The [unfinished] LotusScript Book

By Julian Robichaux

June, 2003

Introduction

Welcome to my unfinished book.

About a year ago (early 2002), I decided that I was going to try to sit down and write a book about LotusScript. My motivation was that I hadn't been able to find any books that were specifically written with "advanced" LotusScript programmers in mind, so I was going to try to write something that might have useful information for people who had already worked with the language for a while. Not that I consider myself a LotusScript "guru" or anything, but I figured I had been programming for long enough to be able to put one or two interesting things down on paper.

So, I wrote out a table of contents and started waking up early every day to add a little more and a little more. After several months, I actually had the rough drafts of a few chapters done, and I was making good progress. Then life swooped in and we had another baby, and the next thing I knew hadn't written anything in about 8 months. So I had a choice: should I try to pick up where I left off, or abandon the project altogether?

I decided that before I put any more time into the book, I would try to assess the feasibility of a LotusScript book from a publisher's standpoint – granted, that was probably something I should have done before I ever started writing in the first place, but better late than never. So I started going to bookstores and visiting online book retailers with that in mind, and I saw that most bookstores didn't even have any Lotus Notes books on the shelves in the first place, and the selection at the online retailers was pretty slim.

I finally decided that even if I could convince some poor publisher to print this book for me, that it would never make it to the bookstore shelves, and it would probably never get promoted in any meaningful way, and (in short) it would be a huge failure and disappointment for me. Furthermore, I had started working on some other side projects that I would have to drop or delay in order to finish the book, and I just didn't have the enthusiasm that I once had to finish it up. The thrill was gone.

At that point, I had to decide what to do with what I had already written. It seemed a shame to just delete the files and throw away about 5 months worth of work, especially because I thought that some of the things I had already written might actually help someone else out there. However, I really didn't want to spend any more time on the book, because if I wasn't going to finish it, then I also wasn't going to labor over revisions of the unfinished chapters.

So, what you have here is the raw, unfinished manuscript of what was to become my LotusScript book. I ended up with 5 chapters, plus some random scripts that I was going to put in a "Miscellaneous" chapter or an Appendix or something. If you're curious, here is the original table of contents that I was working from:

- Introduction
    - Who should read this?
    - What is LotusScript?
    - Why use LotusScript?
    - Versions of Notes
    - Where can I get more information?
- Writing in LotusScript

    - Standard events
    - Global variables and routines
    - Built-in functions, classes, etc.
    - Writing your own functions and subs
    - Writing your own classes
    - Script libraries
    - Using an LSX
    - Using a DLL
    - OLE Automation
- Error handling
- Notes function boilerplates
- String functions
- Lists and Arrays
- Numeric functions
- Date/Time functions
- File manipulation
- ODBC
- OLE/COM
- MS Office
- Accessing DLLs
- Windows API
- Notes API
- Algorithms
- Miscellaneous Notes scripts
- Miscellaneous other scripts

Appendix A: @ Function Equivalents
Appendix B: LotusScript Limits
Appendix C: Tips, Tricks, and Pet Peeves

As you can see, my five draft chapters really only puts me about 25% of the way to completion of the first draft of the book, so I still had a lot more writing to do. In addition, my target audience was users of Lotus Notes and Domino version 4.6 and R5, and by the time I started revisiting the chapters, ND6 was on the market and version 6.5 was already in public beta. If I ever did manage to finish the book, then by the time it got published it would be at least 2 releases behind!

So, once I convinced myself that I wasn't going to make any money off the book in the first place, and I would probably never finish it in such a way that it would be publishable, I decided that the proper thing to do was just to give it away.

Luckily, I already had an established website (http://www.nsftools.com), so "publishing" the book to the web wouldn't be a problem. Just plop everything into a PDF file, and put it out on the site. I don't know that I'll ever make any meaningful updates to this manuscript, but if I do you then will be able to find it at my nsftools.com site. If you're a publisher and you're feeling crazy enough to pay me to finish this thing up, please contact me at the site (contact information is on the home page, or on the “About” page).

Everything in this document was written entirely by me (Julian Robichaux), unless stated otherwise in the text. As such, you may not republish (on the web, in a magazine, in a book) any of the material herein without my consent and without giving proper credit to me – sorry, but I spent a good deal of my personal time writing this material, and I’d at least like to get credit for it. You may, however, use the code and information presented here in any programs or scripts that you would like to use it in, free of charge and without permission, although I’d ask that you are kind enough to mention where it came from in the comments of your source code.

In case there are any lawyers reading this, the information presented in this document is provided "as-is", and should be used at your own risk. I make no express or implied warranty about anything, and I will not be responsible or liable for any damage caused by the use or misuse of anything related to this document. I make no guarantees about anything, and some of the information herein might be incorrect. Please thoroughly test all of the information and code that you find here before you attempt to use it in your production environment. Finally, you are not allowed to use this code for the propagation of weapons of mass destruction, nor can you use it in conjunction with harming small animals or forest creatures…these things are right out.

Okay, I think that about covers it. I apologize for the unfinished quality of any of the information that follows; however, you got this for free, and you get what you pay for. I hope you find this document useful in some way.

April 2003
Julian Robichaux
http://www.nsftools.com

A Note About Efficiency and Error Checking

While most of the scripts in this book were designed to be as efficient as possible (from the standpoint of speed), there is sometimes a tradeoff between efficiency and readability. This book is meant to be somewhat of a teaching tool, and as a result, all of the scripts were written with readability as a primary objective. While this does not mean that someone with only a basic knowledge of LotusScript will be able to look at all of the scripts and understand them completely, it does mean that I tried not to skip or combine too many steps, I tried to use complete blocks and evaluation formulas for loops and if-then statements, and I tried to keep the individual lines of script as short as possible. In several cases, the scripts could have been a little shorter and possibly a little more efficient by (for example) combining several lines of script into one, but such practice was avoided when it would have hampered the reader's ability to understand the logic of the script.

On the other hand, error checking was used rather conservatively throughout this book. While I generally believe that every function or subroutine should have an error-handling block, I left out the error block in many of the scripts in order to keep them reasonably short. In cases where error checking or data type checking seemed necessary or important, however, I normally left the checking logic in place.

String Manipulation

You'll be dealing with strings in almost every aspect of your LotusScript programming. Text strings are everywhere. Luckily, the LotusScript language has quite a few built-in functions and statements that allow you to perform a lot of string manipulation natively. A summary of these functions is below.

Function/Statement	Usage	Example
Asc	Returns the numeric ASCII character code for the first character in a string.	
Chr[$]	Returns the character represented by a numeric ASCII character code.	
CStr	Converts a number or date value to a string.	
Format($)	Converts a number, date, or string value to a string of a given format.	
FullTrim	(New in R5) Removes whitespace from a string (trailing, leading, and multiple spaces), or removes empty entries from an array.	
InStr	Returns the position of one string within another string.	
InStrB	Returns the byte position of one string within another string (technically replaced by InStrBP)	
InStrBP	Returns the byte position of one string within another string, using the platform-native character set	
InStrC	(New in R5) Returns the column position of one string within another string, for column-based writing systems such as Thai.	

LCase[$]	Returns a string converted to lowercase.	
Left[$]	Returns the given number of characters from the left (beginning) of a string.	
LeftB[$]	Returns the given number of bytes from the left (beginning) of a string (technically replaced by LeftBP)	
LeftBP[$]	Returns the given number of bytes from the left (beginning) of a string, using the platform-specified character set.	
LeftC	(New in R5) Returns the given number of columns from the left (beginning) of a string, for column-based writing systems, such as Thai.	
Len	Returns the number of characters in a string.	
LenB	Returns the number of bytes in a string (technically replaced by LenBP).	
LenBP	Returns the number of bytes in a string, using the platform-native character set.	
LenC	(New in R5) Returns the number of columns in a string, for column based writing systems such as Thai.	
Like	Returns a Boolean indicating whether or not a string matches a given pattern.	
LSet	Sets a string variable to the leftmost x characters of another string, where x is the original length of the string variable being	

	set.	
LTrim	Returns a string with no leading spaces.	
Mid[$]	As a function, returns a specified number of characters from a specified location in a string. As a statement, replaces a specified number of characters of a string with another string.	
MidB[$]	Returns a specified number of bytes from a specified location in a string (technically replaced by MidBP)	
MidBP[$]	Returns a specified number of bytes from a specified location in a string, using the platform-specified character set	
MidC	(New in R5) Returns a specified number of bytes from a specified location in a string, for column-based writing systems such as Thai.	
Right[$]	Returns the given number of characters from the right (end) of a string.	
RightB[$]	Returns the given number of bytes from the right (end) of a string (technically replaced by RightBP).	
RightBP[$]	Returns the given number of characters from the right (end) of a string, using the platform-specified character set.	
RightC	Returns the given number of characters from the right (end) of a string, for column-based writing systems such as Thai	

	systems such as Thai.	
RSet	Sets a string variable to the rightmost x characters of another string, where x is the original length of the string variable being set.	
RTrim[$]	Returns a string with no trailing spaces.	
Space[$]	Returns a string consisting of a specified number of spaces.	
Str[$]	Converts a number to a string (adding a leading space for positive numbers)	
StrComp	Same as StringCompare	
StrCompare	Compares two strings and returns either NULL, -1, 0, or 1.	
StrConv	Returns a string converted to a different case (upper, lower, or proper-case) or a different character set.	
StrLeft	(New in R5) Returns the part of a string to the left of the first occurrence of a given substring.	
StrLeftBack	(New in R5) Returns the part of a string to the left of the last occurrence of a given substring.	
StrRight	(New in R5) Returns the part of a string to the right of the first occurrence of a given substring.	
StrRightBack	(New in R5) Returns the part of a string to the right of the last occurrence of a given substring.	

String[$]	Returns a string consisting of a specified ASCII character, repeated a specified number of times.	
Trim[$]	Returns a string with no trailing or leading spaces.	
UCase[$]	Returns a string converted to uppercase.	
UChr[$]	Returns the character represented by a numeric Unicode character code.	
Uni	Returns the numeric Unicode character code for the first character in a string.	
UString[$]	Returns a string consisting of a specified Unicode character, repeated a specified number of times.	
Val	Converts a string to a number.	

While this is a pretty formidable list of functions, it should be a good indication of how often you'll be manipulating strings in LotusScript. You should also make sure you review this list of built-in functions before you decide to write a string manipulation routine yourself. For example, before you decide to start writing and debugging a function to simulate the @ProperCase function, keep in mind that the StrConv function will do that natively.

Representations of Strings

LotusScript stores strings internally using 2-byte characters. While you might think that this is just technical nitpicking, it actually comes into play in certain instances. This is why you can have a LotusScript string of up to 64k in size, but you can only have a string length of up to 32k characters (because each character is 2-bytes long). The 2-byte representation is actually the same as the way that Unicode characters are stored. So even though you don't realize it, LotusScript is actually storing strings as Unicode.

So what is Unicode? Well, the "old" representation for characters in strings is ASCII, which is a single-byte representation. Using only one byte for a character only gives you 256 different possibilities (one byte is 8 bits, and 2 to the eighth power is 256). This isn't bad if you're just storing

English characters, but if you want to store characters from many different languages (like English and Traditional Chinese and Japanese), then you'll need a lot more than 256 possibilities.

Unicode is a double-byte representation of string characters, which gives you 65,536 different possibilities (256 x 256). This is much better for dealing with international characters. However, if you're expecting a single-byte character, and you end up getting a double-byte character, you'll have a lot of empty space in your string. Specifically, there will be a null character (Chr(0)) between every letter in the string. For example, the ASCII representation of "A" is 65, but the Unicode representation is 0065, so if you're looking at a Unicode string but you're expecting an ASCII string, you'll see an "A" and a null character instead of just an "A".

So when does this come into play? Well, when you're just dealing with Lotus Notes forms, formulas, and scripts, all the conversion and the storage and the on-screen display happens in the background for you, so it never really matters how the strings are represented. However, if you're reading data from an external source, you can occasionally run into trouble.

For example, if you're reading string information from a file, LotusScript expects files opened in Sequential mode to have ASCII (single-byte) characters, and it expects files opened in Binary mode to have Unicode (double-byte) characters. So if you issue an Input command to get text from a Unicode file that's open in Sequential mode, you'll get a string that has null characters between all the letters. Or if you read ASCII strings from a Binary mode file using the Get command, you'll get a bunch of funny characters, because the single-byte characters are being read as double-byte characters. Conversion functions for both of these situations are provided at the end of this chapter.

Tips, Tricks, and Things to Watch Out For

There are quite a few things you can do to make your string manipulation routines work faster and more efficiently. Especially if you end up working with large strings (10,000 characters or greater), making small modifications in your scripts based on these tips can result in huge performance gains.

Concatenating Strings

String concatenation can be an expensive operation if you have to do it a large number of times, because every time the string grows in size, LotusScript has to reallocate memory to accommodate the new string size. Now, this isn't to say that you should never concatenate strings over and over to create a new string. For example, consider the following script:

```
tempString$ = Space$(10000)

startTime! = Timer()
For i% = 1 To 10000
	Mid$(tempString$, i, 1) = "a"
Next
Print "Time Elapsed: " & Round(Timer() - startTime!, 2) & " seconds"

startTime! = Timer()
tempString$ = ""
For i% = 1 To 10000
	tempString$ = tempString$ & "a"
Next
Print "Time Elapsed: " & Round(Timer() - startTime!, 2) & " seconds"
```

The string concatenation in this case ends up being much faster. This is for two reasons: first, because the Mid function has to perform a search and replace operation in order to create the new string in the first loop, and this kind of repetitive action on a large string ends up taking quite a bit of time. Second, the initial creation of the string (using the Space function) pre-allocated the memory for the large string that we were going to create, which makes the concatenation go much faster.

To see what a difference it makes to initialize a string before performing multiple concatenations, try this script out:

```
tempString$ = ""
startTime! = Timer()
For i% = 1 To 10000
    tempString$ = tempString$ & "a"
Next
Print "Time Elapsed: " & Round(Timer() - startTime!, 2) & " seconds"

tempString2$ = Space$(10000)
tempString2$ = ""
startTime! = Timer()
For i% = 1 To 10000
    tempString2$ = tempString2$ & "a"
Next
Print "Time Elapsed: " & Round(Timer() - startTime!, 2) & " seconds"
```

The second loop runs in about 70% of the time it takes for the first loop to run. If you're going to create a large string using concatenation, and you know the approximate size of the string you'll end up with, you'll often get better performance if you initialize the string you're creating with the Space function before you start the concatenation. When you're dealing with smaller strings or small numbers of concatenations, the difference in time becomes negligible.

Using & vs. + for String Concatenation

The decision to use & or + for string concatenation is somewhat a matter of personal preference, but the "proper" way to do it is to use the ampersand (&). One of the advantages to using & instead of + is that you don't have to worry about converting non-string values when you're adding them to a string. For example, in the two lines of code:

```
Print "Using + gives us" + 1 + 2
Print "Using & gives us " & 1 & 2
```

the first line will give an error, while the second is perfectly legal (and will output "Using & gives us 12"). In order to make the first line of script work, we'd have to enter:

```
Print "Using + gives us" + Cstr(1) + Cstr(2)
```

which is just a lot more typing. Also, if you're adding information to a string using variables, the nature of the & concatenation operator that automatically converts information to strings means that you don't have to worry about the data type of the variable as you're adding it.

Using = vs. StrCompare to Compare Strings

When comparing strings, the equals (=) operator is slightly faster than the StrCompare()function. For example, with the two pieces of code:

```
isMatch% = (string1$ = "some string")
isMatch% = StrCompare(string1$, "some string")
```

the first comparison is faster than the second.

The advantage to using the StrCompare() function is that it has an optional 3rd argument that allows you to specify what kind of match you're testing for (case- or pitch-sensitive). If you use this function, however, be aware that if two strings match, a zero is returned, which is normally interpreted as a boolean False!

If you're checking for a single character instead of an entire string, then it's much faster to compare using the Asc() function, instead of comparing strings. With the two pieces of code:

```
isMatch% = (letter1$ = "A")
isMatch% = (Asc(letter1$) = 65)
```

the second comparison is a great deal faster than the first. However, when comparing the two lines of code:

```
isMatch% = (letter1$ = "A")
isMatch% = (Asc(letter1$) = Asc("A"))
```

the difference in speed is negligible. So if you're comparing the value of a single character against a constant ASCII value, you should convert the character using the Asc() function and compare it to the numeric ASCII value. If you're comparing two characters that may vary in value, you can just compare using an equals sign and save yourself all the conversions.

Using the $ Version of Functions

Many string functions have two versions: one that returns a variant string and one that returns a string (indicated by a $ at the end of the function name). In my testing, there is generally no speed difference between the two versions of these functions to produce a string. For example, comparing the two lines of code:

```
leftString$ = Left(string1$, 100)
leftString$ = Left$(string1$, 100)
```

the second line is approximately the same speed as the first, despite the fact that there is an implicit conversion from a variant to a string value in the first operation.

Using Instr vs. Stepping Through the String

As you might expect, the fastest way to find an occurrence of a string within another string is by using the LotusScript Instr function. No surprises there. However, there is a special case when it can actually be faster to step through a string character by character, rather than using several Instr calls. Consider the following piece of script:

```
tempString$ = Space$(30000)

startTime! = Timer()
pos% = Instr(tempString$, " ")
count% = 0
Do Until (pos% = 0)
    count% = count% + 1
    pos% = Instr(pos% + 1, tempString$, " ")
Loop
Print "Instr Time Elapsed: " & Round(Timer() - startTime!, 2) & " seconds; " & count%

startTime! = Timer()
strLen% = Len(tempString$)
count% = 0
For i% = 1 To strLen%
    If (Asc(Mid$(tempString$, i%, 1)) = 32) Then
        count% = count% + 1
    End If
Next
Print "For Time Elapsed: " & Round(Timer() - startTime!, 2) & " seconds; " & count%
```

In this case, it's actually faster to step through the string to find all the occurrences of the space character than it is to perform multiple calls to Instr. Why? Because Instr is always doing a string comparison to find the character, which (as we discussed before) is much slower than doing an Integer comparison of the ASCII value of a character. So in this specific example, where you need to find a specific character multiple times in a large string that contains many instances of that character, it's actually quicker to step through the string character by character.

This may seem like far too unusual of a situation to even be worth mentioning, but it's actually come up for me on two different occasions. On the first, I had to write a script that counted the number of lines in a text file. The fastest way to do this (for large files) was to read large pieces of the file into a string and count the number of Chr(13) characters in the string. On the second occasion, I had to parse very long strings with single-character delimiters. Using this technique gave me some performance benefits in both instances.

Using Recursion with Large Strings

Using recursion to perform operations on large strings can sometimes give you huge performance benefits, although it can also sometimes slow you down (or crash your program). Several of the functions in this chapter will show you instances when it might be good (or bad) to write a recursive function.

There's no general rule as to when recursion will work well and when it won't, because it's different for every situation. If you do decide to use recursion, make sure you do a lot of testing for extreme circumstances, so that your function doesn't run out of stack space during heavy operations (many functions run out of stack space after 50 to 100 recursions, depending on the function).

Using || or {} as String Literal Delimiters

Normally when you assign a value to a string literal, you enclose the string in quotation marks. For example:

```
myString$ = "This is my string. It looks like many other strings. But this one is
mine."
```

You could also perform the same assignment using vertical bars (||) or braces ({}) as the delimiters.

```
myString$ = |This is my string. It looks like many other strings. But this one is
mine.|
myString$ = {This is my string. It looks like many other strings. But this one is
mine.}
```

This can come in handy in two cases. One is when you are creating a string that has a lot of quotation marks within it. If you use a vertical bar or a brace, you avoid the need to use double quotation marks within the string. For example:

```
myQuotedString$ = |This is a "quoted" string.|
```

This is especially good to use if you' re using the Evaluate statement, which often requires you to pass several sets of quotation marks within the formula you want to evaluate. The second case where the "alternate" delimiters are useful is when you are creating a string with many carriage returns in it. If you use vertical bars or braces to delimit a string, you can embed carriage returns without having to concatenate Chr(13) & Chr(10) characters all over the place. For example:

```
myMultiLineString$ = {This string has multiple lines.
Here's the second line.
And the third.}
```

This technique is good for writing Javascript to a Web page, or for writing multi-line statements to a form field.

Custom Routines

The rest of this chapter will consist of custom subs and functions that will demonstrate ways to manipulate strings.

Repeat Function

This will end up being a kind of obscure example to start the chapter off with, but we might as well jump in feet first. The obvious way to code a function that mimics the @Repeat function is this:

Script – Simple (slow) Repeat function

```
Function RepeatSimple (text As String, numTimes As integer) as String
  Dim tempString As string
  Dim i As Integer

  For i = 1 To numTimes
      tempString = tempString & text
  Next

  RepeatSimple = tempString
End Function
```

Add a little error handling, and you' ve got a perfectly serviceable script. However, it' s really slow when you start repeating your characters a large number of times. Is there a better way? Of course there is:

Script – Very fast Repeat function

```
Function Repeat (text As String, numTimes As Integer) As String
  '** EXTREMELY FAST
  '** This function repeats the text string the specified number of times.
  '** For example, Repeat("abc", 3) = "abcabcabc"
  '** While the code for this function may seem overly complex, it is much
  '** more efficient (faster) than simply calling:
  '**    tempstring = tempstring & text
  '**  when dealing with large numbers of repetitions.
```

```
 On Error Goto processError

 Dim tempString As String
 Dim binLength As Integer
 Dim leftOver As Integer
 Dim i As Integer

 If (text = "") Or (numTimes < 1) Then
     Repeat = ""
     Exit Function
 End If

 '** The most efficient way to create our resulting string is to keep
    '** doubling tempString until we're close to the string length that we
    '** want, then adding any remaining repetitions to the end. The number
    '** of times we can double the string without going over is the power
    '** of 2 represented by the most significant 1 in the binary
    '** representation of numTimes.
 binLength = Len(Bin(numTimes)) - 1
 leftOver = numTimes - (2 ^ binLength)
 tempString = text

 '** Double the string as many times as we can
 If (binLength > 0) Then
     For i = 1 To binLength
         tempString = tempString & tempString
     Next
 End If

 '** Then add any remaining repititions by making a recursive call
 If (leftOver > 0) Then
     tempString = tempString & Repeat(text, leftOver)
 End If

 Repeat = tempString
 Exit Function


processError:
 Dim errMess As String
 errMess = Error$
 Repeat = ""
 Exit Function


End Function
```

Okay, so what's this function actually doing? For starters, we can easily figure out that it's a lot faster to create a large string by doubling a string over and over than by adding new pieces to the string little by little. This script starts by figuring out how many times you can double the text that you were given, in order to create a large string without creating a string larger than what we want. Once we've done this, we should have a string close to the size that we're looking for, but there will usually be a few more repetitions of the original text string that we need to tack on there.

The best way to add the remaining repetitions of the string on to the end is to make a recursive call to this function. This will apply the same "doubling" logic to the number of repetitions that we have left, and keep doing this until we have the string we're looking for. In this case, the recursive call is "safe", because we'll never recurse more than 14 times (the largest integer value is 32767, which is 111111111111111 in binary, which would cause this script to recurse 14 times to get down to 1).

Finding the Last Occurrence of a Substring Within a String

We know that Instr will give us the first occurrence of a substring within a string, but to find the last occurrence of the substring means that we have to call Instr several times until we get to the end of the string. The "easy" way to code a function that finds the last occurrence of a substring is this:

Script – Find the last substring in a string (easy & slow)

```
Function InstrLastEasy (startPos As Integer, fullString As String, _
searchString As String, caseSensitive As Integer) As Integer
 Dim pos As Integer, lastPos As Integer

 pos = Instr(startPos, fullString, searchString, caseSensitive)
 Do While (pos > 0)
     lastPos = pos
     pos = Instr(lastPos + 1, fullString, searchString, caseSensitive)
 Loop

 InstrLastEasy = lastPos
End Function
```

This does the basic job, it's easy to follow, and it works just fine for a string with a small number of occurrences of the substring. However, it gets really slow if you have a large string with a large number of occurrences of the substring. A much faster way to write this function is like this:

Script – Find the last lsubstring in a string (fast)

```
Function InstrLast (startPos As Integer, fullString As String, _
searchString As String, caseSensitive As Integer) As Integer
 '** find the last position of a substring within a string
 Dim stringLength As Integer
 Dim pos As Integer
 Dim halfPos As Integer
 Dim posBefore As Integer, posAfter As Integer
 Dim lastPos As Integer, endPos As Integer
 Dim tempString As String

 stringLength = Len(fullString)

 '** exit early if there's nothing to search
 If (stringLength = 0) Or (Len(searchString) = 0) Then
     InstrLast = 0
     Exit Function
 End If

 pos = Instr(startPos, fullString, searchString, caseSensitive)
 If (pos = 0) Then
     '** also exit early if searchString isn't in fullString
     InstrLast = 0
     Exit Function
 Else
     '** otherwise, find a point halfway between the last known
     '** position of a match and the end of the string, and search
     '** both segments
     halfPos = pos + ((stringLength - pos) \ 2)
     If (halfPos > Len(searchString) + 1) Then
         posAfter = Instr(halfPos - Len(searchString), fullString, searchString,
caseSensitive)
         '** only check the first half if there was no match in
                 '** the second half
         If (posAfter = 0) Then
             posBefore = Instr(pos + 1, theString, searchString, caseSensitive)
```

```
            End If
        Else
            posBefore = Instr(pos + 1, fullString, searchString, caseSensitive)
        End If

        If (posAfter > posBefore) Then
            lastPos = posAfter
            endPos = stringLength
        Elseif (posBefore > 0) Then
            lastPos = posBefore
            endPos = halfPos
        End If

        '** if we found a match in either segment, recurse and look
             '** again within that segment
        If (lastPos > 0) Then
            If (lastPos = stringLength) Then
                InstrLast = lastPos
            Else
                tempString = Mid$(fullString, lastPos + 1, endPos - lastPos + 2)
                InstrLast = lastPos + InstrLast(1, tempString, searchString,
caseSensitive)
            End If
        Else
            InstrLast = pos
        End If
  End If

End Function
```

This version of the function gains speed and efficiency by doing the following: it finds the first occurrence of the substring, then it splits the rest of the string in half and finds the next occurrence in both halves, and it takes the higher of the two possible positions that it found and uses recursion to keep doing this until we can' t find the substring anymore. Because we' re halving the string every time we search, we don' t have to actually search the entire string from beginning to end, we can keep trying to jump forward halfway to the end as a part of our search.

If we calculate our recursion in a worst-case scenario (32k string, all spaces, searching for the last space character), we don' t really have to recurse too many times. The first calculation will split the string in half and find an occurrence at position #15999, the second (first recursion) will find an occurrence at position #8000, the third (second recursion) will find an occurrence at position #4000, and so on, which mathematically extends to only 16 total recursions until we get to the last occurrence. That shouldn' t be too much of a strain on our computer, and it' s a whole lot less searching than calculating the Instr value 32,000 times.

LotusScript Versions of @Left, @LeftBack, @Right, @RightBack

Okay, before you start complaining that R5 already has native LotusScript versions of these functions, let' s consider a few things. First of all, not everyone has upgraded to R5 yet. Heck, at the time I' m writing this (early 2002), there are still companies out there running ccMail. Those people need these functions. Second, you may need to write slightly modified versions of these functions that take special situations into account (we' ll see an example of that later in the chapter). These functions will give us a basis for writing those modifications.

Script – LotusScript version of @Left

```
Function StringLeft (text As String, searchTerm As String, caseSensitive As Integer)
As String
 '** Example: StringLeft("www.lotus.com", ".", True) would return "www"
 '** If the searchTerm isn't found, nothing is returned

 Dim pos As Integer

 '** If there is no searchTerm, just exit
 If (searchTerm = "") Then
     StringLeft = text
     Exit Function
 End If

 '** try to find the searchTerm
 pos = Instr(1, text, searchTerm, caseSensitive)

 If (pos > 0) Then
     StringLeft = Left$(text, pos - 1)
 Else
     StringLeft = ""
 End If

End Function
```

Script – LotusScript version of @LeftBack

```
Function StringLeftBack (text As String, searchTerm As String, caseSensitive As
Integer) As String
 '** Example: StringLeftBack("www.lotus.com", ".", True) would
 '** return "www.lotus"
 '** If the searchTerm isn't found, nothing is returned

 Dim pos As Integer

 '** If there is no searchTerm, just exit
 If (searchTerm = "") Then
     StringLeftBack = text
     Exit Function
 End If

 '** try to find the searchTerm, using the custom
 '** InstrLast function
 pos = InstrLast(1, text, searchTerm, caseSensitive)

 If (pos > 0) Then
     StringLeftBack = Left$(text, pos - 1)
 Else
     StringLeftBack = ""
 End If

End Function
```

Script – LotusScript version of @Right

```
Function StringRight (text As String, searchTerm As String, caseSensitive As Integer)
As String
 '** Example: StringRight("www.lotus.com", ".", True) would
 '** return "lotus.com"
 '** If the searchTerm isn't found, nothing is returned

 Dim pos As Integer

 '** If there is no searchTerm, just exit
 If (searchTerm = "") Then
```

```
        StringRight = text
        Exit Function
    End If

    '** try to find the searchTerm
    pos = Instr(1, text, searchTerm, caseSensitive)

    If (pos > 0) Then
        StringRight = Mid$(text, pos + 1)
    Else
        StringRight = ""
    End If

End Function
```

Script – LotusScript version of @RightBack

```
Function StringRightBack (text As String, searchTerm As String, caseSensitive As
Integer) As String
    '** Example: StringRightBack("www.lotus.com", ".", True) would
    '** return "com"
    '** If the searchTerm isn't found, nothing is returned

    Dim pos As Integer

    '** If there is no searchTerm, just exit
    If (searchTerm = "") Then
        StringRightBack = text
        Exit Function
    End If

    '** try to find the searchTerm, using the custom
    '** InstrLast function
    pos = InstrLast(1, text, searchTerm, caseSensitive)

    If (pos > 0) Then
        StringRightBack = Mid$(text, pos + 1)
    Else
        StringRightBack = ""
    End If

End Function
```

As you can see, all of the functions are essentially the same, with the exception of the use of the Instr or the InstrLast functions (remember InstrLast from earlier in the chapter?), and the way that the final string is calculated if a match is found.

