150) that allows copying data without navigating to the source: `{Let}`, `{PutBlock}`, `{Put}`, `{BlockValues}`, `{BlockCopy}`, or `{SetObjectProperty}`.

Annotated Example: Cell-by-Cell

To recapitulate this discussion, Table 18.14 a model macro to run a report that includes only those entries on the Data sheet in which the quantity in the D column

Table 18.14: Recapitulation: Reporting quantities less than 2.25, cell-by-cell

C:	A	B	C
1	D-Row	{For C:A1,2,@COUNT(D:A1..A1000),1,C:C2}	
2	D-Col		{If @@("D:"&(@ADDRESS(C:A1,4)))<2.25}{Return}
3	B-Row		{Let C:A3,@COUNT(B:A1..A1000)+1}
4			{For C:A2,1,@COLS(D:A1..E1000),1,B5}
5		{Let ("B:"&@ADDRESS(C:A3,C:A2)),@@("D:"&@ADDRESS(C:A1,C:A2))}	

Note: C4 and B5 could be replaced by:
{BlockValues ("D:"&@ADDRESS(C:A1,1)&".."& @ADDRESS(C:A1,5)),
("B:"&@ADDRESS(C:A3,1))}

is greater than or equal to 2.25. All functions and commands use the screen grid of coordinates.

In B1, the `{For}` command stores the current row of data in C:A1. That row counter begins at row 2 (which excludes the header row, since that was already placed on the Report page). The `{For}` loop cycles through the rows of D:A1..E1000 until it reaches the last row with data (instead of cycling until row 1000). The last row is determined by the `@COUNT` function, though we could have used an `{If}` test and the `{Forbreak}` command instead. For each row, this `{For}` command calls the macro at C2.

In C2, the `@ADDRESS` function gets the address of the cell containing the quantity, column D (or 4) on the current row, and the `@@` function returns the value in that cell. Using the default rule of inclusion, if that quantity is less than 2.25, the `{Return}` executes and this row of data is skipped. Otherwise, the macro proceeds to C3.

In C3, the macro stores in cell C:A3 the next blank row on the Report page using the `@COUNT` command again. If 20 rows of data have already been filled, the next blank row is 20+1. Since `@COUNT` would return 20, we add +1.

In C4, the macro starts a second `{For}` loop, which cycles through the cells on the current row of data by column number, which column number is stored in C:A2. For each cell, the macro calls the command at B5.

In B5, the macro uses `{Let}` to locate the destination cell on the report page by column (A2) and row (A3), and to place in that cell the data returned by the `@@` function.

Performance test results

I tested all major variations of the commands for placing content in cells for transferring 20 rows of data from the D to the B sheets, both on the cell-by-cell basis shown above, and in the case of `{BlockValues}` and `{BlockCopy}`, on the basis of copying an entire row of data at a time. In optimal conditions, the cell-by-cell commands transferred all the data in a range between 0.09 to 0.17 of a second, and the commands that transferred an entire row of 5 cells at one time transferred the 20 rows in between 0.05 and 0.06 of a second. At first, testing resulted in some large discrepancies, but when I controlled for them by either closing other notebooks or changing recalculation settings to manual at the beginning of each test, the results had a more uniform and acceptable range. If your report is slow, see the discussion of speed and performance at page 113.

The commands used on a cell-by-cell basis, beginning with the fastest, were:

1. `{Put B:A1..E1000,A2-1,A1-1,`
   `@@("D:"&@ADDRESS(A1,A2))}`
2. `{Let ("B:"&@ADDRESS(A1,A2)),`
   `@@("D:"&@ADDRESS(A1,A2))}`
3. `{SetObjectProperty ("B:"&@ADDRESS(A1,A2)`
   `&".Value"),("D:"&@ADDRESS(A1,A2)&".Value")}`
4. `{PutBlock @@("D:"&@ADDRESS(A1,A2)),`
   `("B:"&@ADDRESS(A1,A2))}`
5. `{BlockCopy ("D:"&@ADDRESS(A1,A2)),`
   `("B:"&@ADDRESS(A1,A2)),1,0,1,0,0,0,0,0}`
6. `{PutBlock2 @@("D:"&@ADDRESS(A1,A2)),`
   `("B:"&@ADDRESS(A1,A2))}`
7. `{BlockCopy ("D:"&@ADDRESS(A1,A2)),`
   `("B:"&@ADDRESS(A1,A2))}`

The commands that transferred an entire row, in the order of speed, were:

1. `{BlockValues ("D:"&@ADDRESS(A1,1)&".."`
   `&@ADDRESS(A1,5)),("B:"&@ADDRESS(A1,1))}`
2. `{BlockCopy ("D:"&@ADDRESS(A1,1)&".."`
   `&@ADDRESS(A1,5)),("B:"&@ADDRESS(A1,1)),1,0,1,0,`
   `0,0,0,0}`
3. `{BlockCopy ("D:"&@ADDRESS(A1,1)&".."`
   `&@ADDRESS(A1,5)),("B:"&@ADDRESS(A1,1))}`

In each of these examples, if a row is set in A3 for placing data on sheet B, A3 should replace A1 in reference to placing data on sheet B.

Constructing the macros and functions

I found it somewhat difficult to keep track of when and how to alter variables with *+1* or *-1* when going back and forth between systems that start number rows and columns with 0 or 1, and with functions and macro commands that do likewise. Therefore, I compiled the chart in Table 18.15 based on the example used above. For each task, it shows the relevant commands functions in two categories, based on whether we are using a starting point of 0,0 or 1,1.

How to transpose data from the same row on different pages to the same page

In WPU 38483, a user asked how to take data that occupies the same three cells on a single row on three consecutive pages and transpose it onto a single page of 3 rows. There appears to be no generic command, so Table 18.16 shows one way to do it, using the techniques discussed above.[1]

The user types the 3D block to be searched in cell `A1`.

The starting position for the output goes into cell `A2`. It is important to express this as a block (`A10..A10`) rather than just `A10` for the functions to work.

[1]My initial answer in the forum worked for the data I had located in A:A1..C:C1. But that worked only because the data started on the first page and first column. I've varied the initial answer so that it works no matter where the data starts.

Table 18.15: Format of database report functions for both origins

To get/do this	Origin	Commands/functions consistent with the starting point
For loop in data	0,0	{For C:A1,0,@COUNT(D:A1...A1000)-1, ...
	1,1	{For C:A1,1,@COUNT(D:A1...A1000), ...
Last row of data	0,0	@COUNT(D:A1..A1000)-1
	1,1	@COUNT(D:A1..A1000)
Current data row	0,0	@OFFSET(D:A1,C:A1,0,1,5) "D:"&@ADDRESS(C:A1+1,1) &".."&@ADDRESS(C:A1+1,5)
	1,1	@OFFSET(D:A1,C:A1-1,0,1,5) "D:"&@ADDRESS(C:A1,1) &".."&@ADDRESS(C:A1,5)
Current data cell	0,0	@OFFSET(D:A1,C:A1,C:A2) "D:"&@ADDRESS(C:A1+1,C:A2+1)
	1,1	@OFFSET(D:A1,C:A1-1,C:A2-1) "D:"&@ADDRESS(C:A1,C:A2)
Contents of current data cell	0,0	@INDEX(D:A1..E1000,C:A2,C:A1) @@("D:"&@ADDRESS(A1+1,A2+1))
	1,1	@INDEX(D:A1..E1000,C:A2-1,C:A1-1) @@("D:"&@ADDRESS(C:A1,C:A2))
For loop in cells	0,0	{For C:A2,0,4 ...
	1,1	{For C:A2,1,5 ...
Row in Report	0,0	@COUNT(B:A1..A1000)
	1,1	@COUNT(B:A1..A1000)+1
Target in Report	0,0	@OFFSET(B:A1,C:A3,0) "B:"&@ADDRESS(C:A3+1,1)
	1,1	@OFFSET(B:A1,C:A3-1,0) "B:"&@ADDRESS(C:A3,1)
Current Report Cell	0,0	@OFFSET(B:A1,C:A3,C:A2) "B:"&@ADDRESS(A3+1,A2+1) {Put B:A1..E1000,C:A2,C:A3, ...
	1,1	@OFFSET(B:A1,C:A3-1,C:A2-1) "B:"&@ADDRESS(A3,A2) {Put B:A1..E1000,C:A2-1,C:A3-1, ...

The command in `B1` sets the starting sheet number for the data to be retrieved, using `@Cell` and places it into cell `A3`. The command in `B2` sets the starting column number and places it into cell `A4`.

The `{For}` command in `B3` loops through the pages, starting from the first sheet set in `A3` and running to the last sheet, which is calculated as the starting sheet number, plus the number of sheets in the block, less one. For each sheet, it invokes the loop in `B5`.

The `{For}` command in `B5` loops through each column, starting from the first column set in `A4` and running to the last column, which is calculated as the starting column number, plus the number of columns in the block, less one. For each column, it invokes the command in `B7`.

The command in `B7` uses the `{Let}` command to take data retrieved by the `@Index` function, which allows us to get data from a multi-page block, and place

Table 18.16: Transposing data from consecutive pages onto consecutive rows

	A	B
1	C:C4..H:F4	{Let A3,@Cell("sheet",@@(A1))}
2	A:A10..A10	{Let A4,@Cell("col",@@(A1))}
3		{For A5,A3,A3+@Sheets(@@(A1))-1,1,B5}
4		
5		{For A6,A4,A4+@Cols(@@(A1))-1,1,B7}
6		
7		{Let @Offset(@@(A2),A5-A3,A6-A4),@Index(@@(A1),A6-A4,0,A5-A3)}

it in a cell determined by the `@Offset` function. Both functions use the same offset numbers.

Building reports with @MATCH

The preceding section dealt with using a `{For}` loop to cycle through every row of the database to determine whether the row should be written to the report page. Though fast enough for most purposes, it is possible to make an even faster loop using `@MATCH` to skip immediately from one matching row to the next.

How much faster? I tested both methods for a particular quantity on a Data sheet that contained 979 rows of data, in which there were 53 matches to be transferred to the report sheet. The `{For}` loop produced the results in 1.36 seconds. The `@MATCH` technique here produced the same results in about one-third of the time, 0.42 seconds.

The basic concept is this. `@MATCH` is given a value and then finds the offset of the first matching value in a column of values. If there is no match, it returns `ERR`, so that is the sign that it is time to end the loop. If there is a match, the row on the data sheet is equal to the first row in the column we search plus the offset number that `@MATCH` returns. That allows us to transfer the row to the report sheet by now-familiar means. After the transfer, we have to reset the starting row for the next search to the next row in the column of data. Then we branch back to perform the search again, and this continues until `@MATCH` returns an `ERR`. Table 18.17 showswhat it looks like.

Table 18.17: Finding all rows with a Quantity of 4.32 in column D, using @MATCH

C:	A	B	C
1	D-Start	{Let C:A1,1}	
2	D-Row	{If @ISERR(@MATCH(4.32,@@("D:"&@ADDRESS(C:A1,4)&"..D1000"),0))}{Return}	
3	D-Col	{Let C:A2,C:A1+@MATCH(4.32,@@("D:"&@ADDRESS(C:A1,4)& "..D1000"),0)}	
4	B-Row	{Let C:A4,@COUNT(B:A1..A1000)+1}	
5		{For C:A3,1,@COLS(D:A1..E1000),1,C:C6}	
6		{Let C:A1,C:A2+1}	{Let ("B:"&@ADDRESS(C:A4,C:A3)),@@("D:"&@ADDRESS(C:A2,C:A3))}
7		{Branch C:B2}	

In B1, the macro sets the beginning data row at 1, which is stored in A1. The starting row number will be updated each time a match is found. The starting number is critical in formulating the @MATCH function, which here looks for the number 4.32 in the D column on the data sheet:

```
@MATCH(4.32,@@("D:"&@ADDRESS(C:A1,4)&"..D1000"),0)
```

The `"D:"&@ADDRESS(C:A1,4)&"..D1000"` component evaluates as *D:D1..D1000*, and the `@@` wrapper converts that text into coordinates to be searched by `@MATCH`. (As in other cases, 4.32 would more likely be placed in a different cell, say A5 here, and that cell would be written into the function.)

In B2, the loop begins. The first command is an `{If}` test, which tests if there is a match in the range. If not, the `@MATCH` function returns `ERR`, and the `{Return}` (or `{Quit}`) command suitably stops the loop. If a match is found, the macro proceeds to B3

In B3, the macro determines the row number of data on which the match occurs by adding the offset number that `@MATCH` returns to the starting number in A1, and it stores that row number in A2.

In B4, the macro determines the next blank row on the report page and stores it in A4.

In B5, the macro runs a `{For}` loop like those in the examples of the last section, which loops through each cell by column in the row of data and calls the macro in C6, which simply writes it to the next blank row of the report sheet.

In B6, the macro resets the starting range for the next `@MATCH` search as the row below the one on which a match was found.

In B7, the macro loops back to B2, and the process repeats until `@MATCH` returns an `ERR`.

The `@MATCH` technique works only if you can search for an exact match in the relevant column of data, so more complex testing must be done either on a row-by-row basis or by creating a helper column of data that can be searched by an exact-match method.

The above technique constructed the data block to search (C3) using the `@ADDRESS` function. `@OFFSET` can be used as well. After setting the initial search block (here, D:D1..D1000) and setting the initial offset to 0, the search block can be recalculated after each search by recalculating the offset. If the offset is stored in C:A5 and the number of rows in C:A6 (here, 1000), this formula will always return the searchable block: `@OFFSET(D:D1,C:A5,0,C:A6-C:A5,1)`. Using `@MATCH` on that range will either return `ERR`, in which case the loop terminates, or a number, which we can store in C:A7. The new offset to be put in C:A5 will be `C:A5+C:A7+1`. The searchable block formula will give the correct new searchable block, and loop can continue until an `ERR` occurs.

Notebook queries

QP has various ways of making reports by other means, chief of which is the Notebook Query, available under Tools > Data Tools > Notebook Query .... This technique requires three tables: (1) the Database itself; (2) a Criteria Table, which contains formulas that QP uses to select records from the Database; and (3) an Output Table, which contains the headers of the Database table that should be displayed in the report. For a Database like the example used here in Table 18.18, I would typically have a separate page (Here E:) with my criteria table on the top rows and the Output Table a few rows below it. It might look like this:

In the dialog box, one would manually select the Database Cells (here

Table 18.18: Sample Criteria Table in Green, Output Table in Orange

E:	A	B	C	D	E
1	Invoice	Date	Customer	Quantity	Price
2				4.32	
3					
4					
5	Invoice	Date	Customer	Quantity	Price
6					

D:A1..E1000), the Criteria Table (here E:A1..E2), and the Output Cells (here, E:A5..E5) and press the `[Extract]` button. Entering 4.32 into cell E:D2, the query would place every record with 4.32 in the Quantity column into row 6 and below. A macro can automate that process, as shown in Table 18.19.

Table 18.19: Standard search macro

	A
1	{Query.Reset}
2	{Query.Database_Block "D:A1..E1000"}
3	{Query.Criteria_Table "E:A1..E2"}
4	{Query.Output_Block "E:A5..E5"}
5	{Query.Extract}

Criteria can be combined in various ways, including more ways than the Help file might indicate. The help file suggests that one searches for a combination of items by placing terms capable of exact matches in E:A2..E5, and that to search for one of two terms in the same category, you need to put both terms in two rows under the correct category (e.g., E:D2..D3) and expand the Criteria Table. Though those concepts work, they are neither necessary nor desirable. Instead, a placement like 4.32 in E:D2 above is really a shorthand for *+D:D2=4.32*, which QP tests for successive rows (testing first D:D2, then D:D3, etc.).

The real rule is that a row is included in the output if all formulas in E:A2..E2 evaluate as TRUE (and blank cells evaluate as true); otherwise it is excluded. Therefore, you can combine any number of tests into any of the cells in E:A2..E2, whether they relate to the category in E:A1..E1 above or not. Hence, the formula you could put a complex formula into E:A2 that has nothing to do with invoice numbers, a formula like this:

```
+D:D2<4.32#AND#D:B2>=$A:$A$4
```

which looks for all rows with quantities less than 4.32 and dates greater than the date in the fixed cell A:A4. (As QP evaluates each successive row, any reference to a cell is adjusted to the cell below it, unless the cell is given an absolute address.) A macro can begin by placing such search terms into a criteria table and then getting the results accordingly.

In automating the use of notebook queries, the programmer must remember that not all macro commands can be used to put a *formula* into the criteria block. Many would convert a formula into a value. Therefore, the programmer should use a command that will leave the formula intact. As shown in Chapter 16, those commands are `{PutCell2}` and `{PutBlock2}`.

Bug. Unfortunately, there is a flaw in the system that occurs when different queries have output in the same columns, whether those columns are on the same sheet, different sheets, or even different notebooks. See discussions at OC 16049, WPU 36717, and linked pages. The flaw is that, sometimes, data placed under one field in the output block comes from the wrong field in the database. This bug can be worked around by setting the output range for different queries to different columns, or by using an alternative to notebook queries. Once the error has occurred, the only way to fix it is to close QP entirely and restart it. For alternatives to notebook queries, see the rest of this chapter.