LotusScript Version of @ReplaceSubstring

The @ReplaceSubstring function is another useful function that appears in the @Command language but not in LotusScript. I'll present 3 different ways of coding this function.

Script – LotusScript version of ReplaceSubstring, using the Evaluate function

```
Function ReplaceSubstringEvaluate (Byval fullString As String, oldString As String,
newString As String) As String
    Dim session As New NotesSession
    Dim db As NotesDatabase
    Dim doc As NotesDocument
    Dim var As Variant

    Set db = session.CurrentDatabase
    Set doc = New NotesDocument(db)

    Call doc.ReplaceItemValue("FullString", fullString)
    Call doc.ReplaceItemValue("OldString", oldString)
    Call doc.ReplaceItemValue("NewString", newString)

    var = Evaluate("@ReplaceSubstring(fullString; oldString; newString)", doc)

    ReplaceSubstringEvaluate = var(0)

    '** clean up the memory we used
    Set doc = Nothing
    Set db = Nothing

  End Function
```

This version of the function uses the Evaluate function to call the actual @ReplaceSubstring function and run it against our variables. In order to do this, we have to create a temporary document that holds our variables in fields, and then we can run @ReplaceSubstring against those fields. We have to do this because in pre-R5 versions of Notes, Evaluate can only run using either string literals or fields from a document. In a situation like this, we don't know any of the string values ahead of time, so we need to run Evaluate against fields that we populate. (If you know that all of your users will be using R5, you can simply pass the parameters directly.)

Despite the fact that there is some overhead involved with opening the database and creating a document, this method is still by far the fastest way (from a performance standpoint) to emulate @ReplaceSubstring.

Unfortunately, @ReplaceSubstring can only do case-sensitive searches, so if you need case-insensitive search-and-replace functionality, you'll need to use one of the other functions below. There can be some problems with this version of the function, too. First of all, if the user who's running the function doesn't have authority to create documents in the database, the function will fail. Second, some of the earlier versions of the 4.5 and 4.6 clients had problems with memory leaks when using the Evaluate function. If you're using one of those versions of Notes, you should avoid this function.

Another option is to perform the entire process with LotusScript, as in the following example:

Script – LotusScript version of @ReplaceSubstring

```
  Function ReplaceSubstring (Byval fullString As String, oldString As String, newString
  As String) As String
    On Error Goto processError

    Dim tempString As String
    Dim tempString2 As String
    Dim lenOldString As Integer
    Dim pos As Integer

    '** If the user passes us bogus values, just exit
    If (fullString = "") Or (oldString = "") Then
        ReplaceSubstring = fullString
        Exit Function
    End If

    '** initialize the variables
    tempString = fullString
```

```
 lenOldString = Len(oldString)
 pos = Instr(tempString, oldString)

 '** initialize tempString2, to speed things up a little
 If ((Len(fullString) * Len(NewString)) > 32000) Then
     tempString2 = Space$(32000)
 Else
     tempString2 = Space$(Len(fullString) * Len(NewString))
 End If
 tempString2 = ""

 '** get all the matches in the string, building a new string as we go
 Do While (pos > 0)
     tempString2 = tempString2 & Left$(tempString, pos - 1) & newString
     tempString =  Mid$(tempString, pos + lenOldString)
     pos = Instr(tempString, oldString)
 Loop

 '** add anything that's left in the original string to the end of
    '** the return string
 tempString2 = tempString2 & tempString

 ReplaceSubstring = tempString2
 Exit Function

processError:
 '** error 228 is String Too Large
 Dim errMess As String
 errMess = "Error " & Err & ": " & Error$
 ReplaceSubstring = fullString
 Exit Function

End Function
```

There are a couple different ways of actually doing the searching and replacing within the string: you can either go through character by character to find your substring or you can use Instr, and when you're creating the new string you can either modify the original string as you go or you can create a new string additively, as in the example shown.

Generally, the quickest way to perform this operation is to use Instr to search, and to create the new string additively, because it takes less overhead to create a new string than it does to modify an existing string over and over – that is, assuming that you pre-initialize the string, as we did in the function above. By initially setting the string to it's largest possible value using the Space() function, and then resetting it to nothing before we start, we can force LotusScript to allocate the memory for the final string all at once, early in the script, so that it won't have to keep allocating the string memory little by little as the string grows. To see the difference, run the script in separate agents with and without the pre-initialization, on a long string with a lot of replacements.

One thing you might want to change about the function above, depending upon your needs, is the value returned by the function in the case of an error. In the version shown above, the original string is returned if an error occurs while the function is running. Depending on your situation, it may be more useful to return either an empty string, or the portion of the string that was successfully operated upon prior to the occurrence of the error.

In case you were wondering about the performance of a ReplaceSubstring function that uses recursion, you can try out the following script:

Script – LotusScript version of @ReplaceSubstring, using recursion

```
Function ReplaceSubstringRecursive (Byval fullString As String, oldString As String, 
newString As String, retPos As Integer) As String
 '** recursive -- tricky math to calculate retPos
 On Error Goto processError
 
 Dim tempString As String
 Dim tempString2 As String, tempString3 As String
 Dim lenOldString As Integer
 Dim position As Integer
 Dim offset As Integer, halfPos As Integer
 Dim tempLeft As Integer
 Dim lastPos As Integer, lastPos2 As Integer
 
 '** If the user passes us bogus values, just exit
 If (fullString = "") Or (oldString = "") Then
     ReplaceSubstringRecursive = fullString
     Exit Function
 End If
 
 tempString = fullString
 lenOldString = Len(oldString)
 position = Instr(fullString, oldString)
 
 If (position > 0) Then
     tempString = Left$(fullString, position - 1) & newString
     tempString2 = Mid$(fullString, position + lenOldString)
     
     retPos = position
     halfPos = Len(tempString2) \ 2
     
     If (halfPos > lenOldString) Then
         tempString3 = ReplaceSubstringRecursive(Left$(tempString2, halfPos), 
oldString, newString, lastPos)
         If (lastPos = 0) Then
             offset = halfPos - lenOldString + 1
             tempLeft = offset
         Else
             offset = lastPos + lenOldString
             tempLeft = Len(tempString3) - (halfPos - (offset - 1))
         End If
         tempString3 = Left$(tempString3, tempLeft)
     Else
         offset = 1
     End If
     
     tempString2 = ReplaceSubstringRecursive(Mid$(tempString2, offset), oldString, 
newString, lastPos2)
     
     If (lastPos2 > 0) Then
         retPos = position + (lenOldString - 1) + (offset - 1) + lastPos2
     Elseif (lastPos > 0) Then
         retPos = position + (lenOldString - 1) + lastPos
     End If
 End If
 
 ReplaceSubstringRecursive = tempString & tempString3 & tempString2
 Exit Function
 
processError:
 Print "Error " & Err & ": " & Error$
 ReplaceSubstringRecursive = fullString
 Exit Function
```

```
   End Function
```

This version of the function uses recursion to continue splitting the string in half over and over, so that the ReplaceSubstring function can operate on smaller and smaller chunks of the string, and then add all of those small chunks together to form the new string. There are only a few unusual situations where this version of the function will give you better performance than the previous version, so you don't generally need to add this level of complexity to this particular function.

However, so you know how the function works, here's the logic. The only reason why this would work more quickly as a recursive function is if we thought that the efficiency we gain by having LotusScript work on small strings instead of large strings would outweigh the overhead of having to continuously break up the string, run the function against it, and combine it back together when we're done. In order to do this recursively, we can't just replace the first occurrence in the string and then call the function again with the remainder of the string, because on a large string with a lot of replacements we'd run out of stack space very quickly.

The other choice is to make our first replacement in the string, and then make a recursive call to the beginning half of the string, followed by a recursive call to the rest of the string. This will keep us from blowing the stack while allowing us to operate on smaller strings. The one thing you have to watch out for when operating on the string in this fashion is having a matching piece of string getting split between the two parts of the string that you're working with.

For example, let's say that the portion of string after the first match is "dogdogdog", and you're looking to replace the word "dog". If you split this piece of string in half, the first half will be "dogd" and the second half will be "ogdog". If you simply did a search and replace on the first and second half of the string, you'd miss the word "dog" that got split between the two substrings. So what you'll have to do is to perform a search and replace on the first half of the string, and then perform a search and replace on the rest of the string, starting from the end position of the last match we found.

This leads to some slightly tricky math, because in order for this function to work properly in a recursive fashion, it will have to calculate and return the position of the last match in the string it's working on. At first glance, this seems like a straightforward calculation, but as you're adding the last positions together as they're being returned from the recursive calls, you have to make sure you account for the fact that each recursive call will actually add one to the total of your last count. This isn't because of the recursive call itself, it's actually because of the fact that when we pass a piece of the string, we've already incremented the position count by one (because the new string starts at the last position plus one), so we have to subtract one for each position that one of the recursive calls returns. See the calculation of the retPos variable at the end of the function for more details.

As noted earlier, this isn't generally the best way to implement the @ReplaceSubstring function (although the performance isn't terrible). It's included mainly as another example of writing a "safe" recursive function, and it should help demonstrate some of the considerations you'll need to take into account when writing your own recursive functions.

Matching String Patterns

LotusScript has a built-in function called "Like" that allows you to do basic string pattern matching. However, it will only declare a match if the entire string matches the pattern. For example, if your search pattern is "d?g", then "dog" will be a match, but "doggy" and "My dog doesn't bite" will not be.

What follows is a group of functions that will allow you to mimic the Like function, but it will let you search for a string pattern anywhere within a string. There's kind of a lot here, so make yourself comfortable.

Script – Function for matching string patterns (main function)

```
Function FindWildcard (fullString As String, Byval searchString As String, _
startPos As Integer, endPos As Integer) As Integer
  '** find a substring within a string, between startPos and endPos,
  '** using wildcard operators (*, ?, and #) in your search expression.
  '** You can also search for lists or ranges of characters (or one of
  '** the wildcard operators as a character) by enclosing them in
  '** brackets []. The rules there are essentially the same as the
  '** rules for the Like operator, except you can additionally search for
  '** a close bracket character inside a set of brackets by doubling it
  '** ( like [asdf]]] will search for the characters a, s, d, f, and ].
  '** If a match is found, the return value will be True and the startPos
  '** and endPos will be reset to give the start and end position of the
  '** matching string

  Dim beginStar As Integer, qMarkOffset As Integer
  Dim newChar As String
  Dim ascChar As Integer
  Dim matchChar As String, lastMatchChar As String
  Dim isMatch As Integer
  Dim matchPos As Integer
  Dim getPos As Integer
  Dim retStartPos As Integer, retEndPos As Integer
  Dim ascStar As Integer, ascNum As Integer
     Dim ascQuest As Integer, ascBracket As Integer
  Dim isLastMatchStar As Integer
  Dim bracketText As String
  Dim bracketEndPos As Integer
  Dim isBracketMatchInverse As Integer
  Dim i As Integer

  '** exit if there's no search string
  If (Len(fullString) = 0) Or (Len(searchString) = 0) Then
      FindWildcard = False
      startPos = 0
      endPos = 0
      Exit Function
  End If

  '** validate the startPos and endPos
  Select Case startPos
  Case Is < 1
      startPos = 1
  Case Is > Len(fullString)
      startPos = Len(fullString)
  End Select

  Select Case endPos
  Case Is < 1
```

```
        endPos = Len(fullString)
    Case Is < startPos
        endPos = startPos
    Case Is > Len(fullString)
        endPos = Len(fullString)
    End Select

    '** set the starting values of our variables
    FindWildcard = False
    isMatch = False
    ascStar = Asc("*")
    ascNum = Asc("#")
    ascQuest = Asc("?")
    ascBracket = Asc("[")
    isLastMatchStar = False

    '** if the searchString begins with wildcard characters, including *,
    '** then we can strip them from the searchString and avoid having to
    '** use them in our search every time (if we end up finding a match,
    '** we can adjust the start position at the end)
    If (Instr(searchString, "*") > 0) Then
        Do Until (matchPos > Len(searchString))
            matchPos = matchPos + 1
            matchChar = Mid$(searchString, matchPos, 1)
            If (matchChar = "*") Then
                beginStar = True
            Elseif (matchChar = "?") Then
                qMarkOffset = qMarkOffset + 1
            Else
                Exit Do
            End If
        Loop

        '** adjust searchString if we found anything
        searchString = Mid$(searchString, matchPos)
    End If

    For i = (startPos + qMarkOffset) To endPos
        newChar = Mid(fullString, i, 1)
        ascChar = Asc(newChar)

        '** check the character in the string against the next character
        '** in the searchString
        If Not isMatch Then
            matchPos = 1
            retStartPos = 0
            matchChar = Mid$(searchString, matchPos, 1)
            lastMatchChar = ""
            isLastMatchStar = False
        End If

        Select Case Asc(matchChar)
        Case ascStar    '* = 42
            '** handle this outside the Select Case statement, in case
            '** we have a match plus a * plus nothing else, which means
            '** we should match the rest of the string

        Case ascQuest   '? = 63
            '** ? always matches exactly one character
            isMatch = True

            '** get the next searchString character
            lastMatchChar = matchChar
            isLastMatchStar = False
            matchPos = matchPos + 1
            matchChar = Mid$(searchString, matchPos, 1)

        Case ascNum '# = 35
            '** # always matches exactly one number
            If (Instr("0123456789", newChar) > 0) Then
                isMatch = True

                '** get the next searchString character
                lastMatchChar = matchChar
                isLastMatchStar = False
                matchPos = matchPos + 1
                matchChar = Mid$(searchString, matchPos, 1)
            Elseif isLastMatchStar Then
                '** if we were previously matching with "*", we
                '** should still consider this to be a match, but
                '** we shouldn't advance the matchPos
                isMatch = True
            Else
                isMatch = False
            End If

        Case ascBracket '[ = 91
            '** if it's a bracketed match, search for any the
            '** characters within the brackets
            isMatch = False
            bracketText = ExplodeBracket(Mid$(searchString, matchPos), _
            bracketEndPos, isBracketMatchInverse)

            If (Instr(bracketText, newChar) > 0) Then
                isMatch = True
            End If

            '** adjust the match, based on bracketMatchInverse
            isMatch = isMatch Xor isBracketMatchInverse

            If isMatch Then
                '** get the next searchString character
                lastMatchChar = "[]"
                isLastMatchStar = False
                matchPos = matchPos + bracketEndPos
                matchChar = Mid$(searchString, matchPos, 1)
            Elseif isLastMatchStar Then
                '** if we were previously matching with "*", we
                '** should still consider this to be a match, but
                '** we shouldn't advance the matchPos
                isMatch = True
            End If

        Case ascChar
            '** this should be an exact match
            isMatch = True
            '** get the next searchString character
            lastMatchChar = matchChar
            isLastMatchStar = False
            matchPos = matchPos + 1
            matchChar = Mid$(searchString, matchPos, 1)

        Case Else
            If isLastMatchStar Then
                '** if we were previously matching with "*", we
                '** should still consider this to be a match, but
                '** we shouldn't advance the matchPos
                isMatch = True
            Else
                isMatch = False
            End If

        End Select

        '** handle the special case where matchChar = *
        If (matchChar = "*") Then
            '** the * is always a match
            isMatch = True
            lastMatchChar = "*"
            isLastMatchStar = True
            '** get the next searchString character that's not
            '** a wildcard
            Do Until (matchPos > Len(searchString))
                matchPos = matchPos + 1
                matchChar = Mid$(searchString, matchPos, 1)
                If (matchChar <> "*") And (matchChar <> "?") Then
                    Exit Do
                End If
            Loop

        End If

        '** if we're in the middle of a match, make sure the startPos
        '** variable is set, and check to see if we found a full match
        If isMatch Then
            '** set the return startPos, if necessary
            If (retStartPos = 0) Then
                retStartPos = i
            End If

            '** if we got to the end of the searchString, then we
            '** found a match
            If (matchPos > Len(searchString)) Then
                If isLastMatchStar Then
                    '** if the last character in the searchString
                    '** is a "*", then this matches everything
                    '** from startPos to the end of the line
                    retEndPos = endPos
                Else
                    retEndPos = i
                End If

                FindWildcard = True
                Exit For
            Elseif isLastMatchStar Then
                '** speed things up a little by finding the next
                '** character match, if we're looking for *something
                getPos = GetFirstPossibleMatchPos(Mid$(fullString, i + 1),
    Mid$(searchString, matchPos))
                If (getPos = 0) Then
                    Exit For
                Else
                    i = i + getPos - 1
                End If
            End If
        Else
            '** couldn't find a match, so let's advance to the
            '** next possible spot
            getPos = GetFirstPossibleMatchPos(Mid$(fullString, i + 1), searchString)
```

```
            If (getPos = 0) Then
                Exit For
            Else
                i = i + getPos - 1
            End If
        End If

    Next i

    If FindWildcard Then
        If Not beginStar Then
            startPos = retStartPos - qMarkOffset
            '** Else startPos is whatever the user originally passed us
        End If
        endPos = retEndPos
    Else
        startPos = 0
        endPos = 0
    End If


End Function
```

Script – GetFirstMatchPos function (helper for main FindWildcard function)

```
Function GetFirstPossibleMatchPos (fullString As String, searchString As String) As
Integer
 '** get the first position of a possible match within fullString, based on
 '** the wildcard match specifications in searchString
 Dim matchPos As Integer
 Dim matchChar As String
 Dim ascMatchChar As Integer
 Dim bracketText As String
 Dim searchPos As Integer
 Dim firstSearchPos As Integer
 Dim ascStar As Integer, ascNum As Integer
   Dim ascQuest As Integer, ascBracket As Integer
 Dim i As Integer

 '** exit early if we got a bogus string
 If (Len(fullString) = 0) Then
     GetFirstPossibleMatchPos = 0
     Exit Function
 End If

 '** initialize the variables
 ascStar = Asc("*")
 ascNum = Asc("#")
 ascQuest = Asc("?")
 ascBracket = Asc("[")

 '** start searching
 firstSearchPos = 0
 matchPos = 1

 Do Until (matchPos > Len(searchString))
     matchChar = Mid$(searchString, matchPos, 1)
     ascMatchChar = Asc(matchChar)

     Select Case ascMatchChar
     Case ascStar
         '** loop
     Case ascQuest
         '** loop
     Case ascBracket
         '** if we're matching stuff inside brackets, get the
         '** exploded bracket string and search for all the
         '** characters inside
         bracketText = ExplodeBracket(Mid$(searchString, matchPos), 0, 0)
         For i = 1 To Len(bracketText)
             searchPos = Instr(fullString, Mid$(bracketText, i, 1))
             If (searchPos > 0) Then
                 If (searchPos < firstSearchPos) Or (firstSearchPos = 0) Then
                     firstSearchPos = searchPos
                 End If
             End If
         Next
         Exit Do
     Case ascNum
         '** if we're looking for a number, look for the first
         '** number in the string
         For i = 0 To 9
             searchPos = Instr(fullString, Cstr(i))
             If (searchPos > 0) Then
                 If (searchPos < firstSearchPos) Or (firstSearchPos = 0) Then
                     firstSearchPos = searchPos
                 End If
             End If
         Next
         Exit Do
     Case Else
         '** we're just looking for a single character
         firstSearchPos = Instr(fullString, matchChar)
         Exit Do
     End Select

     matchPos = matchPos + 1
 Loop

 '** if we're at the end of the searchString, then we're either searching
 '** for nothing or the entire searchString is just * or ?, so the first
 '** possible match is the beginning of the string
 If (matchPos > Len(searchString)) Then
     GetFirstPossibleMatchPos = 1
     Exit Function
 End If

 GetFirstPossibleMatchPos = firstSearchPos

End Function
```

Script – ExplodeBracket (helper for main FindWildcard function)

```
Function ExplodeBracket (bracketString As String, retEndPos As Integer, isInverse As Integer) As String
 '** Convert the given bracketed characters in a wildcard match string
 '** to a string of all characters that would count as a match.
 '** Returns the end position of the brackets, if the user wants them.
 '** Also returns whether the string match is an inverse match.
 Dim char As String
 Dim pos As Integer
 Dim i As Integer
 Dim isBracketRange As Integer
 Dim bracketStartRange As Integer, bracketEndRange As Integer

 '** if we're not looking at a bracketed string, just return
 If Not (Left$(bracketString, 1) = "[") Then
     retEndPos = 0
```

```
        isInverse = False
        ExplodeBracket = ""
        Exit Function
    End If

    '** save some information as static information, so we don't end up
    '** exploding the same bracket string over and over again
    Static lastString As String
    Static lastEndPos As Integer
    Static lastInverse As Integer
    Static lastResult As String

    '** if we're looking at the same thing we looked at last time,
    '** just return the same results from last time
    If (bracketString = lastString) Then
        retEndPos = lastEndPos
        isInverse = lastInverse
        ExplodeBracket = lastResult
        Exit Function
    Else
        lastString = bracketString
    End If

    If (Mid$(bracketString, 2, 1) = "!") Then
        '** if the first character is a "!", then we're finding anything
        '** that doesn't match the characters in the brackets
        pos = 3
        isInverse = False
    Else
        pos = 2
        isInverse = False
    End If

    '** initialize the variables and start going through the brackets
    retEndPos = 2
    isBracketRange = False

    Do Until (pos > Len(bracketString))
        char = Mid$(bracketString, pos, 1)

        Select Case char
        Case "]"
            If (Mid$(bracketString, pos + 1, 1) = "]") Then
                pos = pos + 1
                char = Mid$(bracketString, pos, 1)
            Else
                ' we're at the end of the brackets
                retEndPos = pos
                Exit Do
            End If
        Case "-"
            If (bracketStartRange >= 0) Then
                isBracketRange = True
            End If
        Case Else
            '** continue on
        End Select

        '** set the start and end range of the character(s) we're
        '** looking for
        If isBracketRange And (char <> "-") Then
            '** if we're dealing with a range of characters, get the
            '** start and end of the range and list them out
```

```
         If (Asc(char) > bracketStartRange) Then
             bracketEndRange = Asc(char)
         Else
             bracketEndRange = bracketStartRange
             bracketStartRange = Asc(char)
         End If

         For i = bracketStartRange To bracketEndRange
             ExplodeBracket = ExplodeBracket & Chr$(i)
         Next
         isBracketRange = False

     Elseif Not isBracketRange Then
         ExplodeBracket = ExplodeBracket & char
     End If

     '** advance the position and continue
     pos = pos + 1

 Loop

 '** if we got all the way to the end without finding the close bracket,
 '** just return the end of the string as the end of the brackets
 If (pos > Len(bracketString)) Then
     retEndPos = Len(bracketString)
 End If

 '** set the static variables for the next time
 lastEndPos = retEndPos
 lastInverse = isInverse
 lastResult = ExplodeBracket

End Function
```

Okay, let's start with the FindWildcard function. As input, it takes the string you want to search ("fullString"), the string pattern you're searching for ("searchString"), and the start and end positions of the portion of fullString that you want to search ("startPos" and "endPos"). It returns a Boolean value indicating whether or not a match was found in the string, and if a match was found, then startPos and endPos are set to the values of the starting and ending positions of where the match was found within fullString. This is important because if we're matching a pattern like "d*g", we don't know how long the matching string is going to be (it could be "dog", "doing", or "do the shag").

The function starts with some error checking, to make sure our initial startPos and endPos numbers aren't going to cause errors. We also gain a little efficiency by stripping any wildcard characters from the beginning of the pattern we're looking for, because a * or ? will always match anything, so there's no need to keep trying to match those. We then start trying to match characters in the string.

Every time we find a match, we'll advance our position in the pattern and look at the next character in fullString. We keep going until we're either at the end of the pattern (which means we found a match and our search is over), or a match condition is negative. If we're not in the middle of a match, then we can speed things up by using the GetFirstPossibleMatchPos function. That function finds the next possible match in fullString by finding the next occurrence of the first character (or characters) that we're matching in the pattern. For example, if the pattern is "d*g", it will find the next "d" in fullString; if the pattern is "[1-9] course meal", it will find the next 1, 2, 3, …, 9 in fullString. Using this function helps us from having to step through every single character in the string while we're matching.

The special case when we're matching is when there are multiple characters enclosed in brackets. The bracketed characters can be either a list of characters, a range of characters, or a combination of the two, and the user can optionally choose to search for things that aren't in the brackets by using an exclamation point. In order to help us with this situation, we have the ExplodeBracket function, which "explodes" the bracketed expression into a string of all the characters represented by the expression, so we can use a simple Instr to determine if a character matches the bracketed expression.

Because there's a lot going on in this function, there are a few efficiencies built-in to make sure it doesn't run too slowly when operating on large strings. First, the GetFirstPossibleMatchPos function helps us avoid having to examine every single character in the string while we're searching. The fewer potential matches in the string, the more efficiency we gain. The second thing that speeds things up is checking for a character match using the numeric ASCII value of the character instead of the string representation. This is sped up even more by the fact that we store the ASCII value of the special characters (*, ?, #, and [) in variables at the beginning of the function, so we don't have to keep figuring out what they are every time we're looking at a character. Third, we try to "jump ahead" as much as possible when we're matching a * in the string pattern. We do this by stripping the * from the beginning of a pattern, if one exists, and when we run into a * in the middle of a pattern, we use GetFirstPossibleMatchPos to find the next character, instead of stepping through the string character by character until we find it.

I've also experimented with using Static variables in the ExplodeBracket function, so if there's only one bracketed term inside in the pattern, we won't have to "re-explode" the term every time we look at it. However, while stress testing the functions, use of Static variables for this purpose didn't have any noticeable impact on the performance of the function.

Replacing Wildcard Patterns

Now that we have a function that finds wildcard patterns in a string, we can extend the functionality of some of our other string handling functions (StringLeft, StringRight, etc.) to handle wildcard cases as well. Here's a modification of ReplaceSubstring that uses the FindWildcard function.

Script – ReplaceSubstringWildcard

```
Function ReplaceSubstringWildcard (Byval fullString As String, oldString As String,
newString As String) As String
  '** ReplaceSubstringWildcard, using the FindWildcard function
  On Error Goto processError

  Dim tempString As String
  Dim tempString2 As String
  Dim startPos As Integer, endPos As Integer

  '** If the user passes us bogus values, just exit
  If (fullString = "") Or (oldString = "") Then
      ReplaceSubstringWildcard = fullString
      Exit Function
  End If

  '** initialize the variables
```

```
    tempString = fullString
    startPos = 1
    endPos = Len(tempString)

    '** initialize tempString2, to speed things up a little
    If ((Len(fullString) * Len(NewString)) > 32000) Then
        tempString2 = Space$(32000)
    Else
        tempString2 = Space$(Len(fullString) * Len(NewString))
    End If
    tempString2 = ""

    '** get all the matches in the string, building a new string as we go
    Do While (startPos > 0)
        If FindWildcard(tempString, oldString, startPos, endPos) Then
            tempString2 = tempString2 & Left$(tempString, startPos - 1) & newString
            tempString =  Mid$(tempString, endPos + 1)
            startPos = 1
            endPos = Len(tempString)
        End If
    Loop

    '** add anything that's left in the original string to the end of the
    '** return string
    tempString2 = tempString2 & tempString

    ReplaceSubstringWildcard = tempString2
    Exit Function

   processError:
    '** error 228 is String Too Large
    Dim errMess As String
    errMess = "Error " & Err & ": " & Error$
    ReplaceSubstringWildcard = fullString
    Exit Function

   End Function
```

Here you can see why it was so important to return the beginning and the ending position of the match, so that we can replace the proper characters. Because of the nature of the FindWildcard function, we can actually use this function as a full replacement for the ReplaceSubstring function that we wrote earlier. However, it will run more slowly because of the extra processing that the FindWildcard function introduces.

Just for fun, here' s the recursive version of the ReplaceSubstringWildcard function:

Script – ReplaceSubstringWildcardRecursive

```
   Function ReplaceSubstringWildcardRecurs (Byval fullString As String, oldString As
   String, _
   newString As String, retPos As Integer) As String
    '** ReplaceSubstringWildcardRecurs, using the FindWildcardMatch2 function
    On Error Goto processError

    Dim tempString As String
    Dim tempString2 As String
    Dim tempString3 As String
    Dim lenOldString As Integer
    Dim startPos As Integer, endPos As Integer
    Dim lastPos As Integer, lastPos2 As Integer
    Dim halfPos As Integer, offset As Integer
```

```
 Dim tempLeft As Integer

 '** If the user passes us bogus values, just exit
 If (fullString = "") Or (oldString = "") Then
     ReplaceSubstringWildcardRecurs = fullString
     Exit Function
 End If

 '** initialize the variables
 tempString = fullString
 lenOldString = Len(oldString)
 startPos = 1
 endPos = Len(tempString)

 If FindWildcard(tempString, oldString, startPos, endPos) Then
     tempString = Left$(fullString, startPos - 1) & newString
     tempString2 = Mid$(fullString, endPos + 1)

     retPos = endPos
     halfPos = Len(tempString2) \ 2

     If (halfPos > lenOldString) Then
         tempString3 = ReplaceSubstringWildcardRecurs(Left$(tempString2, halfPos),
oldString, newString, lastPos)
         If (lastPos = 0) Then
             offset = 1
             tempLeft = 0
         Else
             offset = lastPos + 1
             tempLeft = Len(tempString3) - (halfPos - (offset - 1))
         End If
         tempString3 = Left$(tempString3, tempLeft)
     Else
         offset = 1
     End If

     tempString2 = ReplaceSubstringWildcardRecurs(Mid$(tempString2, offset),
oldString, newString, lastPos2)

     If (lastPos2 > 0) Then
         retPos = endPos + (offset - 1) + lastPos2
     Elseif (lastPos > 0) Then
         retPos = endPos + lastPos
     End If
 End If

 ReplaceSubstringWildcardRecurs = tempString & tempString3 & tempString2

 Exit Function

processError:
 '** error 228 is String Too Large
 Dim errMess As String
 errMess = "Error " & Err & ": " & Error$
 Print errMess
 ReplaceSubstringWildcardRecurs = fullString
 Exit Function

End Function
```

Like the previous recursive ReplaceSubstring function in this chapter, this particular function doesn't run quite as quickly as the non-recursive version, but it's interesting to see how the function is structured.

Finding the First and Last Occurrence of a String Pattern

We don't really have to write a pattern-matching version of the Instr function, because that is essentially what the FindWildcard function is. If you wanted to write a wrapper function for FindWildcard in order to mimic the syntax more accurately, though, you could write something like this:

Script – Wrapper function for FindWildcard, which mimics Instr syntax

```
Function InstrWildcard (startPos As Integer, fullString As String, _
searchString As String, compMethod As Integer) As Integer
  If (compMethod = 0) Or (compMethod = 4) Then
      InstrWildcard = FindWildcard (fullString, searchString, _
           startPos, endPos)
    Else
      InstrWildcard = FindWildcard (LCase(fullString), _
           LCase(searchString), startPos, endPos)
    End If
End Function
```

Notice that we sort of cheated on the fourth parameter for Instr, which indicates whether or not the match is case- or pitch-sensitive. This is because we didn't write any pitch-sensitive code into the original FindWildcard function, so all we can do is make the fullString and the searchString both lowercase for any kind of case-insensitive search. You should be able to use the "Option Compare" statement to specify pitch-sensitivity, though.

If you want to find the last occurrence of a string pattern within a string, you can make a few modifications to the InstrLast function that we wrote earlier.

Script – Finding the last occurrence of a string pattern

```
Function FindWildcardLast (theString As String, searchString As String, _
startPos As Integer, endPos As Integer) As Integer
  '** find the last position of a substring within a string
  Dim stringLength As Integer
  Dim isFound As Integer
  Dim origStartPos As Integer, origEndPos As Integer
  Dim halfPos As Integer
  Dim wStartPos1 As Integer, wStartPos2 As Integer
  Dim wEndPos1 As Integer, wEndPos2 As Integer
  Dim posBefore As Integer, posAfter As Integer
  Dim lastStartPos As Integer, lastEndPos As Integer
  Dim tempString As String
  Dim beginStar As Integer, endStar As Integer
  Dim matchPos As Integer, matchChar As String

  stringLength = Len(theString)
  origStartPos = startPos
  origEndPos = endPos

  '** exit early if there's nothing to search
  If (stringLength = 0) Or (Len(searchString) = 0) Then
      FindWildcardLast = False
      Exit Function
  End If
```

```
isFound = FindWildcard(theString, searchString, startPos, endPos)
If Not isFound Then
    '** also exit early if searchString isn't in theString
    FindWildcardLast = False
    Exit Function
Else
    '** if we're here, we found at least one match
    FindWildcardLast = True
    '** otherwise, find a point halfway between the last known
         '** position of a match and the end of the string, and search
         '** both segments
    halfPos = endPos + ((stringLength - endPos) \ 2)
    wStartPos1 = endPos + 1
    wStartPos2 = halfPos
    wEndPos1 = halfPos
    wEndPos2 = origEndPos

    If (FindWildcard(theString, searchString, wStartPos2, wEndPos2)) Then
        '** if we found at least one match in the last half of
                '** the string, use that position as a basis for a
                '** recursive search
        lastStartPos = wStartPos2
        lastEndPos = wEndPos2
        endPos = stringLength
    Else
        '** otherwise, search the first half
        If (FindWildcard(theString, searchString, wStartPos1, wEndPos1)) Then
            lastStartPos = wStartPos1
            lastEndPos = wEndPos1
            endPos = halfPos
        End If
    End If

    '** if we found a match in either segment, recurse and look again
    '** within that segment
    If (lastStartPos > 0) Then
        If (lastEndPos = endPos) Then
            '** we're at the end
            startPos = lastStartPos
        Else
            tempString = Mid$(theString, lastEndPos + 1, endPos - lastEndPos + 2)
            startPos = 1
            endPos = Len(tempString)
            Call FindWildcardLast(tempString, searchString, startPos, endPos)
            If (startPos > 0) Then
                startPos = startPos + lastEndPos
                endPos = endPos + lastEndPos
            Else
                startPos = lastStartPos
                endPos = lastEndPos
            End If
        End If
    Else
        '** startPos and endPos should already be set
    End If
End If

'** make sure we didn't miss anything (which could happen if
'** a match got split where we cut the string in half, or there's a
'** match within the match we found)
If (startPos + 1< origEndPos) Then
    lastEndPos = origEndPos
```

```
        If FindWildcard(theString, searchString, startPos + 1, lastEndPos) Then
            startPos = startPos + 1
            endPos = origEndPos
            Call FindWildcardLast(theString, searchString, startPos, endPos)
        End If
    End If

    '** now that we're at the end, check the searchString to see if
    '** it begins or ends with an asterisk (if it does, then we should
    '** return everything starting from the beginning of the string or
    '** everything ending at the end of the string)
    matchPos = 0
    Do Until (matchPos > Len(searchString))
        matchPos = matchPos + 1
        matchChar = Mid$(searchString, matchPos, 1)
        If (matchChar = "*") Then
            beginStar = True
        Elseif Not (matchChar = "?") Then
            Exit Do
        End If
    Loop

    matchPos = Len(searchString)
    Do Until (matchPos = 0)
        matchChar = Mid$(searchString, matchPos, 1)
        If (matchChar = "*") Then
            endStar = True
        Elseif Not (matchChar = "?") Then
            Exit Do
        End If
        matchPos = matchPos - 1
    Loop

    '** adjust if there was an asterisk at the beginning or the end
    If beginStar Then
        startPos = origStartPos
    End If

    If endStar Then
        endPos = origEndPos
    End If

  End Function
```

This is a little more complex than the previous InstrLast function, although it uses the same logic. The first difference is that we have to do an extra check for a match near the end of the function, in case the term that we're searching for gets split when we're cutting the string in half, which could cause us to incorrectly come up with no match. This is not a consideration with the original InstrLast function, because in that case we knew exactly how long the search string was, and we could account for that when we were splitting the string. With a wildcard search such as "d*g", our matching search string could be anywhere from 2 to 30,000 characters long.

The second difference is that we have to adjust our start and/or end positions at the end of the function, in case our search term begins or ends with a *.

The only caveat of the way this function is written is that there are certain cases where the last match you find isn't quite the same as the last match you might find if you're just searching forwards through the string. For example, if you're searching for the string pattern "d*g", and your string is "dogddddddddddddog", then a forward search will give you a last string starting position of 4 (matching "dddddddddddog"), but the function above will give a position of 14 (matching just "dog" at the end). That's because the function above actually ends up simulating a reverse search through the string, not a forward search.

Fuzzy Searching (or, Approximate String Matching)

Another type of search that you might want to do within a string is a "fuzzy" search. This is usually defined as a search that matches character combinations that are similar to the combination you're looking for, in addition to exact matches. For example, you might want a search for "there" to match similar combinations like "their" and "they're".

There are three common ways of implementing a fuzzy search to provide approximate string matching capabilities. One way is to use a "dictionary" match, in which the word you're looking for is first found in a dictionary list of words, and then all the string variations found in that dictionary list are used in the search in addition to the word itself (this will allow you to look up "run", "running", "ran", etc. in a single search). A second way to search is to use a variation of the "Soundex" method, in which the letters of the word you want to search for are converted to a small set of numbers (as is the string you're searching), and you check for a number match to see if your fuzzy search is successful or not. A common variation of Soundex that's used for this purpose is called "Monophone". A third way is to calculate the "edit distance" (sometimes called the Levenschtein distance) between two strings, which is the number of letters that would have to be added, deleted, or changed in order to convert one string to another. A popular implementation of this is the Wu-Manber search.

In this chapter, we'll only be looking at the Soundex method for approximate string matching, because it's easy to understand and implement, it's fast, and it generally provides good results.

Fuzzy Searching Using Soundex

The "Soundex" method of fuzzy searching was actually created during an early US Census (???) as a way of matching common misspellings of surnames. It's still used quite often in name matching situations, such as with genealogy, and even in the Notes Name and Address Book! Here are the rules for converting a string to its Soundex equivalent.

1. Leading blanks in the input string are ignored.

2. The uppercase of the first letter in the string becomes the first character of the 4-character string. If the first non-blank in the string is not a letter, the code "0000" is returned.

3. After the first letter, the letters A, E, H, I, O, U, W, and Y are ignored in producing the code.

4. The remaining letters are assigned a code as follows:

   B, F, P, V = 1
   C, G, J, K, Q, S, X, Z = 2
   D, T = 3
   L = 4
   M, N = 5

R = 6

5. The code for the next letter is added to the output string unless it is a repeat of the code of the previous source string character, in which case it's ignored.

6. The scan stops at the first non-alpha character (including blank), and the code is padded with "0" if it ends up being less than 4 characters in length.

Here's a LotusScript implementation of this methodology:

Script – LotusScript version of the Soundex function

```
Function Soundex (Byval text As String) As String
 '** implementation of the traditional Soundex function
 Dim char As String
 Dim newString As String
 Dim convChar As String
 Dim i As Integer
 
 text = Trim(Ucase(text))
 char = Left$(text, 1)
 
 If (Asc(char) < Asc("A")) Or (Asc(char) > Asc("Z")) Then
     Soundex = "0000"
     Exit Function
 End If
 
 newString = char
 For i = 2 To Len(text)
     char = Mid$(text, i, 1)
     
     '** convert the next character in the string to its Soundex
     '** equivalent
     Select Case char
     Case "A", "E", "H", "I", "O", "U", "W", "Y"
         convChar = ""
     Case "'"
         '** non-standard, but you get better name matches
         '** if you treat this as a valid character that can
         '** safely be ignored
         convChar = ""
     Case "B", "F", "P", "V"
         convChar = "1"
     Case "C", "G", "J", "K", "Q", "S", "X", "Z"
         convChar = "2"
     Case "D", "T"
         convChar = "3"
     Case "L"
         convChar = "4"
     Case "M", "N"
         convChar = "5"
     Case "R"
         convChar = "6"
     Case Else
         Exit For
     End Select
     
     '** if the converted character is different from the last
          '** converted character in the string, append it
     If Not (Right$(newString, 1) = convChar) Then
         newString = newString & convChar
```

```
      End If

      '** once we've created a string of length 4, we're done
      If (Len(newString) = 4) Then
          Exit For
      End If
  Next

  '** add zeros to the end, in case the string isn't long enough
  newString = Left$(newString & "0000", 4)

  Soundex = newString

End Function
```

As noted in the function, we've included one non-standard element in our algorithm, and that is to treat a single quote as a valid character that should be ignored (as opposed to a non-valid character that should terminate the string). For example, think of the last name O'Leary or d'Angelo. Other than the special case with the single quote, this function should give you the same output for a character string as any other traditional implementation of the Soundex function does (like @Soundex).

If we want to make a few modifications, we can extend this function to write our own simple fuzzy search function. First, here's a slightly modified version of the Soundex function from above:

Script – Soundex function with minor modifications

```
Function SoundexPlus (Byval text As String, returnLength As Integer, _
allowSpaces As Integer, retLastPos As Integer) As String
  '** the Soundex function, with a few modifications
  Dim char As String
  Dim newString As String
  Dim convChar As String
  Dim i As Integer

  text = Trim(Ucase(text))
  char = Left$(text, 1)

  If (Asc(char) < Asc("A")) Or (Asc(char) > Asc("Z")) Then
      SoundexPlus = "0000"
      retLastPos = 0
      Exit Function
  End If

  '** initialize newString
  If (returnLength = 1) Then
      SoundexPlus = char
      retLastPos = 1
      Exit Function
  Else
      newString = Space$(returnLength)
      newString = char
  End If

  For i = 2 To Len(text)
      char = Mid$(text, i, 1)

      '** convert the next character in the string to its Soundex
      '** equivalent
      Select Case char
      Case "A", "E", "H", "I", "O", "U", "W", "Y"
```

```
         convChar = ""
     Case "'", "-"
         '** string concatenation characters that we can ignore
         convChar = ""
     Case "B", "F", "P", "V"
         convChar = "1"
     Case "C", "G", "J", "K", "Q", "S", "X", "Z"
         convChar = "2"
     Case "D", "T"
         convChar = "3"
     Case "L"
         convChar = "4"
     Case "M", "N"
         convChar = "5"
     Case "R"
         convChar = "6"
     Case " ", Chr(9), Chr(13), Chr(10)
         If allowSpaces Then
             convChar = ""
         Else
             Exit For
         End If
     Case Else
         Exit For
     End Select

     '** if the converted character is different from the last
     '** converted character in the string, append it
     If Not (Right$(newString, 1) = convChar) Then
         newString = newString & convChar
     End If

     '** once we've created a string of the length we want, we're done
     If (returnLength > 0) And (Len(newString) = returnLength) Then
         Exit For
     End If
 Next

 '** add zeros to the end, in case the string isn't long enough
 If (returnLength > 0) And (Len(newString) < returnLength) Then
     For i = (returnLength - Len(newString)) To returnLength
         newString = newString & "0"
     Next
 End If

 '** calculate retLastPos
 If (i <= Len(text)) Then
     retLastPos = i
 Else
     retLastPos = Len(text)
 End If

 '** and return
 SoundexPlus = newString

End Function
```

The enhancements in this function are that it allows you to specify the length of your return Soundex string (if you want something different than the standard 4-character return string), it allows you to specify whether or not you want to include spaces as valid characters (instead of terminating characters), and it returns the position of the last character used to create the Soundex string. The position of the last character is important for the same reason it's important in the FindWildcard function: if we want to extend this function for use in the other string functions we've been using in this chapter, we'll need to know how long the original string we're dealing with is, because it's not going to be a fixed length.

Here's an example of using this function in a modified version of the Instr function:

Script – An implementation of the Instr function, using Soundex for fuzzy matches

```
Function InstrSoundex (startPos As Integer, fullString As String, _
searchString As String, allowSpaces As Integer, lastPos As Integer) As Integer
  '** get the Instr position of the first Soundex match
  Dim tempString As String
  Dim soundexSearch As String
  Dim ssLength As Integer
  Dim ssStart As Integer
  Dim pos As Integer
  Dim i As Integer

  '** initialize the variables
  tempString = Ucase(fullString)
  soundexSearch = SoundexPlus(searchString, 0, allowSpaces, 0)
  ssLength = Len(soundexSearch)
  ssStart = Asc(Left$(soundexSearch, 1))
  InstrSoundex = 0
  lastPos = 0

  '** start searching from the first possible match, if one exists
  pos = Instr(startPos, tempString, Chr$(ssStart))
  If (pos = 0) Then
      Exit Function
  End If

  '** if we got this far, start searching the string for a Soundex match,
  '** and return as soon as we found one
  For i = pos To Len(tempString)
      If (Asc(Mid$(tempString, i, 1)) = ssStart) Then
          If (SoundexPlus(Mid$(tempString, i), ssLength, allowSpaces, lastPos) =
soundexSearch) Then
              InstrSoundex = i
              Exit For
          End If
      End If
  Next

End Function
```

This will search for the first Soundex match of the search string within the string, and will return both the start and end position of the first match. If you don't particularly care about the end position, you can just pass a number as the lastPos variable.

This particular implementation of the function searches the string character by character for a match, although you could easily change it to perform repetitive Instr calls as well. Both methods are quite fast, in this case.

Converting Double-Byte and Single-Byte Strings

Earlier in the chapter, we discussed the difference between double-byte and single-byte representations of strings, and how that can cause problems if you are expecting one and get the other. Here are two functions that allow you to go between the two representations.

Script – Convert Single-byte string to Double-byte

```
Function DoubleByteToSingleByte (sbString As String) As String
  Dim newLen As Long
  Dim returnString As String
  Dim i As Integer

  newLen = Len(sbString) * 2

  '** initialize the returnString
  returnString = Space$(newLen)
  returnString = ""

  For i = 1 To newLen
      returnString = returnString & Midb$(sbString, i, 1)
  Next

  DoubleByteToSingleByte = returnString

End Function
```

Script – Convert Double-byte string to Single-byte

```
Function DoubleByteToSingleByte (dbString As String) As String
  Dim newLen As Long
  Dim returnString As String
  Dim i As Integer

  '** the single-byte return string will be half the length
  '** of the double-byte string, rounded up
  newLen = Cint(Fix(Len(dbString) / 2)) + (Len(dbString) Mod 2)
  returnString = Space$(newLen)

  For i = 1 To Len(dbString)
      Midb(returnString, i) = Mid$(dbString, i, 1)
  Next

  DoubleByteToSingleByte = returnString

End Function
```

These function names are a little deceiving, because LotusScript always stores a string as a double-byte string, regardless of what data is in it. What these functions actually do is perform the necessary conversions if you happen to read data into a string variable in the wrong format. This can happen if you're reading information from a file, and you're expecting Unicode and you get ASCII, or vice versa.

Reading Large Strings from Form Fields

You may occasionally find that a field in a form has more string data than you can fit into a LotusScript string. This will happen if you try to read certain pieces of information from documents in the LOG.NSF file on a server (like the "Events" field in the "Miscellaneous Events" form). Here's a way around that:

Script – Reading large strings from form fields

```
Function GetLargeTextField (doc As NotesDocument, fieldName As String) As Variant
  '** For a text field that's larger than 32K, this function will
  '** return the field contents as two elements of a string array:
  '** the first element will be the first 32K of data, and the second
  '** element will be the rest.
  On Error Goto processError

  Dim eString1 As Variant, eString2 As Variant
  Dim largeTextString(0 To 1) As String

  eString1 = Evaluate( |@Left(| & fieldName & |; 32000)|, doc)
  eString2 = Evaluate( |@If(@Length(| & fieldName & |) > 32000; @Right(| & fieldName &
|; @Length(| & fieldName & |) - 32000); "")|, doc)

  largeTextString(0) = eString1(0)
  largeTextString(1) = eString2(0)

  GetLargeTextField = largeTextString

  Exit Function

processError:
  LastError$ = "Error " & Cstr(Err) & ": " & Error$
  largeTextString(0) = ""
  largeTextString(1) = ""
  GetLargeTextField = largeTextString
  Exit Function

End Function
```

This function will only work in R5 the way it was written, because earlier releases of Notes required the formulas in Evaluate statements to be known at the time the script is compiled, so you can't use the unknown "fieldName" variable in the formula. However, if you know the field that you'll be reading when you're writing the script, you could simply hard-code the field name in the formula.

Arrays and Lists

Arrays and lists are structured collections of data elements. If the array or list is defined as a Variant array or list, then it can hold many different types of data; otherwise, it will hold multiple items of the same data type or class.

The native LotusScript functions for dealing with arrays and lists are:

Function/Statement	Usage	Example
ArrayAppend (R5)	Creates a new array by adding the contents of one array to the end of a second array.	
ArrayReplace (R5)	Copies an array element by element to the result array.	
ArrayGetIndex (R5)	Searches for a value in an array, and returns the number of the first matching array element it finds.	
Erase	Deletes an element of a list, or removes all contents of an array or a list.	
FullTrim (R5)	Removes empty entries from an array.	
IsArray	Returns a Boolean value indicating whether or not the given variable or expression is an array.	
IsElement	Returns a Boolean value indicating whether or not the given value is a Tag in the given list. If you are looking for a String value, this search will be case-sensitive, unless "Option Compare Nocase" is in effect.	
IsList	Returns a Boolean value indicating whether or not the given variable or expression is a list.	

LBound	Returns the lower bound for an array. You can optionally indicate which dimension of the array you want the lower bound for.	
ListTag	Returns the name of the Tag for the list element currently being processed by a ForAll loop.	
ReDim	Declares or resizes a dynamic array.	
UBound	Returns the upper bound for an array. You can optionally indicate which dimension of the array you want the upper bound for.	

Using Lists

Declaring and using lists is generally easier than doing the same with an array, because lists are partially searchable (using the IsElement function), and they can resize automatically. To create a new list, you use the syntax:

```
Dim listName List As DataType
```

where "listName" is the name of your variable, and "DataType" is the data type you want to use. For example, this will create a String list:

```
Dim myList List As String
```

Initially, the list will have no data and no elements, but you can add an element by simply coming up with a unique "Tag" for your element, and entering the data. For example:

```
myList(1) = "The number 1"
myList("Two") = "The number 2"
```

The Tag (in the example above, the Tags are 1 and "Two") can be any native LotusScript data type, regardless of the way the list was defined, and you can mix Tag data types within a list – normally you won' t want to mix them, but you can. If an item with that Tag already exists in the List, then the item will be overwritten; if an item with that Tag doesn' t already exist, then a new List element will be created.

To refer to an element in a list, you simply use its Tag as an identifier, like this:

```
myString$ = myList("Two")
```

One thing to watch out for with Tags: if "Option Compare Nocase" is in effect for your script, then the list element myList("A") will be the same as myList("a"). If it' s not, which is the default case, then myList("A") and myList("a") will be different elements. If you want your list Tags to be non-case sensitive, but you don' t want to turn "Option Compare Nocase" on, you should make sure to create and refer to your Tags exactly the same time throughout your script, or you should always create and refer to them in all upper- or all lower-case.

To remove an element in a list, you can use the Erase function. Be very careful, though, because this command will remove the element in the list with a Tag of "Two":

```
Erase myList("Two")
```

but this command will remove all elements of the list:

```
Erase MyList()
```

Also, if you try to remove or refer to an element that doesn' t exist in the list, you will get an error 120.

Using Fixed Arrays

A fixed array is an array that has a fixed number of elements and dimensions, which are set when the array is defined. For example:

```
Dim myArray(5) As String
```

will (by default) create an array with 6 elements (0 to 5), and all elements will be of the String data type. Here are some notes about array creation:

- By default, the first element (the lower bound) of an array is element zero (0). This can be overridden in one of two ways: either you can define the lower bound in the array definition, with a statement like "Dim myArray (1 To 5)", which will set the lower bound to 1 and the upper bound to 5; or you can use Option Base statement to set the default lower bound to either zero or one.

- The upper and lower bounds (also referred to as the "subscript" bounds) of an array can be any number that is a valid Integer in LotusScript.

- An array that is defined with no upper or lower bounds is a dynamic array, which will be discussed shortly.

- An array can have up to 8 dimensions, with each dimension being a comma-separated value in the array definition. For example, myArray(5, 2) is a two-dimensional array.

- All elements in an array are initialized to whatever the default value is for the data type of the array. So the array declaration "Dim myArray(0 To 5) As Integer" will create an array with all elements equal to zero, until they are assigned values explicitly.

Assigning and reading elements of an array is the same as assigning and reading elements of a list: you reference the element with the element number, which is the same as the list Tag except it is always an Integer value. For example:

```
myArray(3) = "The quick brown fox"
someString$ = myArray(3)
```

You can't really erase an element of an array; all you can do is explicitly set the element to a value that is in the range of the data type you're using. If you use the Erase function on a fixed array, it will reinitialize all the elements of the array back to the initial state for the data type you're using.

Using Dynamic Arrays

A dynamic array is an array that can be resized. You cannot change a fixed array to a dynamic array; the array has to be defined as a dynamic array in one of two ways, either:

```
Dim dynamicArray() As String
```

in which the array is defined with no upper or lower bounds, or like this:

```
Redim dynamicArray(0) As String
```

in which you use the Redim statement to define the array. To change the bounds of a dynamic array, you use the Redim statement. You will normally want to use the Preserve option with Redim, because that will keep the existing elements of your array intact – without the Preserve option, the array will be resized, but all of the array elements will be reinitialized. For example, look at the following script fragment:

```
Redim dynamicArray(0 To 1) As String
dynamicArray(0) = "zero"
dynamicArray(1) = "one"
Redim dynamicArray(0 To 2) As String
```

At the end of the fragment, the array will have 3 elements instead of the original 2, but all the elements will have been reinitialized (and the assignments you just made will have been lost). However, if you change the last line to:

```
Redim Preserve dynamicArray(0 To 2) As String
```

then the Preserve option will keep all the existing element assignments as-is. The only time you will lose data with the Preserve option of Redim is if you resize the array to a smaller size than it used to be, in which case the elements that are no longer within the legal bounds of an array will go away.

You can use Redim to change either the upper or the lower bound of an array. For example, you could do something like this:

```
Redim dynamicArray(0) As String
Redim dynamicArray(-1 To 0) As String
```

which would give you an extra element at the lower end of the array. If you use the Preserve option with Redim, however, you can only change the upper bound.

Unfortunately, you cannot use the Redim statement to change the data type of an array.

You should keep in mind that Redim is an "expensive" operation in terms of the efficiency of your script. Every time you resize an array, LotusScript has to reallocate memory, and if you do that many times in a script you will notice a performance hit. When possible, you should try to approximate the final size of your dynamic array ahead of time, and adjust the size as few times as possible.

One final note: if you use the Erase statement against a dynamic array, all the elements will be removed, and you will be left with an array with no elements.

Arrays and Lists of User-Defined Data Types and Classes

Besides creating arrays and lists of standard LotusScript data types (like Strings, Integers, etc.), you can also create arrays and lists of user-defined data types. For example:

```
Type Book
 ISBN As Long
 Author As String
 Title As String
End Type

Dim bookList List As Book
Dim bookArray(0 To 10) As Book
```

This will create a list and an array of Book objects. To assign or retrieve any of the elements of the Book type, you use dot-notation:

```
bookList(1).Author = "John Smith"
title2$ = bookArray(2).Title
```

In much the same way, you can create arrays and lists of classes:

```
Dim dbArray(3) As NotesDatabase
Set dbArray(0) = New NotesDatabase("", "names.nsf")
Print dbArray(0).FilePath
```

You can use either native LotusScript classes or user-defined classes. The normal rules of construction and access apply.

Passing Arrays and Lists as Function Parameters and Results

Special rules apply if you want to use an array or a list as a function or sub parameter, or as the result of a function.

- If you want to pass a list as a parameter, the parameter must be either a Variant or a list of the same data type as the list you' re passing. The exception is if you are passing a list of a user-defined data type, in which case the parameter must be a list of the same user-defined data type.

- If you want to pass an array as a parameter, the parameter must be either a Variant or an array of the same data type as the array you' re passing, with no upper or lower bounds definitions. The exception is if you are passing an array of a user-defined data type, in which case the parameter must be an array of the same user-defined data type.

- Arrays and lists that are passed as parameters are passed by reference. This means that if you change the array or list data in the function or sub, the data in the original array or list gets changed as well.

If you want to return an array or a list as the result of a function, you should declare the function as type Variant. You cannot, however, return an array or list of a user-defined data type as the result of a function. If you wish to pass this type of array or list back to the calling routine, then you must return it as a parameter instead of a result.

Use Caution Passing Lists as Function or Sub Parameters

You should be very careful when you are passing lists around as function or sub parameters. Many versions of Notes have problems passing lists more than once as a parameter, and as a result, you can sometimes lose data or corrupt the list as the list is passed from the calling routine to the function or sub and back to the calling routine. Consider the following script:

Script – Data loss passing lists as parameters

```
Sub Initialize
 Dim newlist As Variant
 Dim listsize As Integer

 newlist = CreateList
 listsize = CountList(newlist)

End Sub

Function CreateList () As Variant
 Dim thislist List As Integer
 thislist(1) = 1
 thislist(2) = 2
 thislist(3) = 3
 thislist(4) = 4
 thislist(5) = 5
 thislist(6) = 6

 CreateList = thislist

End Function

Function CountList (thislist As Variant) As Integer
 Forall stuff In thislist
     count% = count% + 1
 End Forall

 Forall things In thislist
     count2% = count2% + 1
     Print "Forall loop #1 counted " & count% & _
           ". Forall loop #2 is counting " & count2% & "..."
     If (count2% = 50) Then
         Print "Forall loop #2 exiting because a count of " & _
                 count2% & " was reached."
         Exit Forall
     End If
 End Forall

 CountList = count%
End Function
```

If you compile this agent and run it, you will see that in the CountList function, the first loop through the list will return a count of 2, and the second loop through the list will loop forever if you don't stop it.