How to formulate search terms for text in the database

In this connection, it is useful to review the logic for finding a text match within the database using the `@FIND` command. `@FIND` looks for a particular string of text in a cell, and it returns a number in the case a match is found, but an `ERR` if no match is found. Therefore, to look for the name "Smith" in column C of the sample database, the function,

```
@ISERR(@FIND("Smith",C2,0))
```

will return 1 (TRUE) if the match is not found; 0 (FALSE) if the match is found. Thus:

```
@NOT(@ISERR(@FIND("Smith",C2,0)))
```

returns 1 if a match is found, 0 if not found. On the basis of this, we can generalize:

Mandatory terms

To get all rows with "Name1" in column X:

```
@NOT(@ISERR(@FIND("Name1",X2,0)))
```

To get all rows with "Name1" in column X and "Name2" in column Y:

```
@NOT(@ISERR(@FIND("Name1",X2,0)))
#AND#@NOT(@ISERR(@FIND(")Name2",Y2,0)))
```

To get all rows with either "Name1" in column X or "Name2" in column Y:

```
@NOT(@ISERR(@FIND("Name1",X2,0)))
#OR#@NOT(@ISERR(@FIND("Name2",Y2,0)))
```

Excluded terms

To get all rows except those with "Name1" in column X:

```
@ISERR(@FIND("Name1",X2,0))
```

To get all rows except those with either "Name1" in column X or "Name2" in column Y:

```
@ISERR(@FIND("Name1",X2,0))
#AND#@ISERR(@FIND("Name2",Y2,0))
```

To get all rows except those with both “Name1” in column X and (if) “Name2” in column Y:

```
@ISERR(@FIND("Name1",X2,0))
#OR#@ISERR(@FIND("Name2",Y2,0))
```

Mixture

To get all rows with “Name1” in column X but (if) not “Name2” in column Y:

```
@NOT(@ISERR(@FIND("Name1",X2,0)))
#AND#@ISERR(@FIND("Name2",Y2,0))
```

To get all rows except those with “Name1” in column X unless “Name2” is in column Y:

```
@ISERR(@FIND("Name1",X2,0))
#OR#@NOT(@ISERR(@FIND("Name2",Y2,0)))
```

How to replicate a simple notebook query with a For loop

The macro commands for doing what a *simple* Notebook Query does are surprisingly straightforward, as shown in Table 18.20. A simple query does not use formulas in the criteria block; it simply places values that the data either match or do not. This macro-based technique is slower than QP’s built-in method, so it is less suitable for large databases, but for small ones (say, less than 100 rows of data), it is a candidate.

Table 18.20: Macro alternative to simple Database Queries

C:	A	B
1	D:A1..E10000	{; initializing, including blanking A4..A7, setting blocks in A1..A3}
2	E:A1..E2	{Let A4,A4+1}
3	E:A5..E5	{If @INDEX(@@(A1),0,A4)=""}{Quit}
4	DRow	{If A4>=@ROWS(@@(A1))}{Quit}
5	CCol	{Let A5,0}
6	ORow	{If @CELL("type",@@(@OFFSET(@@(A2),1,A5)))="b"}{Branch B9}

7	OCol	{If @CELL("type",@@(@OFFSET(@@(A2),1,A5)))="l"}
		{IF @ISERR(@FIND(@INDEX(@@(A2),A5,1),
		@HLOOKUP(@INDEX(@@(A2),A5,0),@@(A1),A4,1),0))}{Branch B2}

8		{If @CELL("type",@@(@OFFSET(@@(A2),1,A5)))="v"}
		{IF @INDEX(@@(A2),A5,1)<>
		@HLOOKUP(@INDEX(@@(A2),A5,0),@@(A1),A4,1)}{Branch B2}

9		{If A5<@COLS(@@(A2)-1)}{Let A5,A5+1}{Branch B6}
10		{Let A6,@COUNT(@@(@OFFSET(@@(A3),0,0,10000,1)))}
11		{Let A7,0}

12		{If @HLOOKUP(@INDEX(@@(A3),A7,0),@@(A1),A4,0)<>""}
		{Let @OFFSET(@@(A3),A6,A7),
		@HLOOKUP(@INDEX(@@(A3),A7,0),@@(A1),A4,0)}

13		{If A7<@COLS(@@(A3))-1}{Let A7,A7+1}{Branch B12}
14		{Branch B2}

The macro writer will need to initialize the macro by putting the database into A1, the criteria range into A2, the output range into A3, the three blocks required by QP's database query, and by blanking the cells A4..A7. These and other steps are implicit in B1.

B2 sets the row of the database to be tested, so when a row is fully tested, the macro loops back here. B3 and B4 test conditions for stopping the macro.

B5 resets the testing by starting with the leftmost cell of the criteria table; the macro will test those cells from left to right, and loop back here for the next row of data. B6 through B8 test whether there is a test in the criteria table. If not (B6), the macro proceeds to the next cell (B9). But if the cell contains text (B7) or a number (B8), the macro looks to see if the criterion text is contained in text in the relevant database cell or the number is equal to the number in the relevant database cell. If not, the macro ends testing of this row and goes to the next row of data (B2). If so, the macro repeats this loop for each column of the criteria table (B9), and if it reaches the last cell of the criteria table without failing a test, it proceeds to output.

Note that the test for text in B7 looks for text on a case-sensitive basis. If one wanted to search on a case-insensitive basis, this formula should be used after the if-test:

```
{IF @ISERR(@FIND(@UPPER(@INDEX(@@(A2),A5,1)),
@UPPER(@HLOOKUP(@INDEX(@@(A2),A5,0),
@@(A1),A4,1)),0))}
```

And if one sought only an exact match, this simpler formula should be used:

```
{IF @INDEX(@@(A2),A5,1)<>@HLOOKUP(@INDEX(@@(A2),
A5,0),@@(A1),A4,1)}
```

B10 sets the correct row for the output data, and B11 resets the column to begin writing to the output table. B12 actually writes the data. The core idea is to get the field name from the output table, and look for the value at the intersection of that field name in the database and the current database row. B12 begins with an if test to determine whether the value at that intersection is blank or not; if not, it prints it to the output, but if it is blank, the macro bypasses it. The reason for doing so is that the best function for obtaining that information would print a blank as a zero. B13 repeats the B12 process for each cell in the output table, and when the end is reached, it loops back to the beginning (B14) to test the next row of data.

How to replicate a notebook query with @Match and a helper column

Table 18.21 presents a more complex version of the same basic idea, but because it uses `@Match` to find matches rather than testing every row, it will often be quicker than a `{For}` loop. It uses a number of functions in column B to calculate a single block formula that will create an array of 1s (matches) and 0s (non-matches). (Due to the complexity of elements concatenated in B12 and B13, the elements are set off in alternating colors.) It then uses searches for each match and prints the corresponding row.

The normal three blocks are set in B1..B3. (Columns A and C, which are used for documentation, are omitted here for space reasons.) B4 calculates the number of rows in the database. The macro initializes in D1..D2, which turns off recalculation.

Table 18.21: Alternative to Query using @Match and Helper Column

	B	D
1	Data:A1..E1000	{Notebook.Recalc_Settings "Manual;Natural; 1;Yes;No"}{WindowsOff}
2	Query:A1..A2	{Blank H1..H10000}{Blank B15}{Let B5,0}
3	Query:A5..E5	{recalc B6..B13}
4	@Count(@@(@Offset(@@(B1), 0,0,@Rows(@@(B1)),1)))	{if B6="b"}{Branch D7}
5	Criteria counter	{if B6="l"}{Let B15,B15&B13}
6	@Cell("type", @@(@Offset(@@(B2),1,B5)))	{if B6="v"}{Let B15,B15&B12}
7	@Index(@@(B2),B5,0)	{If B5<@Cols(@@(B2)-1)}{Let B5,B5+1} {Branch D3}
8	@Match(B8,@@(B9),0)	{Putblock B15,H1}{BlockCopy H1,+B16}
9	@Offset(@@(B1),0,0,1, @Cols(@@(B1)))	{Recalc B16}
10	@Match(B8,@@(B9),0)	{Let B17,0}
11	@Offset(@@(B1),0,B10,B4,1)	{Recalc B18..B19}
12	@Concatenate(@If(B15<>"","#AND#",""), "(",B11,"=", @If(B6="b",@Char(34),""),B7, @If(B6="b",@Char(34),""),")")	{if @iserr(B19)}{Branch D20}
13	@Concatenate(@If(B15<>"","#AND#",""), "(@IsErr(@Find",@Char(34), B7,@Char(34),",",B11, ",0))=0)")	{Let B20,@Count (@@(@Offset(@@(B3),0,0,10000,1)))}
14		{Let B21,0}{Recalc B22}
15	Block formula for the helper column	{if @HLookup(B22,@@(B1), B17+B19,0)<>""} {Let @Offset(@@(B3),B20,B21), @HLookup(B22,@@(B1),B17+B19,0)}
16	@Offset(H1,0,0,B4,1)	{If B21<@Cols(@@(B3))-1} {Let B21,B21+1}{Recalc B22} {Branch D15}
17	Cumulative offset	{Let B17,B17+B19+1}
18	@Offset(H1,B17,0,B4-B17,1)	{Branch D11}
19	@Match(1,@@(B18),0)	
20	Output Row	{Notebook.Recalc_Settings "Background; Natural;1;Yes;No"}
21	Output Column	{Blank +B16}
22	@Index(@@(B3),B21,0)	

D3..D7 have the ultimate goal of building a single block formula to be stored in B15. It would be composed of elements created by formulas in B12 and B13, which are in turn given content by the formulas in B6..B11, which are recalculated for each cell in the criteria table by the loop in D3..D7, which ends when the macro operates on the last cell of the criteria table.

D8..D9 place the formula created in B15 into H1, and copy it in parallel with the database. It consists of an array of 1s and 0s, which can be searched with `@Match`.

D10..D18 creates the engine that searches for each match, and within this range, upon finding a match, D13..D16 prints the data from the matching row to the output block. The technique for printing that row is given in the preceding section, and the technique for searching for matches using `@Match` and `@Offset` was described in the section on building reports using `@Match`, above.

When ultimately no more matches can be found, B19 will return an ERR, and the macro proceeds to D20 to restore normal recalculation and to clear the helper column.

Chapter 19

Databases: Adding or Modifying Data

The basic methods of adding data to a database or modifying data in it were discussed in connection with commands relating to cell content (see Chapter starting at page 150) and methods of building reports starting at page 171.

How to add a row of data to the bottom of a database

Using the same database structure as in the prior chapter on retrieving data from a database (page 170), let us assume that we want to add this row of data (Table 19.1) from our interface sheet (A) to the bottom of the database.

Table 19.1: Sample data to be added to database on Data sheet

A:	A	B	C	D	E
5	Invoice	Date	Customer	Quantity	Price
6	1078	04/05/15	Doe, John	1.78	$219.74
7					

As before, the strategy in the simple macro in Table 19.2 is to find the row on the first blank row on the data sheet, and to add the data from A6..E6 there.

Table 19.2: Simple macro to add a row of data to the bottom of a database

	A	B		C
1	Data-Row	{Let A1,@Count(Data:A1..A1000)+1}		
2	A-Col	{For A2,1,@Cols(A:A6..E6),1,C3}		
3			{Let ("Data:"&@Address(A1,A2)),@@("A:"&@Address(6,A2))}	

To vary this slightly, lets say that we don't have the invoice number in A6, but simply want to add the data in A:B6..E6 with the next available invoice number. If so, preface this sequence of commands with:

```
{Let A:A6,@MAX(Data:A1..A1000)+1}
```

The programmer can automate the front-loading process further. One can easily imagine a routine like Table 19.3, which first fills out the data range in A:A6..E6. It gets the next invoice number automatially (B1), then gets the date of the transaction (B2..B4), the customer's name (B5), and the quantity (B6), and then calculates a price from the quantity (B7). After transferring this row to the bottom of the database on the D sheet (B8..B9 and B12), it then loops back (B10) to allow the user to add the next invoice. However, at that point, it gives the user the opportunity of ending the data entry session by typing x, at which time the macro stops (B3).

Table 19.3: Adding rows to bottom of database repetitiously

	A	B
1	Data-Row	{Let A:A6,@Max(Data:A1..A1000)+1}
2	A-Col	{GetLabel "Enter Date of Sale (MM/DD/YY) or x to cancel",A3}
3	Input	{If A3="x"}{Quit}
4		{Let A:B6,@DateValue(A3)}
5		{GetLabel "Enter Customer (LastName, FirstName)",A:C6}
6		{GetNumber "Enter Quantity",A:D6}
7		{Let A:E6,A:D6*123.45}
8		{Let A1,@Count(Data:A1..A1000)+1}
9		{For A2,1,@Cols(A:A6..E6),1,B12}
10		{Branch B1}
11		
12		{Let ("Data:"&@Address(A1,A2)),@@("A:"&@Address(6,A2))}

Generalizing, Table 19.4 shows how to add a row of data to a database by taking the coordinates out of the macro and placing them into cells.

The coordinates of the row of data to be added are placed in A1.

The coordinates of the database are in A2; be sure that it extends far enough down the sheet that it will include the data to be added.

B1 places the row of the database where the new data will be added into A3.

B2 runs a `{For}` loop that places data into each cell within the row, using the `{Put}` command in B4.

Table 19.4: A generalized macro to add data to the bottom of a database

	A	B
1	NewRow	{let A3,@COUNT(@@(@OFFSET(@@(A2),0,0,10000,1)))}
2	DBBlock	{for A4,0,@cols(@@(A1))-1,1,B4}
3	DBRow	
4	Col	{put +A2,A4,A3,@INDEX(@@(A1),A4,0)}

How to add data between the end of the data and totaling functions

Here we want to insert data between that last row of data and the totals immediately under it. It is not difficult to identify the row where the data should be placed and to insert a new blank row there. The only significant addition here is in B2, which inserts a blank row precisely where we say in B1 that it should be.

Table 19.5: Inserting a row of data between last row of data and totals beneath it

	A	B
1	Data Row	{Let A1,@Count(Data:A1..A1000)}
2	A-Col	{BlockInsert.Rows ("Data:"&@Address(A1,1));Entire}
3		{For A2,1,@Cols(A:A6..E6),1,B5}
4		
5		{Let ("Data:"&@Address(A1,A2)),@@("A:"&@Address(6,C:A2))}

Caution 1. The database must be set up properly. The problem is that most totals are created by a simple `@SUM` function that doesn't change the range that it sums if a row is inserted *where the function is*. The result would be that newly inserted data is not within the range, which means that the `@SUM` function does not reflect all the data it is intended to reflect.

One option is to have a blank row between the data and the totals, and the new row is always inserted where the blank row is. That would have the effect of inserting a row within the range of the `@SUM` function, which would adjust appropriately. The simplicity of this approach may compensate for its lack of aesthetic appeal.

My preference, however, is to replace the `@SUM` functions with those identified earlier that always return coordinates relative to the current cell (see discussion at 86). Thus, if there are say 300 rows of data, instead of placing `@SUM(Data:E1..E300)` into cell Data:E301, place this function there:

```
@SUM(@@(@OFFSET(Data:E1,0,0,@ROW-1,1)))
```

As long as columns are not inserted to the left of this row, this formula will always sum the numbers above it in that column.

The line where the insertion should occur is the line containing the `@SUM` formula. Finding it depends on whether there is an entry in the A column on that row (e.g., "Totals:"). If so, `@COUNT(Data:A1..A1000)` returns that row number; but if it is blank, we need `@COUNT(Data:A1..A1000)+1`. Let's assume here that "Totals:" is in that cell. Table 19.5 shows how to insert the row of data.

Caution 2. Whenever adding or deleting columns or rows, one must consider whether it has some effect on data outside the database but on the same sheet of the notebook. See the discussion of changing database structure at page 196.

How to add data with {Form}

A quick and easy way to add data to the bottom of a database involves using QP's Form (see Tools > Data Tools > Form ...). This can be run by a simple macro, and the macro can be run from a button. The only command that needs to be executed is `{Form` *`DatabaseBlock`*`}`, and `DatabaseBlock` can be easily specified by an `@OFFSET` function. In the sample we are using, the Database is `@OFFSET(Data:A1,0,0,@COUNT(Data:A1..A10000),5)`, so the method is simple:

1. Run the macro `{Form @OFFSET(Data:A1,0,0,` `@COUNT(Data:A1..A10000),5)}`.
2. Press the `[New]` button on the resulting dialog box.
3. Enter data in the five categories (the five columns).
4. Press `[New]` again.

The data is now at the bottom of the database and the dialog box is cleared, ready for entry of the next row of data (step 3). You can continue adding data until you press [Close].

The only downside is that no further automation is possible. The user will need to know what the invoice numbers should be and how to make any calculations correctly.

How to automate cursor movement during manual data entry

Much of data entry can be tedious, with repetitively pressing [Enter] and [Right] and then, at the end of the row, going back to the left and down a row. There are many ways to assist in automating this process. Table 19.6 works with the database in use here by navigating to the first blank on column A and pausing for data entry. Each time you press [Enter] after entering the data, the cursor moves to the right for the next entry, until it gets past column E (5), at which time it navigates back to the first blank in the A column. The user can stop it by pressing [Ctrl+Break] at any time.

Table 19.6: Macro assisting manual entry of data at bottom of database

	A
1	{SelectBlock ("Data:"&@Address(@COUNT(Data:A1..A1000)+1,1))}
2	{?}{Right}{If @Cellpointer("col")<6}{Branch c(0)r(0)}
3	{Branch c(0)r(-2)}

And if you simply want to navigate to the cell in the database in order to add a row manually, the first cell in that macro will do, as will @OFFSET alternative:

```
{SelectBlock @Offset(Data:A1,@Count(Data:A1..A1000),0)}
```

Same, with prompts, default responses, and error checking

This example presents a more complex example of the same principle. It automates addition of data to a database composed of three columns, but it adds prompts and default responses in each cell, and it then checks the entry for errors before moving to the next cell. I use this particular application to prepare indexes for Bible references from books that lack indexes. The data are on sheet A. Column A on that sheet houses the book of the Bible, B contains the verses, and C contains the page of the book that I am indexing. Each column is constrained as text ("Labels Only") so that the final product can be sent as text to a different program that would impose the structure of an index upon the data.

The macros are on sheet B. The main macro is in column B, as reflected in Table 19.7. Its error-checking subroutines are in column D, as reflected in Table.

Cell B1 starts the macro by finding the first blank line in the A column to begin. This is where the loop begins for each new row; the loop for testing individual cells begins in B2.