If you need to pass lists back and forth to functions and subs, it is better to do this using a Global list variable rather than as a function or sub parameter.

Copying Arrays and Lists

Normally, you can copy arrays and lists to Variant variables without problem. Copying to a Variant will make an actual copy of the array or list (instead of creating a pointer), so after you' ve made your copy, and changes to the original array or list will not affect the new copy.

The exception is an array or list of a user-defined data type, which cannot be treated as a Variant and cannot be copied unless you copy each element of the array or list individually.

Using LotusScript Arrays with Evaluate and @Functions

One really nice thing that the designers of LotusScript did was to provide an automatic conversion of LotusScript arrays to the list data type that is used on Forms. The following statements are valid, and will result a Form fields that is a text list:

```
Dim myArray(3) As String
myArray(1) = "Some string"
Call doc.ReplaceItemValue("FormList1", myArray)
```

You can make number and date lists in the same way. This feature also allows you to easily use arrays in @Functions, and call the Evaluate function in LotusScript to get the result. @Functions provide a much richer set of functions for handling arrays than LotusScript does, so this can come in very handy.

There are a few limitations that you should be aware of, though. One is that this functionality is only available for LotusScript arrays, because LotusScript lists do not automatically convert to Form field lists. Another is that Form field lists do not support mixed data types, so your array shouldn' t be a Variant array of mixed types. A related issue is that a Variant array that hasn' t had any elements initialized yet (or one that' s just been cleared with the Erase statement) will be passed as a single empty string.

Custom Routines

The rest of the chapter will consist of custom functions, subs, and scripts that you might find useful when dealing with lists and arrays.

Convert String to List or Array

These functions will convert a string to a list or an array, based on a delimiter of your choosing. They will produce a result similar to the @Explode function.

Script – String to List

```
Function StringToList (thisText As String, delim As String) As Variant
  '** convert a string to a list, separating at the specified delimiter.
  '** this is often easier to use than StringToArray, since you don't have
  '** to worry about dimensioning your return variable
  Dim templist List As String
  Dim tempstring As String
  Dim delimlength As Integer
  Dim pos As Integer
  Dim i As Integer
  tempstring = thisText
  delimlength = Len(delim)
  pos = Instr(1, tempstring, delim, 5)
  i = 0

  Do While (pos > 0)
      '** get the substring
      templist(i) = Left$(tempstring, pos - 1)

      '** reset the variables
      tempstring = Right$(tempstring, Len(tempstring) - pos - delimlength + 1)
      pos = Instr(1, tempstring, delim, 5)
      i = i + 1
  Loop

  '** make sure you get the stuff at the end of the string
  templist(i) = tempstring

  '** return the array as a result
  StringToList = templist

End Function
```

Script – String to Array

```
Function StringToArray (thisText As String, delim As String) As Variant
  '** convert a string to an array, separating at the specified delimiter
  Dim temparray() As String
  Dim tempstring As String
  Dim delimlength As Integer
  Dim pos As Integer
  Dim i As Integer

  tempstring = thisText
  delimlength = Len(delim)
  pos = Instr(1, tempstring, delim, 5)
  i = 0

  Do While (pos > 0)
      '** add a placeholder in the array for the new element
      Redim Preserve temparray(i) As String

      '** get the substring
      temparray(i) = Left$(tempstring, pos - 1)

      '** reset the variables
      tempstring = Right$(tempstring, Len(tempstring) - pos - delimlength + 1)
      pos = Instr(1, tempstring, delim, 5)
      i = i + 1
  Loop

  '** make sure you get the stuff at the end of the string
  Redim Preserve temparray(i) As String
  temparray(i) = tempstring$

  '** return the array as a result
  StringToArray = temparray

End Function
```

You can also create a similar function by calling the @Explode function directly:

Script – String to Array, using @Explode

```
Function StringToArrayEval (thisText As String, delim As String) As Variant
  '** convert a string to an array using the Evaluate function
  '** Keep in mind that if delim is multiple characters, then each
  '** of those characters is treated as a delimiter, not the whole
  '** word.
  Dim session As New NotesSession
  Dim db As NotesDatabase
  Dim doc As NotesDocument
  Dim var As Variant

  Set db = session.CurrentDatabase
  Set doc = New NotesDocument(db)

  Call doc.ReplaceItemValue("thisText", thisText)
  Call doc.ReplaceItemValue("delim", delim)

  StringToArrayEval = Evaluate("@Explode(thisText; delim)", doc)

  '** clean up the memory we used
  Set doc = Nothing
  Set db = Nothing

End Function
```

The main difference between the first two functions and the last one is that a multi-character delimiter is treated differently. In the first two functions, a multi-character delimiter is treated as a word, so that if you use "and" as a delimiter, the string will be broken up wherever the word "and" appears. In the last function, a multi-character delimiter is treated as a group of single-character delimiters, so that if you use "and" as a delimiter there, the string will be broken up wherever there's an "a", "n", or "d". There are cases where both of these bits of functionality are desirable.

The first two functions are also much easier to modify, in case you need to customize the parsing (like if you want to include the delimiter in the results or something like that).

Getting the Data Type of an Array or List

The LotusScript DataType function is useful for finding out what kind of data type a variable contains. If you check an array or list for its data type, however, an additional number is added to the DataType number, indicating whether the variable is a list, a dynamic array, or a fixed array. Sometimes you just want to know what the data type is (regardless of whether it's an array or a list or a scalar value), and the function below will do that for you.

Script – Get the data type of an array or list

```
Function ArrayDataType (a1 As Variant) As Integer
  '** determine the data type of an array or list
  Dim dType As Integer
  dType = Datatype(a1)

  Select Case dType
  Case Is >= 8704 :
      '** dynamic array
      dType = dType - 8704
  Case Is >= 8192 :
      '** fixed array
      dType = dType - 8192
  Case Is >= 2048 :
      '** list
      dType = dType - 2048
  Case Else :
      '** scalar value; no adjustment needed
  End Select

  ArrayDataType = dType
End Function
```

You can also simplify (or inline) this function by using a bitmask to strip the high bits (see the chapter on Numbers for more discussion about bitmasking).

Script – Array or list data type using a bitmask

```
  Function ArrayDataType2 (a1 As Variant) As Integer
    ArrayDataType2 = Datatype(a1) And 63
  End Function
```

However, the first method is a little easier to understand, and it allows you to do a little more customization of the function if you need to (like if you want to return something else if you' re checking a scalar value, or if you want to indicate both the data type and whether it' s a list or an array, or something like that).

Determining Whether or not an Array or List is Empty

Sometimes you need a way to determine whether or not an array or list is "empty". Part of that determination depends on what you consider "empty" to mean. Let' s look at a function that checks for empty arrays or lists.

Script – Determine whether an array or list is empty

```
  Function IsEmptyArray (a1 As Variant) As Integer
    '** determine whether or not the passed value is
    '** an empty array or list
    On Error Goto processError

    Dim checkVal As Variant

    '** calculate what an empty value for this array/list
    '** should be
    Select Case (Datatype(a1) And 63)
    Case 1 To 6, 11 :
        checkVal = 0
    Case 7 :
        checkVal = Cdat(0)
    Case 8 :
        checkVal = ""
    End Select

    '** and check all the elements of a1 to see if they match
    '** this value
    Forall stuff In a1
        '** uncomment this block out if you think that only lists with
        '** no elements should be considered empty
        'If Islist(a1) Then
        '   IsEmptyArray = False
        '   Exit Function
        'End If

        If Not (stuff = checkVal) Then
            IsEmptyArray = False
```

```
          Exit Function
      End If
  End Forall

  '** if we got all the way to the end, then a1 must have been empty
  IsEmptyArray = True
  Exit Function

processError:
  If (Err = 200) Then
      '** this is an "Attempt to access uninitialized array" error,
      '** which we can treat as a sign that a1 is empty
      IsEmptyArray = True
      Exit Function
  Else
      Print "Error " & Err & ": " & Error$
      IsEmptyArray = False
      Exit Function
  End If

End Function
```

First, we determine what an "empty" value is, based on the data type of the array or list we were given. This will be either the uninitialized value of that data type, or Nothing (for variants, user-defined types, classes, etc.). In this function, we call an array or list empty if all of its entries are equal to the uninitialized data type value, or if it is an uninitialized dynamic array, which would fall down to the error block. There' s also a block of commented code that you can uncomment if you want a list to be considered empty only if it contains no elements (similar to an uninitialized array).

Removing Empty Elements

You can use a similar logic in your code to write routines that remove empty elements from an array or a list. You' ll need two routines, though: one to handle arrays, and one to handle lists.

Script – Remove empty elements from an array

```
Function RemoveEmptiesInArray (a1 As Variant) As Variant
  '** remove all empty values in an array
  On Error Goto processError

  Dim checkVal As Variant
  Dim tempArray As Variant
  Dim count As Integer

  '** create an array that we can store our return values in
  If Not Isarray(a1) Then
      RemoveEmptiesInArray = a1
      Exit Function
  Else
      tempArray = a1
      count = Lbound(a1)
  End If

  '** calculate what an empty value for this array
  '** should be
  Select Case (Datatype(a1) And 63)
  Case 1 To 6, 11 :
      checkVal = 0
  Case 7 :
      checkVal = Cdat(0)
  Case 8 :
      checkVal = ""
  End Select

  '** check all the elements of a1 to see if they are empty or not
  Forall stuff In a1
      If Not (stuff = checkVal) Then
          tempArray(count) = stuff
          count = count + 1
      End If
  End Forall

  '** redimension the tempArray and return it to the user
  If (count > Lbound(a1)) Then
      Redim Preserve tempArray(Lbound(a1) To count - 1)
  Else
      Erase tempArray
  End If

  RemoveEmptiesInArray = tempArray
  Exit Function

 processError:
  Print "Error " & Err & ": " & Error$
  RemoveEmptiesInArray = a1
  Exit Function

 End Function
```

Script – Remove empty elements from a list

```
   Sub RemoveEmptiesInList (a1 As Variant)
    '** remove all empty values in a list
    Dim checkVal As Variant

    '** make sure we're dealing with a list
    If Not Islist(a1) Then
        Exit Sub
    End If

    '** calculate what an empty value for this List
    '** should be
    Select Case (Datatype(a1) And 63)
    Case 1 To 6, 11 :
        checkVal = 0
    Case 7 :
        checkVal = Cdat(0)
    Case 8 :
        checkVal = ""
    End Select

    '** check all the elements of a1 to see if they are empty or not
    Forall stuff In a1
        If (stuff = checkVal) Then
            Erase a1(Listtag(stuff))
        End If
    End Forall

   End Sub
```

In these examples, RemoveEmptiesInArray is a Function, and RemoveEmptiesInList is a Sub. The routine that operates on arrays could also have easily been written as a Sub, simply by copying the tempArray variable to the a1 variable that was originally passed. You should be careful if you try to convert the routine that operates on lists to a Function, however, because of the potential problems with lists passed as parameters (discussed earlier in this chapter).

There are two other "built-in" methods for removing empties. One is to use Evaluate to run the @Trim function against an array. This is a very efficient method for removing empties from a text list, but you can only use it against an array of Strings. If you're using R5 or higher, you can (and probably should) use the FullTrim function to remove empties. It doesn't work on lists, though, and if you have any "special" needs (like you have some extended definition of what it means to have an empty element), then you'll want to write a routine like the ones listed here.

Removing Duplicate Elements of an Array or List

A similar function you may wish to write is one that removes duplicate elements in an array or list. Below are routines that remove duplicates from arrays and lists.

Script – Remove duplicate entries from an array

```
Function RemoveDuplicatesInArray (a1 As Variant) As Variant
  Dim tempList List As Integer
  Dim returnArray As Variant
  Dim count As Integer

  '** make sure we're dealing with a list
  If Not Isarray(a1) Then
      Exit Function
  End If

  returnArray = a1
  count = Lbound(a1)

  '** copy the elements one-by-one to a temporary list;
  '** if the elements have already been copied over,
  '** they're duplicate values that can be erased
  Forall stuff In a1
      If Not Iselement(tempList(stuff)) Then
          tempList(stuff) = 0
          returnArray(count) = stuff
          count = count + 1
      End If
  End Forall

  '** resize the returnArray and send it back to the user
  If (count = Lbound(a1)) Then
      Erase returnArray
  Else
      Redim Preserve returnArray(0 To count - 1)
  End If

  RemoveDuplicatesInArray = returnArray

End Function
```

Script – Remove duplicate entries from a list

```
Sub RemoveDuplicatesInList (a1 As Variant)
  Dim tempList List As Integer

  '** make sure we're dealing with a list
  If Not Islist(a1) Then
      Exit Sub
  End If
```

```
    '** copy the elements one-by-one to a temporary list;
    '** if the elements have already been copied over,
    '** they're duplicate values that can be erased
    Forall stuff In a1
        If Iselement(tempList(stuff)) Then
            Erase a1(Listtag(stuff))
        Else
            tempList(stuff) = 0
        End If
    End Forall

  End Sub
```

As with the scripts for removing empty elements (and for the same reason), the routine that operates on an array is a function, and the one that operates on a list is a sub. The logic is that you take each item in the array or list that you are passed and copy it to a temporary list as a list tag. You can then use the IsElement function to easily check the temporary list to see if it already contains that item, and if it doesn' t then the item is not a duplicate. The nice thing about using the IsElement function in this capacity is that it observes whatever Option Case rules you happen to be following in your script, so you can maintain consistency comparing string values.

Adding Two Arrays

When we talk about "adding" arrays here, were actually talking about concatenating a pair of arrays, so that the elements of the second array are appended to the first array, resulting in an array that is as long as the length of the first array plus the length of the second array. There are, of course, other ways of adding arrays (such as concatenating the first element of the first array with the first element of the second array, the second element of the first array with the second element of the second array, etc.), but we' re only concerned with appending right now.

If you' reusing R5 or higher, you can use the ArrayAppend function to achieve a similar result, but this function has the advantage of being able to add a pair of arrays, lists, or scalar values (in any combination) together, while ArrayAppend requires the first value to be an array and does not allow lists.

Script – Adding two arrays, lists, or scalar values to get a new array

```
  Function AddArrays (a1 As Variant, a2 As Variant) As Variant
    '** return a new array, consisting of the elements of a1,
    '** followed by the elements of a2
    On Error Goto processError

    Dim newArray As Variant
    Dim count As Integer
    Dim i As Integer

    '** start with a1
    If Isscalar(a1) Then
        '** if a1 is a scalar value, just add a single entry to
            '** the new array
        Redim newArray(0)
        newArray(0) = a1
        count = 1
    Elseif Islist(a1) Then
        '** if a1 is a list, we'll want to convert to an array
        count = 0
        Forall stuff In a1
            Redim Preserve newArray(0 To count)
            newArray(count) = stuff
            count = count + 1
        End Forall
    Else
        '** otherwise, treat it like an array
        If (ArrayDataType(a1) = ArrayDataType(a2)) Then
            '** if a1 and a2 are the same data type, we can return
                    '** an array of that data type (our comparison is done
                    '** with the user-defined ArrayDataType function)
            newArray = a1
            count = Ubound(a1) + 1
        Else
            '** if we've got different data types, return a
                    '** variant array
            count = Lbound(a1)
            Redim newArray(Lbound(a1) To Ubound(a1))
            Forall stuff In a1
                newArray(count) = stuff
                count = count + 1
            End Forall
        End If
    End If

    '** and append a2 to the end
    If Isscalar(a2) Then
        '** scalar value; just add a single element
        Redim Preserve newArray(Lbound(newArray) To count)
        newArray(count) = a2
    Elseif Isarray(a2) Then
        '** array; so we only have to Redim the newArray once
        Redim Preserve newArray(Lbound(newArray) To count + (Ubound(a2) - Lbound(a2)))
        Forall stuff In a2
            newArray(count) = stuff
            count = count + 1
        End Forall
    Else
        '** list; so we have to keep redimensioning newArray
        Forall stuff In a2
            Redim Preserve newArray(Lbound(newArray) To count)
            newArray(count) = stuff
            count = count + 1
        End Forall
    End If

    AddArrays = newArray
    Exit Function


   processError:
    '** if there was an error, just return an empty array
    Print "AddArrays Error " & Err & ": " & Error$
    Redim emptyArray(0) As Variant
    AddArrays = emptyArray
    Exit Function

   End Function
```

There are a few things I' d like to point out in this function. First, after we check the data types of a1 and a2, if we determine that they are the same data type (and a1 is an array), we can simply create the return array by setting the newArray variable equal to a1. The interesting part of this is that even if a1 is a fixed array, newArray will end up being a dynamic array that we can resize later. Second, if we redimension a dynamic array without explicitly assigning a data type to it, the array automatically retains its current data type (which is handy, because we don' t necessarily know in advance what the data type is in this case). Also, we split up the way we append the items at the end if a2 is a list or an array. This is because we can easily determine how many elements an array has, which allows us to redimension newArray only once to allocate storage for all the elements of a2; otherwise (for a list), we have to keep redimensioning newArray for each element of the list, which is a much more expensive operation.

You can use @Functions to do the same type of thing, if you want:

Script – Using Evaluate and @Functions to add two arrays, lists, or scalar values

```
Function AddArraysEval (a1 As Variant, a2 As Variant) As Variant
  '** add two arrays or scalar values using @Functions
  Dim session As New NotesSession
  Dim db As NotesDatabase
  Dim doc As NotesDocument
  Dim var As Variant

  Set db = session.CurrentDatabase
  Set doc = New NotesDocument(db)

  Call doc.ReplaceItemValue("a1", a1)
  Call doc.ReplaceItemValue("a2", a2)

  AddArraysEval = Evaluate("a1 : a2", doc)

  '** clean up the memory we used
  Set doc = Nothing
  Set db = Nothing

End Function
```

One limitation here is that you can' t use this function if either a1 or a2 is a List. Also, a1 and a2 have to be the same data type, or the Evaluate statement will fail. You could get around this by rewriting the Evaluate statement as:

```
AddArraysEval = Evaluate("@Text(a1) : @Text(a2)", doc)
```

which is okay if you don' t mind always getting String arrays as your function result.

Getting Common Elements of Two Arrays or Lists

Here' s a function that will return a Variant array of elements that are common to a pair of arrays, lists, or scalar values.

Script – Get common elements of two arrays, lists, or scalar values

```
Function SameArrayItems (a1 As Variant, a2 As Variant) As Variant
  '** return an array that has all the elements in array a1
  '** that are also in array a2
  On Error Goto processError

  Dim returnArray As Variant
  Dim tempList List As Integer
  Dim count As Integer
```

```
    '** convert a2 into a list, for easier searching
    If Isscalar(a2) Then
        tempList(a2) = 0
    Else
        Forall stuff In a2
            tempList(stuff) = 0
            count = count + 1
        End Forall
    End If

    '** initialize the return array
    Redim returnArray(0 To count)

    '** check all the elements in a1 against the new list we made
    count = 0
    If Isscalar(a1) Then
        If Iselement(tempList(a1)) Then
            returnArray(count) = a1
            count = count + 1
        End If
    Else
        Forall element In a1
            If Iselement(tempList(element)) Then
                returnArray(count) = element
                count = count + 1
            End If
        End Forall
    End If

    '** resize the return array down, so it will have the right
    '** number of elements
    If (count = 0) Then
        Erase returnArray
    Else
        Redim Preserve returnArray(0 To count - 1)
    End If

    '** give the returnArray back to the user
    SameArrayItems = returnArray
    Exit Function

   processError:
    Print "Error " & Err & ": " & Error$
    Redim returnArray(0)
    Erase returnArray
    SameArrayItems = returnArray
    Exit Function

   End Function
```

The trick here is to minimize the amount of times we step through each array or list (or scalar value, which is treated as a single-valued array) in order to make our comparisons and achieve our result. We can actually do this only once for each parameter, because we first convert the a2 parameter to a list of zeros (which keeps the memory usage of the list as small as possible), using the elements of a2 as list tags, and then we use the IsElement function to perform quick searches against that list. This way, we only have to iterate through each array or list once, and we let IsElement take care of all the searching.

If there are no matches, or if the function errors out (like if you pass an object as one of the parameters or something), then the function returns an empty array.

Getting Different Elements of Two Arrays or Lists

The function to get different elements between two arrays or lists is a little more complicated than the function to get common elements, because we have to check both arrays or lists for differences, not just one. The implementation only adds a few more lines of code, though.

Script – Getting different elements of two arrays or lists

```
Function DifferentArrayItems (a1 As Variant, a2 As Variant) As Variant
  '** return an array that has all the elements in array a1
  '** array a2 that are different
  On Error Goto processError
  
  Dim returnArray As Variant
  Dim tempList List As Variant
  Dim count As Integer
  
  '** convert a2 into a list, for easier searching
  If Isscalar(a2) Then
      tempList(a2) = a2
  Else
      Forall stuff In a2
          tempList(stuff) = stuff
          count = count + 1
      End Forall
  End If
  
  '** initialize the return array
  Redim returnArray(0 To count)
  
  '** check all the elements in a1 against the new list we made
  count = 0
  If Isscalar(a1) Then
      If Iselement(tempList(a1)) Then
          Erase tempList(a1)
      Else
          returnArray(count) = a1
          count = count + 1
      End If
  Else
      Forall element In a1
          If Iselement(tempList(element)) Then
              Erase tempList(element)
          Else
              returnArray(count) = element
              count = count + 1
          End If
          
          '** make sure we have room in the returnArray
          If (count = Ubound(returnArray)) Then
              Redim Preserve returnArray(0 To count + 50)
          End If
      End Forall
  End If
  
  '** now add anything that's still left in tempList, which will be
  '** things in a2 that weren't in a1
  Forall leftover In tempList
      '** Listtag always returns a String, so we want to use the
         '** list element instead
      'returnArray(count) = Listtag(leftover)
      returnArray(count) = leftover
      count = count + 1
```

```
     '** make sure we have room in the returnArray
     If (count = Ubound(returnArray)) Then
         Redim Preserve returnArray(0 To count + 50)
     End If
 End Forall

 '** resize the return array down, so it will have the right
 '** number of elements
 If (count = 0) Then
     Erase returnArray
 Else
     Redim Preserve returnArray(0 To count - 1)
 End If

 '** give the returnArray back to the user
 DifferentArrayItems = returnArray
 Exit Function

processError:
 Print "Error " & Err & ": " & Error$
 Redim returnArray(0)
 Erase returnArray
 DifferentArrayItems = returnArray
 Exit Function

End Function
```

This function starts off the same as the function that gets the common elements of two arrays, with the small difference that the list that we create contains the actual data elements of a2, instead of zeros. This makes the list a little bigger and a little less memory-efficient, but we'll need the information in that format later as we're stepping through the list. As we check to see if any of the elements of a1 are in the list, we can use the Erase function to remove common elements from the list as we go – this is another nice thing about using a list to check the data elements, because you can use Erase to delete individual components of a list without having to redimension or reorder the list.

Finally, after we're done checking the elements of a1, we can take any remaining elements of the temporary list (remember that all the common elements have already been removed) and add them to our return array as well. As we're doing this, you'll see why we chose to store the data elements of a2 as list elements and not just list tags, because the ListTag function always returns a String value, and we want to populate our return array with elements that are the same data type as the original arrays or lists.

One other thing you might notice in this function is that there's no good quick way of finding out what the possible size of the return array is (we can't use Ubound - Lbound for the two parameters we've been passed, because they're not necessarily arrays), so we end up having to redimension the return array in our Forall loops. Instead of redimensioning the array for every single element that we're adding, though, we're growing the array 50 elements at a time. This ends up being a lot more efficient from a memory management standpoint, because redimensioning an array is an expensive operation, so we want to do it as few times in the script as possible. At the end of the function, we shrink the return array back down to whatever size it needs to be anyway, so there's no real penalty for adding extra elements as we redimension.

Sorting Arrays

The subject of how to sort the elements of an array most quickly has fascinated programmers for some time now. There are two things that you have to balance when you write an array-sorting algorithm: you want to modify the elements of the array as little as possible, and you want to examine the individual elements of the array as few times as possible. That being said, it's a lot faster to look at an array element than it is to change an element, so you'll generally want to lean towards the side of checking versus changing for the bulk of your algorithm.

While it's a lot of fun to try to create and refine your own sorting algorithm, you'll do best to use one of the "classic" sorting algorithms in your programs for maximum efficiency. The two that I've chosen to present here are the QuickSort algorithm and the Shell Sort algorithm, although there are many others that will also work. For a good discussion about different sorting algorithms, along with LotusScript implementations for them, take a look at the Lotus Redbook titled "Performance Considerations for Domino Applications".

Please note that I am certainly *not* the original author of the sorting algorithms that follow, and in no way am I trying to take credit for writing them. They are simply LotusScript implementations of classic algorithms that are in the public domain.

Script – QuickSort algorithm for array sorting

```
Sub QuickSort (PassedArray As Variant, LowerBound As Integer, UpperBound As Integer)
  '** the classic QuickSort algorithm for sorting an array
  Dim CurValue As Variant
  Dim SwapValue As Variant
  Dim i As Integer
  Dim k As Integer

  '** if there's nothing to sort, don't do anything
  If (UpperBound <= LowerBound) Then
      Exit Sub
  End If

  CurValue = PassedArray(LowerBound)
  i = LowerBound
  k = UpperBound

  While (i < k)
      '** find a value on the low side that's greater than CurValue
      While (PassedArray(i) <= CurValue) And (i < UpperBound)
          i = i + 1
      Wend

      '** find a value on the high side that's smaller than CurValue
      While (PassedArray(k) > CurValue)
          k = k - 1
      Wend

      '** the two values we ended up with need to get swapped
      If (i < k)  Then
          SwapValue = PassedArray(i)
          PassedArray(i) = PassedArray(k)
          PassedArray(k) = SwapValue
      End If
  Wend

  '** do one last swap, since we skipped PassedArray(LowerBound)
```

```
   SwapValue = PassedArray(LowerBound)
   PassedArray(LowerBound) = PassedArray(k)
   PassedArray(k) = SwapValue

   '** Now all the data on the low side of myArray(k) should be smaller
      '** than CurValue, and all the data on the high side of myArray(k)
      '** should be larger. We can use recursion to sort data on either
      '** side of k.
   Call QuickSort (PassedArray, LowerBound, k - 1)
   Call QuickSort (PassedArray, (k + 1), UpperBound)

  End Sub
```

Script – Shell Sort algorithm for sorting an array

```
  Sub ShellSort (PassedArray As Variant)
   '** the classic ShellSort algorithm for sorting an array

   Dim i As Integer
   Dim k As Integer
   Dim LowerBound As Integer
   Dim UpperBound As Integer
   Dim TempValue As Variant
   Dim SwapValue As Variant

   LowerBound = Lbound(PassedArray)
   UpperBound = Ubound(PassedArray)

   '** start with elements that are half the length of the array apart
   k = (UpperBound - LowerBound + 1) \ 2

   Do While k > 0
       '** from the bottom of the array to the top, swap everything
       '** that's k elements apart and out of order
       For i = LowerBound To UpperBound - k
           If ( PassedArray(i) > PassedArray(i + k) ) Then
               TempValue = PassedArray(i)
               PassedArray(i) = PassedArray(i + k)
               PassedArray(i + k) = TempValue
           End If
       Next i

       '** now go back down the array, swapping again
       For i = UpperBound - k To LowerBound Step -1
           If ( PassedArray(i) > PassedArray(i + k) ) Then
               TempValue = PassedArray(i)
               PassedArray(i) = PassedArray(i + k)
               PassedArray(i + k) = TempValue
           End If
       Next i

       '** divide k in half and do it again, until we get down to 0
       k = k \ 2
   Loop

  End Sub
```

Most of what I know about how these algorithms work is documented in the code comments above. Both routines work quite fast, although the QuickSort algorithm is often just a tiny bit faster. Some people like to use QuickSort because of the small increase in speed, while others like ShellSort because it doesn't use recursion and you have no chance of blowing the stack when you use it. Either one should work well for virtually any programming situation.

Numeric Functions

LotusScript has the following 5 numeric data types:

Data type	Suffix	Value range	Size
Integer	%	-32,768 to 32,767	2 bytes
Long	&	-2,147,483,648 to 2,147,483,647	4 bytes
Single	!	-3.402823E+38 to 3.402823E+38 smallest non-zero value is 1.175494351E-38	4 bytes
Double	#	-1.7976931348623158E+308 to 1.7976931348623158E+308 smallest non-zero value is 2.2250738585072014E-308	8 bytes
Currency	@	-922,337,203,685,477.5807 to 922,337,203,685,477.5807 smallest non-zero value is .0001	8 bytes

All numeric variables initialize to a value of zero. If you do not need to use decimal places, you can use an Integer or a Long data type. If you do need decimals, or if you are just dealing with numbers larger than what a Long can handle, you can use Currency, Single, or Double (often, a numeric data type with decimal places is called a "float").