Table 19.7: Adding data, with prompts, default values, and error checking

	B
1	{Selectblock @Offset(Data:A1,@Count(Data:A1..A10000),0)}
2	{If @Cellpointer("col")=1}{let []c(0)r(0),[]c(0)r(-1)}{Edit} {Ctrl+Shift+Home}
3	{If @Cellpointer("col")=2}VERSES{Edit} {Ctrl+Shift+Home}
4	{If @Cellpointer("col")=3}{let []c(0)r(0),[]c(0)r(-1)} {Edit}{Ctrl+Shift+Home}
5	{If @Cellpointer("col")>3}{Branch B1}
6	{?}
7	{If @CountBlank(@@(@Address(@Cell("row",[]c(0)r(0)),1)&".. "&@Cell("Address",[]c(0)r(0))))>0}{D1}
8	{If @Cellpointer("col")=1} {If @IsErr(@Match([]c(0)r(0),$C:$A$1..$A$1000,0))}{Branch D3}
9	{If @Cellpointer("col")=2}{If @IsErr(@Find(".",[]c(0)r(0),0))} {Branch D7}
10	{If @Cellpointer("col")=3}{If @IsErr(@Value([]c(0)r(0)))} {Branch D10}
11	{If @Cellpointer("col")=3}{If @Value([]c(0)r(0))<@Value([]c(0)r(-1))} {Branch D13}
12	{Right}{Branch B2}

B2..B5 test the column where the cursor is and supply default prompts or responses appropriate to that column.

If the cursor is in column A, B2 by default inserts the value in the cell above it, on the idea that one will often have multiple citations in a row from the same Bible book. It goes into edit mode and selects the reference, so that if it is satisfactory, pressing `[Enter]` accepts it, but if not, the user simply types the desired book.

If the cursor is in column B, B3 enters the prompt VERSES, goes into edit mode, and selects the word; the user simply types the desired verses.

If the cursor is in column C, B4 by default inserts the value in the cell above it, on the idea that one will often have multiple citations on the same page of the book one indexes. It goes into edit mode and selects the page, so that if it is satisfactory, pressing `[Enter]` accepts it, but if not, the user simply types the desired page.

If the cursor is not in columns A..C, B5 loops back to cell B1, and the user starts a new row. This is an infinite loop, so the user stops it at any time by pressing `[Ctrl+Break]`.

As long as the cursor is in columns A..C, cell B6 gets the user's entry with the `{?}` command. B7..B11 then conduct error checking.

B7 checks for whether there are any blanks in cells from the A column to where the cursor currently is; if so, it reports an error and the macro branches to B:D1 (below) for handling.

B8 checks whether the entry of a book in the A column matches some entry in a list of Bible books that is stored in column A on sheet C. If so, it passes this test, but if not, it branches to B:D3 for handling.

B9 checks whether the entry of verses in the B column corresponds to my format, which requires at least on period separating a chapter from a verse. If not, it branches to B:D7 for handling.

B10 checks whether the entry of a page number in the C column is number-like. If not, it branches to B:D10 for handling.

B11 checks whether the entry of a page number in the C column is a number less than the number immediately above it. If it is, it branches to B:D13 for handling.

If the entry passes those tests, B12 moves the cursor one cell to the right and loops back to B2.

Table 19.8: Same: Error checking subroutines

	D
1	{Beep 5}{Alert "Blanks","There should be no blanks in this row!",A1}{Branch B6}
2	
3	{Beep 5}{Alert "Book name?",([]c(0)r(0)&" is not in the book list. Add it?"),A1,4}
4	{If A1=6}{Let @Offset(C:A1,@Count(C:A1..A1000),0),[]c(0)r(0)}{Branch B12}
5	{Edit}{Ctrl+Shift+Home}{?}{Branch B7}
6	
7	{Beep 5}{Alert "Verses?",([]c(0)r(0)&" is not in cc.vv format"),A1,0}
8	{Edit}{Ctrl+Shift+Home}{?}{Branch B7}
9	
10	{Beep 5}{Alert "Reference?",([]c(0)r(0)&" is not a page reference"),A1,0}
11	{Edit}{Ctrl+Shift+Home}{?}{Branch B7}
12	
13	{Beep 5}{Alert "Right?",([]c(0)r(0)&" is less than the line above it. Correct it?"),A1,4}
14	{If A1=6}{Edit}{Ctrl+Shift+Home}{?}{Branch B7}
15	{Branch B12}

Each of these error checking routines begins with an audible beep and an `{Alert}` message, storing the response to the message in B:A1.

D1 simply branches back to the paused state, awaiting user input, in B6. The user can fill any blanks in the row, press `[Enter]`, and resume data entry.

D3..D5 asks whether the book should be entered into the listing of Bible books. If so, D4 adds it, and then branches to B12 so the cursor moves to the next cell for resumed data entry. If not, D5 re-selects the entry so that the user can make appropriate corrections, and it then re-checks the entry for errors.

D7..D8 re-highlight the entry so that the user can put the verses in the B column into the correct format, and it then branches back to B7 for error checking.

D10..D11 does the same for page references in the C column.

D13..D15 asks whether the user wants to correct the page reference. If so, the entry is re-highlighted, and when the user makes a new entry in the cell, the macro branches back to B7 for error checking. If not, the macro simply moves to the next cell.

How to update a row or a cell in a database

As in the cases of retrieving rows of data and adding them, the task of updating a row of data turns on finding the correct row. The macro in Table 19.9 uses @MATCH to search for the row in the database corresponding to the invoice number in A:A11, and then it places the data in A:B11..E11 into the correct row of the Database.

Table 19.9: Macro to find particular row in database and put potentially different data in it

	A	B
1	D-Row	{Let A1,@Match(A:A11,Data:A1..A1000,0)+1}
2	A-Col	{For A2,2,@Cols(A:A11..E11),1,B4}
3		
4		{Let ("Data:"&@Address(A1,A2)),@@("A:"&@Address(11,A2))}

In B1, `@MATCH` returns the offset from Data:A1 at which we find the value equal to A:A6, but because that uses an offset of 0 for the first row, to convert it to the row on the screen, we add `+1`. B2 runs the by-now familiar `{For}` loop through the cells in A6..E6, here beginning with column 2 (B), and calling the command at B4 that overwrites the existing data in the database.

If you want to update only one item of information in one row in a well-formed database, it is often better to do so with a single command. Thus, for instance, if you want to change the name in column C on an invoice, and the invoice number is stored at A:A6, and the new name is stored at A:C6, then a single command is easy:

```
{Put Data:A1..E1000,2,
@MATCH(A:A11,Data:A1..A1000,0),A:C11}
```

This puts the data at A:C11 into the database in column 2 (meaning column C, since `{Put}` uses offsets of 0), at the row where the invoice number in A:A11 can be found.

If you simply want to navigate to the cell in the database in order to modify the name manually, use a `{Selectblock}` command with some addressing macro. Here, it would be:

```
{SelectBlock @OFFSET(Data:A1,
@MATCH(A:A11,Data:A1..A1000,0),2)}
```

How to modify a cell in each row if another condition in the row is true

A macro can easily conditionally modify data in each row. Let us assume that you want to change the invoice price in column E to give a 5% discount if the quantity in column D is 3 or more. Here is a straightforward macro.

The macro in B1 runs the macro in B3 for each row of the database. It uses the starting offset of 0 because the commands in B3 and B4 use those offsets.

In B3, the macro tests whether the value in column D (column 3, if the starting offset is 0) is less than 3, and if so, it skips the row by the `{Return}` command. If not, it proceeds to B4.

In B4, it puts into the E column 0.95 of the value that is already there, thus

Table 19.10: Applying 5% discount to certain values in a column

	A	B
1	D-Row	{For A1,0,@Count(Data:A1..A1000)-1,1,B3}
2		
3		{If @Index(Data:A1..E1000,3,A1)<3}{Return}
4		{Put Data:A1..E1000,4,A1,@Index(Data:A1..E1000,4,A1)*0.95}

making a 5% reduction in the price. The overall macro does this for every row where the amount in the D column is 3 or more.

Chapter 20

Databases: Structural Change

Altering Order: {Sort}

Use the `{Sort}` command to alter the structure of a database or other set of data. It is usually used in the context of a series of commands of the format `{Sort.`*`Parameter`*`}` that set the parameters for the sort, and as often happens, those parameters are poorly documented.

As I see it, after experimenting, the parameters to be selected depend mostly on whether you want to include a "Heading" or not. The only advantage of including a Heading is that you can specify a sort key in terms of text in the (last row of the) Heading block; otherwise, you can specify it in terms of a cell.

In terms of logical sequence, this is how I would recommend structuring the sequence of commands:

•`{Sort.Reset}` will clear the settings from a prior search. By using this at the start, you can insure that the following settings will obtain consistent results. Without it, settings in prior searches may carry over into the current search and affect the outcome.

•`{Sort.Heading `*`1or0`*`}` determines whether the search block will (1) or will not (0) include a certain number of rows of heading. The choice affects how the following parameters are set and the options that the programmer has in specifying the sort key.

•`{Sort.HeadingSize `*`Number`*`}` sets the number of rows in the header. If the `{Sort.Heading}` parameter was set at 0, this number simply does not matter, and can be set at 0 or 1000.

But if that parameter is set at 1, this number must equal the number of rows in the block that will not be sorted, or in other words, the number of rows in that block that should remain in place.

•`{Sort.Block `*`Block`*`}` sets the block to be sorted. If the `{Sort.Heading}` parameter was set at 0, this block should be entirely limited to the data that will be sorted. If it included the header, the rows in the header will also be sorted, which is almost certainly unintended results.

But if that parameter is set at 1, the block should include the number of rows set by the `{Sort.HeadingSize}` parameter.

So, for example, if the data that will be sorted is in cells A5..D100, the block should be adjusted based on settings as shown in Table 20.1.

Table 20.1: Valid Sort Settings for Data in A5..D100

Heading 0	A5..D100
Heading 1 HeadingSize 1	A4..D100
Heading 1 HeadingSize 2	A3..D100
Heading 1 HeadingSize 3	A2..D100
Heading 1 HeadingSize 4	A1..D100

•`{Sort.Key_1` *`CellOrText`*`}` identifies the primary column to be sorted. Subordinate columns can be specified by similar commands up to a fifth column.

If the `{Sort.Heading}` parameter was set at 0, this column can be specified by reference to any cell in the column. Thus, if data in column B are to be sorted, the key could be cell B1, B2, or B10000, whether the key is in the sort block or not.

If the `{Sort.Heading}` parameter was set at 1, however, the key must be either (a) a cell in the primary column to be sorted and within its header rows, or (b) the text in the same column on the last header row. Thus, for example, if the sort block A1..D100 consists of four header rows (A1..D4) and sortable data of A5..D100, column B can be designated as the primary sorting column by using as the key B1, B2, B3, B4, or the text in B4.

•`{Sort.Order_1` *`AscendingOrDescending`*`}` identifies whether the sort orders data from least to greatest (Ascending) or from greatest to least (Descending). Subordinate columns can be ordered using the same options up to a fifth column.

An ascending sort will place the least number in the first row of data, the second smallest in the next row, and so on. It will also order text alphabetically, with entries beginning with A preceding entries beginning with B, and so on. The descending sort simply reverses the order.

•`{Sort.Go}` then executes the sort, so it belongs after all parameters have been set. Parameters other than the above can be set, as sketched in the help file, but these are the ones I find most useful.

Changing Database Structure: {BlockInsert}, {BlockDelete}

You can alter the structure of a database by inserting or deleting rows and columns. The commands are:

- To insert row(s):
  `{BlockInsert.Rows CellOrBlock,EntireOrPartial}`
- To insert column(s):
  `{BlockInsert.Columns CellOrBlock,EntireOrPartial}`
- To delete row(s):
  `{BlockDelete.Rows CellOrBlock,EntireOrPartial}`
- To delete column(s):
  `{BlockDelete.Columns CellOrBlock,EntireOrPartial}`

`CellOrBlock`. This argument defines the place at which the insertion or deletion will occur.

If the block is more than one cell high, the commands to insert or delete rows will insert or delete the same number of rows.

If the block is more than one cell wide, the commands to insert or delete columns will insert or delete the same number of columns.

`EntireOrPartial`. This argument requires the programmer to choose whether to insert or delete an *entire* row or column (or multiple rows or columns), or only a *partial* row or column.

`Entire`. Choosing the entire option is easier to program, since you only need to identify a single cell as the `CellOrBlock`. But you must be certain that inserting an entire row or column will not disrupt data elsewhere on the sheet

`Partial`. Choosing the `partial` option requires identifying the precise block to be inserted or deleted.

- In the case of inserting rows, the effect is simply to move data in `CellOrBlock` downward outside the block.
- In the case of deleting rows, the effect is simply to move data below `CellOrBlock` upward into the block.
- In the case of inserting columns, the effect is simply to move data in `CellOrBlock` to the right outside the block.
- In the case of deleting columns, the effect is simply to move data to the right of `CellOrBlock` leftward into the block.

Examples

- To insert an entire row where row 10 currently is:
  `{BlockInsert.Rows A10,Entire}`
- To insert five rows where row 10 is:
  `{BlockInsert.Rows A10..A14,Entire}`
- To insert a row where the cursor currently is:
  `{BlockInsert.Rows c(0)r(0),Entire}`
- To insert five rows at the current place:
  `{BlockInsert.Rows c(0)r(0)..c(0)r(4),Entire}`
- To insert a row in D:A10..E10 only:
  `{BlockInsert.Rows D:A10..E10,Partial}`

How to delete rows with ERR in a column

Let's say that you want to remove rows that have an `ERR` in a particular column. Deleting rows confuses efforts to keep track of row numbers, so the macro in Table 20.2 will assume that the user has selected the top cell in the column. It will go down the column, and when it confronts an `ERR`, it will delete the row.

Table 20.2: Deleting rows with ERR

	A
1	{If @CELLPOINTER("type")="b"}{Quit}
2	{If @ISERR(@CELLPOINTER("contents"))=1} {BlockDelete.Rows c(0)r(0);Entire}{Branch c(0)r(-1)}
3	{Down}{Branch c(0)r(-2)}

The macro begins in A1 with a test for stopping the loop when the cursor reaches a blank cell.

A2 then tests whether the content of the cell is ERR. If it is, the macro deletes the entire row and then loops back to the test for whether the current cell (which had been the cell below the one containing ERR) is blank. It will continue in this loop as long as it continues encountering cells containing ERR. When it doesn't, the macro proceeds to A3.

A3 moves the cursor down, and then loops back to A1. This has the effect of testing every cell in the column until a blank is reached, and deleting all rows that have ERR in the column.

How to delete rows with blank lines

The utility macro in Table 20.3 goes down a column of entries and deletes rows when it comes up on a blank, but it knows when to stop. A1 tests whether we have passed the last cell in the column with data, and if so, it quits. A2 tests whether the current cell is blank, and if so, it deletes the entire row and loops back to A1. A3 loops back to A1.

Table 20.3: Deleting blank rows

	A

1	{if @cellpointer("row")>@MAX((@ISBLANK(A:B1..B10000)=0)*@ROW(A:B1..B10000))}
	{home}{quit}

2	{if @CELLPOINTER("type")="b"}{BlockDelete.Rows c(0)r(0);Entire}
	{branch c(0)r(-1)}

3	{d}{branch c(0)r(-2)}

Chapter 21

Input and Output: Printing, Text, Other Formats

Here we look at some issues of input and output: importing data from text sources, printing data, and exporting blocks to text, separate spreadsheets, or other sorts of files. Particular problems in exporting to WordPerfect will be addressed in the next chapter.

Printing to Printers: {Print}

Use the `{Print}` command to send a block or page to a printer. It is usually used in the context of a series of commands of the format `{Print.`*`Parameter`*`}` that set the parameters for the print job, and as often happens, those parameters are poorly documented.

If you always print essentially the same size and shape of a document, you may not need any of these commands except the last, but if you potentially have different types of print jobs in the same session, setting the main parameters each time will give you the consistent and intended print job.

In terms of logical sequence, this is how I would recommend structuring the sequence of commands:

•`{Print.Paper_Type` *`PaperType`*`}` selects a paper type that is available from the File > Page Setup dialog box. Mine is most commonly set to `Letter`, without quotation marks.

•`{Print.Orientation` *`PortraitOrLandscape`*`}` makes the standard selection of `Portrait` or `Landscape` orientation, again without quotation marks.

•`{Print.Top_Margin` *`TopMargin`*`}` and `{Print.Left_Margin` *`LeftMargin`*`}` set the top and left margins, obviously. My default margins are `"0.25 in"` for both, this time inside quotation marks. I find that for most print jobs that are so standardized that they are suitable for a macro, these are the only two margins that need to be set, but bottom and right margins can be set as well.

•Headers can be printed with commands such as `{Print.CreateHeader Yes}`, `{Print.Header_Margin` *`HeaderMargin`*`}`, `{Print.Headers_Font` *`FontSettings`*`}`, and `{Print.Header` *`HeaderString`*`}`. The `Yes` argument in the first command is the gateway to using the others. The header can be removed in later print jobs by the command `{Print.CreateHeader No}`.

The `HeaderMargin` is expressed in a height such as `"0.5 in"`.

The `FontSettings` argument is a string of six elements, separated by semicolons, and enclosed in quotation marks. The first element is the font name, and the second is the point size of the font. The remaining four elements are either `Yes` or `No`, reflecting whether or not the font should be bold, italic, underlined, or strikeout.

The `HeaderString` argument is a string enclosed in quotation marks that includes normal text and optional codes provided by QP. The alignment of text in the header is controlled by the pipe character (|): text before a pipe is left-aligned; text after that pipe is center-aligned; and text following a second pipe is right-aligned.