If you are trying to store a smaller data type into a larger data type – for example, if you want to store an Integer value in a Long variable – LotusScript will allow you to do this without any fuss (this is a process known as "implicit casting"). If you want to go the other way and store a larger data type in a variable of a smaller data type, LotusScript will actually allow you to do this implicitly as long as the value will fit in the smaller data type. For example:

```
longVal& = 1000
intVal% = longVal&
```

will work without errors, but:

```
longVal& = 100000
intVal% = longVal&
```

will give an Error 6: Overflow, because the number 100,000 is larger than the biggest acceptable Integer value. In practice, however, you will normally want to do some error checking before you "downcast" from a larger data type to a smaller one. An easy way to do this is to define some constants in the Declarations section of your script, and compare against those.

```
'** (in Declarations)
Const MAX_INTEGER = 32767
Const MAX_LONG = 2147483647
Const MAX_SINGLE = 3.402823E+38
Const MAX_DOUBLE = 1.7976931348623158E+308
Const MAX_CURRENCY = 922337203685477.5807

'** (in your script)
longVal& = 100000
If (longVal& <= MAX_INTEGER) Then
 intVal% = longVal&
End If
```

The advantage to using constants rather than the actual maximum number in the script is that you don't have to type the maximum value numbers for a data type more than once in your script, which will save you a lot of headaches and typos. It's also a lot easier to look at.

The native LotusScript functions for numeric manipulation are:

Function/Statement	Usage	Example
Abs	Returns the absolute value of a number.	
ACos	Returns the arccosine, in radians, of a number between -1 and 1, inclusive.	
ASin	Returns the arcsine, in radians, of a number between -1 and 1, inclusive.	
Atn	Returns the arctangent, in radians, of a number.	
Atn2	Returns the polar coordinate angle, in radians, of a point in the Cartesian plane.	
Bin[$]	Returns the binary value of a number (as a string).	
CCur	Returns a value converted to Currency.	
CDbl	Returns a value converted to a Double.	

CInt	Returns a value converted to an Integer.	
CLng	Returns a value converted to a Long.	
Cos	Calculates the cosine of an angle.	
CSng	Returns a value converted to a Single.	
Exp	Returns the exponential (base e) of a number.	
Fix	Strips the decimal places from a number and returns only the integer portion.	
Fraction	Returns only the decimal part of a number.	
Hex[$]	Returns the hex value of a number (as a string).	
Int	Returns an integer value that is less than or equal to a number.	
IsNumeric	Determines whether an expression is numeric, or can be converted to a numeric value. Keep in mind that this doesn' t necessarily mean that something is already a numeric value, because the string "1234" will also return True.	
Len	For a numeric variable or value, returns the number of bytes used to hold that value, as seen in the table in the beginning of this chapter.	
LenB	For a numeric value, returns the same as Len (above).	
LenBP	For a numeric value, returns the same as Len (above).	

Log	Returns the natural (base e) logarithm of a number.	
Mod	Divides two numbers and returns the remainder (modulus).	
Oct[$]	Returns the octal value of a number (as a string).	
PI	Returns the mathematical constant PI (3.14…).	
Randomize	Initializes the internal random number generator.	
Rnd	Returns a random number greater than 0 and less than 1 (multiply the result by powers of 10 in order to return larger numbers).	
Round	Rounds a number to a specified number of decimal places.	
Sgn	Indicates whether a number is positive (1), negative (-1), or zero (0). Note that this does not return a Boolean value.	
Sin	Returns the sine, in radians, of an angle.	
Sqr	Returns the square root of a number.	
Tan	Returns the tangent, in radians, of an angle.	
Val	Converts a string value to a number.	

In addition, LotusScript provides the following functions for working with Boolean values:

Function/Statement	Usage	Example
And	Performs a conjunction on two values (logical or bitwise).	
Eqv	Performs an equivalence on two values (logical or bitwise).	
FALSE	The Boolean value FALSE (0).	
Imp	Performs an implication on two values (logical or bitwise).	
Not	Performs negation on a value (logical or bitwise).	
Or	Performs a disjunction on two values (logical or bitwise).	
TRUE	The Boolean value TRUE (technically -1, although any non-zero number is considered to be TRUE, for comparison purposes).	
Xor	Performs an exclusion on two values (logical or bitwise).	

Because LotusScript (as of R5) does not have a Boolean data type, you will normally store and pass Boolean values around as Integers. There are a few things to keep in mind here:

- If you explicitly set a variable to True in LotusScript, its value is –1 (not 1, like in some other languages). However, if you are determining whether a value or an expression evaluates to True or False, any non-zero number evaluates to True.

- The Boolean operators shown above (And, Or, Xor, etc.) are both logical and bitwise operators. This means that if you are operating on two strictly Boolean values, you will get the expected Boolean result. However, if you are operating on one or more "non-Boolean" values (like some non-zero number that evaluated to True), the comparison will be bitwise (binary). For example, "True Xor True" evaluates to "False", but "20 Xor 5" evaluates to "17" (which is True in the Boolean world). More on binary operations later in the chapter.

Mathematical Operators and Operator Precedence

The mathematical operators used in LotusScript are as follows:

Operator	Operation Performed
^	Exponentiation
*	Multiplication
/	Regular division (returns a floating point result)
\	Integer division (rounds the operands to integers, divides, and returns only the integer part of the answer)
+	Addition
-	Subtraction or Negation (depending on context)

If you have multiple mathematical operators in a single expression, then LotusScript adheres to some fairly standard rules for "order of operation" when it evaluates the statement. What this means is, the operators are not evaluated strictly from left to right, they are evaluated preferentially from left to right. For example, in the expression:

```
2 + 3 * 4
```

the result is 14, because the multiplication has a higher preference than the addition does, and it therefore gets evaluated first (so we multiply 3 times 4, and then add 2). This is not just some strange thing that LotusScript does, though. It' s just how math works. If you enter this expression into any kind of calculator, you' ll get the same result.

The order of operation for mathematical operators is:

( )	Expressions bracketed in Parenthesis always get evaluated first
^	Exponentiation
-	Negation (not subtraction, which is farther down)
* /	Multiplication and division

\	Integer division
Mod	Modulo division (remainder)
- +	Subtraction and addition
& +	String concatenation
=, <>, ><, <, <=, =<, >, >=, =>, Like	Numeric or string comparison
Not	Logical or bitwise Negation
And	Logical or bitwise And
Or	Logical or bitwise Or
Xor	Logical or bitwise Exclusive-Or
Eqv	Logical or bitwise Equivalence
Imp	Logical or bitwise Implication
Is	Object reference comparison

Before you get too frustrated trying to think of how you'll remember this list of operator preference, keep in mind that you can always just put parenthesis around your expressions to make sure they evaluate in the order you want them to. That's really the recommended way to do it, because it not only makes the code easier to read, but it also keeps you from having to memorize the above list.

For instance, in the example we had earlier, if we really wanted the addition to take place prior to the multiplication, we could just write:

```
((2 + 3) * 4)
```

and we will get our (expected?) answer of 20. Likewise, to make sure that the expression is easily maintainable from the coding standpoint, if you did want the multiplication to happen first, you should really write the expression like this:

```
(2 + (3 * 4))
```

This just makes the statement much more obvious to interpret (and therefore, less error-prone).

Binary, Hex, and Octal Numbers

With the exception of a few very unusual people, we tend to think of numbers as either integers (like 500) or decimals (like 3.14). In mathematical terms, these are "base 10" numbers. What this essentially means is that there are 10 possible characters (0 through 9) that you can use to construct a number.

There are certain cases where it makes more sense to think of numbers in other, non-base 10 numbering systems. LotusScript provides the capability to natively represent numbers in 3 other numbering systems: binary (base 2), octal (base 8), and hexadecimal (base 16). Of these, binary and hexadecimal (usually referred to as "hex") are most common. These numbers can be stored in either Integer or Long data types, and they can be entered like this:

```
binaryNumber& = &B1001001    '** prefixed by &B or &b
octalNumber& = &O1786        '** prefixed by &O or &o (the letter "o")
hexNumber& = &Hffee0         '** prefixed by &H or &h
```

An interesting (and sometimes confusing) thing to keep in mind is that any binary, octal, or hex number that is larger than the maximum allowable value for the data type that it's using (either Integer or Long) is treated as a negative number. So the hex representation of the number –1 is FFFFFFFF, -32,768 is FFFF8000, and -2,147,483,648 is 80000000. This is because of the "sign bit" on the number, which is discussed later in the chapter.

Also, even though you might assign a variable using the notation for, say, a binary number, if you use or display the variable, you'll see it as a regular base 10 value. For example, if you run the following script:

```
myInt% = &B1001
Print myInt%
```

the output will be "9" (which is the binary number 1001 converted to base 10), not "1001". If you want to see the number in its binary, octal, or hex representation, you need to use the Bin, Oct, or Hex functions. If you want to convert a binary, octal, or hex string back to a base 10 number, we'll provide you with a function for doing this later in the chapter.

While you may not immediately be able to think of a time where it would even make sense to deal with non-base 10 numbers, you will occasionally run across situations where it makes your life a lot easier. Network subnets and bitmasking (often used in C programming, and sometimes used by the LotusScript programmer when accessing DLLs) are two common examples. You will see others later in this chapter, and in the chapters on accessing DLLs.

Very Large and Very Small Number Representation

When representing very large or very small numbers, Notes often uses scientific notation. For example, the largest value for a variable of type Single is 3.402823E+38. That "E+38" at the end of the number means that the decimal place in 3.402823 needs to be moved over 38 places to the right, which actually translates into the number 340,282,300,000,000,000,000,000,000,000,000,000,000.

A simpler example to grasp is that the number 208 can be expressed in scientific notation as 2.08E+2. This could also technically be written as 2.08E2 (the plus sign is implied), but it's generally good practice to specify the plus sign.

You can also express very small numbers in scientific notation by using E-# instead of E+#. For example, the number 0.000999 can also be expressed as 9.99E-4. The E-4 means that you should move the decimal place over 4 places to the left.

Using the Mod Function

The Mod function is one of those things that can be very useful, but is underutilized by a lot of people because they don't really understand what it does. By definition, the Mod function will give you the remainder from two numbers that are divided. So, 5 Mod 2 = 1, because 5 divided by 2 equals 2.1, and 1 is the remainder. This seems simple but useless.

However, there are many situations where this is actually a nice shortcut. For example, if you're stepping through thousands of documents in a view and you want to output a running count of documents, you could write some code like this:

```
Do Until (doc Is Nothing)
  '** code here…
  count& = count& + 1
  If (count& Mod 100 = 0) Then
      Print "Processing document # " & count&
  End If
  Set doc = view.GetNextDocument(doc)
Loop
```

This will tell you about every hundredth document that gets processed, which is much nicer and more memory-efficient (if there are a lot of documents) than being told about each one.

There will be other practical examples of the Mod function later in the chapter.

Custom Routines

The rest of this chapter will consist of custom subs and functions that will demonstrate ways to manipulate numbers.

Determining Whether a Number is Odd or Even

We'll start off with something fairly simple here: figuring out whether a number is odd or even. This will be a good use of the Mod function, which we discussed earlier.

Script – Determine if a number is even

```
Function IsEven (num As Variant) As Integer
  Dim tempNum As Double

  '** if this isn't a number, just exit
  If Isnumeric(num) Then
      tempNum = Cdbl(num)
  Else
      IsEven = False
      Exit Function
  End If

  '** use Mod to figure out if the number is
  '** even or odd
  If (tempNum Mod 2 = 0) Then
      IsEven = True
  Else
```

```
        IsEven = False
    End If
  End Function
```

Script – Determine if a number is odd

```
  Function IsOdd (num As Variant) As Integer
    Dim tempNum As Double

    '** if this isn't a number, just exit
    If Isnumeric(num) Then
        tempNum = Cdbl(num)
    Else
        IsOdd = False
        Exit Function
    End If

    '** use Mod to figure out if the number is
    '** even or odd
    If (tempNum Mod 2 = 1) Then
        IsOdd = True
    Else
        IsOdd = False
    End If
  End Function
```

Okay, here are some of the finer points of the code. First, by allowing the user to pass the value they want to check as a Variant, they can check any of the numeric data types with one function, so you don't have to write a separate function for Integers, Longs, etc. We just use the built-in IsNumeric function to make sure we're actually dealing with a number. Second, we're using Mod to see if a number is even or odd. Because Mod gives you the remainder from the division of 2 numbers, and we know that any even number divided by 2 has a zero remainder, we can also know that if a number Mod 2 equals zero, then it's an even number.

Granted, these functions aren't written very elegantly. You could rewrite the last If-Then block in the IsEven function as simply:

```
  IsEven = Not(tempNum Mod 2)
```

However, that's much harder to understand just by looking at it, and so it doesn't serve as a good book example in this case. Also, the IsOdd function is redundant, since you could really just figure out a number is odd by running the IsEven function and inverting the result (if you believe that anything that's not even is odd).

Remove Non-Numeric Elements from a String

Often, you will end up with a string that's supposed to represent a number, but it may contain non-numeric characters (like formatted phone numbers or social security numbers). Here's a function that will get rid of the non-numeric elements for you.

Script – Remove non-numeric elements from a string

```
  Function StripNonNumeric (numberString As String) As String
    '** convert a text string to a string of numbers, stripping any
    '** non-numeric characters.
    '** for example, StripNonNumeric("123-45-6789") = "123456789"
    Dim i As Integer
    Dim tempChar As String
    Dim returnString As String

    For i = 1 To Len(numberString)
        tempChar = Mid$(numberString, i, 1)
        If (Asc(tempChar) >= 48) And (Asc(tempChar) <= 57) Then
            '** we found a number
            returnString = returnString & tempChar
        End If
    Next

    StripNonNumeric = returnString

  End Function
```

The logic is pretty simple: go through the string one character at a time, and create a new string using only the characters that are numbers. In the if-then block, the number 48 is the ASCII code for 0, and 57 is the ASCII code for 9.

It' s also easy to extend the function a little if you have special cases. For example, you might want a version of the function that optionally allows you to have negative numbers and decimal places, as follows.

Script – Remove non-numeric elements, allowing negative numbers and a decimal place

```
Function StripNonNumeric2 (numberString As String, allowNegative As Integer,
allowDecimal As Integer) As String
  '** convert a text string to a string of numbers, stripping any
  '** non-numeric characters. Optionally allows you to keep
  '** a leading negative sign and/or a decimal place (if any)
  '** For example, StripNonNumeric2("123-45-6789") = 123456789
  Dim i As Integer
  Dim tempChar As String
  Dim returnString As String

  For i = 1 To Len(numberString)
      tempChar = Mid$(numberString, i, 1)
      If (Asc(tempChar) >= 48) And (Asc(tempChar) <= 57) Then
          '** we found a number
          returnString = returnString & tempChar
      Else
          '** special non-number cases
          If allowNegative And (tempChar = "-") And (returnString = "") Then
              returnString = returnString & tempChar
          Elseif allowDecimal And (tempChar = ".") Then
              returnString = returnString & tempChar
              '** set allowDecimal to False, so we only have
                          '** one decimal place
              allowDecimal = False
          End If
      End If
  Next

  StripNonNumeric2 = returnString

End Function
```

You could add similar logic to allow for currency symbols, scientific notation, etc.

Formatting Numbers

LotusScript has a built-in Format function that' s really good for formatting numbers as strings. The Designer Help database has all the information you' ll need, as far as the options and parameters for the function are concerned, but here are a few examples to get you started.

Script – Adding commas to a number string

```
numberWithCommas = Format$(num&, "#,#")
```

Script – Displaying a number string as currency

```
currencyNumber = Format$(num@, "Currency")
```

Script – Forcing a certain number of decimal places

```
decimal4 = Format$(num!, "#,#.0000")
```

Script – Add leading zeros, if necessary

```
leadingZeros = Format$(num%, "00000")
```

Script – Force scientific notation display

```
scientific = Format$(num!, "#.###E+")
```

There are many other options and combinations you can use – please consult the help file for more information. Just keep in mind that the Format function is very flexible, and check out its capabilities before you decide to write a custom routine to, say, add commas to a number string output.

By the way, the "#,#" format string will add a thousands separator every 3 digits. At first glance, you might think that it will only display the first 2 numbers separated by a comma, but that' s not the case. Try it out!

Converting Numbers to and from Base 10

LotusScript has built-in functions to convert numbers from base 10 (normal, everyday integer-type numbers that we' re used to seeing and counting with) to either base 2 (binary), base 8 (octal), or base 16 (hex). In the case that you need to convert any of these non-base 10 numbers back to base 10 – or, for some reason, to convert to and from another base – here are some conversion functions.

Script – Convert from any base (<= 36) to base 10

```
Function ConvertToBase10 (thisNumString As String, cBase As Integer) As Long
  '** This function takes a non-base 10 number (of base cBase) and
  '** converts it to a base 10 Long number.
  On Error Goto processError

  Dim tempString As String
  tempString = Trim(Ucase(thisNumString))

  '** if someone passes us an empty string, just return 0
  If (tempString = "") Then
      ConvertToBase10 = 0
      Exit Function
  End If

  '** also, if someone wants to convert from a base bigger than 36,
     '**return 0
  If (cBase < 2) Or (cBase > 36) Then
      ConvertToBase10 = 0
```

```
        Exit Function
    End If

    '** do some special conversions for Binary, Octal, and Hex numbers, '** which might
  be represented in a different format
    If (Len(tempString) > 1) Then
        If (cBase = 2) And (Left$(tempString, 2) = "&B") Then
            tempString = Right$(tempString, Len(tempString) - 2)
        Elseif (cBase = 8) And (Left$(tempString, 2) = "&O") Then
            tempString = Right$(tempString, Len(tempString) - 2)
        Elseif (cBase = 16) And (Left$(tempString, 2) = "&H") Then
            tempString = Right$(tempString, Len(tempString) - 2)
        Elseif (cBase = 16) And (Ucase(Left$(tempString, 2)) = "0X") Then
            tempString = Right$(tempString, Len(tempString) - 2)
        End If
    End If

    Dim convString As String
    convString = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    Dim checkstring As String
    Dim tempnum As Long
    Dim convnum As Integer

    '** for each of the digits in tempString, convert them to base 10
    '** by getting their value as an integer and raising them to the
    '** appropriate power of cBase. Stop when you either get to the end
    '** or a non-convertable character
    i% = 0
    For k% = Len(tempString) To 1 Step -1
        checkstring = Mid$(tempString, k%, 1)
        checknum = Instr(1, convString, checkstring, 5)

        If (checknum < 1) Or (checknum > cBase) Then
            Exit For
        Else
            tempnum = tempnum + ((checknum - 1) * (cBase ^ i%))
            i% = i% + 1
        End If
    Next

    ConvertToBase10 = tempnum
    Exit Function


  processError:
    Dim lastError As String
    lastError = "Error " & Cstr(Err) & ": " & Error$
    ConvertToBase10 = 0
    Exit Function


  End Function
```

Script – Convert from base 10 to any base (<= 36)

```
  Function ConvertFromBase10 (thisNum As Long, cBase As Integer) As String
    '** converts a base 10 number to another base (cBase),
    '** up to base 36
    On Error Goto processError

    '** if someone passes us a number less than 1, just return "0"
    If (thisNum < 1) Then
        ConvertFromBase10 = "0"
        Exit Function
    End If

    '** also, if someone wants to convert to a base bigger than 36,
       '** return "0"
    If (cBase < 2) Or (cBase > 36) Then
        ConvertFromBase10 = "0"
        Exit Function
    End If

    Dim convString As String
    convString = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    Dim numList List As Integer
    Dim curDigit As Integer
    Dim nextDigit As Long
    Dim tempnum As Long

    '** convert the base 10 number to a list of integers (numList),
    '** each one being a digit of the new base cBase number.
    i% = 0
    tempnum = thisNum
    nextDigit = (tempnum \ cBase)

    Do While (tempnum > cBase - 1)
        curDigit = (tempnum Mod cBase)
        numList(i%) = curDigit
        i% = i% + 1

        tempnum = nextDigit
        nextDigit = (tempnum \ cBase)
    Loop

    '** also include whatever's left over
    numList(i%) = tempnum

    '** now each element of numList will be a base cBase digit (from
       '** 0 to cBase-1), that we can convert to a cBase digit. We pick
       '** the digits from the list in the convString string, such that
       '** the number 0 is the first element of the string, the number 1
    '** is the second element, etc.
    Dim tempString As String
    Forall digit In numList
        tempString = Mid$(convString, digit + 1, 1) & tempString
    End Forall

    ConvertFromBase10 = tempString
    Erase numList
    Exit Function

   processError:
    Dim lastError As String
    lastError = "Error " & Cstr(Err) & ": " & Error$
    ConvertFromBase10 = ""
    Erase numList
    Exit Function


   End Function
```

The assumption here is that a non-base 10 numbering system will use digits that increase in value from 0 to 9, and then from A to Z (just like hex numbers do). If this is not true, you will need to adjust the "convString" variable, so that it contains all the digits in your number system, from lowest to highest.

Otherwise, this is just straight math, which unfortunately is easier to explain by looking at the code than by trying to tell you what all the calculations are. It's really not that complex once you understand what you're supposed to do, but, like all things mathematical, knowing what to do is the hard part.

Using AND, OR, and XOR as Bitwise Operators

When you use a "bitwise" operator, this means that the operator is comparing the bits of the binary representation of two numbers. Below is a table of bitwise operator calculations:

bit 1	bit 2	AND	OR	XOR	EQV	IMP
1	1	1	1	0	1	1
1	0	0	1	1	0	0
0	1	0	1	1	0	1
0	0	0	0	0	1	1

Of these operators, you will use AND, OR, and XOR the most. Let's look at some examples to try to make this a little easier to grasp.

Let's say you have two binary numbers &B0011 and &B1010, and you want to apply the AND operator to them and get the result. Here's a table of what happens:

bit position	first number	second number	result of 2 bits being ANDed together (from calculation table above)
0	1	0	0
1	1	1	1
2	0	0	0
3	0	1	0

So your resulting number is &B0010. To get the resulting bit in position 0 (the position farthest to the left), you look at the bits that are in that position in both numbers (1 and 0), and you get the result of "1 AND 0" from the previous table. Then you move on and do that for all the bits in all the positions. It's a lot like adding two numbers (from left to right, you determine from an addition table what the sum of two digits are), except you don't carry anything.

The same rules apply for all the other bitwise operators: calculate each bit position one at a time, using the table on the previous page. If one of the two numbers is "shorter" than another (it has fewer significant bits), you simply left-pad the shorter number with zeros to make them equal in length.

By the way, the reason I'm referring to these as "bitwise" operators is because the AND, OR, XOR, etc. operators can also operate on logical/Boolean values (in which case, they're referred to as "logical" operators). You can use the same table to calculate the results, but you simply replace "1" with "True" and "0" with "False" in the calculation and the result.

You need to be a little careful about the difference between a bitwise and a logical operator when you're working with logic in a script, however. For example, a lot of people like to shorten their scripts by using the fact that LotusScript treats all non-zero numbers as Boolean True, as in this statement:

```
someNum% = a + b
If (someNum%) Then
  '** if someNum% is not equal to zero, do something here…
End If
```

This works as expected, evaluating the If-Then statement when the "someNum" variable is a non-zero value. However, if you try to do this:

```
someNum% = a + b
If Not (someNum%) Then
  '** you want to get here if someNum% is equal to zero,
  '** but it's not going to work this way
End If
```

then it will only work if the "someNum" variable is exactly equal to a binary number that is all ones (like &B111). That is because the Not is used as a bitwise operator in that context, since it is operating on a number, not a true Boolean value. Be careful!

One last point before we move on: the bitwise operators only work properly on number systems that are Base 2 or a base that is a power of 2 (like Octal/Base 8 and Hex/Base 16). Hex is often very convenient to use when working with large numbers, because "F" is actually the binary "1111", so you can easily construct a long string of ones by appending a few F's together. However, if you try to use bitwise operators on Base 10 integers, you may get some unexpected results.

Using Bitwise AND for "Masking" Numbers

A number "mask" is a method of removing or modifying certain significant digits from another number, so that you can see or operate on other specific digits in the number. In programming, number masking is often used to allow the programmer to look at individual bits of a binary number or at different ranges of hexadecimal numbers, when each bit or range might specify something different (for example, with error codes).

We'll start with an easy example. Let's say you have a binary number in your program, where the third bit to the left (like the "1" in the binary number "00100") indicates whether or not a program option is enabled. To check this, we can set up a mask of binary "100" (&B100), and apply it to the number using the AND operator.

```
Const MY_MASK = &B100
maskedNum% = checkNum% And MY_MASK

If (maskedNum% = MY_MASK) Then
    isThirdBitSet% = True
Else
    isThirdBitSet% = False
End If
```

The way the mask works is that when you apply the mask with the AND operator, every place there's a zero (which is everywhere before and after the "1", in this case) becomes a zero, and every place there's a one remains a one as long as that place was already one. In other words, if the number has ones in every place the mask has a one, then the number satisfies the mask condition, and after the AND operation, it will equal the mask.

You might often see the above script segment written as this:

```
maskedNum% = checkNum% And MY_MASK
If (maskedNum%) Then
    isThirdBitSet% = True
Else
    isThirdBitSet% = False
End If
```

or, even less responsibly:

```
maskedNum% = checkNum% And MY_MASK
isThirdBitSet% = maskedNum%
```

Be very careful when you start pretending that all non-zero numbers are Boolean True values. In the above case, this sort of works because we're only checking one bit, but if the mask was checking for multiple bits (like &B0101), then you would end up with a false "True" result even if only one of the bits matched. If that's what you want, I suppose it works out fine, but you can really easily spend a lot of time debugging around that sort of thing.

Thinking about it one way, masking allows you to check to see if certain bits have been "set", as in the example above (you can check multiple bits just by adding ones in other places). Thinking about it another way, masking can allow you to discard certain bits from a number so you only have to deal with the other bits. A good example of this is subnet masking, which is used quite often in TCP/IP networks.

A common subnet mask is 255.255.255.0. In binary, this corresponds to 11111111.11111111.11111111.00000000. When you apply that mask to an IP address – say, 192.168.1.123 – it returns the first 3 octets of the address (192.168.1.0), which is the subnet that the address is in. For a piece of networking equipment on the WAN, those first 3 octets are the only ones that are significant (on that particular network) in terms of knowing where to send a packet of data. If your subnet mask is 255.255.255.128 (where 128 is 10000000 in binary), then all the IP addresses get masked to either #.#.#.128 or #.#.#.0, so you've doubled the number of subnets you can look at.

Using Bitwise OR for Combining Numbers

While the bitwise AND operator can be used to strip (or mask) individual bits from a number, the OR operator can be used to combine the bits of numbers. For example, let's say you have a pair of binary numbers, &B1000 and &B0001. By "ORing" the two numbers together, you get &B1001. The OR operator will put a "1" in every bit position where there's a "1" in either (or both) numbers, and will leave "0" everywhere else.

The concept of combining number bits comes into play every so often when you're working with external DLL functions and you need to specify a certain combination of options. For example, here's a call to the Windows API function that opens a file:

```
Const OF_READ = &H0
Const OF_SHARE_DENY_NONE = &H40

hFile = OpenFile(fileName, FileStruct, OF_READ Or OF_SHARE_DENY_NONE)
```

In this function call, the options to open a file for read mode (OF_READ) and in share mode (OF_SHARE_DENY_NONE) are combined so that the file is opened for both reading and sharing.

Using Bitwise XOR for "Flipping" Bits

The nature of the XOR operation makes it good for "flipping" one or more bits in a number. In other words, if the bit in a certain position is 0, you can "flip" it to 1, and if it's 1, you can "flip" it to 0. As an example, let's say you want to change the bit in bit position zero. You can simply do this:

```
myNumber% Xor &B0001
```

and the bit in position zero will be flipped. If you perform the identical operation again, the bit will be flipped back. This type of functionality is often used in data encryption (because it's reversible) and binary addition, among other things.

Signed and Unsigned Numbers, and the Sign Bit

In the binary representation of a number, the most significant bit (the one farthest to the left) indicates whether a number is odd or even. For example, the Integer number &B0000000000000001 is positive, but the Integer number &B1000000000000000 is negative. For a Long number, the sign bit is twice as far to the right (bit position 31 instead of 15).

Also, smaller negative numbers are left-padded with ones, so that the representation of –1 as an Integer is &B1111111111111111 (or, more conveniently, &HFFFF), while &B1000000000000001 represents –32,767.

Getting and Setting the Bit in a Given Position

Let's put all this information about bit manipulation together with a few examples. First, here's a function that will return the value of a bit in a specified position.

Script – Function that gets the bit value in a given position

```
Function GetBitVal (theNum As Long, whichBit As Integer) As Integer
```

```
    '** gets the value of the bit (0 or 1) in the given position
    '** (from 0 to 31)
    Dim tempNum As Long

    If (whichBit > 31) Or (whichBit < 0) Then
        GetBitVal = -1
        Exit Function
    End If

    If (whichBit = 31) Then
        '** this is the sign bit
        tempNum = theNum And &H80000000
    Else
        tempNum = theNum And (2 ^ whichBit)
    End If

    If (tempNum = 0) Then
        GetBitVal = 0
    Else
        GetBitVal = 1
    End If

  End Function
```

With this function, we pass a Long number value and the bit position we want to look at (from 0 to 31), and value of the bit in the specified position is returned (either 0 or 1, or –1 if the bit position is outside the legal range of values). We count from 0 to 31 instead of from 1 to 32 because of the way that binary numbers are represented: the bit position that' s farthest to the right represents 2 ^ 0 when it' s set, so it' s position 0, the next one to the left represents 2 ^ 1 when it' s set, so it' s position 1, and so on.