The codes available for the `HeaderString` argument can be found at Editing and formatting spreadsheets > Formatting spreadsheets > Reference: Formatting spreadsheets > Header and footer codes. Among the more important ones are:

#f - File name without path
#F - File name with path
#n - Moves to a new line
#p - Current page number
#P - Total pages
#d - Date, short format
#D - Date, long format
#t - Time, short format
#T - Time, long format

Hence, to construct a header in which the date and time are on separate rows on the left, the full file name is centered, and Page-n-of-y is on the right, a string like this would work: `"Date:  #D #nTime:  #T | #F | Page #p of #P"`.

Parallel commands exist for Footers.

•`{Print.Block` ***`Block`***`}` and `{Print.Area Selection}` select a block for printing. The `Block` argument is set within quotation marks, but it can be a variable block as established by reference to a cell that contains the block or a block defined by the `@OFFSET` function.

Note that the first command alone is not always sufficient: if the Print dialog has defaulted back to printing the entire page, for instance, the specified block will be ignored. The second command enforces the selection of the block.

•`{Print.DoPrint}` then sends the block to the printer, so it belongs after all parameters have been set. Parameters other than the above can be set, as sketched in the help file, but these are the ones I find most useful.

Saving in other formats: {FileSaveAs}

The `{FileSaveAs}` command saves QP files in other formats. The command has only two active arguments, separated by two meaningless arguments that may have at one time been significant. `{FileSaveAs` ***`FileName,,,FileFormat`***`}` saves the entire current file as `FileName` and in a particular `FileFormat`. Techniques for saving a block of data in a particular format will be discussed below.

Table 21.1 shows how to save a QP notebook to some of the more important file formats, among many others. In some cases, only the current page can be saved in the different file format. In most cases, extra objects in the file cannot be saved to a file different format.

The difference between CSV format and ASCII is this: When QP saves the file in CSV format, it separates each cell on a row by commas, but places quotation

Table 21.1: Sample export commands

Desired file format	Command
Comma-delimited CSV	{FileSaveAs "C:\Temp\MyData.csv",,,CSV}
Tab-delimited	{FileSaveAs "C:\Temp\MyData.txt",,,ASCII Text}
QP's earlier format	{FileSaveAs "C:\Temp\MyData.wb3",,,QPW v7/v8}
Basic Excel format	{FileSaveAs "C:\Temp\MyData.xls",,,Microsoft Excel v5/v7}
Basic Lotus 123 format	{FileSaveAs "C:\Temp\MyData.wk1",,,"Lotus 1-2-3 v2.x"}

marks only around those text cells that contain commas. (The term "CSV" applies to various systems, some of which place quotations around every text cell, and some around every separate item.) Saving the file as tab-delimited text separates the cells by tabs (ASCII character 9), of course, but does not wrap text cells in quotation marks.

Caution 1: No ending Return. If you save a file as CSV or ASCII, the final row is not ended by a Hard Return. Some commands (such as Readln) assume that a row of data is always ended by a return, and it will not read the last row of data without a return.

Caution 2: Macro Recording Problems. When manually saving a file as ASCII text, the dialog box gives two options, one for saving as comma-delimited text and another for saving as tab-delimited text. When used manually, both options create differently formatted text files. But when the user records macros making either selection, the recorder specifies the `FileFormat` argument in each as *ASCII Text*, and when played, that macro produces only tab-delimited text. To have it the recorder generate a macro that saves a file as comma-delimited text, use *CSV* as the `FileFormat` argument.

Help File Problems. The help file says that the argument after `FileName` calls on you to choose a particular behavior — Confirm or Replace or Backup — but testing shows that the choice makes no difference whatever, and the slot can be left blank. The next argument is simply reserved.

Moreover, the formats listed in the help file do not exactly match those available in the *Save As Type* dropdown box from the File > Save As ... menu.

Extracting blocks to a new spreadsheet: {FileNew} and {FileExtract}

There are several ways in which you can place data blocks from your spreadsheet into new spreadsheets.[1] The choice of method will depend on how the data will be used. One thing to consider in making the choice is the desired format of the data in the new file: many of these commands transfer the data with either all formatting properties of the source cells and data, or none of them. See the summary grid at page 158. You may, for instance, want to store the data with no formatting other than numeric formatting. See the discussion at page 161.

1. The user may open a new file with the `{FileNew}` command and write data

[1]Later sections in this chapter will explain how to write macros that will convert a database into text files.

to it in the manner explained in Chapter 18. After a file is opened in this way, it can be closed with `{FileClose}`.

2. The user may create a single-purpose notebook (here, C:\MyFiles\Links.qpw) that contains links (such as `+[SourceFile]A1`) to the critical data from the old one, and update it in QP. Applying the `{BlockValues}` command to the links in this file would convert the formulas into values. This intermediate notebook file would be preset with the numeric and font formats appropriate for later use, such as importing into WP.[2] To re-use the notebook with links, it should then be saved with a new name. The example in Table 21.2 addresses a notebook that has links in its cells A1..Z50. After those are converted to values, the file is saved with a new name to preserve the Links file.

Table 21.2: Creating an intermediate correctly formatted file

	A
1	{FileOpen "C:\MyFiles\Links.qpw","Update References"}
2	{BlockValues []A:A1..E1000,[]A:A1..E1000}
3	{FileSaveAs "C:\MyFiles\ExportToWP.qpw"}
4	{FileClose}

3. The user may employ the `{FileExtract}` command. `{FileExtract` ***`ValuesOrFormulas,Block,FileName`***`}` extracts a `Block` to a file called `FileName`. The `ValuesOrFormulas` argument requires the programmer to specify whether formulas in the `Block` will remain as formulas or whether they will be converted to values. The `Block` will appear with all formats of the cells and data that were in the original file. Unfortunately, there is no option for saving the data in other formats, such as delimited text.

Quirks. Unfortunately, though the `Values` argument does nicely in converting numeric formulas to values, it converts formulas that produce text to zeros. This is unlike `{BlockValues}`, which converts both numeric and text formulas correctly. There is no reason why the data could not be extracted to a new file with `Formulas`, the new file opened, the block be converted to values by `{BlockValues}`, and the file re-saved, but that should not be necessary. See the discussion at WPU 30463.

Also, `{FileExtract}` saves the cursor's current position in the original file as the current position in the new file, which seems like an odd choice when opening a new file that may only be a block of unrelated text.

Samples. Using the prior example of a database that consists of a certain number of rows filled within the block D:A1..E1000, the block with data in it can be identified as `@OFFSET(A1,0,0,@COUNT(A1..A1000),5)`. This command extracts that block of data into a file called *C:\Temp\Extract.qpw*, preserving formulas in the original:

```
{FileExtract Formulas,@OFFSET(A1,0,0,
@COUNT(A1..A1000),5),"C:\Temp\Extract.qpw"}
```

To work around the inability to extract data in a different file format, the new

[2]The new notebook can also be saved in some format other than QPW (page 201), and it perhaps should be saved in a different format if it is to be merged into WordPerfect (see the next chapter at page 211).

file could then be opened with `{FileOpen}`, converted to some other format with `{FileSaveAs}`, and then closed with `{FileClose}`.

Handling text files with macro commands: {Open}, {ReadLn}, {WriteLn}, etc.

QP has a number of commands for handling export to and input from a text file. The main value of these commands is to be able to transfer data to and from other programs that can deal with delimited text files. WP does not import such files automatically (though a PerfectScript macro could be written to do so), but other programs like QP do.

This section covers only the basic commands and options for sequentially processing lines in text files, leaving more complex applications to other programs and programmers. In the cases considered here, the delimiter between items of data on a row will be the caret (^) and rows will end with a standard carriage return.

In programming, recognize that these commands deal with only one text file at a time.

The most important commands are:

`{Open TextFileName,RorWorA}` which opens a file specified as text (i.e., in quotation marks or in a cell containing the file name as text). It opens the file in one of several modes, the two most important of which are:

- `R` (read-only for an existing file);
- `W` (writing, which creates a new file or entirely overwrites an existing one);
- `A` (appending to the end of an existing file).

Other modes are available for more complex uses.

`{Close}` closes the open text file, and as noted above, there can be only one.

`{WriteLn Text1<,Text2,Text3,...>}` writes the Text argument(s) to a single line in the text file opened in `W` or `A` mode. It adds a carriage return at the end, so that the next text exported will appear on the next line of the text file.

`{Write Text1<,Text2,Text3,...>}` command does the same, but without the carriage return that goes to the next line. Since I am presenting these commands as a means of producing delimited text files to be processed line-by-line, I recommend either that you use only `{WriteLn}` or, after one or more `{Write}` commands, you conclude with `{WriteLn ""}`, which would add a carriage return at the end of the line.

`{ReadLn Cell}` reads an entire line of text from the file opened in `R` mode and places that content in a QP `Cell`. The macro can process the line further by acting on the contents of `Cell`. The `{Read}` command reads only a specified number of bytes, which requires greater complexity in programming than is needed here.

`{FileSize Cell}` stores the size in bytes of the currently open text file in `Cell`.

`{GetPos Cell}` stores the current position of the byte pointer in the open text file in `Cell`. The byte pointer is the point from which `{ReadLn}` would read a line of text and to which `{WriteLn}` would write a line of text. For more complex programming, the byte pointer can be moved around by the `{SetPos}` command, but for the sequential line processing covered here, the byte pointer will always be at the start of a line of text or at the end of the text file.

Quirks. Unlike most or all other QP macro commands, when these are executed, no further commands in the same cell are executed, and the macro proceeds with commands in the next cell below. The programmer should recognize this anomaly in planning the structure of the macro.

I also find that when executing these text-file input and output commands, QP does not like to use subroutines, but it can use `{Branch}` commands. Therefore, in programming, don't use a command like `{M1}`, but use a command like `{Branch M1}`, and find a way to branch back.

How to import a delimited text file without {ParseExpert}

QP has a tool called Quick Columns (Tools > Data Tools > Quick Columns...) that allows the user to take a source of data, including delimited text files, and to divide it (parse it) into columns. Once you get the knack of it, it works nicely. However, it has been observed that when you record the macro commands that do so, playing the recorded `{ParseExpert}` commands does not work. This has apparently been a problem since QP10 (WPU 9267) and it appears to remain a problem (WPU 34095).

Fortunately, the macro commands for importing text files help with delimited text. The example in Table 21.3 will essentially reverse the last one. We'll open the text file *C:\Temp\DelimitedText.txt* for reading, read each line of it, separate it into distinct units delimited by the caret, and add each to the database in Data, converting number-like text into numbers along the way.

Table 21.3: Reading a delimited text file into a database

	A	B
1	D-Row	{Open "C:\Temp\DelimitedText.txt",R}
2	D-Col	{FileSize A3}
3	FileSize	{GetPos A4}
4	FilePtr	{If A4>=A3}{Close}{Quit}
5	Text	{Let A1,@COUNT(Data:A1..A1000)}
6	#Fields	{ReadLn A5}
7	Data	{Let A6,@LENGTH(A5)-@LENGTH(@SUBSTITUTE(A5,"^",""))+1}
8		{For A2,1,A6,1,B11}
9		{Branch B3}
10		
11		{Let A7,@FIELD(A5,A2,"^")}
12		{If @ISERR(@VALUE(A7))}{Let @OFFSET(Data:A1,A1,A2-1),A7}{Return}
13		{Let @OFFSET(Data:A1,A1,A2-1),@VALUE(A7)}

B1 opens the delimited text file and B2 stores the file size in A3. This number, which marks the end of the file, is necessary in order to know when we have reached the end of the file and can stop looping through each row of it.

B3 is the place where the loop begins, and it begins by storing into cell A4 the current byte pointer in the text file. B4 then tests whether that pointer is greater than or equal to the file size, in which case the macro has processed the entire file, which is then closed and the macro terminates.

B5 gets the next blank line in the target database and stores it in A1.

B6 reads the entire current line of text from the text file into A5.

B7 counts the number of fields that are delimited by ˆ in A5. The number of fields on a row in a properly formatted delimited text file should be the number of delimiters plus 1. This formula counts the number of delimiters by subtracting from the length of the text in A5 the length it would have without delimiters (as established by the `@SUBSTITUTE` function). It adds 1 and stores the sum in A2, where these macros have stored the column number.

B8 then runs a `{For}` loop for each item in the text, as B7 stored in A6. For each item, it calls the macro at B11, and when that cycle is complete, B9 causes the macro to branch back to B3 to handle the next line of text.

B11 takes each delimited item in the text row, as separated by the `@FIELD` function, and places it in cell A7. The purpose of doing so is to test that cell for whether it is a number-like cell or not. If it is not, we will put it into the corresponding cell of the database, but if it is, we need to convert it to a value, because unless we do so, the number will be imported purely as text.

B12 performs the test on whether the delimited item is numberlike. If it is numberlike, using `@VALUE` on it will return a value, but if not, using `@VALUE` on it will return `ERR`. If the test in B12 shows that the item is not numberlike, it uses `@OFFSET` to locate the corresponding cell in the database and places it there with a `{Let}` command, and then returns control back to the `{For}` loop to test the next item. If the item is numberlike, the macro proceeds to B13.

B13 places the numeric value of the delimited item into the current cell, using `@VALUE` to convert the number-like text into a number.

How to append text to an existing file, if any, or create a new one

Say that you want to add data from the spreadsheet to a text file called "C:\Temp\MyData.txt," but you don't know if that file exists; if it does not, you want to create the file new. The only native QP command that creates a new text file is `{Open}` with the `W` (write) parameter, but opening a file with the `W` parameter will overwrite data in the file if it already exists. Opening the file with the `A` (append) parameter does what you want, but it will not be effective if the file does not exist.

You must first test whether the file exists. There are at least two ways to do so. The easiest is to use the `@FILEEXISTS` function. But since I wrote this article before discovering that function, we'll begin with an only slightly more complex method using the `{GetDirectoryContents}` command.

1. `{GetDirectoryContents` *`OutputCell,FileName`*`}` lists all files matching the `FileName` argument in a column starting at the `OutputCell`. The help file erroneously states that `FileName` must include wildcards; a search for a `FileName` without wildcards will return the `FileName` if it exists, and it will not change the `OutputCell` otherwise.

The solution in Table 21.4 uses `{GetDirectoryContents}` to determine whether C:\Temp\MyData.txt exists. If so, it opens the file to append data to it. If not, it creates the file. It then transfers three columns of entries from Data:A1..C10000 to the new or pre-existing file.

B1 clears cell A2, and then places there any existing file that matches the file

Table 21.4: Appending text to an existing file, or creating a new one

	A	B
1	D-Row	{blank A2}{Getdirectorycontents A2,+A3}
2	Match?	{If A2=""}{Open +A3,W}
3	C:\Temp\MyData.txt	{If A2<>""}{Open +A3,A}
4		{For A1,0,@COUNT(Data:A1..A10000)-1,1,B7}
5		{Close}
6		
7		{WriteLn @INDEX(Data:A1..C10000,0,A1)," ", @INDEX(Data:A1..C10000,1,A1)," ", @INDEX(Data:A1..C10000,2,A1)}

shown in A3. If that file exists, A2 will have content; if it does not, A2 will remain blank.

B2 and B3 then text whether A2 is blank. If it is, the file does not exist, and B2 creates the file by opening it with the `W` parameter. If not, B3 opens the file to append data to it. Both proceed to the `{For}` loop in B4.

The `{For}` loop in B4 takes every row in the Database and writes it to the open text file. Note that this assumes that the data is entirely text. The next example shows how to deal with data that includes numbers.

When the loop finishes, the command in B5 closes the text file and the macro stops.

2. The second solution is slightly easier because cells B1 and A2 are not needed. B2 can be replaced with `{If @FILEEXISTS(A3)}{Open +A3,A}`, and B3 can be replaced with `{If @NOT(@FILEEXISTS(A3))}{Open +A3,W}`. Unlike in the first solution, the order of testing here is significant. If the order of B2 and B3 were reversed and the file did not already exist, the B2 command would create it and open it, and then the B3 command would find that it exists, and attempt to open it (again) to append to it.

How to save a database as a delimited text file

When dealing with delimited text files, I generally prefer to use the caret (ˆ) as the delimiter rather than either commas or tabs. The caret is at least as easy to program for, and it will never appear in my data on its own. It is also QP's default "custom" delimiter in the dialog box that comes from Tools > Data Tools > QuickColumns . . . .

The task here will be to convert the sample database we have been using for some time now (starting in the chapter on retrieving data from a database at page 170) to a delimited text file, using ˆ as the delimiter between cells. The macro in Table 21.5 does the job.

Cell B1 opens the file *C:\Temp\DelimitedText.txt* for writing. B2 runs a `{For}` loop on each row of the database Data:A1..E1000 in the now-familiar way, storing the row offset in A1, and for each row, calling the macro at B4. When the loop finishes, the `{Close}` command closes output file and the macro stops.

B4 runs a `{For}` loop through each cell on the row of data by column, storing the column offset in A2. It calls the macro at B6 for each cell.

Table 21.5: Saving database to delimited text file

	A	B
1	D-Row	{Open "C:\Temp\DelimitedText.txt",W}
2	D-Col	{For A1,0,@COUNT(Data:A1..A1000)-1,1,B4}{Close}
3		
4		{For A2,0,@COLS(Data:A1..E1000)-1,1,B6}
5		
6		{Contents A3,@OFFSET(Data:A1,A1,A2),1000}
7		{If A2=@COLS(Data:A1..E1000)-1}{WriteLn A3}
8		{If A2<@COLS(Data:A1..E1000)-1}{Write A3,"^"}

B6 uses the `{Contents}` command to transfer the content of the cell to A3 as text. Any other command that converts numbers to text would work, but numbers must be converted to text because `{Write}` and `{WriteLn}` take only text as arguments, not numbers.

B7 and B8 test whether the cell is the last (rightmost) cell in the database. If it is, B7 writes it to the output file with a carriage return. If it is not, B8 writes it to the output file, followed by the ^ delimiter.

An alternative to the structure in B6..B8 is to use a series of cells for each item of data on the source row that will go into the delimited text file. Another cell would use the `@Concatenate` function, which converts numbers to text, to combine those cells and delimiters. Then, the latter cell could be written to the text file with a single {WriteLn} command.

How to save a database as a CSV file

Here, the task is to save a particular block of QP data into a comma-delimited (CSV) text file in the same manner that `{FileSaveAs}` does. These examples assume that the database in question is Data:A1..E1000.

If there is no extraneous data on the Data sheet, the simplest method will be to navigate to the Data sheet and use a command like:

```
{FileSaveAs "C:\Temp\Export1.csv",,,CSV}
```

QP will warn that only the current page will be saved in that format, but it will save the entire page that way. What, however, if this method is undesirable, either because there is extraneous data on the Data sheet or otherwise? Unfortunately, the `{FileSaveAs}` command saves the entire current page in a given format, and unfortunately, `{FileExtract}` can save a block in the file to another file, but only in QPW format. Therefore, workarounds are needed.

Method 1: Copy to new file

Table 21.6 shows what is probably the simplest method. The macro copies the source data in A1..A2. A3 simply selects a random cell (otherwise, the entire database remains selected). A4..A5 opens a new file and pasts the database. A6..A7 save the database in CSV format and closes it.

Table 21.6: Copying the database block to a new file, before saving as CSV

C:	A
1	{EditGoto Data:A1..E1000}
2	{EditCopy}
3	{SelectBlock Data:A1}
4	{FileNew}
5	{EditPaste}
6	{FileSaveAs "c:\temp\export1.csv",,,CSV}
7	{FileClose}

Method 2: Writing to a CSV text file

The method in Table 21.7 exports to CSV in perhaps the same way that QP's coding does for the entire file. QP's version of CSV appears to convert each cell to text, and it writes that text as is to the CSV file if the text contains no comma, but if the text contains a comma, it encloses the text in double-quotes.

Table 21.7: Writing a CSV file, using QP's version of CSV format

C:	A	B	C
1	Data:A1..E1000	{Open "C:\temp\export2.csv",W}	
2	Row	{For C:A2,0,@Rows(@@(C:A1))-1,1,C:C3}	

3	Col	{Close}	{If @Cell("type",@@(@Offset(@@(C:A1),C:A2,0)))="b"}
			{ForBreak}

4	Content		{For C:A3,0,@Cols(@@(C:A1))-1,1,C:B5}
5		{Contents C:A4,@Offset(@@(C:A1),C:A2,C:A3),1000}	
6		{If @Not(@IsErr(@Find(",",C:A4,0)))=0}{Write C:A4}	

7		{If @Not(@IsErr(@Find(",",C:A4,0)))=1}
		{Write @Char(34),C:A4,@Char(34)}	

8		{If C:A3<@Cols(@@(C:A1))-1}{Write ","}	
9		{If C:A3=@Cols(@@(C:A1))-1}{WriteLn ""}	

The user enters the database block into A1 and then runs the macro at B1. B1 simply opens a text file and B3 closes it. The intervening cell, B2, runs a `{For}` loop for each row of the database, calling the macro at C3, which first tests whether we've reached a blank row (in which case the `{For}` loop terminates), and if not, it runs the macro at B5..B9 for each cell in the row, column-by-column.

B5 places the contents of the cell as appears on screen as text into A4. B6..B7 test whether there is a comma in that text in A4; if not, B6 writes it as-is to the CSV file; if so, B7 writes it between double quotes to the CSV file. B8..B9 test whether the current cell is the last one in the row; if not, it writes a comma to the CSV file; if so, it uses `{WriteLn}` to write the end of the line.

How to save a file in another format automatically

In WPU 37382, a QP user wanted to save QP notebooks to some format that she could view on an Android app, since Corel has written no viewer for QP. The discussion there lays out some issues for handling that need, and Roy Lewis wrote a general-purpose PerfectScript macro that could be attached to a toolbar icon.

Here, we examine a way to do the same automatically, without needing to remember to press the toolbar button, using only native QP commands. The core macro commands are fairly straightforward. If one knows the desired name of the file and folder, it can be hard-coded into the macro as shown in Table 21.8. The only trick is to name the starting cell of the macro (A1 here) as `_NBExitMacro`, which will cause the macro to run when the file is closed, and thus automate this process. See the discussion of automatic startup and shutdown macros at page 116.

Table 21.8: Saving the file automatically to XLS

	A
1	{FileSave}
2	{FileSaveAs "C:\Temp\Backup.XLS";;;Microsoft Excel v5/v7}
3	{FileClose 0}

If, however, the desired name is to determined automatically, the macro can use a function to set that name. See the discussion of functions to modify the file name at page 47 above. The desired function can be substituted into the macro, or it can be placed in another cell of the Notebook, and the macro can then reference the other cell.

How to save a database as an HTML file

QP has a series of macro commands based on `{SaveHTML}` that will generate an HTML file from a QP database, but it is not very flexible and, in some cases, inserted very odd formatting. In lieu of that, I recommend a variation of the method of writing CSVs shown in Table 21.9.

Table 21.9: Writing an html file

	A	B
1	Data:A1..E1000	{Open "C:\Temp\MyWebPage.html",W}
2	Row	{Writeln "<html><head><title>My Title</title></head>"}
3	Col	{Writeln "<body><H1 align=center>My Title</H1> <table border=1>"}
4	Content	{For A2,0,@ROWS(@@(A1)),1,B8}
5		{Writeln "</table></body></html>"}
6		{Close}
7		
8		{If @CELL("type",@@(@OFFSET(@@(A1),A2,0)))="b"} {ForBreak}
9		{For A3,0,@COLS(@@(A1))-1,1,B11}
10		
11		{Contents A4,@OFFSET(@@(A1),A2,A3),1000}
12		{If A3=0}{Writeln "<tr>"}
13		{Writeln "<td>",A4,"</td>"}
14		{If A3=@COLS(@@(A1))}{Writeln "</tr>"}

Chapter 22

Exporting QP Data to WordPerfect

This chapter reviews the ways that QP data can be transferred to a WP document. Numeric data should be calculated and formatted in QP before the transfer, simply because it is the better product for that purpose. The final product will be best beautified and printed from WP.

Caveat. Every available way to transfer data from QP to WP should be easy and bulletproof, but unfortunately, I have experienced a number of quite different problems. This chapter is the result of extensive testing from 2013-2015 to find foolproof and completely reliable ways of transferring data in an efficient way from a complex QP Notebook, Zenas, my QP-based case management system (see WPU 34158). Unfortunately, I found at the time no one "best" method. In particular, straightforward attempts in WordPerfect to merge directly from data in the Zenas notebook occasionally caused crashes. So I experimented with a number of alternatives and workarounds. This chapter will itemize pros and cons of each method.[1]

Caveat to the caveat. Before proceeding, I should note that there have been two later versions of the WordPerfect Suite since my testing, and it is possible that some of the problems I note below have been fixed. Therefore, the user may experiment with methods that I question below. If the problems do not occur, I would be very happy to hear about it and report it. If they do, try the workarounds in this chapter.

Overview of methods. There are essentially four basic methods to transfer QP data to WP. In figure 22.1, the methods are highlighted in yellow, and the file types are in green.

1. Pasting (or PasteSpecial) after copying in QP. See p. 212.
2. Importing (under Insert > Spreadsheet). See p. 214.
3. Merging (under Tools > Merge) in several ways: directly from QP, via a different spreadsheet format, via WP's .DAT format, and via the clipboard. See p. 216.
4. Placing by a PerfectScript Macro. See p. 221.[2]

Factors for deciding on the best method. Variations on these methods should be evaluated in terms these factors:

- Stability – does the process sometimes crash, and what must be done to avoid crashing?
- Numeric formats – what must be done to keep them as QP has them, particularly dates and currency? Should other QP cell formats appear in WP?

[1]In addition, there are ways to merge QP databases into printable documents without using WordPerfect. See page 233 for a sample.

[2]See also the fuller discussion of PerfectScript in Chapter 24, beginning at page 235.

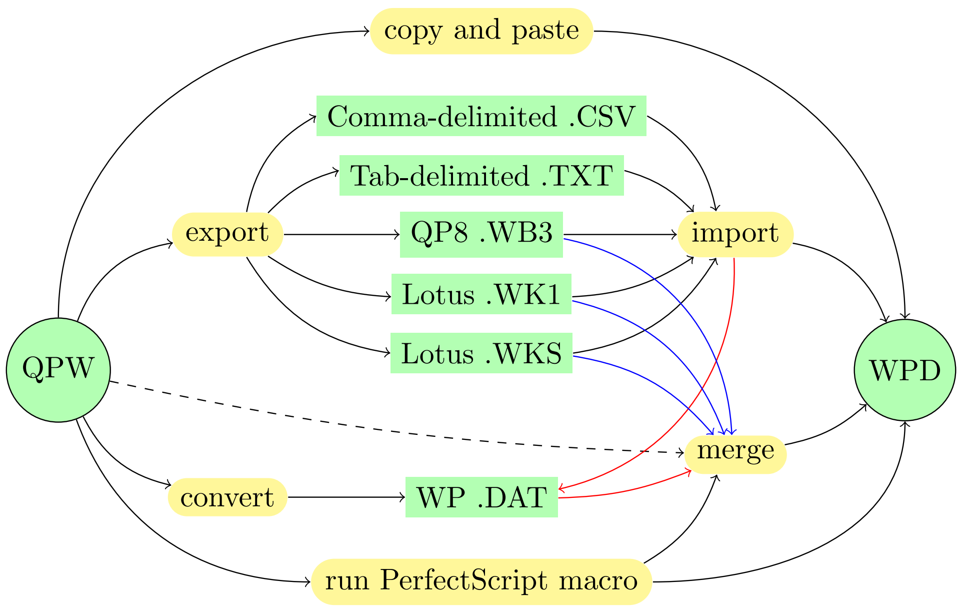


Figure 22.1: Pathways for exporting QP data to WordPerfect

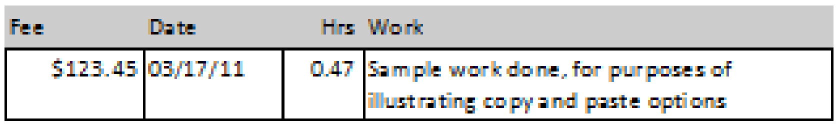
- Post-processing – what else is needed after placement in WP to get it in the desired shape?
- Flexibility – can the data be plugged in any place or order, or must it retain the structure in QP?
- Movement – does it require moving the cursor around in open QP files?
- Simplicity – is it easy to use, or complex?

None of these methods is perfect, and the desired outcome determines which is the best method for a particular purpose.

Copy and Paste

Blocks of cells can be copied in QP and pasted in WP in several formats. The process can be automated by PerfectScript macros, called by QP `{PlayPerfectScript}` commands.

Figure 22.2 is sample data to be copied from a QP spreadsheet and pasted into WordPerfect. The Date column is left-aligned (including dates). The Hrs column is right-aligned (including text). The cells in the Work column are set to allow text-wrapping. Rows are top-aligned. Borders are on all cells in the bottom row, but not the top row.



Fee	Date	Hrs	Work
$123.45	03/17/11	0.47	Sample work done, for purposes of illustrating copy and paste options

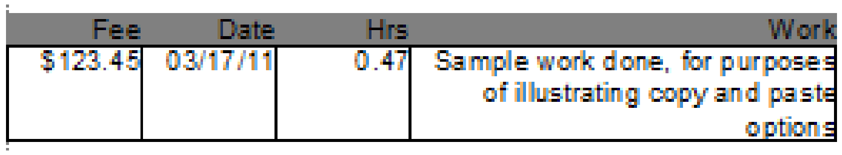
Figure 22.2: QPW Data copied to be pasted into WordPerfect

Here's how the three methods place the data in WP.[3]

[3]And beyond simply placing the cells as a table in WP, that table can be converted to a .DAT merge data file, as explained below at p. 220.

Method #1: Simple paste (same as PasteSpecial as RTF)

Simply pasting the sample along the left margin produces the result in 22.3. Note that the alignments are wrong in several places. The table is also placed oddly, at one tab stop from the left margin is also odd. The spacing (padding between the cell walls and the data within the cell disappeared. The shading and lines are retained, whether or not desired.



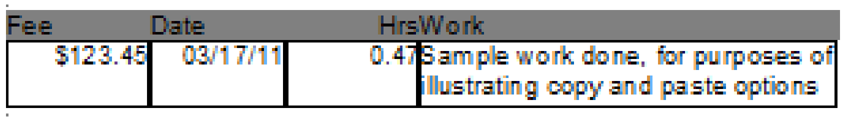
Fee	Date	Hrs	Work
$123.45	03/17/11	0.47	Sample work done, for purposes of illustrating copy and paste options

Figure 22.3: Simple Pasting, and Pasting as RTF

Creating a table in WP first and then pasting into that table put all of this data into the single current cell of the table.

Method #2: PasteSpecial as QP Format

Using PasteSpecial and selecting QP Format in the left margin results in Figure 22.4. Here, the justification and position was correct, but the lines changed. Upon first placement, the table horizontally divides the cells in the first and second columns, though tabbing through those cells causes the division to disappear. Pasting into the cell of a pre-existing table places all the data in the one cell.

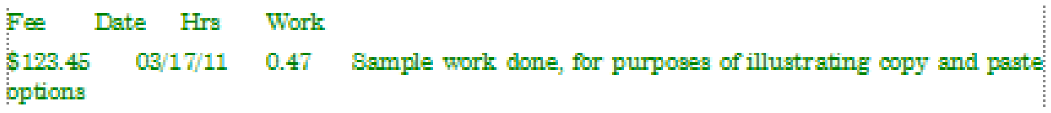


Fee	Date	Hrs	Work
$123.45	03/17/11	0.47	Sample work done, for purposes of illustrating copy and paste options

Figure 22.4: Paste Special in QP Format

Method #3: Paste as Unformatted Text

Using PasteSpecial and selecting Unformatted Text in the left margin produces Figure 22.5. (The text was colored green to set it off from the rest of this text.) This pastes the text with tabs (`[Left Tab]` in Reveal Codes) between the data. All alignment, shading, and borders are gone. As in the other cases, pasting into a table cell places all the data in that cell.



```
Fee      Date     Hrs     Work
$123.45      03/17/11     0.47     Sample work done, for purposes of illustrating copy and paste
options
```

Figure 22.5: Paste as Unformatted Text

Summary of Pros and Cons.

Pros: This is the simplest in concept, and it is stable. It keeps numeric formats nicely.

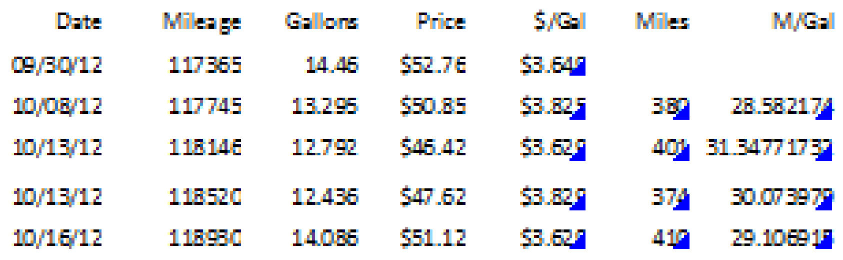
Cons: It requires movement to the affected cells in QP, copying, movement to the WP file, and pasting. It can only be pasted as a cell or block of cells, so it is cannot easily be used outside that context, and even in that context, alignments and cell-padding may require further work. All versions of copy-and-paste are likely to place stray font and formatting codes in the WP document that require further processing.

Importing - ImportDoImport (Caution!)

WP allows a user to import data from QP manually into a WP file (or a preexisting table within the file), using Insert > Spreadsheet > Import (and selecting *Table* under *Import As*) or using the macro `ImportDoImport`. I and a number of other people believe that, unfortunately, this method is too unstable for regular use on complex QPW notebooks. The following are examples of the results when it works.

Method #1. Importing into the main text

Importing a particular block from the only sheet of a particular QPW into the main body of the text yields Figure 22.6, which is close to the original file's format.



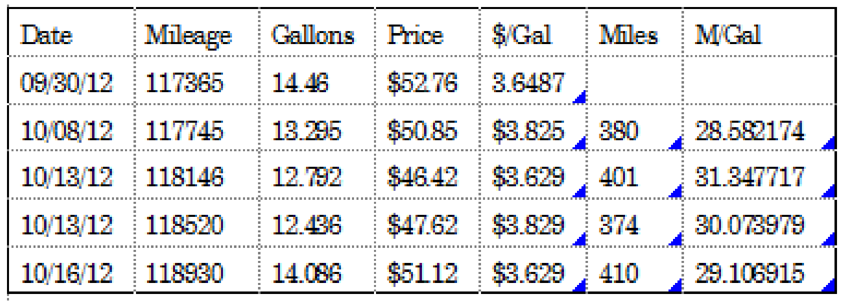
Date	Mileage	Gallons	Price	$/Gal	Miles	M/Gal
09/30/12	117365	14.46	$52.76	$3.649		
10/08/12	117745	13.295	$50.85	$3.825	380	28.582174
10/13/12	118146	12.792	$46.42	$3.629	401	31.34771732
10/13/12	118520	12.436	$47.62	$3.829	374	30.073979
10/16/12	118930	14.086	$51.12	$3.629	410	29.106915

Figure 22.6: Import of spreadsheet into the main text

Data in the last three columns of the WP file have blue triangles in the bottom right, indicating that formulas are present. These do not appear when the file is printed.

Method #2. Importing into a WP table

Importing the same block into a pre-existing WP table yields this Figure 22.7, which uses the WP table's pre-set format.