As we mentioned earlier, the bit in the farthest position to the left is the sign bit. We can' t check for that bit as 2 ^ 31, because that will overflow the function. Technically, that bit (as a Long number) is -2,147,483,648; however, it' s easier to refer to it as &H80000000, as we did in the script.

You can also write an Integer version of this function, where the maximum bit position is 15, and the sign bit is &H8000.

The complementary function to this is one that sets the value of a bit in a given position and returns the modified number. This process is slightly more complicated, but it' s not too bad.

Script – Function that sets the bit value in a given position

```
  Function SetBitVal (theNum As Long, whichBit As Integer, newVal As Integer) As Long
    '** set the value of a particular bit
    Dim tempNum As Long

    If (whichBit > 31) Or (whichBit < 0) Then
        SetBitVal = theNum
        Exit Function
    End If

    If (newVal <> 0) Then
        newVal = 1
    End If

    '** using OR will always set the value to 1. We can then use XOR
    '** to flip the bit, if it's really supposed to be zero
    If (whichBit = 31) Then
        '** this is the sign bit
        tempNum = theNum Or &H80000000
        If (newVal = 0) Then
            tempNum = theNum Xor &H80000000
        End If
    Else
        tempNum = theNum Or (2 ^ whichBit)
        If (newVal = 0) Then
            tempNum = theNum Xor (2 ^ whichBit)
        End If
    End If

    SetBitVal = tempNum

  End Function
```

So, when we're getting a bit value, we use the AND operator to apply a mask to the number and see what we get. When we're setting a bit value, we use OR to change the bit to 1 (while leaving the other bits alone), and then use XOR to flip that bit if it's really supposed to be 0.

Bit Shifting

"Bit shifting" is the process of taking all the bits in a binary number and moving them a certain number of positions to the left or to the right. For example, if you have the Binary number 00001010 and you shift it to the left 1 position, it becomes 00010100; if you shift it to the left 2 positions, it becomes 00101000, and so on. Likewise, if you shift it to the right 1 position, it becomes 00000101; if you shift it right 2 times, it becomes 00000010, etc.

Each time you shift a binary number to the left, you're technically multiplying that number by 2. While this may not seem too intuitive at first, think about numbers in a normal Base 10 number system. If you multiply a number by 10, you're really just moving all the numbers over to the left and sticking a zero on the end. In the same way, in a Binary (Base 2) system of numbers, multiplying by 2 will move all the numbers to the left and add a zero to the end. Conversely, shifting a Binary number to the right is the same thing as dividing by 2.

So why would you ever want to shift bits one way or another? Well, one common use of bit shifting is with cryptographic encryption and hashing algorithms. Or you might need to convert some C code into LotusScript, and if the code employs any bit-shifting operations, you'll find that LotusScript (at least as of R5) doesn't have any native bit-shifting functions.

Here are some basic bit-shifting functions to look at:

Script – Bit-shift left, optionally maintaining the sign of the number

```
  Function BitShiftLeft (theNum As Long, howMany As Integer, maintainSign As Integer) As
  Long
    '** shift the bits of a number to the left a specified number of times,
    '** optionally maintaining the sign bit of the number
    Dim tempNum As Long
    Dim isSigned As Integer
    Dim leftBit As Long
    Dim mask As Long

    '** exit early if the shift is out of range
    If (howMany > 31) Then
```

```
        BitShiftLeft = 0
        Exit Function
    Elseif (howMany <= 0) Then
        BitShiftLeft = theNum
        Exit Function
    End If

    '** strip the sign bit, and figure out if this is a signed number
    tempNum = theNum And &H7FFFFFFF
    If Not (tempNum = theNum) Then
        isSigned = True
    End If

    '** figure out what the left-most bit will be (we'll have to strip it off
    '** before we multiply)
    leftBit = theNum And (2 ^ (31 - howMany))

    '** strip away the bits that will be shifted away, so we don't overflow
    mask = (2 ^ (31 - howMany)) - 1
    tempNum = tempNum And mask

    '** shift over by multiplying by 2
    tempNum = tempNum * (2 ^ howMany)

    '** adjust the sign bit as necessary
    If (maintainSign And isSigned) Then
        '** set the sign bit if this is a negative number that should
             '** stay negative
        tempNum = tempNum Or &H80000000
    Elseif (leftBit > 0) And Not maintainSign Then
        '** also set the sign bit if we don't care about the sign, and
             '** the left-most bit should be a one
        tempNum = tempNum Or &H80000000
    End If

    BitShiftLeft = tempNum
  End Function
```

Script – Bit-shift right, optionally maintaining the sign of the number

```
  Function BitShiftRight (theNum As Long, howMany As Integer, maintainSign As Integer)
  As Long
    '** shift the bits of a number to the right a specified number of times,
    '** optionally maintaining the sign bit of the number
    Dim tempNum As Long
    Dim isSigned As Integer

    '** exit early if the shift is out of range
    If (howMany > 31) Then
        BitShiftRight = 0
        Exit Function
    Elseif (howMany = 31) Then
        '** special case if we're shifting 31 times
        If (theNum >= 0) Then
            BitShiftRight = 0
        Elseif (maintainSign) And ((theNum And &H40000000) = 0) Then
            BitShiftRight = 1
        Else
            BitShiftRight = 0
        End If
        Exit Function
    Elseif (howMany <= 0) Then
        BitShiftRight = theNum
        Exit Function
    End If

    '** strip the sign bit, if necessary
    If Not maintainSign Then
        tempNum = theNum And &H7FFFFFFF
        If Not (tempNum = theNum) Then
            isSigned = True
        End If
    Else
        tempNum = theNum
    End If

    '** shift over by dividing by 2
    tempNum = tempNum \ (2 ^ howMany)

    '** if we had stripped the sign bit before, we'll want to
    '** add it back to whatever spot it would have shifted to
    If (isSigned And Not maintainSign) Then
        tempNum = tempNum Or (2 ^ (31 - howMany))
    End If

    BitShiftRight = tempNum
  End Function
```

Both functions take as their input the number that we're operating on (theNum), the number of times we want to shift (howMany), and a Boolean value indicating whether or not we want to maintain the sign of the number as we're shifting (maintainSign). The maintainSign flag is important, because the result you get will be different depending on whether or not you want to keep the sign bit of the number constant. This is especially important for when you shift negative numbers to the right. If you want to maintain the sign of the number to keep it negative, then the resulting number (unless it's zero) will be left-padded with ones. If you don't want to maintain the sign, it will be left-padded with zeros.

To watch these functions in action, add them to an agent and add this bit of script to the Initialize sub:

```
    Dim testNum As Long
    Dim zeros As String
    testNum = -1234567
    zeros = String(32, "0")

    Print Right$(zeros & Bin$(testNum), 32)

    Print "Left Shift"
    For i% = 0 To 32
        Print Right$(zeros & Bin(BitShiftLeft(testNum, i%, True)), 32), _
        Right$(zeros & Bin(BitShiftLeft(testNum, i%, False)), 32)
    Next

    Print "Right Shift"
    For i% = 0 To 32
        Print Right$(zeros & Bin(BitShiftRight(testNum, i%, True)), 32), _
        Right$(zeros & Bin(BitShiftRight(testNum, i%, False)), 32)
    Next
```

The basic way to shift left or right is to multiply or divide by 2. However, there are two special things we have to account for. First, the sign bit (in position number 31) will not automatically shift as we're multiplying or dividing. If we want to maintain the sign of a negative number as we're shifting right, this isn't a problem; otherwise, we have to move or manipulate the sign bit manually. For a left-shift, this means figuring out whether or not the bit that would end up in position 31 would be a one or not, and setting the bit with an OR operator if it would. For a right-shift, this means if we're shifting a negative number and we don't want to maintain the sign, we have strip the sign bit and manually set the bit at the position (31 – howMany) after we're done shifting.

The second thing to watch out for is overflowing the result as we're multiplying to left-shift a number. To avoid this, we strip off howMany bits on the left side of the number using an AND mask before we multiply. This ensures that the left-most bits of the number that's being shifted will be zeros, and shifting over a zero is a "safe" operation.

Bit Rotation

Something that's even more obscure than bit-shifting is a concept called bit rotation. With bit shifting, the bits "fall off" at the end of the number when they are shifted beyond the legal range of bit positions. With bit rotation, if a bit would have "fallen off" the left side of the number, it is instead moved to the far right of the number. Likewise, if it would have "fallen off" the right side, it gets moved to the far left. For example:

Original binary number	11100000000000100101101011010000
Left-shifted one position	11000000000001001011010110100000
Left-rotated one position	1100000000000100101101011010001

This is a concept used in some hashing functions, in order to scramble the text of a number. Just in case you ever run into this situation, here's a couple of functions that will perform a left or right rotation of a number for you:

Script – Rotate a number a certain number of bits to the left

```
Function BitRotateLeft (theNum As Long, Byval howMany As Integer) As Long
  '** rotate the bits of a number to the left a specified number of times
  Dim leftSide As Long
  Dim rightSide As Long

  '** adjust if we're supposed to shift more than 31 times
  howMany = howMany Mod 32

  '** calculate the left and right sides of the number
  leftSide = BitShiftLeft(theNum, howMany, False)
  rightSide = BitShiftRight(theNum, (32 - howMany), False)

  '** and OR them together for the result
  BitRotateLeft = leftSide Or rightSide
End Function
```

Script – Rotate a number a certain number of bits to the right

```
Function BitRotateRight (theNum As Long, Byval howMany As Integer) As Long
  '** rotate the bits of a number to the right a specified number of times
  Dim leftSide As Long
  Dim rightSide As Long
  '** adjust if we're supposed to shift more than 31 times
  howMany = howMany Mod 32

  '** calculate the left and right sides of the number
  leftSide = BitShiftLeft(theNum, (32 - howMany), False)
  rightSide = BitShiftRight(theNum, howMany, False)

  '** and OR them together for the result
  BitRotateRight = leftSide Or rightSide
End Function
```

These functions make use of the left- and right-shift functions we wrote earlier to figure out what the left and right sides of the resulting number should be, and then uses the OR operator to combine the left and right sides.

Dealing with Date/Time Values

Date and time values present some of the biggest challenges when programming in LotusScript. This is because there are really 5 different representations of date/time values:

- The NotesDateTime class
- The LotusScript Date Variant (type 7)
- A numeric value (LotusScript data type Double)
- A date/time value stored in a Notes document field
- A simple text representation, as a string

While a date/time value can normally be converted between all of these representations, you have to be very careful to know what type of value you're working with when you want to manipulate or compare values. The most common representations you'll deal with are NotesDateTime, Variant, and string, so we'll address those in the most detail. Below are some brief descriptions of the different representations, along with some common conversions you'll end up using.

NotesDateTime Representation

The NotesDateTime class is a native LotusScript class used specifically for working with date/time values. As such, it has quite a few built-in properties and methods that are handy to use, and it probably offers the highest degree of flexibility of any of the representations.

For a NotesDateTime value ndtDateTime, the following conversions are available:

Variant	String	Number (Double)
ndtDateTime.LSLocalTime ndtDateTime.LSGMTTime	ndtDateTime.DateOnly ndtDateTime.TimeOnly ndtDateTime.LocalTime ndtDateTime.GMTTime	Cdbl(ndtDateTime.LSLocalTime)

One thing you' ll want to watch out for when using multiple NotesDateTime values: if you set two NotesDateTime values equal to each other using the Set command, and you then go back and change one of the values, then the other one will change as well. For this reason (and because of the confusion it causes), you' ll want to set two NotesDateTime values to be equal to each other by using the following technique:

Script – Setting two NotesDateTime values equal to each other

```
Set ndtDateTime1 = New NotesDateTime(Now)
Set ndtDateTime2 = New NotesDateTime(ndtDateTime2.LocalTime)
```

Variant Date Type Representation

The Variant date type (data type 7) is also fairly easy to use, although manipulating the value of a date/time variant is a little more difficult than trying to do the same thing with a NotesDateTime value. However, a lot of date-related LotusScript operations rely on the variant date type, so it ends up being fairly common to work with. The statements Today, Now, Date, and Time all return variant values, and functions like Format and DateValue work with variants but not the NotesDateTime class.

For a variant date/time value varDateTime, the following conversions are available:

NotesDateTime	String	Number (Double)
Set ndtDateTime = New NotesDateTime(varDateTime)	Cstr(varDateTime) Format$(varDateTime, "mm/dd/yyyy")	Cdbl(varDateTime)

String Representation

A string value is a common way to enter or output date/time values, but if you're doing any kind of data manipulation with the value, you'll probably find strings very tedious to work with. The biggest thing to watch out for with string values is the formatting of the value, especially the date portion. Remember that when you convert a string to another date representation, or you convert another representation to a string, LotusScript will automatically use the operating system's Short Date format to interpret or output the value. For example, the string:

"11/02/2001"

Will be interpreted as November 2, 2001 on an American server or workstation, but it will be interpreted as February 11, 2001 on an Italian machine. This really becomes an issue with multinational Notes deployments, especially when you're allowing the user to enter dates or when you have dates hard-coded in scripts.

For a string strDateTime that has a valid date/time format, the following conversions are available:

NotesDateTime	Variant	Number (Double)

Set ndtDateTime = New NotesDateTime(strDateTime)	Cdat(strDateTime)
	DateValue(strDateTime)
	TimeValue(strDateTime)
		Cdbl(Cdat(strDateTime))

Numeric Representation

You normally won't want to manipulate your date/time values as numeric values unless you have some mathematical reason for doing so. This is mostly because the numeric representation of a date/time is kind of confusing, especially to the naked eye.

A Variant date is actually stored internally as a number (although you never see it that way), where the integer portion of the number represents the number of days since December 30, 1899, and the decimal portion of the number represents the fraction of the day that has elapsed since midnight. For example, the date/time January 1, 2001 at 4:00 PM is represented numerically as:

36892.5833333333

As you can see, this isn't a very intuitive way of looking at a date, but it does become useful if you're doing a quick calculation of how many days there are between two dates. For example:

Script 1 – Number of days between two dates

```
total_day% = Fix(Cdbl(Cdat(“07/08/2001”))) - Fix(Cdbl(Cdat(“02/15/2001”)))
```

will give you the number of days between those two dates.

If you are using a numeric date/time value that needs to be converted to another type of date/time representation, you should first convert it to a Variant by using the CDat function, and then convert it using one of the methods described above.

Notes Document Representation

The representation of a date/time value in a Notes document is essentially a variant value, but you usually need to take special care when handling date/time values in Notes documents. This is for two reasons:

- First, what you think is a date/time value in a Notes document will sometimes be a text representation of a date, which will throw you off if you are only expecting a variant date/time value. This can happen if, for example, an agent that hasn't been carefully constructed is used to populate the date/time field.
- Second, when you are updating a Notes document date/time field with script, you can inadvertently add a text value instead of a date/time value if you're not really careful with your script.

The safest way to access and store date/time values in a Notes document is by using the DateTimeValue property of the NotesItem that you're dealing with. This property will return the NotesDateTime value of the item, when applicable. Below are some examples of good practices when getting date/time values from a Notes document, and storing date/time values in a Notes document:

Script 2 – Getting a date/time value from a Notes document

```
Dim doc As NotesDocument
Dim item As NotesItem
Dim ndtDateTime As New NotesDateTime("")
Set item = doc.GetFirstItem("FieldName")
Set ndtDateTime = item.DateTimeValue
```

Script 3 – Adding a date/time value to a Notes document

```
Dim doc As NotesDocument
Dim item As NotesItem
Dim ndtDateTime As New NotesDateTime(Today)
Set item = doc.GetFirstItem("FieldName")
Set item.DateTimeValue = ndtDateTime
```

An even safer (and slightly different) way of getting the date/time value from a Note document is to write a function that does some error-handling for you, as seen in the next script:

Script 4 – Function that gets the date/time value from a Notes document field

```
Function GetFieldDateText (thisDoc As NotesDocument, _
thisField As String) As String
  '** Some versions of Notes will return "12/30/1899" for empty
     '** date fields. This function will circumvent that.

  On Error Goto processError
  Dim dateItem As NotesItem
  Set dateItem = thisDoc.GetFirstItem(thisField)

  If (dateItem Is Nothing) Then
      GetFieldDateText = ""
  Elseif (dateItem.Text = "") Then
      GetFieldDateText = ""
  Else
      GetFieldDateText = Format$(dateItem.Text, "General Date")
  End If

  Exit Function

processError:
  Dim errMess as String
  errMess = Error$
  GetFieldDateText = ""
  Exit Function

End Function
```

This function provides a few useful features. First, it doesn't care if the document field is truly a date/time value or a text value – it simply takes the text value of the field and attempts to format it as a date/time string. Second, it handles any errors that occur if the field doesn't exist, or the field isn't in a proper date/time format, or whatever. Third, it handles the special case where some versions of Notes will return "12/30/1899" as the value of an empty date/time field. You don't normally want to trap for this condition in your scripts, so by using this function you can simply check to see if the returned value is an empty string (""), in which case you can treat the field as though it has no data.

Native LotusScript Date/Time Functions

Before we start writing our own LotusScript functions, let's look at the functions that LotusScript already has for dealing with dates and times.

Function/Statement	Usage	Example
CDat	Used to convert a numeric or string date/time value to a variant.	CDat("03/02/2001") => 03/02/2001
Date[$]	Used to return the current date as a variant (Date) or a string (Date$).	Date$ = "03/02/2001"
DateNumber	Used to convert a set of year, month, and day integer values to a date/time variant.	DateNumber(2001, 3, 2) => 03/02/2001
DateSerial	Same as DateNumber	
DateValue	Used to convert a string date/time value to a variant (returns the date only).	DateValue("03/02/2001 04:05") => 03/02/2001
Day	Used to return the day of the month indicated by a numeric, string, or variant date/time value.	Day("03/02/2001") => 2
Format[$]	Used to format a numeric, string, or variant date/time value as a string in the indicated format (see the Notes help for format options).	Format$("03/02/2001", "mm-dd-yy") => "03-02-01"
Hour	Used to return the hour indicated by a numeric, string, or variant date/time	Hour("04:05 PM") => 16

	value.	
IsDate	Used to indicate whether a string or variant is a valid date/time value.	IsDate("03/02/2001") => True
Minute	Used to return the minute indicated by a numeric, string, or variant date/time value.	Minute("04:05 PM") => 5
Month	Used to return the month indicated by a numeric, string, or variant date/time value.	Month("03/02/2001") => 1
Now	Used to return the current date and time as a variant.	Now => 03/02/2001 04:05:45 PM
Second	Used to return the second indicated by a numeric, string, or variant date/time value.	Second("04:05:45 PM") => 45
Time[$]	Used to return the current time as a variant (Time) or a string (Time$).	Time$ = "04:05:45 PM"
TimeNumber	Used to convert a set of hour, minute, and second integer values to a date/time variant.	TimeNumber(16, 5, 45) => 04:05:45 PM
Timer	Used to get the current time elapsed since midnight.	
TimeSerial	Same as TimeNumber.	

TimeValue	Used to convert a string date/time value to a variant (returns the time only).	TimeValue("03/02/2001 04:05 PM") => 16:05:00
Today	Same as Date	
Weekday	Used to return the weekday indicated by a numeric, string, or variant date/time value (Sunday is 1, Saturday is 7).	Weekday("03/02/2001") => 6
Year	Used to return the year indicated by a numeric, string, or variant date/time value.	Year("03/02/2001") => 2001

In addition, here are the properties and methods of the NotesDateTime class. For the examples, we will use a variable ndtDateTime, which has been set by:

```
Set ndtDateTime = New NotesDateTime("3/2/2001 12:30:00 PM")
```

Property/Method	Usage	Example
DateOnly	A read-only string that' s the date representation of the value.	ndtDateTime.DateOnly => "03/02/2001"
GMTTime	A read-only string that' s the date representation of the value, converted to Greenwich Mean Time.	ndtDateTime.DateOnly => "03/02/2001 04:30:00 PM GMT"
IsDST	A read-only Boolean integer that indicates whether the value reflects daylight savings time.	ndtDateTime.IsDST => False

LocalTime	A read-write string representing the value in the Local time zone.	ndtDateTime.LocalTime => "03/02/2001 12:30:00 PM"
LSGMTTime	A read-only variant that' s the date representation of the value, converted to Greenwich Mean Time.	ndtDateTime.DateOnly => 03/02/2001 04:30:00 PM
LSLocalTime	A read-write string representing the value in the Local time zone.	ndtDateTime.LocalTime => 03/02/2001 12:30:00 PM
TimeOnly	A read-only string that' s the time representation of the value.	ndtDateTime.TimeOnly => "12:30:00 PM"
TimeZone	A read-only integer that represents the time zone of the value.	ndtDateTime.TimeZone => 5 (if this is in EST)
ZoneTime	A read-only string that' s the same as LocalTime, unless ConvertToZone is used.	ndtDateTime.LocalTime => "03/02/2001 12:30:00 PM EST"
AdjustDay	Adds or subtracts the specified number of days from the value.	Call ndtDateTime.AdjustDay(7)
AdjustHour	Adds or subtracts the specified number of hours from the value.	Call ndtDateTime.AdjustHour(12)
AdjustMinute	Adds or subtracts the specified number of minutes	Call ndtDateTime.AdjustMinute(30)

	from the value.	
AdjustMonth	Adds or subtracts the specified number of months from the value.	Call ndtDateTime.AdjustMonth(3)
AdjustSecond	Adds or subtracts the specified number of seconds from the value.	Call ndtDateTime.AdjustSecond(15)
AdjustYear	Adds or subtracts the specified number of years from the value.	Call ndtDateTime.AdjustYear(50)
ConvertToZone	Changes the TimeZone and IsDST properties of the value.	Call ndtDateTime.ConvertToZone(10, False)
New	Used to create a new NotesDateTime value. You can set the value using a string, variant, or the special values "Today" or "Yesterday".	Set ndtDateTime = New NotesDateTime("Today")
SetAnyDate	Changes the date portion of the value to a wildcard, so that the value will match any date.	Call ndtDateTime.SetAnyDate
SetAnyTime	Changes the time portion of the value to a wildcard, so that the value will match any date.	Call ndtDateTime.SetAnyTime
SetNow	Changes the value to the current date	Call ndtDateTime.SetNow

	and time.	
TimeDifference	The difference between two NotesDateTime values, in seconds.	ndtDateTime.TimeDifference(ndtDateTime2) => 3600, if ndtDateTime minus ndtDateTime2 is a difference of one hour

Custom Routines

The rest of the chapter will consist of custom functions, subs, and scripts that you might find useful when dealing with date/time values.

Day of Year Function

This function calculates the day of the year a particular date/time value represents. For example, January 1st is day 1, February 1st is day 32, etc. The input for this function can be a string, variant, or number date/time value.

Script – Get the day of the year

```
Function GetDayOfYear (dateString As Variant) As Integer
    '** calculate the day of the year (from 1 to 365) for a given day
    On Error Goto processError

    Dim firstDay As String
    firstDay = "01/01/" & Cstr(Year(dateString))

    GetDayOfYear = Cint( Cdbl(Datevalue(dateString)) -
Cdbl(Datevalue(firstDay)) ) + 1
    Exit Function

processError:
    Dim errMsg As String
    errMsg = "Error " & Cstr(Err) & ": " & Error$
    GetDayOfYear = 0
    Exit Function

End Function
```

The logic behind this function is fairly simple. We subtract the date we were given with January 1st of that year, and the number of days difference between the dates is the day of year for the date. In order to subtract the dates, we convert them to the numeric representations, subtract them, and get the integer portion of the remainder. The only special thing we want to do is to add one to the result, because we want to return a number from 1 to 365, not from 0 to 364 (think about how January 1st minus January 1st is zero, but January 1st is the first day).

Week of Year Function

This is almost identical to the GetDayOfYear function, except it calculates the week of the year that a given date/time represents.

Script – Get the day of the year

```
Function GetWeekOfYear (dateString As Variant) As Integer
    '** calculate the week of the year (from 1 to 52) for a given day
    On Error Goto processError

    Dim firstDay As String
  Dim dayOfYear as Integer

    firstDay = "01/01/" & Cstr(Year(dateString))
    dayOfYear = Cint( Cdbl(Datevalue(dateString)) -
Cdbl(Datevalue(firstDay)) )

  GetWeekOfYear = (dayOfYear \ 7) + 1
    Exit Function

processError:
    Dim errMsg As String
    errMsg = "Error " & Cstr(Err) & ": " & Error$
    GetWeekOfYear = 0
    Exit Function

End Function
```

The same logic applies to this function as it did for the GetDayOfYear function, but after we find out the day of the year we do an integer division by 7 to determine the week. Also, when we're calculating the day of the year we don't add one, because the first 7 days are the first week, and we need to make sure that all 7 of those days are less than the number 7 (do a few calculations on paper if that doesn't make sense).

Get Day of Week as String

This is really simple, but it's something you'll probably have to do.

Script – Get the string value of the Weekday

```
Function GetWeekdayString(dateString As Variant) As String
  '** Get the Day of the Week as a string
    On Error Goto processError
  Dim dayString as String

  Select Case Weekday(dateString)
  Case 1 : dayString = "Sunday"
  Case 2 : dayString = "Monday"
  Case 3 : dayString = "Tuesday"
  Case 4 : dayString = "Wednesday"
  Case 5 : dayString = "Thursday"
  Case 6 : dayString = "Friday"
  Case 7 : dayString = "Saturday"
  End Select

  GetWeekdayString = dayString
    Exit Function

processError:
    Dim errMsg As String
    errMsg = "Error " & Cstr(Err) & ": " & Error$
    GetWeekdayString = ""
    Exit Function

End Function
```

I don't think any explanation is really needed.

Get Month as String

Another very simple function, that translates the month value of a date into a string.

Script – Get month as string

```
Function GetMonthString(dateString As Variant) As String
  '** Get the Month as a string
  On Error Goto processError
  Dim monthString as String

  Select Case Month(thisdate.LSLocalTime)
  Case 1 : monthString = "January"
  Case 2 : monthString = "February"
  Case 3 : monthString = "March"
  Case 4 : monthString = "April"
  Case 5 : monthString = "May"
  Case 6 : monthString = "June"
  Case 7 : monthString = "July"
  Case 8 : monthString = "August"
  Case 9 : monthString = "September"
  Case 10 : monthString = "October"
  Case 11 : monthString = "November"
  Case 12 : monthString = "December"
  End Select

  GetMonthString = monthString
  Exit Function

processError:
  Dim errMsg As String
  errMsg = "Error " & Cstr(Err) & ": " & Error$
  GetMonthString = ""
  Exit Function

End Function
```

Again, this one is probably so simple that no explanation is necessary.

Calculate the Number of Days Between Two Dates

This script was mentioned earlier in the discussion of the numeric representation of dates. It' s a way to determine the number of days between two date values.

Script – Calculate the number of days between two dates

```
Function DifferenceOfDates (date1 as Variant, _
date2 as Variant) as Long
  '** Get the number of days between two dates
  On Error Goto processError

  DifferenceOfDates = Fix(Cdbl(Cdat(date1))) - Fix(Cdbl(Cdat(date2)))
  Exit Function

processError:
  Dim errMsg As String
  errMsg = "Error " & Cstr(Err) & ": " & Error$
  DifferenceOfDates = 0
  Exit Function

End Function
```

As we discussed earlier, the numeric value of a date is a Double, where the integer part of the number is a date and the decimal part of the number is the time. The Fix function strips the decimal from the date/time numbers, so that we can just subtract two whole numbers and get the number of days in between.

Calculate the Amount of Time Between Two Times

Similar to the last function, but with a slightly different technique, this function will take a pair of time values and return the number of seconds between them.

Script – Calculate the number of seconds between two time values

```
Function DifferenceOfTimes (time1 as Variant, time2 as Variant) as Long
  '** Get the number of seconds between two times
  On Error Goto processError
  Dim seconds1 as Long
  Dim seconds2 as Long

  seconds1 = (Hour(CDat(time1))*24*60) + (Minute(CDat(time1))*60) + Second(CDat(time1))
  seconds2 = (Hour(CDat(time2))*24*60) + (Minute(CDat(time2))*60) + Second(CDat(time2))

  DifferenceOfTimes = seconds1 - seconds2
  Exit Function

processError:
  Dim errMsg As String
  errMsg = "Error " & Cstr(Err) & ": " & Error$
  DifferenceOfTimes = 0
  Exit Function

End Function
```

The way this one works is by converting the time value to hours, minutes, and seconds, and figuring out the difference of seconds from there.

Determine the Number of Weekend Days Between Two Dates

This function starts off using the same technique as the DifferenceOfDates function above, but it goes on to determine only the number of weekend days that fell between the two dates.