Date	Mileage	Gallons	Price	$/Gal	Miles	M/Gal
09/30/12	117365	14.46	$52.76	3.6487		
10/08/12	117745	13.295	$50.85	$3.825	380	28.582174
10/13/12	118146	12.792	$46.42	$3.629	401	31.347717
10/13/12	118520	12.436	$47.62	$3.829	374	30.073979
10/16/12	118930	14.086	$51.12	$3.629	410	29.106915

Figure 22.7: Import of spreadsheet into a WordPerfect table

Though again not apparent in the PDF version of this text, data in the last three columns of the WP file have blue triangles in the bottom right, indicating the formulas are present.

Automating the Import

Table 22.1 is a PerfectScript Macro for importing QP data into WordPerfect. Let's say that the source QPW is called C:\MyFiles\ExportToWP.qpw, and it is composed of a single sheet (A), with three columns of data in columns A, B and C. It is open. Let's say that the WP file is open, and it contains a table called *Table A* that consists of one row of three cells.

Table 22.1: Automating Data Import

```
Application (wp; "WordPerfect"; Default!; "EN")
Application (qp; "QuattroPro"; Default!)
vRows=qp.Eval("@Count([ExportToWP]a1..A10000)")
vBlock="A:A1..A:C"+vRows -".0"
PosTableCell (Cell: "TABLE A.A1")
TableSize (vRows; 3)
ImportSetFileName ("C:\MyFiles\ExportToWP.qpw")
ImportSetSource (Spreadsheet!)
ImportSetDestination (WPTable!)
ImportSetRange (vBlock)
ImportSetMacroVariableName (VariableName: "")
ImportDoImport ()
```

This PerfectScript macro:

- Reads the open QPW file and gets the number of rows of data in the A column using the `Eval` command and stores it in the `vRows` variable.
- Composes the block address with that number and stores it in the `vBlock` variable. The `-".0"` component of the command is due to the fact that the `Eval` command adds one digit after the decimal point, even for integers like row numbers; this removes it.
- Enters the WP table; expands it to contain the number of rows needed for import.
- Identifies the QPW file from which to import by full path (safest).
- Sets all other variables, and runs the import.

See the fuller discussion of PerfectScript in Chapter 24, beginning at page 235.

Quirks, Flaws, Bugs, and Workarounds.

A minor quirk is that the block to be imported must be specified in a non-standard manner. The block cannot be identified by a name assigned to it, only by coordinates. Both beginning and ending coordinates must include sheets, and the sheets can only be addressed by letter (not sheet name). Thus, even though this sheet was called "Mileage," it had to be addressed as "A," and the entire block imported had to be called A:A2..A:G7.

The help file tells us that "When you import a spreadsheet workbook, WordPerfect imports only the first sheet," but that is not actually true. I have imported data from later pages of some spreadsheets, but not all.

I have observed that when the source QPW file lacks data in some cells in the top row, the import incorrectly moves data into the gap that should be to the right of the gap. WPU 35226. One workaround is to have a header row or ensure that all cells in the top row have data. Another is to import into a merge data file rather than a table. Importing into a merge data file does not result in the same error; see the discussion below (page 217) of merging via a different format.

Importing a QPW file with a certain number of rows somehow forces WP's merge functions to assume that the next QPW file to be merged will have the same number of rows, whether it actually has more or less (thereby either merging less than all records or extra blank records). WPU 35226.

A major problem is that for some yet-to-be-determined reason, WP fails (and sometimes crashes) on attempting to import such data from more complex spreadsheets. Error messages include suggestions that there is "invalid data" in the QPW file or that it is "corrupt." For background and documentation of the problem, see WPU 32687. The problem does not appear to exist when importing from single-sheet spreadsheets that contain only data. See the discussion above of methods for extracting data to single-sheet spreadsheets (page 202).

Summary of Pros and Cons

Pros: This method keeps numeric formats nicely. It can be executed remotely. It works for importing single-sheet notebooks that have only data into pre-formatted tables. When it works, it is fairly simple and quick to use. It is particularly useful for importing data as a WP merge data file (.DAT), as noted at page 218 below.

Cons: This method is unstable for importing from complicated notebooks, and thus workarounds are needed. The spreadsheet is imported in a table format, so it is cannot easily be used outside that context, and even in that context, alignments and cell-padding may require further work.

Merge from QP (Caution!)

WP can merge from certain types of QP spreadsheets under certain conditions, but unfortunately, there remain flaws in the process. In most cases, these will run best if the QP file is a single sheet, rather than a complex notebook, if no other operations are importing from spreadsheets, etc. However, because of a large number of problems, I have concluded that "one cannot reliably merge data directly from QP to WP."[4] See WPU 35226. Fortunately, there are workarounds.

Quirks, Flaws, and Bugs

Importing a QPW file with a certain number of rows somehow forces WP's merge functions to assume that the next QPW file to be merged will have the same number of rows, whether it actually has more or less (thereby either merging less than all records or extra blank records). See WPU 35226.

There are occasionally mysterious "invalid data in the input file" reports. See WPU 36400, citing WPU 15426, WPU 24008, and WPU 29342.

[4]But see the "Caveat to the caveat" at the start of this chapter.

Occasionally, after running one good merge, WP will crash on running the next. See WPU 35761, citing WPU 35090 and WPU 29713.

Often, WP retains the font from QP in the first cell in the merged document. See WPU 36211 and WPU 36806.

Workarounds

- Save relevant portions of complex files to single QPW sheets, and convert those into DAT files. The downside is that two additional files have to be created for something that should require neither.
- Save relevant portions of complex files to some other intermediate format. See WPU 36480, referring to clipboard as preferred (citing WPU 36400), but suggesting that one save in WB3, Lotus WK1, XLS, XLSX, Paradox, comma-delimited, or tab-delimited, citing WPU 35226.
- Place the relevant portions of complex files in the clipboard, and then merge from the clipboard. See WPU 36286 and WPU 36400, both citing WPU 34097.
- Devise a macro that places most things directly from QP into WP.

In short, the workarounds for these problems are essentially to use other methods documented here.

Summary of Pros and Cons

Pros: This is the simplest in concept, and it really, really, really ought to work.
Cons: It is unreliable. Use a workaround.

Merge from QP via a different format

Since merging directly from .QPW files is problematic, the question becomes whether the user can export data from QP to some other format that WP can accept. Our candidates are WP's .DAT files and other spreadsheet, database, or delimited text formats. We need to extract the data from complex QP sheets to single-purpose sheets and ensure that the numeric formats are correct, because WP merges them in the format it finds them.

Merge directly from other formats

I have not experimented at length with merging directly from the other formats, but in limited experiments, these appear to be the major options, all of which can be created by `{FileSaveAs}`.

- QP's older .WB3 and Lotus's .WK1 and .WKS formats merge every row (even if the top row is a header rather than data). Formatting of the WP form file controls everything except numeric formatting and text color. I recommend these, and if the header row in QP is not to be merged, it should not be in the data exported to one of these formats.
- Text formats like tab-delimited .TXT and comma-delimited .CSV files are not always interpreted correctly; sometimes they are interpreted as one single block of data to merge. When that error is avoided, the merge conflates some or all items on the same row in tab-delimited files, and it seems to skip the last row of comma-delimited files.

- Paradox .DB does not merge the top row (at least if it is a header row), but it converts numeric formats to other formats, which defeats the purpose.
- Excel .XLS and .XLSX files reformat dates to add times and change the default font size of the merged data. .XLS files had to be resaved in Excel in order to be usable.

A typical PerfectScript macro merging a WB3 file identified in cell A:A1 into a form file identified in call A:A2 might look like Table 22.2.

Table 22.2: Merging with a QP .WB3 file

```
Application (WordPerfect; "WordPerfect"; Default!; "EN")
Application (qp; "QuattroPro"; Default!)
vDataFile=qp.GetCellValue("A:A1")
vFormFile=qp.GetCellValue("A:A2")
MergePageBreak (On!)
MergeRepeat (1)
MergeBlankField (Remove!)
MergeCodesDisplayRun (Show!)
MergeSelect (All!)
MergeRun (FormFile!;vFormFile; DataFile!; vDataFile; ToNewDoc!)
```

Merge from other formats via WP .DAT files

The best format for the data to be merged by WP is WP's own .DAT format. Unfortunately, Corel has not provided a way to save a QP spreadsheet directly into the DAT format. However, many file formats (including numerous formats that QP can create) can be imported into a WP .DAT file:

- Here, tab-delimited .TXT and comma-delimited .CSV text files work best, and because I find .CSV files easier to read, they are my recommendation. The data are assimilated into a .DAT file with no additional modifications, and thus in WP's default format.[5] A PerfectScript macro importing it looks like Table 22.3. The macro obtains the name of the file to import from cell A:A1 of the currently open spreadsheet. The programmer can obviously set the name by other means. And if the first row of data contains field names, the argument for `ImportSetFirstRecFieldnames` should be `Yes!`.
- Data from Lotus's .WK1 and .WKS formats are placed the same way, except that merge inserts a Times New Roman font code at the start of the data. That is easy to delete by macro commands.
- Data from QP's earlier .WB3 format is placed the same way, except that the merge inserts font and font-size codes from the .WB3 file at the start of the data. These are also easy to delete by macro commands. Font color codes from the source data were carried over into the merge.
- Dates in Excel .XLS and .XLSX files were reformatted to add times. In other respects, they merged like .WB3, except that the XLS file had to be re-saved in Excel to be usable.

[5]If the top line of text should supply field names, however, the system must convert them into field names or otherwise deal with them.

Table 22.3: Merging from text files

```
Application (wp; "WordPerfect"; Default!; "EN")
Application (qp; "QuattroPro"; Default!)
vDataFile=qp.GetCellValue("A:A1")
ImportSetFileName (vDataFile)
ImportSetSource (ASCII!)
ImportSetDestination (MergeData!)
ImportSetSizeToFit (No!)
ImportSetFirstRecFieldnames (No!)
ImportSetAsciiFieldDelimiter ("")
ImportSetAsciiRecordDelimiter ("")
ImportSetAsciiStrip ("")
ImportSetAsciiEncap ("")
ImportSetMacroVariableName ("")
ViewDraft (Yes!)
ImportDoImport ()
```

- Paradox .DB would get the nod here, because it converts the top row to field names automatically (at least if that is what they are in the source), but it converts numeric formats to other formats, which defeats the purpose.

A typical PerfectScript macro that imports data from a block set in cell A:A2 in a spreadsheet identified in cell A:A1 (in any of the spreadsheet formats identified above) might look like Table 22.4.

Table 22.4: Merging from other spreadsheets

```
Application (wp; "WordPerfect"; Default!; "EN")
Application (qp; "QuattroPro"; Default!)
vDataFile=qp.GetCellValue("A:A1")
vBlock=qp.GetCellValue("A:A2")
ImportSetFileName (vDataFile)
ImportSetSource (Spreadsheet!)
ImportSetDestination (MergeData!)
ImportSetRange (vBlock)
ImportSetMacroVariableName ("")
ViewDraft (Yes!)
ImportDoImport ()
```

A downside to doing all of this is the potential proliferation of files to perform what should be a simple merge. We can start with (1) a large complex QPW, from which we extract (2) a single-purpose QPW file, which we convert to (3) a tab-delimited text file (or some other middle format), which we import into (4) a DAT file, which we merge with a form file, to create (5) a new WP document. Until Corel incorporates a way to convert QPW text directly into a DAT file, we cannot skip between files (1) or (2) directly to (4). The only way to save a step in going reliably from a large .QPW file to a .DAT is to skip step (2), and when we create the single-purpose spreadsheet, instead of saving it as a QPW file, we save it as a .CSV file or in some other format.

Merge from QP via WP .DAT files

It has recently come to my attention that one can create a .DAT file by copying a block from QP into a new WP document, placing the cursor inside the table, and using the menu Table > Convert, which gives the option to convert the table into a .DAT file and the further option to use the first row as field names.

To automate this process, one copies the block of cells in QP, opens a new file in WP, pastes the data, which creates a table in WP. Since the cursor is then outside the table, the macro must cause it to move inside the table; a `PosLineUp` command should do. At that point, the macro command `TableDeleteTable (ConvertToMergeNames!)` will convert the table into a .DAT file that uses the first row of the table as field names. If the first row contains only data, not field names, then use instead the command `TableDeleteTable (ConvertToMerge!)`. As recorded, the macro adds two lines that will disambiguate this process and make the resulting .DAT file easier to read: `MergeFileType (TextData!)` and `ViewDraft (Yes!)`.

The resulting file contains stray font codes, but those can be cleaned up by appropriate macro commands as well.

Summary of Pros and Cons

Pros: As long as one stays in the prescribed paths, one can reliably create intermediate files with proper numeric formats, which can be reliably merged into WP. Though complex to do manually at each stage, PerfectScript macros can automate the process.

Cons: This requires is a cumbersome process to extract a block of data from a complex QP notebook in order to create a .DAT file or other file that WP can reliably merge.

Merge from QP via the clipboard

By contrast with the other merges, this is simple and straightforward. The macro in Table 22.5 navigates to the block in question, copies it with `{EditCopy}`, and then runs a PerfectScript macro that merges the clipboard with the desired form file.

Table 22.5: Merging from the clipboard - Selecting QP data

	A
1	{EditGoto "Data:A1..E1000"}
2	{EditCopy}
3	{PlayPerfectScript "C:\WPMacros\MailMerge.wcm"}

To customize it, you would substitute your range of data in the first cell, perhaps using a function that identifies the block, such as `@OFFSET(Data:A1,0,0, @COUNT(A1..A1000),5)`. You would identify the PerfectScript macro in the third cell. I also to add other commands that end the selection of the entire data block.

The PerfectScript macro would then look like Table 22.6, with your form file specified in the last line.

One potential problem is that some undesired font attributes in the QP file are carried over into the newly merged file. It may be possible to deal with those by

Table 22.6: Merging from the clipboard - Selecting QP data

```
Application (wp; "WordPerfect"; Default!; "EN")
MergePageBreak (State: On!)
MergeRepeat (NumberToRepeat: 1)
MergeBlankField (State: Remove!)
MergeCodesDisplayRun (Display: Show!)
MergeRun (FormFile!; "C:\MyForms\Envelope.frm"; Clipboard!; ; ToNewDoc!)
```

later PerfectScript commands, but a workaround was noted in WPU 38051: namely, to paste the cells first into a newly created blank QP file; recopy them from there; and then proceed with the merge.

Summary of Pros and Cons

Pros: This method is quite simple to accomplish and it reliably merges data.
Cons: Apart from the minor inconvenience of navigating to the desired block in the source QPW file, merging from the clipboard places font and font size codes from the source file at the place where merging begins, and if a cell in the source file has unusual formatting codes (such as color, italics, bold, font, or font size), merging from the clipboard places that formatting into the resulting document. The programmer will likely need to employ one workaround or another to eliminate any undesired codes.

Perfectscript macro alone (Document assembly)

Finally, a PerfectScript macro can be written to transfer directly into a WordPerfect file from the source QP file. This is the most complex, but also the most flexible and powerful of methods of transferring QP data into WordPerfect. I have used such a macro (ZenasMerger.wcm) for document assembly in *Zenas*, my QP-based case management system, though I now assemble documents largely in QP or LaTeX. A simple document assembly system was provided in the attachment to my post at WPU 38062.

As noted in the introduction to the topic above at page 235, PerfectScript obtains values from a QP notebook by the `GetCellValue` command, and it runs QP functions by the `Eval` command. The macro would navigate to the correct places in the WP file and get the correct data from the QP file, modify that content or its format in any desired way, and then type it into the WP file.

In that connection, the programmer should note that PerfectScript and QP handle numbers differently. `GetCellValue` gets numbers without numeric formatting. They use different date numbering systems. Thus,

- Instead of returning the number *$100.00* from QP, it returns *100*.
- Instead of returning the date *04/06/15* from QP, it returns the number *42100*
- To convert a QP date number to the number that PerfectScript would evaluate as referring to the same date, add 693593. Thus, QP numbers April 6, 2015 as day 42100, but PerfectScript numbers it 735693. To use WP's better date formatting, the command `DateString(qp.GetCellValue("C4") +693593;Long!;"MMMM dd, yyyy")` makes the conversion. See Kenneth Hobson's elaboration at WPU 36843.

- PerfectScript may interpret a cell's content as text rather than a number. If you need for it to be treated as a number, use a technique for making PerfectScript interpret it that way. I use a technique like prefacing it with a mathematical statement like this:

```
vNumber = 0 + qp.GetCellValue("A:A1")
```

Here are some observations on how to use this method as a generic document assembly tool. I draft a WordPerfect form file with markers for the macro to find and interpret. The markers are carets (ˆ) on both sides of an instruction. The macro looks for one, gets the instruction between it and the next caret, performs the instruction, and then loops to look for the next one, until all instructions have been performed.

- If the instruction is a cell in the active QP file, the macro deletes the instruction, gets the content of that cell, and types it into the same location. Thus an instruction like *ˆData:A1ˆ* would be replaced by whatever is in cell *Data:A1* in the current QP file.
- If I want to get a formatted numeric value from a cell in the QP file, I precede the instruction with # to signal the macro to return the formatted numeric value. An instruction like *ˆ#Data:A1ˆ* would cause QP to convert the numeric value in Data:A1 into formatted text in another cell by using one of the data conversion commands (see page 68). It would place the resulting text from the other cell into the text of the WordPerfect document.
- Other special functions can be written between carets, and the macro would return other data or perform other functions. Thus, the instruction *ˆDATEˆ* would place today's date at that location. The instruction *ˆ?Messageˆ* would display the *Message* and ask for the user's keyboard input. Any process or procedure that can be coded in PerfectScript could be invoked. PerfectScript can also cause QP to execute one of its native macros with the `ExecMacro` command and use the results.

Summary of Pros and Cons

Pros: This is the most powerful and flexible. It can provide general document assembly functions.
Cons: This requires the most programming.

How to avoid or remove stray formatting codes in WordPerfect

Some of the methods discussed in this chapter will place unwanted formatting codes from the spreadsheet into the WordPerfect text. Apart from choosing methods that do not do so, there are several strategies for avoiding or removing them.