Script – Calculating the number of weekend days between dates

```
Function CalculateWeekendDays (date1 as Variant, _
date2 as Variant) as Long
  '** Calculate the number of weekend days
  '** between two dates
  On Error Goto processError
  Dim topDate as Variant, bottomDate as Variant
  Dim totalDays as Long, weekendDays as Long

  If (Cdat(date1) > Cdat(date2)) Then
      topDate = Cdat(date1)
      bottomDate = Cdat(date2)
  Else
      topDate = Cdat(date2)
      bottomDate = Cdat(date1)
  End If

  totalDays = Fix(Cdbl(topDate)) - Fix(Cdbl(bottomDate))

  '** divide the total days by 7 and multiply by 2.
  '** This will give the total weekend days that
  '** probably elapsed
  weekendDays = (totalDays \ 7) * 2

  '** We should add a weekend if the day of the week
  '** that topDate represents is less than the day
  '** of the week that bottomDate represents
  If (Weekday(topDate) < Weekday(bottomDate)) Then
      weekendDays = weekendDays + 2
  End If

  CalculateWeekendDays = weekendDays

  Exit Function

processError:
  Dim errMsg As String
  errMsg = "Error " & Cstr(Err) & ": " & Error$
  CalculateWeekendDays = 0
  Exit Function

End Function
```

First we calculate the total number of days between the dates. Then we divide this by seven to get the number of weeks between the dates – there will be one weekend for each full week that's elapsed between the dates, so multiplying the number of full weeks times 2 (since there's 2 weekend days in every weekend) will give us the number of weekend days between the days.

The only unusual thing we have to do otherwise is to add a weekend if the higher date is a lesser day of the week than the lower date. Think of the situation where the lower date is a Friday, and the higher date is the next Monday: there are 3 days between the dates, so 3 \ 7 will be zero, but there are 2 weekend days to account for.

Get a Specified Day of a Month

This function will get the specified day of a month (for example, the 2nd Monday in January, or the last Thursday in November). It's useful in determinig holidays, where a holiday is a regular day of the month.

Script – Getting the specified day of a month

```
Function GetSpecifiedDay (aMonth As Integer, aYear As Integer, aDay As String,
whichDay As Integer) As String
  '** Get the specified day of the month -- for example, the 2nd Monday
  '** of January
  If (aMonth < 1) Or (aMonth > 12) Then
      GetSpecifiedDay = ""
      Exit Function
  End If

  '** get a date to start with
  Dim tempDate As New NotesDateTime(aMonth & "/01/" & aYear)
  Dim firstDate As NotesDateTime
  Dim dayInt As Integer
  Dim count As Integer

  '** figure out which day we're looking for
  Select Case Ucase(aDay)
  Case "MONDAY"
      dayInt = 2
```

```
Case "TUESDAY"
    dayInt = 3
Case "WEDNESDAY"
    dayInt = 4
Case "THURSDAY"
    dayInt = 5
Case "FRIDAY"
    dayInt = 6
Case "SATURDAY"
    dayInt = 7
Case "SUNDAY"
    dayInt = 1
Case Else
    GetSpecifiedDay = ""
    Exit Function
End Select

'** get the first day of the month of the day type we're looking for
Dim tempWeekday As Integer
Dim dayAdjustment As Integer
tempWeekday = Weekday(tempDate.LSLocalTime)

If Not (tempWeekday = dayInt) Then
    Call tempDate.AdjustDay(dayInt - tempWeekday)
    If (Month(tempDate.LSLocalTime) <> aMonth) Then
        Call tempDate.AdjustDay(7)
    End If
End If

'** if we're only looking for the first day of this type for the month,
   '** then we found it
If (whichDay = 1) Or (whichDay = 0) Then
    GetSpecifiedDay = tempDate.DateOnly
    Exit Function
Else
    Set firstDate = New NotesDateTime(tempDate.DateOnly)
End If

'** otherwise, keep going up in 7 day intervals until we've either found
   '** our day or we're at the end of the month
   Dim findDay as Integer
If (whichDay < 0) Then
    '** handle the special case where we're looking for a negative
         '** number (for example, you want the 2nd to last Friday)
    findDay = 6
Else
    findDay = whichDay
End If

count = 1
Do Until (Month(tempDate.LSLocalTime) <> aMonth)
    Call tempDate.AdjustDay(7)
    count = count + 1
    If (Month(tempDate.LSLocalTime) <> aMonth) Then
        Call tempDate.AdjustDay(-7)
        Exit Do
    Elseif (count = findDay) Then
        Exit Do
    End If
Loop

'** okay, now that we're here we've either found our date or we're at the
   '** end of the month
```

```
  If (whichDay > 0) Then
      GetSpecifiedDay = tempDate.DateOnly
  Else
      '** count backwards for negative whichDays
      Call tempDate.AdjustDay( 7 * (whichDay + 1) )
      If (Month(tempDate.LSLocalTime) <> aMonth) Then
          '** we went back too far, so use the first day
          GetSpecifiedDay = firstDate.DateOnly
      Else
          GetSpecifiedDay = tempDate.DateOnly
      End If
  End If

  Exit Function

processError:
  Dim errMsg As String
  errMsg = "Error " & Cstr(Err) & ": " & Error$
  GetSpecifiedDay = ""
  Exit Function

End Function
```

So here's how you would use this function: if you wanted to get the 2nd Monday in January, 2001, you would call:

```
secondMonday$ = GetSpecifiedDay(1, 2001, "Monday", 2)
```

You can also use a negative number for the last parameter if you want to find the day relative to the end of the month. For example, if you wanted the second to last Thursday in November, you could call:

```
secondToLastThursday$ = GetSpecifiedDay(11, 2001, "Thursday", -2)
```

In this version of the script, we're allowing the user to pass the day of the week as a string, which adds a little more script to the function but makes the calling function a little more readable. The comments within the function itself should provide a good basic description of how it works.

Calculating Easter Sunday

While you can calculate most holidays with either a single line of script or by using the previous function (because they either fall on a particular day of the month or a particular weekday of the month), the calculation of Easter is a little more difficult. Technically, on the Gregorian calendar, Easter Sunday falls on the first Sunday after the Paschal full moon, but that's not a date that's intuitively determined or easily calculated. However, there are algorithms that will perform this calculation for you. Below is a LotusScript rendition of Butcher's method for calculating Easter (see http://www.smart.net/~mmontes/nature1876.html for more information about the history of this function).

Script – Calculating Easter Sunday

```
Function GetEaster (Byval thisYear As Integer) As String
  '** Butcher's method of calculating Easter - see
  '** http://www.smart.net/~mmontes/nature1876.html
  Dim easterMonth As Integer
  Dim easterDay As Integer

  '** adjust for 2-digit years
```

```
    If (thisYear < 100) Then
        If (thisYear < 50) Then
            thisYear = thisYear + 2000
        Else
            thisYear = thisYear + 1900
        End If
    End If

    a = thisYear Mod 19
    b = thisYear \ 100
    c = thisYear Mod 100
    d = b \ 4
    e = b Mod 4
    f = (b + 8) \ 25
    g = (b - f + 1) \ 3
    h = (19 * a + b - d - g + 15) Mod 30
    i = c \ 4
    k = c Mod 4
    l = (32 + 2 * e + 2 * i - h - k) Mod 7
    m = (a + 11 * h + 22 * l) \ 451
    easterMonth = (h + l - 7 * m + 114) \ 31
    p = (h + l - 7 * m + 114) Mod 31
    easterDay = p + 1

    GetEaster = "0" & Cstr(easterMonth) & "/" & Cstr(easterDay) & _
      "/" & thisYear

  End Function
```

Please see the web page referenced in the script comments for a discussion of the algorithm.

Calculating Holidays

Now that we have functions that calculate Easter and the date of particular weekdays in a month, we can write a function that calculates holidays.

Script – Calculating American holidays

```
  Function IsHoliday (thisDate As Variant) As Integer
    '** determine whether the given date is a holiday, based on normal
    '** American holidays
    On Error Goto processError

    Dim thisDateVar As Variant
    Dim thisYear As Integer
    thisDateVar = Datevalue(Cdat(thisDate))
    thisYear = Year(thisDateVar)
    IsHoliday = False

    '** New Year's Day is January 1
    If (thisDateVar = Cdat("01/01/" & thisYear)) Then
        IsHoliday = True
        Exit Function
    End If

    '** MLK Day is 3rd Monday in January
    If (thisDateVar = Cdat(GetSpecifiedDay(1, thisYear, "Monday", 3))) Then
        IsHoliday = True
        Exit Function
    End If

    '** Valentine's Day is February 14
    If (thisDateVar = Cdat("02/14/" & thisYear)) Then
```

```
        IsHoliday = True
        Exit Function
    End If

    '** President's Day is 3rd Monday in February
    If (thisDateVar = Cdat(GetSpecifiedDay(2, thisYear, "Monday", 3))) Then
        IsHoliday = True
        Exit Function
    End If

    '** St. Patrick's Day is March 17
    If (thisDateVar = Cdat("03/17/" & thisYear)) Then
        IsHoliday = True
        Exit Function
    End If

    '** Easter is a Sunday, which can be calculated by algorithm
    Dim easterDay As New NotesDateTime(GetEaster(thisYear))
    If (thisDateVar = Cdat(easterDay.DateOnly)) Then
        IsHoliday = True
        Exit Function
    End If

    '** Ash Wednesday is 46 days before Easter
    Call easterDay.AdjustDay(-46)
    If (thisDateVar = Cdat(easterDay.DateOnly)) Then
        IsHoliday = True
        Exit Function
    End If

    '** Mother's Day is the 2nd Sunday of May
    If (thisDateVar = Cdat(GetSpecifiedDay(5, thisYear, "Sunday", 2))) Then
        IsHoliday = True
        Exit Function
    End If

    '** Memorial Day is the last Monday in May
    If (thisDateVar = Cdat(GetSpecifiedDay(5, thisYear, "Monday", -1))) Then
        IsHoliday = True
        Exit Function
    End If

    '** Father's Day is the 3rd Sunday in June
    If (thisDateVar = Cdat(GetSpecifiedDay(6, thisYear, "Sunday", 3))) Then
        IsHoliday = True
        Exit Function
    End If

    '** Independence Day is July 4
    If (thisDateVar = Cdat("07/04/" & thisYear)) Then
        IsHoliday = True
        Exit Function
    End If

    '** Labor Day is the first Monday in September
    If (thisDateVar = Cdat(GetSpecifiedDay(9, thisYear, "Monday", 1))) Then
        IsHoliday = True
        Exit Function
    End If

    '** Columbus Day is 2nd Monday in October
    If (thisDateVar = Cdat(GetSpecifiedDay(10, thisYear, "Monday", 2))) Then
        IsHoliday = True
```

```
     Exit Function
 End If

 '** Halloween is October 31
 If (thisDateVar = Cdat("10/31/" & thisYear)) Then
     IsHoliday = True
     Exit Function
 End If

 '** Veteran's Day is November 11
 If (thisDateVar = Cdat("11/11/" & thisYear)) Then
     IsHoliday = True
     Exit Function
 End If

 '** Thanksgiving is 4th Thursday in November
 If (thisDateVar = Cdat(GetSpecifiedDay(11, thisYear, "Thursday", 4))) Then
     IsHoliday = True
     Exit Function
 End If

 '** Christmas is December 25th
 If (thisDateVar = Cdat("12/25/" & thisYear)) Then
     IsHoliday = True
     Exit Function
 End If

 Exit Function

processError:
 Dim errMsg As String
 errMsg = "Error " & Cstr(Err) & ": " & Error$
 IsHoliday = False
 Exit Function

End Function
```

This function simply determines whether or not a given date is a holiday or not. You could also modify the function to export all the dates as strings, display them to the user, etc.

Determining Whether or not a Day is a Valid Business Day

A simple extension of the previous function is one that determines whether or not a given day is a valid business day. For this example, we'll consider a business day to be any day that's not a Saturday, Sunday, or a calculated holiday.

Script – Determine whether or not a day is a valid business day

```
Function IsBusinessDay (thisDate As Variant) As Integer
     '** determine whether the given date is a valid business day
     '** (i.e. -- not on a weekend, and not a holiday)
     On Error Goto processError

     thisDateVar = Datevalue(Cdat(thisDate))
     IsBusinessDay = True

     If (Weekday(thisDateVar) = 1) Or (Weekday(thisDateVar) = 7) Then
          IsBusinessDay = False
     Elseif (IsHoliday(thisDateVar)) Then
          '** use the custom IsHoliday function to check for holidays
          IsBusinessDay = False
     End If
     Exit Function

processError:
     Dim errMsg As String
     errMsg = "Error " & Cstr(Err) & ": " & Error$
     IsBusinessDay = False
     Exit Function

End Function
```

The process here is pretty simple: we use the Weekday function so see if the day is a Saturday or a Sunday, and we use the custom IsHoliday function from earlier in the chapter to see if the day is on a holiday. If any of these things are true, then the given date is not a valid business date; otherwise, it is.

Converting a Date/Time to a Valid "Business Time"

Now that we have a function that will determine whether or not a day is a business day or not, it might be interesting to create a function that will check to see if a given date is on a business day, within specified business hours. If not, we can convert it to a valid business day during the specified hours.

Script – Converting a date/time to a business day, during specified business hours

```
Function ConvertToBusinessTime (thisDate As Variant, startTime As Variant, endTime As
Variant) As String
 '** This function converts a given date/time to "business time",
 '** based on the startTime and endTime values you give it.
 On Error Goto processError

 Dim tempTime As NotesDateTime
 Dim thisDateString As String
 Dim startHour As Integer, startMin As Integer, startSec As Integer
 Dim endHour As Integer, endMin As Integer, endSec As Integer

 thisDateString = Cstr(Cdat(thisDate))
 startHour = Hour(Cdat(startTime))
 startMin = Minute(Cdat(startTime))
 startSec = Second(Cdat(startTime))
 endHour = Hour(Cdat(endTime))
 endMin = Minute(Cdat(endTime))
 endSec = Second(Cdat(endTime))

 '** If the value doesn't have a time field, set it to startTime
 If (Instr(thisDateString, ":") > 0) Then
     Set tempTime = New NotesDateTime(thisDateString)
 Else
     Set tempTime = New NotesDateTime(thisDateString & " " &
Cstr(Timevalue(startTime)))
 End If

 '** Convert the initial time to a "business" time, where:
 '**   -  anything before start-of-business time is changed to
 '**      start-of-business time
 '**   -  anything after end-of-business time is changed to
 '**      start-of-business time the next day
 '**   -  anything on a weekend is moved to the next Monday at
 '**      start-of-business time
 Dim tTime As Variant, sTime As Variant, eTime As Variant
 tTime = Fraction(Cdat(tempTime.LSLocalTime))
```

```
  sTime = Fraction(Cdat(startTime))
  eTime = Fraction(Cdat(endTime))

  '** we're comparing both the time values and the string versions
  '** of the time values because there's a bug in some versions of
  '** Notes where the decimal representation of a time value
  '** with a date is slightly different than the decimal representation
  '** of the same time value without a date (for example,
  '** Fraction(Cdbl(Cdat("01/01/2001 5:00 PM"))) = .708333333335759 but
  '** Fraction(Cdbl(Cdat("5:00 PM"))) = .708333333333333)
  If ( tTime < sTime ) And Not (Cstr(tTime) = Cstr(sTime)) Then
      '** If the time is less than startTime, change to startTime
      Call tempTime.AdjustSecond(startSec - Second(tempTime.LSLocalTime), True)
      Call tempTime.AdjustMinute(startMin  - Minute(tempTime.LSLocalTime), True)
      Call tempTime.AdjustHour(startHour - Hour(tempTime.LSLocalTime), True)
  Elseif ( tTime > eTime ) And Not (Cstr(tTime) = Cstr(eTime)) Then
      '** If it's after endhour, set the time to startTime the next morning
      Call tempTime.AdjustSecond(startSec - Second(tempTime.LSLocalTime), True)
      Call tempTime.AdjustMinute(startMin  - Minute(tempTime.LSLocalTime), True)
      Call tempTime.AdjustHour(startHour - Hour(tempTime.LSLocalTime), True)
      Call tempTime.AdjustDay(1, False)
  End If

  '** And finally, adjust for weekends and holidays
  Do While (IsBusinessDay (tempTime.LSLocalTime) = False)
      Call tempTime.AdjustSecond(startSec - Second(tempTime.LSLocalTime), True)
      Call tempTime.AdjustMinute(startMin  - Minute(tempTime.LSLocalTime), True)
      Call tempTime.AdjustHour(startHour - Hour(tempTime.LSLocalTime), True)
      Call tempTime.AdjustDay(1, False)
  Loop

  ConvertToBusinessTime = tempTime.LSLocalTime
  Exit Function


processError:
  Dim errmess As String
  errmess = Error$
  ConvertToBusinessTime = ""
  Exit Function

End Function
```

The inline comments within the script should give you a good idea of how the script works. The only really unusual thing we have to do is double-check our time comparisons, because some versions of Notes internally convert time values with dates slightly differently than time values without dates. For example, the script:

```
Fraction(Cdbl(Cdat("01/01/2001 5:00 PM")))
```

gives us a value of .708333333335759, but the script:

```
Fraction(Cdbl(Cdat("5:00 PM")))
```

gives us a value of .708333333333333. To work around this, we compared both the numeric representations of the times and the string representations.

Dealing with Date Ranges

So far, I've been avoiding a discussion of date/time ranges, which are not natively addressed in LotusScript. There *is* a NotesDateTimeRange class, but all that really does is allow you to take a date range from a Notes document and get the start and end dates. If you really have a need to deal with date ranges, you'll need much more functionality than that.

Below I've included a class that you can use to manipulate not only date/time ranges, but also lists of dates and times as well. Because it's so long, I haven't included too many comments in the script, but each of the routines within the class are fairly short so you should be able to figure out what's going on if you read through it piece by piece.

Script – DateRange class

```
Class DTRange
 '** member variables
 Private hTime As Variant
 Private lTime As Variant
 Private timeList List As String
 Private delim As String

 '** constructor
 Sub New (timeRangeString As String)
     Call InitializeClass()
     Call AddTimeStringToList(timeRangeString)
 End Sub

 '** destructor
 Sub Delete
     Call EraseList()
 End Sub


 '*****************************************************
 '** public properties
 Public Property Get HighTime As String
     '** get the highest date/time value in the list
     If Isnull(hTime) Then
         HighTime = ""
     Else
         HighTime = Cstr(hTime)
     End If
 End Property


 Public Property Get LowTime As String
     '** get the lowest date/time value in the list
     If Isnull(lTime) Then
         LowTime = ""
     Else
         LowTime = Cstr(lTime)
     End If
 End Property


 '*****************************************************
 '** public methods
 Public Sub Add (timeRangeString As String)
     '** add a new time or set of times to the list
     Call AddTimeStringToList(timeRangeString)
 End Sub
```

```
    Public Function AddFromDoc (doc As NotesDocument, fieldName As String) As Integer
        '** get the date or a date range from a field on a NotesDocument
        On Error Goto processError

        Dim item As NotesItem
        Set item = doc.GetFirstItem("DateField")
        Call AddTimeStringToList(item.text)
        Exit Function

   processError:
        Dim errMsg As String
        errMsg = Error$
        Exit Function

    End Function


    Public Function IsInRange (timeRangeString As String) As Integer
        '** check to see if a given time is within the times in timeList
        IsInRange = CheckRangeForTime(timeRangeString)
    End Function


    Public Function ExportListAsString () As String
        '** give the timeList to the user as a string
        Forall element In timeList
            If (ExportListAsString = "") Then
                ExportListAsString = element
            Else
                ExportListAsString = ExportListAsString & delim & element
            End If
        End Forall
    End Function


    Public Function ExportListAsArray () As Variant
        '** give the timeList to the user as an array
        Redim tempArray(0) As String
        Dim count As Integer

        Forall element In timeList
            Redim Preserve tempArray(count) As String
            tempArray(count) = element
            count = count + 1
        End Forall

        ExportListAsArray = tempArray
    End Function


    Public Function RemoveItem (timeRangeString As String) As Integer
        '** remove a specified item from the timeList
        Dim tempString As String, fooVar1 As Variant, fooVar2 As Variant
        tempString = ConvertRange(timeRangeString, fooVar1, fooVar2)

        If Iselement(timeList(tempString)) Then
            Erase timeList(tempString)
            Call SetHighAndLowTimes()
            RemoveItem = True
        Else
            RemoveItem = False
        End If
    End Function


    Public Sub EraseList ()
        '** erase the current timeList
        Erase timeList
        Call InitializeClass()
    End Sub


    '*****************************************************
    '** private internal functions and subs
    Private Sub InitializeClass ()
        delim = ";"
        hTime = Null
        lTime = Null
    End Sub


    Private Function AddTimeStringToList (timeRangeString As String)
        '** timeRangeString should be a semi-colon delimited
        '** list of times or time ranges
        Dim tempString As String, tempElement As String
        Dim pos As Integer, lastPos As Integer
        Dim fooVar1 As Variant, fooVar2 As Variant
        tempString = Trim$(timeRangeString)

        If (tempString = "") Then
            Exit Function
        End If

        '** get all the comma-delimited elements
        lastPos = 1
        pos = Instr(lastPos, tempString, delim)
        Do Until (pos < 1)
            tempElement = Trim$(Mid$(tempString, lastPos, pos - lastPos))
            If IsValidRange(tempElement) Then
                timeList(tempElement) = ConvertRange(tempElement, fooVar1, fooVar2)
            End If
            lastPos = pos + 1
            pos = Instr(lastPos, tempString, delim)
        Loop

        '** get the last element
        tempElement = Trim$(Right$(tempString, Len(tempString) - lastPos))
        If IsValidRange(tempElement) Then
            timeList(tempElement) = ConvertRange(tempElement, fooVar1, fooVar2)
        End If

        '** reset the internal high and low times
        Call SetHighAndLowTimes()
    End Function


    Private Function IsValidRange (timeRangeString As String) As Integer
        '** check to see if a timeRangeString is a valid format
        Dim leftRange As Variant, rightRange As Variant

        Call GetLeftAndRightRanges(timeRangeString, leftRange, rightRange)
        If Isnull(leftRange) And Isnull(rightRange) Then
            IsValidRange = False
        Else
```

```
            IsValidRange = True
        End If

    End Function


    Private Function GetLeftAndRightRanges (timeRangeString As String, _
    retLeftRange As Variant, retRightRange As Variant) As Integer
        '** get the left and the right parts of a time range
        On Error Goto processError
        Dim pos As Integer

        pos = Instr(timeRangeString, "-")
        If (pos > 0) Then
            retLeftRange = Cdat(Trim$(Left$(timeRangeString, pos - 1)))
            retRightRange = Cdat(Trim$(Right$(timeRangeString, Len(timeRangeString) -
pos)))
            GetLeftAndRightRanges = True
        Else
            retLeftRange = Cdat(Trim$(timeRangeString))
            retRightRange = retLeftRange
            GetLeftAndRightRanges = False
        End If

        Exit Function

processError:
        Dim errMsg As String
        errMsg = Error$
        GetLeftAndRightRanges = False
        Exit Function

    End Function


    Private Function ConvertRange (timeRangeString As String, _
    retLeftRange As Variant, retRightRange As Variant) As String
        '** convert a date or date range to a formatted string for storage
        On Error Goto processError
        Dim leftRange As Variant, rightRange As Variant
        retLeftRange = Null
        retRightRange = Null

        Call GetLeftAndRightRanges(timeRangeString, leftRange, rightRange)
        If Isnull(leftRange) And Isnull(rightRange) Then
            ConvertRange = ""
        Elseif (leftRange = rightRange) Then
            ConvertRange = Cstr(leftRange)
            retLeftRange = leftRange
            retRightRange = rightRange
        Elseif (leftRange > rightRange) Then
            ConvertRange = Cstr(rightRange) & " - " & Cstr(leftRange)
            retLeftRange = rightRange
            retRightRange = leftRange
        Else
            ConvertRange = Cstr(leftRange) & " - " & Cstr(rightRange)
            retLeftRange = leftRange
            retRightRange = rightRange
        End If

        Exit Function

processError:
        Dim errMsg As String
        errMsg = Error$
        ConvertRange = ""
        Exit Function

    End Function


    Private Sub SetHighAndLowTimes()
        '** find the high and low date/times in the timeList
        On Error Goto processError
        Dim leftRange As Variant, rightRange As Variant

        Forall element In timeList
            Call GetLeftAndRightRanges(element, leftRange, rightRange)
            If Isnull(hTime) Then
                hTime = rightRange
            Elseif (leftRange > hTime) Then
                hTime = rightRange
            End If

            If Isnull(lTime) Then
                lTime = leftRange
            Elseif (leftRange < lTime) Then
                lTime = leftRange
            End If
        End Forall

        Exit Sub

   processError:
        Dim errMsg As String
        errMsg = Error$
        Exit Sub

    End Sub


    Private Function CheckRangeForTime (timeRangeString As String) As Integer
        '** check to see if a given date/time is in timeList
        On Error Goto processError
        Dim leftRange As Variant, rightRange As Variant
        Dim leftRange2 As Variant, rightRange2 As Variant
        Dim areTwoValues As Integer

        timeRangeString = ConvertRange(timeRangeString, leftRange, rightRange)

        '** do an initial check to see if we're outside the low
        '** and high times in our list. If so, we can exit early
        If (leftRange > hTime) Or (rightRange < lTime) Then
            CheckRangeForTime = False
            Exit Function
        End If

        '** if we got here, we need to check against the whole list
        Forall element In timeList
            areTwoValues = GetLeftAndRightRanges(element, leftRange2, rightRange2)
            If areTwoValues Then
                If Not((leftRange > rightRange2) Or (rightRange < leftRange2)) Then
                    CheckRangeForTime = True
                    Exit Function
                End If
            Else
                If (leftRange = leftRange2) Or (rightRange = rightRange2) Then
                    CheckRangeForTime = True
                    Exit Function
                End If
            End If
        End Forall

        Exit Function


    processError:
        Dim errMsg As String
        errMsg = Error$
        CheckRangeForTime = False
        Exit Function

    End Function

    End Class
```

If you want to try out some of the class functionality, add this class to an agent and put this script in the Initialize sub.

```
Sub Initialize
 Dim testRange As New DTRange("")
 testRange.Add("1/1/02; 1/10/02 - 1/20/02")

 Print "High = " & testRange.HighTime & "; Low = " & testRange.LowTime

 If testRange.IsInRange("01/10/2002") Then
     Print "In Range"
 Else
     Print "Out of Range"
 End If

 Dim testArray As Variant
 testArray = testRange.ExportListAsArray

 Print testRange.ExportListAsString
 testRange.RemoveItem("01/1/02")
 testRange.Add("01/25/2002")
 Print testRange.ExportListAsString

End Sub
```

As always, you can step through the script using the LotusScript debugger to really see what' s going on.

The advantage to using a class like this is that you can have a whole list of dates or times that you want to compare against, and you can then compare against all of them at once, even if some of the dates or times are actually ranges of dates and times. AddFromDoc method even allows you to pull a range of dates in from a multi-value date field on a form, which can prove very useful.

Working with Files

From within LotusScript, you have the ability to access and manipulate files, which allows you to read and write data to external sources rather easily. There are several different ways that LotusScript can access files natively:

- Random – which is a file with structured data, normally only readable through other LotusScript programs
- Binary – which is any type of file, although if you open text files as Binary then you' ll have to make sure to check for non-text characters (like linefeeds)
- Input – which is a read-only text file, starting from the beginning of the file
- Output – which is a write-only text file, although if the file already exists, it is overwritten
- Append – which is a write-only text file, starting from the end of the file (if the file doesn' t exist, an error occurs)

Random and Binary files can be opened as either read-only, write-only, or read-write; the other methods will open files as specified above. If no method for opening a file is specified, then the file will be opened in Random mode. You can also specify the type of file sharing used when the file is opened: either Shared (the default), Lock Read, Lock Write, and Lock Read Write. The general syntax for opening a file for use is:

```
Open fileName
     [ For { Random | Input | Output | Append | Binary } ]
     [ Access { Read | Read Write | Write } ]
     [ { Shared | Lock Read | Lock Read Write | Lock Write } ]
     As [#]fileNumber
     [ Len = recLen ]
```

The pound sign (#) before fileNumber is entirely optional, and the function works the same with or without it. For example, in its simplest form, you can open a file with the statement:

```
Open C:\Myfile.foo As 1
```

This will accept all the defaults for the Open statement, which means that the file will be opened in Random mode, with Read-Write access and Shared file locking and a record length of 128. More typically, you might want to open a file like this:

```
Open C:\Myfile.txt For Input As 1
```

or:

```
Open C:\Myfile.bin For Binary Read As 1
```

A useful function to use in conjunction with the Open statement is the Freefile function, which will automatically get a free file handle for use in opening your file. You can normally access a file without any problems by specifying any number from 1 to 255 as a file number when opening your file, but if you want to be safe, you can write some code like this:

```
Dim fileNum as Integer
fileNum = Freefile
Open C:\Myfile.txt For Output Lock Write As fileNum
```

This will ensure that you have an unused file number when you' re opening the file. It' s also good coding practice, since many of the other file manipulation functions use the file number to access an open file, and using a variable to refer to the file number makes the code much easier to read and maintain.

Before we get too much further, let' s summarize the LotusScript functions and statements that are used to access, manipulate, and get information about files.