In WordPerfect, one can use Reveal Codes to locate the codes and manually delete them. Or one can write a macro (perhaps the continuation of a macro) that automates a search and replace for those codes. Replacing them with the null string ("") will delete them. Or one can write a macro that selects the entire file and applies the same formatting to everything.

On the front end in QP, one can apply the desired font, font size, and font attributes to the cells in the notebook, so that they will blend in the WP file. Or one can clear all font formats from the data in QP so that none are transferred by removing them manually (Edit > Clear > Format) or by the `{ClearFormats 0}` macro command.

Chapter 23

Launching, Linking, and Other Useful Macros

This section collects macros that I found too valuable to omit, but that don't fit into other major categories.

How to run Windows system commands: {Exec}

`{Exec` *`CommandLine,WindowMode`*`}` is used to run external commands, and those include the commands available at the command prompt (formerly known as the DOS prompt) or the Run utility of Windows. `CommandLine` is a command that can be included within quotation marks, and therefore, it should not have internal quotation marks. It must not exceed 100 characters. `WindowMode` is the number 1, 2, or 3, which run the external command in a window of normal, minimized, or maximized size, respectively.

Therefore, to copy a particular file to the D drive (typically a CD or other storage device), the relevant command would typically look like this:

```
{Exec "C:\Windows\system32\cmd.exe /c copy
C:\MyFolder\MyFile.qpw D:\",2}
```

The switch /c causes the command console to disappear when the command has fully executed. This macro command executes the same command that one can execute at the command prompt by typing:

```
copy C:\MyFolder\MyFile.qpw D:\
```

Anything that can be executed from the command line can be run by this means. Thus, running a program would have this form:

```
{Exec "C:\PathToMyProgram\MyProgram.exe",2}
```

Limitation. The functionality of `{Exec}` is limited by the number of characters (100) and the impermissibility of quotation marks inside the `CommandLine` argument. There are several ways to circumvent this problem.

Composing the command line from other cells. The functionality of this command can be extended further by introducing variable elements into it. For example, if instead of hard-coding the file name in the first example, you write it in cell A1, then this formula creates a macro command that accepts the contents of cell A1 as the file to copy:

```
+"{Exec "&@Char(34)&"C:\Windows\system32\cmd.exe /c copy "&A1&"
d:\"&@Char(34)&",2}"
```

Note that the original double quotes have to be replaced with `@Char(34)` in order to appear in the result of the string formula.

You can break it up further by placing other elements into other cells. So, if the command line program components were placed into cell A2, the last example would reduce to:

```
+"{Exec "&@Char(34)&A2&"copy "&A1&" d:\"&@Char(34)&",2}"
```

Alternatively, you can put the entire command as text into one cell and reference that cell in the `{Exec}` command. Thus, if you put `copy C:\MyFolder\MyFile.qpw D:\` into cell A1, this macro command will do the same thing:

```
{Exec +A1,2}
```

That functionality can be extended by using functions or macros to change variables so that the command in `A1` changes.

Writing batch files to be executed. Using techniques for writing text files that are explored in the Chapter on exporting to text files (see Chapter 21), a QP macro could write a batch file that transcends these limitations. This batch file could then be run by a command like:

```
{Exec "c:\windows\system32\cmd.exe /c c:\temp\temp.bat",2}
```

Knowledge of batch file programming helps extend QP's capacities. This will not be a comprehensive look at such programming, but will give the user some ideas.

How to launch non-QP files and emails from QP

The command prompt instruction `start` launches data files in their default program.[1] Typing `start /?` at the command line will list its various parameters, but the two mandatory arguments appear to be a title for the window (usually two double-quotes will do) and the file.[2]

• So, to open `c:\temp\temp.txt` in Notepad.exe or whatever default text viewer that you use, I would recommend doing something like this (as one among many ways to compose the total command line).

First, place the command line `c:\windows\system32\cmd.exe /c start` into one cell, say A1. Second, place the file or commands to be launched in another cell, say A2. Here it would be `c:\temp\temp.txt`. Third, place a function that joins these units in the final product in another cell, say A3. Here, the function would be:

```
@Concatenate(A1," ",@Char(34),@Char(34)," ",A2)
```

Finally, the macro command can be created. It is simply this:

```
{Exec +A3,2}
```

[1]An alternative way to do this, using PerfectScript, is discussed at 240. See also Kenneth Hobson's discussion of this subject at WPU 31079.

[2]In Windows 10, if not earlier, the `start` command apparently no longer requires that the title be entered.

•An email can be sent from the command line using the `mailto` protocol. Various elements can be compiled in various ways. In this example, we send an email to `friend@somewhere.com` with the subject line `Hello`. Using the same system just mentioned, place this command into cell A2:

```
mailto:friend@somewhere.com?subject=Hello
```

Running the same command will launch your default email client, with a new email addressed to that person with the intended subject line.

•The `mailto` protocol does not support attachments. To send an email from QP with an attachment, using Thunderbird as an email client, I must compile a command from (1) the Thunderbird command line, (2) the `-compose` switch, (3) a `to=` address, (4) a `subject=` subject line, and (5) an `attachment=` file attachment, with items 3 through 5 separated by commas but without spaces. Similar techniques may be necessary to utilize other email clients.

How to generate lists of files by extension, beginning at a certain date, in a folder and its subfolders

In this example, I want to get a list of all .WPD and .PDF files in a particular folder (`c:\cases`) and its subfolders that have a file date on or after July 1, 2017. I want it to list them first by date and then by file name, for further viewing or processing. First, we'll create a batch file (`c:\temp\temp.bat`) that will contain commands that direct the Windows system to write the desired files to a text file (`c:\temp\temp.txt`).

The batch command `forfiles` provides a quick way to generate the list. Consult the command `forfiles /?` to get all of the parameters. Here, I need to execute two DOS commands, which I'll put into cells A1 and A2, respectively.

```
forfiles /P c:\cases /M *.pdf /S /D +2017-07-01 /C "cmd /c echo
@fdate @path" >\temp\temp.txt

forfiles /P c:\cases /M *.wpd /S /D +2017-07-01 /C "cmd /c echo
@fdate @path" »\temp\temp.txt
```

To explain the commands, starting from the end, the output is directed to the desired text file. One > creates the file in the first command, overwriting any prior file by that name; the second command uses » to append data to the same file without overwriting it. The command to be executed for each type of file appears in quotes, and the `echo` term simply writes the date of the file (`@fdate`) and its full path and name (`@path`). The `/D+` argument limits the search to files dated on or after the following date. The `/S` parameter finds cases in subfolders. The `/M` parameter filters files by the standard DOS wildcard characters. The `/P` parameter sets the starting path/folder for the search.

To write the batch file and run it is straightforward, as shown in Table 23.1. Run this macro, and the system will generate the list of files in the expected .TXT file. The programmer can then do things with it, such as opening the text file in notepad (see page 225) or import it into the spreadsheet using some variation of the algorithm outlined on page 205.

Table 23.1: Writing and running a batch file

	B
1	{Open "c:\temp\temp.bat",W}
2	{Writeln A1}
3	{Writeln A2}
4	{Close}
5	{Exec "c:\windows\system32\cmd.exe /c c:\temp\temp.bat",2}

How to get Windows environmental variables into QP

In a thread at WPU, a user asked how to get the environmental variable USERPROFILE in a way so that QP could use it. See WPU 37930. There appears to be no direct way for QP to get it, which is not surprising, because the Windows system has changed a lot since QP became a windows-based program, but QP has not attempted to develop commensurately.

Here are two ways:

1. Use a perfectscript macro that is accessed by the {PlayPerfectScript} command. At some appropriate point, the perfectscript macro puts the environmental variable into a QP cell by a {Let} or similar command. Table 23.2 places the UserProfile variable into the cell A1 on the currently active QP sheet.

Table 23.2: Placing an Environmental Variable into A1 using PS

```
Application(qp; "QuattroPro"; Default!)
vEnviro=EnvVariableGet("USERPROFILE")
Let("A1";vEnviro)
```

2. Use the {Exec} command to run DOS commands that get the data. Table 23.3 shows how to do this.

Table 23.3: Placing an Environmental Variable into A1 using QP

	A	B
1		{Exec "c:\windows\system32\cmd.exe /c echo %userprofile% >c:\temp\enviro.txt",2}
2		{Wait @now+@time(0,0,1)}
3		{Open "c:\temp\enviro.txt",R}
4		{ReadLn A:A1}
5		{Close}
6		{Exec "c:\windows\system32\cmd.exe /c del c:\temp\enviro.txt",2}

B1 runs the command that places the environmental variable into a temporary file. B2 pauses for one second, to prevent QP from attempting to read the temporary file before it is closed. B3 opens the file, B4 places its contents into cell A1, B5 closes the file, and B6 deletes it.

Hyperlinking: {Comment.EditURL}

{Comment.EditURL *ExternalTarget,Text,InternalTarget,RelativePath?*} is QP's rather unlikely way of creating a clickable link in a cell. The documentation is also scant.

ExternalTarget is the external file to be activated. If it is a web page, one's default browser is activated. If it is a program, the program is activated. If it is a data file, it is opened in the default program for that type of file. If you wish to link to a target in the QP file, enter two double quotes ("") here.

Text is the text that will appear in the cell, normally colored in blue and underlined, the default appearance for a hyperlink.

InternalTarget is the cell in the current spreadsheet to which to link. The help file says that you can also link to a "bookmark," but that term is not otherwise used in QP. Presumably, a "bookmark" refers to a named range, because entering a named block as the third argument and clicking on it navigates to that block. You cannot jump to a cell in a different spreadsheet by entering it here.[3]

RelativePath? If you link to an external file by filling in the first argument, entering 1 here allows you to use relative paths, and 0 requires the absolute path. Which is the default? Assuming that the macro behaves in the same way as the dialog box that appears when one clicks Tools > Hyperlink, the default is Relative (1).

How to make a launchable list of all files in a folder

Here's another way to create launchable files. The macro in Table 23.4 generates a list in the A column of all desired files in a given folder. The folder (path) is stored in cell B1, and the file filter is stored in B2. Be sure that the folder in B1 ends with a backslash. The *.* filter in B2 will cause the list to include all files.

Table 23.4: Making a launchable listing of files

	A	B
1		c:\temp\
2		*.*
3		{GetDirectoryContents A1,(B1&B2)}
4		{SelectBlock A1}
5		{If []c(0)r(0)=""}{Home}{Quit}
6		{Comment.EditURL (B1)&c(0)r(0),(B1)&c(0)r(0)}
7		{Down}{Branch B5}

The `{GetDirectoryContents}` command in B3 causes all files meeting the criteria in B2 that are in folder B1 to be listed in column A, starting as A1. This command yields only the file name and its extension, not the path that is in B1.

The command in B4 places the cursor in A1. Then the loop begins in B5, which tests whether the current cell is blank, and if so, the cursor goes back to A1 and the macro stops. Otherwise, the macro proceeds to the next step.

[3]Table 23.10 shows a macro that takes a cell address in a different, open spreadsheet and navigates to it.

The {Comment.EditURL} command in B6 creates the hyperlink from each listing in column A. Both arguments in this command are identical, (B1)&c(0)r(0), which combines the path and the file name to create the full name of the file. The first of the arguments is the link to the file in the B1 folder, and the second is what this cell should display. This will have the effect of causing each file name in column A to include the full path.

Finally, the commands in B7 drop the cursor to the next cell below and loop back to B5, to modify each cell in column A in the same way. The result is that each file is a hyperlink that can now be clicked to have the operating system execute that file.

By the way, if the files (or web sites) in the A column contained the full path (or URL), B6 could be replaced by {Comment.EditURL (c(0)r(0))}.

How to make a table of contents to sheets in the QP notebook

This article illustrates two different kinds of Tables of Contents. The first creates the table based on sheet letters (A, B, etc.). The second creates a table of sheet names.

1. The macro in Table 23.5 creates a list of sheet letters from A to Z in cells A1..A26, and each of those can be clicked to jump immediately to cell A1 on the sheet chosen by letter.

Table 23.5: Making a table of contents by sheet letter

	A	B
1		Row counter/Sheet Letter
2		{For B1,65,90,1,B4}{Home}
3		
4		{SelectBlock @OFFSET(A1,B1-65,0)}
5		{Comment.EditURL "",@CHAR(B1),(@CHAR(B1)&":A1")}

The command in B2 runs the {For} loop that fills cells A1..A26. It stores the row counter in B1, but the counter runs from 65 to 90. The reason for these odd numbers is that @CHAR translates them into the capital letters A to Z.,which is the desired result.

The command in B3 places the cursor into the A column, starting at offset 0 (65-65).

The {Comment.EditURL} command in B4 uses nothing for an external link (the first argument), uses the Sheet letter for display in the second argument, and then combines that Sheet letter with the cell A1 in the third argument, which provides an internal link to cell A1 on the given sheet.

The result of the loop is a list of sheets A through Z in A1..A26, and clicking any of those cells takes the user to the page displayed there.

2. But what if you want to create a Table of Contents based on user-given page names for each of your sheets? The following macro (Table 23.6) assumes that you have named each sheet following the first one. The first sheet is named ToC (for Table of Contents), and cell A1 contains the words “Table of Contents.”

The {For} loop in B1 starts with an offset of 1, which will fill in the second line

Table 23.6: Making a table of contents by sheet letter

	A	B
1		{For Macro:A1,1,100,1,Macro:B3}{Home}
2		
3		{If @iserr(@PageName(Macro:A1))}{ForBreak}
4		{SelectBlock @Offset(ToC:A1,Macro:A1,0)}
5		{Comment.EditURL "";@PageName(Macro:A1);(@PageName(Macro:A1)&":A1")}

of the Table on the ToC sheet, and which will fill it in from the second sheet in the notebook. If there are more than 100 named sheets, the second argument should be increased.

The test in B3 causes the `{For}` loop to terminate if it reaches a sheet without a user-given name, which causes the `@PageName` function returns an `ERR`.

B4 then selects the cell of the Table of Contents in which to make the entry, and the `{Comment.EditURL}` command in B5 places a clickable link in cells A2 and below to cell A1 on each named page.

How to name sheets for each month: {Page}

The `{Page}` command allows the macro writer to set details about the active worksheet. Thus, the command `{Page.Name NewName}` would set or reset the name of the current worksheet to `NewName`.

Table 23.7 shows a quick way to name a series of pages after the months of the year. First, fill `A1..A12` with the short names of the month by selecting those cells, and using Edit > Fill > QuickFill | MonthsShort. Then, fill in B1..B4 as shown in the diagram.

Then run the macro in B1. Doing so will name the current page with the first entry in `A1..A12`, then go to the next page, and repeat the process for all twelve entries.

Table 23.7: Naming sheets Jan … Dec

	A	B
1	Jan	{For C1,1,12,1,B3}
2	Feb	
3	Mar	{Page.Name @INDEX(A1..A12,0,C1-1)}
4	Apr	{Ctrl+PgDn}
5	May	
6	Jun	
7	Jul	
8	Aug	
9	Sep	
10	Oct	
11	Nov	
12	Dec	

How to list cells that match search terms and navigate to them.

In WPU 37718, I offered a spreadsheet that (1) list cells is a given block, in the same spreadsheet or another, that contain text matching user-given search terms, (2) that displays the text, and (3) that allows the user to jump to the cell. Apparently, other spreadsheets have this ability already. Here is how to set it up so that it works.

Table 23.8 shows the working part of Sheet A, the interface sheet, where the user enters the block to be searched (B1) and the text to be sought (B2), and where the macro lists the results by location and content below the header at A3..B3. In this example, the macro will search the block `Macs:A1..Z250` in the open `Zenas.qpw` file to find any occurrence of the text `{fileopen`.

Table 23.8: Find-in-file interface

	A	B
1	Search this Block:	[Zenas]Macs:A1..Z250
2	To find this text:	{fileopen
3	Cell	Content
4		

Searching. Table 23.9 is the macro (on a sheet called `Macro`) that searches and returns the results on the interface page. After entering data in A:B1..B2, the user clicks a button that runs this macro.

B2 and B3 determine the number of columns and rows in the search block, and B4 and B7 then run {For} loops that search for matches going down the rows of the first column, then the second, and so on.

In that order, B9 looks to find the search term in each cell on a case-insensitive basis. If it does not find it, the macro proceeds to the next cell.

If it finds it, however, B10 sets the row for output on the interface page, B11 enters the cell in column A, and B12 enters the content of that cell in column B.

Navigating. To go to one of the cells listed in column A, the user places the cursor on that cell and clicks a button, which runs the macro shown in Table 23.10.

D1 places the contents of the current cell in cell C1. D2 conducts error checking to ensure that it is a cell in the A column. If not, it quits (one can add an error message before quitting).

D3 then determines whether the block is in a different file, using the presence of the [] pair. If a different file is indicated, it puts the file's name in C3; otherwise, it blanks C3.

D4 and D5 then isolate the page on which the block is located. D4 places the search block (less file stored in C3) into C4. D5 places the page into C5.

With all of the elements separated, D6..D8 put them together to jump to the desired cell. D6 recalculates cell D8, which will cause the program to jump to the desired page and cell, but D7 first activates the sheet on which that cell is located.

How to redact sensitive data

See David Seitman's excellent macro in the discussion at WPU 34601.

Table 23.9: Macro to find and report matches

	A	B
1		{; startup routines, omitted}
2		{Let Macro:A2,@cols(@@(A:B1))}
3		{Let Macro:A3,@rows(@@(A:B1))}
4		{For Macro:A4,0,Macro:A2-1,1,Macro:B7}
5		{; finalizing routines, omitted}
6		
7		{For Macro:A6,0,Macro:A3-1,1,Macro:B9}
8		
9		{IF @IsErr(@Find(@Upper(A:B2), @Upper(@Index(@@(A:B1),Macro:A4,Macro:A6)),0))}{Return}
10		{Let Macro:A10,@Count(A:A1..A1000)}
11		{Let @Offset(A:A1,Macro:A10,0), @Concatenate(@IndexToLetter(Macro:A4),Macro:A6+1)}
12		{Let @Offset(A:A1,Macro:A10,1), @Index(@@(A:B1),Macro:A4,Macro:A6)}

Table 23.10: Macro to navigate to matches

	C	D
1		{Let Macro:C1,@cellpointer("contents")}
2		{If @IsErr(@Match(C1,A:A4..A1000,0))}{Quit}
3		{Let Macro:C3, @If(@IsErr(@Find("]",A:B1,0)),"", @Mid(A:B1,1,@Find("]",A:B1,0)-1))}
4		{Let Macro:C4, @If(@IsErr(@Find("]",A:B1,0)),A:B1, @Field(A:B1,2,"]"))}
5		{Let Macro:C5,@Field(Macro:C4,1,":")}
6		{Recalc Macro:D8}
7		{Activate (Macro:C3)}
8		+"{SelectBlock "&C4&":"&C1&"}"

How to determine screen size, in twips and cells

David Seitman has another excellent macro for determining the screen size in twips at WPU 35788. I added on the same thread a macro for determining the block of cells that are completely within the screen, under typical conditions.

How to place variable-sized files onto CDs

I provided a macro-based system for this at WPU 35640.

How to merge from a QP database without WordPerfect

In WPU 37705, I offered a little spreadsheet that created a series of twenty letters from a QP database of twenty individuals. It compiles the letters from standard parts and paragraphs that contain variable content. This macro combined all the elements and generated twenty letters in just over one second.

The notebook can be currently found on my website at QPMerge.qpw.

How to save dated versions of a QP notebook

In WPU 38361, a user wanted to know how to have QP save a backup file automatically, as WordPerfect does. The consensus was that QP lacks a similar feature (though it has a timed backup in case of crashes), but third party products may be the best bet for approximating that function.

A slightly different method of doing this, which can be used for other purposes, is to save a current version of the file with the date and time indicated in the file name. The following two macros aim to do that. They could be copied into every file where this feature is desired, but to make it universally available, the instructions below contemplate adding it as a custom macro.

System copy method

1. Click Tools > Customize > Customization > Commands
2. In the dropdown box under `Commands`, click `Macros`. A visually confusing layout appears, but this is where you add macros. The list box on the left contains all macros previously added, and we will want to add a macro to that list.
3. To add a new macro to the listing, click `[Add]`. A new item will appear in the list box to the left. Despite any appearance to the contrary, this item is not yet linked to any macro.
4. At this point, you enter the appropriate macro command in the `Enter Macro` field. I'll explain it below, but at this stage, you could add this:

```
{Exec @CONCATENATE("C:\Windows\System32\cmd.exe /c copy ",
@COMMAND("Notebook.Statistics.Directory"),
@COMMAND("Notebook.Statistics.Filename")," ",
@COMMAND("Notebook.Statistics.Directory"),
@LEFT(@COMMAND("Notebook.Statistics.Filename"),
@LENGTH(@COMMAND("Notebook.Statistics.Filename"))-4),
"_",@YEAR(@NOW)+1900,@IF(@MONTH(@NOW)<10,"0",""),
@MONTH(@NOW),@IF(@DAY(@NOW)<10,"0",""),
@DAY(@NOW ),@IF(@HOUR(@NOW)<10,"0",""),@HOUR(@NOW),
@IF(@MINUTE(@NOW)<10,"0",""),@MINUTE(@NOW),".qpw"),2}
```

   Note that this was all one command on one line — there are no line breaks in this command. If the current file is `c:\Temp\Test.qpw` and the current date and time is June 3, 2017 at 3:00 P.M., this command creates a file called `C:\Temp\Test_201706031500.qpw` by constructing and running the system command:

```
C:\Windows\System32\cmd.exe /c copy C:\Temp\Test.qpw
C:\Temp\Test_201706031500.qpw
```

5. Click `[Apply]` to link that macro command to that item in the dropdown list. At this point, you could drag the command to a toolbar, but I'll leave it out of sight.
6. Click the `Shortcut Keys` tab.
7. In that tab, click in the edit box under "New Shortcut Key:"
8. Click your preferred key combination. In this example, I would suggest `Ctrl+Alt+Shift+S`. Those characters will appear in the edit box.
9. Then click the `[Assign]` button.

The assignment has now been made, so exit out of the dialog boxes. Now, clicking `Ctrl+Alt+Shift+S` should save a separate version of the currently active spreadsheet with the date, hour and minute of the file added to the file name.

Double-FileSaveAs Method

Another option for the command in item 4 is the following, which simply uses 2 consecutive `{FileSaveAs}` commands.

```
{FileSaveAs @CONCATENATE(@COMMAND("Notebook.Statistics.Directory"),
@LEFT(@COMMAND("Notebook.Statistics.Filename" ),
@LENGTH(@COMMAND("Notebook.Statistics.Filename"))-4),
"_",@YEAR(@NOW)+1900,@IF(@MONTH(@NOW)<10,"0",""),
@MONTH(@NOW),@IF(@DAY(@NOW)<10,"0",""),@DAY(@NOW ),
@IF(@HOUR(@NOW)<10,"0",""),@HOUR(@NOW),
@IF(@MINUTE(@NOW)<10,"0",""),
@MINUTE(@NOW),".qpw");Confirm;;QPW v9-X8}
{FileSaveAs @CONCATENATE(@COMMAND("Notebook.Statistics.Directory"),
@LEFT(@COMMAND("Notebook.Statistics.Filename"),
@LENGTH(@COMMAND("Notebook.Statistics.Filename"))-17),
".qpw");Confirm;;QPW v9-X8}
```

The first command saves the current file with the Date, Hour and Minute. The second immediately saves the current file with its original name.

Chapter 24

Quattro Pro Macros in PerfectScript

PerfectScript is Corel's macro-writing language for its Office suite, and the macros are saved as .WCM files. This book is not the place to present a primer on PerfectScript, which would be an extensive subject standing alone. Instead, the book assumes that you know how to compose a PerfectScript macro and provides a few finer points for applying it to QP.

Most of QP's commands have PerfectScript equivalents, and most of those equivalents use the same arguments in the same order as the QP command does. The QP help file contains the commands and arguments that are available. Arguments are always separated by semicolons in PerfectScript.

The Application command

In order for the PerfectScript file to use QP commands and data, it will need to include the `Application` command:

```
Application (qp; "QuattroPro"; Default!)
```

PerfectScript macro commands are clustered in four sets: WordPerfect, QP, Presentations, and PerfectScript. This `Application` command tells PerfectScript to look to find a command in the set of commands for one of those particular applications, QP in this case. If it finds the command in that set, it executes it; otherwise, it looks for the command in a different set. Since several commands in QP and WP have the same name, the `Application` command tells PerfectScript which one to execute. If, on the other hand, the `Application` command identifies WordPerfect as the default source of commands, you can still have PerfectScript apply the QP command by adding the prefix `qp.`, which is the first argument in the `Application` command.

The FileOpen and Activate commands

For PerfectScript to interact with data in a QP spreadsheet, the notebook must generally be open. (See discussions of importing and merging QP data into WordPerfect in Chapter 22.) Because `FileOpen` is a command common to QP, WordPerfect, and Presentations, see the discussion of the `Application` command in order to disambiguate them. The help file's discussion of the arguments for `FileOpen` is particularly opaque, but in my testing, the command works by simply specifying the file to be opened:

```
qp.FileOpen("C:\Myfiles\Target.qpw")
```

In my testing, this is the only command required to ensure that the desired file is open. If the file is not open, the command opens it; if the file is already open, the command is basically ignored.

Multiple open notebooks: Bug and workaround. A problem arises when it is possible that more than one notebook is open at the time the PerfectScript macro starts to interact with QP. If the desired notebook is then open and active, there should be no problem. However, if a different notebook is open and active, PerfectScript would by default look for data in it, yielding incorrect results or causing the macro to crash. One would imagine that the problem can be fixed by making the desired notebook the active notebook, and in that connection, QP and PerfectScript also have an `Activate` command, which functions to make one of several open QP notebooks the active notebook. In QP, `{Activate}` works as expected, but the same command in PerfectScript acts inconsistently. That is, if two or more QP spreadsheets are open at the same time, and the one from which data will be taken is not active, PerfectScript will sometimes interact with the correct, but inactive, one and sometimes interact with the active, but incorrect, one. See the discussions at WPU 34443 and WPU 34671.

The workaround for this situation appears to be to use functions and commands that will act remotely (conversely put, that do not require positioning the cursor in the correct file) and that add references to the file. Thus, the solution is *not* to use the `Activate` command.

```
qp.Activate("C:\Myfiles\Target.qpw")                      Wrong!
```

Instead, open the file and use bracketed references to it in commands that do not require positioning in that file, e.g.,

```
qp.FileOpen("C:\Myfiles\Target.qpw")
qp.Let("[Target.qpw]Data:A1";1)
```

Closing QP or a QP Notebook

In order to close the current QP notebook, use the command `qp.FileClose`. It follows that if an open QP notebook is not currently active, the macro must activate it first. If you desire to close all QP notebooks, use `qp.FileCloseAll`.

By default, these commands display a Save-File dialog if any notebook is currently unsaved. To avoid the display, the programmer can add a No! parameter, as in `qp.FileCloseAll(No!)`. A potentially large problem with doing so is that this will simply close the notebooks without saving them. If saving them is required, the command should be preceded by `qp.FileSave (Replace!)` or `qp.FileSaveAll (Replace!)`.

In order to close the QP application itself, use some variation of `AppClose("Quattro Pro*")`. For a discussion of the issues, see WPU 37498.

The GetCellValue command

To get the data in a target cell (say A:A10 of the currently active QP notebook), a `GetCellValue` command like this works:

```
vData=qp.GetCellValue("A:A10")
```

The address of this function can be created by formulas as well. Thus, if the variable `vRow` has the value of 5, this formula would return the same results:

```
vData=qp.GetCellValue("A:A" +(5+vRow))
```

The address of the function can also be created by other functions, which are converted to a text address by the `Eval` function, discussed below.

Designating the correct cell. The programmer must ensure that the cell in question is adequately identified.

`GetCellValue("A1")` works to get the value in cell A1 in the currently active notebook and currently active sheet. If that notebook might not be open, or if it is not active, or if the notebook is not currently open to the sheet containing the desired data, an error will occur.

`GetCellValue("A:A1")` works to get the value A:A1 in the currently active notebook, even if A is not the active sheet. But if the notebook is not open and active, an error will occur.

`GetCellValue("[Notebk1]A:A1")` works to get the value in A:A1 of an open notebook called Notebk1.qpw, even if that notebook is not active. But if that notebook is not open, an error will occur.

Quirk for numbers. If the value contained by the target cell is a number, `GetCellValue` regards the value as ambiguous and will assume that it is intended to be numberlike text. It is odd that PerfectScript will treat the value as ambiguous when QP does not, but we must play the hand we're dealt. On the plus side, that numberlike text may be freely combined with other text. Fortunately, it is not difficult to have PerfectScript recognize the value as a number. One easy way to this is to add a trailing `+0` to the line.

```
vData=qp.GetCellValue("A:A" +(5+vRow))+0
```

Any other mathematical operation on the value should also tell PerfectScript that we want to use the value as a number.

Help file errors. I note in passing that the help file suggests that `{GetCellValue}` is a native QP equivalent of the PerfectScript command, but that is a false equivalence. `GetCellValue` (and `GetCellFormula`) are PerfectScript commands for which there is no QP equivalent. See the discussion at WPU 32603.

The help file erroneously states that this command retrieves the contents of a cell "as it is displayed, not as its value." On the contrary, a cell with particular numeric formatting is returned in general format.

Caution. If the cell contains a string preceded by a backslash, `GetCellValue` crashes QP.

The Eval command

To have the macro use a QP function to give it data from or about the spreadsheet, place the function as text within an `Eval` command. This example calculates the number of non-blank cells in the block A:A1..A10000 and stores that number in the variable `vNumberOfCells`:

```
vNumberOfCells=qp.Eval("@COUNT(A:A1..A1000)")
```

Thus, one could construct the address of the target cell using the PerfectScript equivalent of the native QP `@OFFSET` function:

```
vTargetCell = qp.Eval("@offset(A:A1;" +vRow +";" +vCol +")" )
```

which returns the cell address that is `vRow` cells below A:A1, and `vCol` cells to the right; this is obviously useful in For loops. Then, the value within the target cell can be obtained either in the next line:

```
vData=qp.GetCellValue(vTargetCell)
```

or by combining both steps in one line:

```
vData=qp.GetCellValue(Eval("@offset(A:A1;" +vRow +";"
+vCol +")" ))
```

Remember to add a trailing `+0`, or something similar, to make sure that numeric data in QP will be treated as such in PerfectScript.

Quirk. An oddity of the `Eval` command is that, even if it returns integers, it returns them with a decimal point and a trailing zero, i.e., ".0". The macro-writer will often need to work around this feature.

Importing a QP block into PerfectScript array

Chapter 6 of Kenneth Hobson's online book on QP macro programming contains a function that can be inserted into a PerfectScript macro that would take the coordinates of a block of QP data and place each element into an array called qpRange. That snippet is reproduced in Table 24.1, but I have modified it to add the qp prefix to qp commands. The coordinates of the QP block are passed to the function as qpName, a text string. The following code would bring the designated block of data ("TimeSheet:A8..E11") into a PerfectScript array, and then display item 1,1 (topmost, leftmost), 1,2 (the cell to the right of the first one), 2,1 (the cell below the first one), and 2,2 (the cell to the right of the last one), and so on.

To invoke the macro, one would include the `Application` command for QP somewhere above, and then create the array with a command like:

```
ListArray[]=QPNameValsToArray("TimeSheet:A8..E11")
```

Items in the array could then be utilized by knowing their relative row and column numbers in the array. In the example just given, cell A8 is in the first row and column of the array, so it is identified by `ListArray[1;1]`. Cell E11 is in the fourth row and fifth column, so it is identified by `ListArray[4;5]`.

Running PerfectScript macros: {PlayPerfectScript}, {Exec}

There are two ways to run or "launch" PerfectScript macros from QP:

• First, you can run the PerfectScript macro from QP with the `{PlayPerfectScript` *`WCMFile`*`}` command, where `WCMFile` is the text name of the file to be run.

Table 24.1: Importing QP data into a PerfectScript Array

```
Function QPNameValsToArray(qpName)
   q=""""
   firstCellinRange=qp.Eval("@Cell("+q+"FullAddress"+q+"; "+qpName+")")
   noRows=qp.Eval("@Rows("+qpName+")")
   noCols=qp.Eval("@Cols("+qpName+")")
   Declare qpRange[noRows; noCols]
   ForNext(i;1;noCols)
      ForNext(j;1;noRows)
         s="@Offset("+firstCellinRange+"; "+(j-1)+"; "+(i-1)+")"
         qpRange[j;i]=GetCellValue(qp.Eval(s))
      EndFor
   EndFor
   Return(qpRange[])
EndFunc
```

Unfortunately, you cannot pass additional parameters to the macro, but you can store them in cells on the QP sheet and write the PerfectScript macro to get them. QP waits until the PerfectScript macro ends before executing the next native QP command.

- You can also run a PerfectScript macro (or any other program that can be launched with a command line) using QP's `{Exec` *`CommandLine,WindowMode`*`}` command. `CommandLine` is the text string that one would insert in the Windows Run box, an executable file with any parameters, enclosed in double quotes. `WindowMode` is a number that allows the program to run normal (1); minimized (2), or maximized (3), but by adding 100 to any of these numbers, QP will wait until the external program/macro finishes before executing; otherwise, the QP macro and the external program/macro will work simultaneously. (Caveat: while the QP native macro is running, the PS macro does not appear to be able to use the changing data in the spreadsheet.) To run a PS macro, this might be a typical command:

```
{Exec "C:\Program Files (x86)\Corel\WordPerfect
Office X7\Programs\ps170.exe
/#/m-c:\mymacro.wcm",2}
```

The `CommandLine` argument first identifies the PS command interpreter (this one is for QP17, earlier versions of QP will be slightly different). The `/#` switch prevents PS from displaying its interface. The `/m-` switch identifies the PS macro to run. The `2` runs the macro minimized.

Using ExecMacro to run QP macros from PerfectScript

PerfectScript can also cause QP to run a native QP macro. If that macro is in cell C:B1 of the notebook that is currently open, this command will operate it:

```
qp.ExecMacro(;"C:B1")
```

And if the file is not open, specifying the file name in the first argument will open the QPW file, run the macro, and then close the file.

```
qp.ExecMacro("C:\Spreadsheets\MyData.qpw";"C:B1")
```

Contrary to the help file, there is no parallel native QP command, at least in current versions of QP. Using `{ExecMacro}` causes an unknown command error message.

How to launch non-QP files and emails using PerfectScript

One simple way to launch data files in their default application is to use the PerfectScript command `AppExecute`.[1] One can place the name of the file to be launched by the operating system into a cell, say A:A1, and then call the PerfectScript macro in Table 24.2.

Table 24.2: Merging from the clipboard - Selecting QP data

```
Application(qp; "QuattroPro"; Default!)
FileToExecute=GetCellValue("A:A1")
AppExecute(FileToExecute)
```

This can be generalized by placing the name of the file into a cell that is named something like `FileToLaunch`, and then substituting `FileToLaunch` for `A:A1` in this macro.

Emails can be sent the same way, using the standard syntax for them. Thus, one could send an email to friend@somewhere.com with a subject line of “Hello” by putting `mailto:friend@somewhere.com?Subject=Hello` into the cell named `FileToLaunch` and running the macro.

[1]A comparable method using only QP’s native commands is discussed at page 225.

Index