Function/Statement	Usage	Example
ChDir	Sets the directory to use for file access (for when a file name is specified but no directory is given).	ChDir "C:\TEMP"
ChDrive	Sets the drive letter to use for file access (for when a file path is specified but no drive is given).	ChDrive "D"
Close	Closes specified open files (writes buffered data first).	Close #1, #2
CurDir[$]	Indicates the current directory.	Print CurDir$
CurDrive[$]	Indicates the current drive letter.	Print CurDrive$
Dir[$]	Returns a file or directory name, based on a name or wildcard specification you give it. Can also be used multiple times to step through a directory.	firstFile$ = Dir$("C:\*.*")
EOF	Boolean value that indicates whether the end of a file has been reached.	If EOF(fileNum%) Then Exit Do
FileAttr	Returns either a number indicating how an open file has been opened (Binary, Random, etc.), or the operating system' s file handle number for an open file.	FileAttr(fileNum%, 1)
FileCopy	Copies a file.	FileCopy "C:\MyFile.txt" "C:\MyNewFile.txt"

FileDateTime	Returns a string with the date and time a specified file was last modified (the file does not have to be currently open)	FileDateTime("C:\MyFile.txt")
FileLen	Returns the length of a file, in bytes (the file does not have to be currently open)	FileLen("C:\MyFile.txt")
FreeFile	Returns the next available unused file number, from 1 to 255.	fileNum% = FreeFile()
Get	Gets data from a binary or random file and puts it into a variable. For a Binary file, you will generally want to use a fixed-length string as your variable. A pound sign (#) before the file number is required.	Get #fileNum%, 1, fileContents
GetAttr	Same as GetFileAttr	
GetFileAttr	Returns the file-system attributes of a file or directory (Read-Only, Hidden, etc.).	GetFileAttr("C:\MyFile.txt")
Input	There are actually two different Input commands. The Input statement reads comma-delimited data from a sequential file into one or more variables, and is the opposite of the Write statement. The Input function reads a fixed amount of data from a sequential or binary file into a string variable. If you' re using the function to get dat from a Binary file, keep in mind that this function retrieves a certain number of characters, not bytes. For the Input statement, a pound sign (#) before the file number is required.	Statement: Input #fileNum%, field1$, field2% Function: textString$ = Input$(500, #fileNum%)
InputB[$]	Reads a specified number of bytes from a sequential or binary file into a string variable.	textString$ = InputB$(500, #fileNum%)

InputBP[$]	Reads a specified number of bytes from a sequential or binary file into a string variable, using the platform-native character set.	textString$ = InputBP$(500, #fileNum%)
Kill	Deletes a file.	Kill "C:\MyFile.txt"
Line Input	Reads a line from a sequential file, up to but not including any carriage returns or linefeeds (a pound sign (#) before the file number is required). This is the opposite of the Print statement.	Line Input #fileNum%, lineOfText$
Loc	Returns your current position in a file (in bytes).	filePos& = LOC(fileNum%)
Lock	Locks either a record (for a Random file), a byte (for a Binary file) or the whole file (for a Sequential file), so that other processes can' t modify the data. Be very sure to use the Unlock statement after you' re done.	Lock #fileNum%, 1 To 20
LOF	Returns the length of a file, in bytes. The file must have already been opened using the Open statement.	fileSize& = LOF(fileNum%)
MkDir	Creates a directory.	MkDir "C:\MyNewDir"
Name	Renames a file or directory.	Name "C:\MyFile.txt" "C:\MyNewFile.txt"
Open	Opens a file, so that you can read or write data.	Open fileName$ For Output As fileNum%
Print #	Writes text to a Sequential file, terminated with a newline character (a pound sign (#) before the file number is required).	Print #fileNum%, "This is a new line of text."
Put	Writes data to a Binary or a Random file (a pound sign (#) before the file number is required).	Put #fileNum%, stuff$

Reset	Closes all open files (writes buffered data first).	Reset
RmDir	Deletes a directory from the file system (the directory must be empty).	RmDir "C:\BadDir"
Seek	Sets or retrieves the byte position (for a Binary file) or the record number (for a Random file).	Seek(fileNum%) or Seek #fileNum%, 1
SetFileAttr	Sets file attributes (Read-Only, Hidden, etc.). Should not be used on a file that's already Open.	SetFileAttr fileName$, ATTR_READONLY + ATTR_HIDDEN
Spc	Used in conjunction with the Print statement to insert a specified number of spaces to a Sequential file.	Print #fileNum%, "some text"; Spc(1); "some more text"
Tab	Used to specify the starting character position of the text written by a Print or Print # statement.	Print #fileNum%, "one"; Tab(10); "two"
Unlock	Unlocks data that has been previously locked by the Lock statement. The parameters used must exactly match the ones used in the corresponding Lock statement.	Unlock #fileNum%, 1 To 20
Width	Sets the line length to be used by the Print statement, when writing to a Sequential file. By default, the line length is unlimited (a pound sign (#) before the file number is required).	Width #fileNum%, 50
Write	Similar to the Print statement, but comma-delimits multiple pieces of data.	Write #fileNum%, field1$, field2%

While the functions and statements above are used to access and manipulate files, there are also some related functions that access the operating system environment. Because we'll be using some of these functions in the scripts later in the chapter, they are also summarized here for reference.

Function/Statement	Usage	Example
ActivateApp	Makes a window the active window, based on the window title.	ActivateApp "Microsoft Word"
AppActivate	Same as ActivateApp	
Beep	Makes the computer beep.	Beep
DoEvents	Same as Yield	
Environ[$]	Returns the contents of an environment variable from the operating system (note that this is an operating system environment variable, not a Notes environment variable).	thePath$ = Environ$("PATH")
Shell	Starts an external program (EXE, BAT, COM, or PIF file). The LotusScript routine will continue operation immediately after the shelled program begins.	Shell("Notepad.exe", 1)
Yield	Frees operating system resources during script execution.	For i% = 1 To 1000 Yield Next

Custom Routines

The functions and subs in this chapter are a combination of routines that interact directly with files and routines that act as "helper" functions.

Get the Default File Separator for the Operating System You're On

If your script is running on only one operating system type, then you won't need this function, which will determine the file separator for the operating system you're on. However, if you're running on multiple OS's, this may come in handy

Script – Get default file separator

```
Function GetFileSeparator () As String
    Dim session As New NotesSession

    Select Case session.Platform
    Case"Macintosh"
           GetFileSeparator = ":"
    Case "UNIX"
           GetFileSeparator = "/"
    Case Else
           GetFileSeparator = "\"
    End Select

End Function
```

We'll be calling this function in several of the scripts later in the chapter, so please be sure to make the appropriate modifications to the scripts as necessary.

Get File Path and File Name from a Path + File Name String

When dealing with files in LotusScript, you'll often run into the situation where you have a file name that includes the full path of the file, and you'll need to parse out the path and the file name from that string. The following functions will do this for you.

Script – Get file path

```
Function GetFilePath (fullFileName As String) As String
  '** given the full file path of a file (like "c:\temp\blah.txt"),
  '** this function will return the path portion of the file path,
  '** including the trailing "\" (like "c:\temp\")
  Dim pos As Integer, lastPos As Integer
  Dim slash as String

  slash = GetFileSeparator()

  pos = InStr(fullFileName, slash)
  lastPos = pos

  Do While (pos > 0)
      lastPos = pos
      pos = InStr(pos + 1, fullFileName, slash)
  Loop

  GetFilePath = Left$(fullFileName, lastPos - 1)

End Function
```

Script – Get file name

```
Function GetFileName (fullFileName As String) As String
  '** given the full file path of a file (like "c:\temp\blah.txt"),
  '** this function will return the file name (like "blah.txt")
  Dim pos As Integer, lastPos As Integer
  Dim tempFileName As String
  Dim slash as String

  slash = GetFileSeparator()

  pos = InStr(fullFileName, slash)
  lastPos = pos

  Do While (pos > 0)
```

```
     lastPos = pos
     pos = InStr(pos + 1, fullFileName, slash)
 Loop

 tempFileName = Right$(fullFileName, Len(fullFileName) - lastPos)

 GetFileName = tempFileName

End Function
```

Both functions do about the same thing, which is to find the position of the last "\" character in the string, and return either everything to the left or everything to the right of it. A similar function that you might need is one that gets the file extension of a file name:

Script – Get file extension

```
Function GetFileExtension (fullFileName As String) As String
 '** given the full file path of a file (like "c:\temp\blah.txt"),
 '** this function will return the file name extension (like "txt")
 Dim pos As Integer, lastPos As Integer
 Dim tempFileName As String
 Dim slash as String

 slash = GetFileSeparator()

 '** first, get the file name itself
 pos = InStr(fullFileName, slash)
 lastPos = pos

 Do While (pos > 0)
     lastPos = pos
     pos = InStr(pos + 1, fullFileName, slash)
 Loop

 tempFileName = Right$(fullFileName, Len(fullFileName) - lastPos)

 '** then, get everything to the right of the last period
 pos = InStr(tempFileName, ".")
 lastPos = pos

 Do While (pos > 0)
     lastPos = pos
     pos = InStr(pos + 1, tempFileName, ".")
 Loop

 If (lastPos < 1) Then
     lastPos = Len(tempFileName)
 End If

 GetFileExtension = Right$(tempFileName, Len(tempFileName) - lastPos)

End Function
```

At first this function may seem like a bit of overkill for what it does, but you need to consider a few things. First, directory names can have dots in them, so you can't just find the first dot and then get everything to the right of it. Second, a file name can have no dots at all, so you can't just find the last dot and get everything to the right. Of course, if you wanted to call the GetFileName function at the beginning of this function, you can skip about half the script.

Also, if you know that you're only going to be using these functions with R5 or higher, you can use StrLeftback and StrRightback to simplify these functions further.

Create a Directory, Including All Subdirectories

Even though the LotusScript MkDir function will create new directories for you, it will fail if any part of the directory tree underneath the directory doesn't exist. The following function will take care if that for you.

Script – Create a directory with subdirectories

```
Function MakeDir (directory As String) As Integer
 '** This function will check for the existence of a directory,
 '** and try to create it if it doesn't exist

 On Error Goto processError

 Dim tempString As String
 Dim pos As Integer, startPos As Integer
 Dim slash as String

 slash = GetFileSeparator()

 If (directory = "") Then
     MakeDir = False
     Exit Function
 End If


 '** Create the directory (at all levels) if it doesn't exist
 '** figure out where the directory path starts
 If (Instr(directory, ":") > 0) Then
     '** it's a drive path of some sort
     startPos = Instr(directory, ":") + 2
 Elseif (Left$(directory, 2) = "\\") Then
     '** it's a UNC path
     startPos = 3
 Elseif (Left$(directory, 1) = slash) Then
     '** it's a relative path
     startPos = 2
 Else
     startPos = 1
 End If

 pos = Instr(startPos, directory, slash)
 Do While (pos > 0)
     tempString = Left$(directory, pos - 1)
     If (Dir$(tempString, 16) = "") Then
         Mkdir tempString
     End If
     pos = Instr(pos + 1, directory, slash)
 Loop

 '** now try to create the whole directory
 If (Dir$(directory, 16) = "") Then
     Mkdir directory
 End If

 MakeDir = True
 Exit Function
```

```
processError:
  Dim errMsg As String
  errMsg = "Error " & Cstr(Err) & ": " & Error$
  MakeDir = False
  Exit Function

End Function
```

The logic is pretty straightforward: find the lowest subdirectory and start building the directory tree from the bottom up.

Now that we have a few helper functions under our belt, let' s start looking at some functions that actually work with files.

Make Sequential Output File

At first glance, it may seem silly to write an entire function just to create a Sequential file for output – after all, you can do that with a single statement, can' t you? The advantage to wrapping this functionality into a function is that you can do some good error checking, and you can do a few things to avoid errors in the process.

Script – Make a sequential output file

```
Function MakeOutputFile (fileName As String, overwrite As Integer) As Integer
  '** This function will create a file based on the filename & path that
     '** you pass. If overwrite is True, it will overwrite any existing file
     '** that is there. If it's False and the file already exists, the
     '** function will append to the file.
  '** All errors will return a number less than 1. Otherwise, the function
     '** returns the file number handle for file access.

  On Error Goto processError

  Dim fileNum As Integer
  Dim tempFileName As String

  '** If no filename is passed, just return
  If (Trim(fileName) = "") Then
      MakeOutputFile = 0
      Exit Function
  End If

  '** Make sure that the directory exists, using the
  '** user-defined MakeDir and GetFilePath functions
  If Not MakeDir(GetFilePath(fileName)) Then
      MakeOutputFile = -1
      Exit Function
  End If

  '** Try to create a file, based on the overwrite setting
  fileNum = Freefile()
  If (overwrite) Then
      Open fileName For Output As fileNum
  Else
      If (Dir$(fileName) = "") Then
          Open fileName For Output As fileNum
      Else
          Open fileName For Append As fileNum
      End If
  End If
  MakeOutputFile = fileNum
  Exit Function


processError:
  Dim errMess As String
  errMess = Error
  MakeOutputFile = 0
  Exit Function

End Function
```

You can see the extra logic applied in this function, that does a little more than just open a file for output. First, it creates the directory path, if it doesn't already exist (using the MakeDir and GetFilePath functions from earlier in the chapter). Then it allows you to decide what to do if the file already exists, either overwrite it or append to it. It's a little extra overhead if you just want to open a file to write to, but it can save you lots of debugging time later.

Getting a Unique File Name

Another thing you might need to do is to make sure that the name of the file you are going to create is unique. The following function shows one way to do that.

Script – Create a unique file name

```
Function GetUniqueFilename (fileName As String) As String
  '** This function will generate a unique fileName in a directory,
     '** based on the original fileName you provide. It scans the given
     '** directory for your fileName, and adds sequential numbers to
     '** the end of the fileName until it is unique.

  On Error Goto processError

  Dim count As Integer
  Dim tempFileName As String
  Dim fileNameLeft As String
  Dim fileNameRight As String
  Dim newFileName As String
  Dim dirName As String
  Dim tempString As String
  Dim slash as String

  slash = GetFileSeparator()

  count = 1

  '** If no fileName is passed, just return
  If (Trim(fileName) = "") Then
      GetUniquefileName = ""
      Exit Function
  End If

  '** Make sure that the directory exists, using the
  '** user-defined functions GetFilePath and MakeDir
  dirName = GetFilePath(fileName)
  If Not MakeDir(dirName) Then
      GetUniqueFileName = ""
      Exit Function
  End If

  '** Use the user-defined functions GetFileName and
```

```
      '** GetFileExtension to get the fileName and extension
    tempString = GetFileName(fileName)
    fileNameRight = GetFileExtension(tempString)
    fileNameLeft = Left$(tempString, Len(tempString) - Len(fileNameRight) - 1)

    If (Dir$(fileName) = "") Then
        '** If the fileName wasn't found in the directory,
             '** return the name we were given
        GetUniqueFileName = fileName
        Exit Function
    Else
        '** If the fileName was found, keep adding a number to
             '** the end of the left part of the fileName until it's
             '** unique. For example, autoexec.bat becomes autoexec(1).bat
        Do While True
            '** Generate a new fileName
            newfileName = dirName & slash & fileNameLeft & "(" & Trim(Str(count)) & ")."
  & fileNameRight

            '** Check to see if the new name is a duplicate
            If (Dir$(newFileName) = "") Then
                Exit Do
            Else
                count = count + 1
            End If
        Loop

        GetUniqueFileName = newFileName
        Exit Function
    End If


  processError:
    Dim errmess As String
    errmess = Error
    GetUniqueFileName = ""
    Exit Function


  End Function
```

The process is: we make sure that the directory exists, then we see if the file name exists, and if it does, we keep adding a number to the name until it' s unique. There are plenty of variations on this theme, too. You can adjust the function so that the file name is always in 8.3 format, or whatever format you want.

Writing to a Text File

Before we get too much farther, we should also show you the basic technique for writing to a text file. This is something you' ll end up using for logging, creating HTML files, etc.

Script – Writing to a text file

```
    '** open a file for writing
    fileNum% = Freefile()
    fileName$ = "C:\SampleFile.txt"
    Open fileName$ For Output As fileNum%

    '** write some stuff
    Print #fileNum%, "Sample Text File Contents"
    Write #fileNum%, "One", "Two", "Three"
```

```
  '** close the file
  Close fileNum%
```

It's a fairly simple process: open a file for Output, use either the Print or the Write statement to write text to it, and close it when you're done. It's very important to close the file when you're done; if you don't, then the file can end up locked so that other processes (including Notes) can't open it until you either shut down Notes or issue a Reset command in another script.

Writing and Running a Batch File

This example will show you the basic technique for writing and running a batch file, which is essentially the same procedure you'll need to use for writing any text file.

Script – Write and run a batch file to delete TMP files

```
Function DeleteTempFiles () As Integer
  '** delete the *.TMP files in the Temp directory, using a batch file
  On Error Goto processError

  Dim tempDir As String
  Dim batFileName As String
  Dim batFileNum As Integer
  Dim taskId As Integer

  '** try to get the path to the Temp directory from the system
  '** environment variables
  tempDir = Environ$("TEMP")

  '** if we can't find the Temp directory, just exit
  If (tempDir = "") Then
      DeleteTempFiles = False
      Exit Function
  End If

  '** We can't send an internal DOS command (like DEL) directly using
  '** the Shell command, so we'll write a little batch file to take care
    '** of it for us. Kind of a pain, but it's still better than using the
    '** Kill command, which doesn't accept wildcards.
  batFileNum = Freefile()
  batFileName = TempDir & "\" & "DelTempFiles.bat"
  Open batFileName For Output As batFileNum
  Print #batFileNum, "REM - TMP File Delete Batch File, created " & Format(Now(), "m-d-
yy h:mm:ss")
  Print #batFileNum, ""
  Print #batFileNum, "DEL " & TempDir & "\*.TMP /Q"
  Close batFileNum

  '** Run the batch file
  taskId = Shell(batFileName, 1)

  DeleteTempFiles = True
  Exit Function


processError:
  Dim errMsg As String
  errMsg = Error$
  Reset
  DeleteTempFiles = False
  Exit Function
```

```
End Function
```

This function gets the path of the Temp directory from the operating system environment variable "TEMP", and it then writes a batch file to delete all the *.TMP files in that directory. In this case, a batch file is a good way of performing this type of operation, because the Shell command doesn't allow you to run DOS commands, and the LotusScript Kill function doesn't accept wildcards.

Notice how the error-handling routine includes a call to the Reset function. This is good practice for a function that creates and closes a file, because if there's an error before the file has been closed, then the file may be inaccessible because it is still being "held" by LotusScript.

Reading from a Text File

The technique for reading from a text file is very similar to writing to a file:

Script – Reading from a text file

```
'** open the file for reading
fileNum% = Freefile()
fileName$ = "C:\SampleFile.txt"
Open fileName$ For Input As fileNum%

'** read some things from the file
string1$ = Input$(30, #fileNum)
Line Input #fileNum, string2$

'** close the file
Close fileNum%
```

It's the same kind of process: open a file for Input, use one of the techniques for reading from the file, and close it.

If you're dealing with fairly small files (less than 10K), you can use the Line Input statement to read one line at a time without too much of a performance hit. Otherwise, you'll generally want to read large "chunks" of the file using the Input$ function and parse out the lines of the file on your own, if necessary. This is because every time you use Line Input, you have to keep going back to the file and read a small piece of it to a string, which takes processing time accessing the file repeatedly as well as reinitializing the string every time you read something new.

Searching for Text in a File

Here's a good example of how to use the technique of reading large pieces of a text file instead of using Line Input to access text in the file. It's a function that searches a file for a string and returns an array containing all the positions in the file where that text was found.

Script – Searching for text in a file

```
Function SearchFile (fileName As String, Byval searchTerm As String, _
maxMatches As Integer) As Variant
  '** This function searches fileName for all occurences of searchTerm,
     '** and returns the results as a list. For every match it finds, it
     '** returns the position in the file where the search term was found,
     '** along with the position of the last linefeed character before the
     '** position, so you can easily return the entire line that the search
     '** term was found in (if desired). The format is:
     '**    tempArray(i) = LinefeedPos;SearchPos
```

```
'**
'** If maxMatches is a number more than 0, only the first x matches
   '** will be returned. This is useful if you're just determining whether
   '** or not a file contains your searchtext, or if you're afraid that a
   '** lot of matches might be found, and you want to keep processing
'** time to a minimum.

On Error Goto processError

Dim fileNum As Integer
Dim count As Integer
Dim fileLength As Long
Dim currentPos As Long
Dim lastPos As Long
Dim text As String
Dim buffer As String
Dim bufferLen As Integer
Dim bufferChunk As String
Dim bufferPos As Integer
Dim lastCRPos As Integer
Dim searchTermLen As Integer

'** Exit if there is no search term
If (Trim(searchTerm) = "") Then
    Redim Preserve tempArray(0) As String
    tempArray(0) = ""
    SearchFile = tempArray
    Exit Function
Else
    searchTermLen = Len(searchTerm)
    searchTerm = Ucase(searchTerm)
End If

'** Start the timer
Dim startTime As Single
Dim elapsedTime As Single
startTime! = Timer()
'** Set maxTime as the maximum amount of time we'll search
'** (in seconds) before exiting the function
Dim maxTime As Integer
maxTime = 20

'** Get the file
fileNum = Freefile()
Open fileName For Input As fileNum

'** Search the file
count = 0
fileLength = Lof(fileNum)

Do While Not Eof(fileNum)
    '** Read the next 30,000 bytes into a string, plus the next
          '** full line after that. It is MUCH faster to read the data
          '** into a string and search it than it is to go through the
          '** entire file line-by-line.
    currentPos = Seek(fileNum)
    lastPos = currentPos
    If (fileLength = currentPos) Then
        Exit Do
    Elseif ((fileLength - currentPos) > 30000) Then
        '** if we're more than 30000 bytes from the end of
                '** the file, get the next 30000 bytes, up to the
                '** next linefeed
```

```
         buffer = Input$(30000, #fileNum)
         Line Input #fileNum, text
         buffer = buffer & text
     Else
         '** if we're more than 30000 bytes from the end of
                 '** the file, get the rest of the file
         buffer = Input$(fileLength - currentPos + 1, #fileNum)
     End If

     bufferLen = Len(buffer)
     buffer = Ucase(buffer)

     '** Check for the search term in the buffer. If it's found, return
     '** its position in the file.

     '** If there are a large number of "hits" in a string, it's
           '** actually more efficient to go through the string character
           '** by character than it is to repeatedly call the Instr function
     bufferPos = 1
     lastCRPos = lastPos
     Do Until (bufferPos > bufferLen)
         bufferChunk = Mid$(buffer, bufferPos, searchTermLen)
         If (bufferChunk = searchTerm) Then
             '** If we found a match, add to the array
             If (count < maxMatches) Or (maxMatches < 1) Then
                 Redim Preserve tempArray(count) As String
                 tempArray(count) = Cstr(lastCRPos) & ";" & _
                               Cstr(bufferPos + lastPos - 1)
             Else
                 Exit Do
             End If
             count = count + 1
         End If

         If (Asc(bufferChunk) = 10) Or (Asc(bufferChunk) = 13) Then
             lastCRPos = bufferPos + lastPos
         End If

         '** Exit if this is taking too long
         If ((Timer() - startTime!) > maxTime) Then
             Exit Do
         End If

         bufferPos = bufferPos + 1
     Loop    '** end of buffer search loop

     '** Exit if there are more than maxMatch matches
     If (count >= maxMatches) And (maxMatches > 0) Then
         Exit Do
     End If

     '** Exit if this is taking too long
     If ((Timer() - startTime!) > maxTime) Then
         Redim Preserve tempArray(count) As String
         tempArray(count) = "Error: Search timed out at " & Trim( Cstr( (Timer() -
startTime!) ) ) & " seconds."
         Exit Do
     End If
 Loop    '** EOF loop

 '** If no results were found, return nothing
 If (count = 0) Then
     Redim Preserve tempArray(0) As String
```

```
        tempArray(0) = ""
    End If

    Close fileNum
    SearchFile = tempArray
    Exit Function


processError:
    Dim errMess As String
    errMess = Error$

    Redim Preserve tempArray(count) As String
    tempArray(count) = "Error: " & errMess
    SearchFile = tempArray

    If (fileNum > 0) Then
        Close fileNum
    End If

    Exit Function

End Function
```

This function opens a file for Input, grabs 30,000 bytes of it at a time, and searches those 30,000 bytes for the search string. Every time it finds an instance of the search string, it adds its position to an array, along with the position of the last linefeed in the 30,000 byte chunk (which is useful if you want to retrieve the entire line that the search result is in). To watch it work, try copying the function into an agent and typing the following into the Initialize section:

```
fileName$ = "C:\Notes\Notes.ini"   '** your Notes.ini file here

searchResults = SearchFile(fileName$, "notes", 0)

fileNum% = Freefile()
Open fileName$ For Input As fileNum%
Forall pos In searchResults
    Seek #fileNum%, Cint( Left$(pos, Instr(pos, ";") - 1) )
    Line Input #fileNum%, text$
    Print text$
End Forall

Close #fileNum%
```

When you run the agent, it will print all the lines in your Notes.ini file that contain the word "notes" to the message bar at the bottom of the Notes client window.

Some possible enhancements you could make to this function are giving the user the ability to search for multiple terms, perform wildcard searches (see the wildcard search function in the chapter on strings), or specify whether or not a search should be case-sensitive.

GetNextToken – an Alternative to Line Input

The "Line Input" function is pretty handy for getting the next line of text in a sequential file. Unfortunately, sometimes you want very similar but just slightly different functionality. For example, you might have a text file that has information that' s delimited by something other than carriage returns or linefeeds. Or you might have Chr(0) characters embedded in your text file –

which would be not only incorrectly interpreted as linefeeds, but the Line Input function will actually skip the rest of the line (not sure if this is a bug or a feature).

The following function will simulate the Line Input function, but it allows you to specify what you want your delimiter to be. If you want this to act just like Line Input, you can use Chr(13) & Chr(10) as your delimiter.

Script – Get the next delimited string in a text file

```
Function GetNextToken (fileNum As Integer, delim As String, isEOF As Integer) As
String
 '** get the next "token" (string of text, delimited by delim)
 '** in a sequential file
 On Error Goto processError

 '** static variables used to maintain state
 Static buffer As String
 Static lastPos As Integer

 Dim dataLength As Integer
 Dim moreText As String
 Dim pos As Integer
 Dim result As String

 isEOF = False

 If (buffer = "") Then
     '** try to get more data from the file if our buffer is empty
     Select Case ( Lof(fileNum) - Seek(fileNum) )
     Case Is <= 0
         isEOF = True
         GetNextToken = ""
         Exit Function
     Case Is > 30000
         dataLength = 30000
     Case Else
         dataLength = Lof(fileNum) - Seek(fileNum)
     End Select

     buffer = Input$(dataLength, fileNum)
 End If

 '** get the next occurence of the delimiter, and make sure
 '** lastPos is valid
 If (lastPos < 1) Then
     lastPos = 1
 End If
 pos = Instr(lastPos, buffer, delim)

 If (pos > 0) Then
     '** if we found the delimiter, we can easily return the text
     '** between the previous delimiter and the one we just found
     result = Mid$(buffer, lastPos, pos - lastPos)
     lastPos = pos + Len(delim)
     If (lastPos >= Len(buffer)) Then
         lastPos = 0
         buffer = ""
     End If
 Else
     '** if we couldn't find the delimiter, we at least want everything
     '** up to the end of the string
     result = Mid$(buffer, lastPos)
```

```
      '** we can then reset the static variables and make a recursive
      '** call, which will get the first part of the next buffer, up to
      '** the delimiter (or nothing, if we're at the end of the file)
      lastPos = 0
      buffer = ""
      result = result & GetNextToken(fileNum, delim, isEOF)
  End If

  GetNextToken = result
  Exit Function

processError:
  '** the most common error will be an overflow (#228), which will
  '** happen if the delimiter hasn't been found for a large stretch
  '** of the file. You can deal with that as a special case if you want.
  '** Also, fileNum may be invalid.
  GetNextToken = result
  Exit Function

End Function
```

This function takes as input the file number of an open sequential file and the delimiter that you' re looking for, and it returns the isEOF variable (which indicates whether or not you' ve reached the end of the file) and the delimited string. It uses static variables to maintain state, although you should keep in mind that if another function uses the same file number to change the cursor position in the file, your results could become skewed. You could use another static variable to make sure your cursor position remains constant, if you' re worried about that.

Also, this function makes a recursive call in order to get the end of the token string, because that' s a little easier than duplicating the file-reading logic again at the end of the function, or trying to build a loop to return to the top of the function to read the next buffer. If you want to see more examples of recursive functions in action, please see the chapter on Strings.

Creating a Listing of Notes ID Files in a Directory

Here' s an example that puts together several of the concepts in this chapter. It will get all the *.ID files in a given directory and output, to a comma-delimited file, information about the ID files it finds.

Script – List information about Notes ID files in a given directory

```
Function ListIDFileInfo (idDir As String, outputFileName As String) As Integer
  '** go through all the ID files in a directory and output information
  '** about them in comma-delimited format to outputFileName
  On Error Goto processError

  Dim outputFileNum As Integer
  Dim fileOpened As Integer
  Dim idFileNum As Integer
  Dim idFileName As String
  Dim idFileNameWhole As String
  Dim searchResults As Variant
  Dim nameString As String
  Dim slash as String

  slash = GetFileSeparator()

  '** open the output file
  outputFileNum = Freefile()
  Open outputFileName For Output As outputFileNum
  fileOpened = True
  Print #outputFileNum, "ID File Information for " & idDir & "; created " &
Format(Now(), "m-d-yy h:mm:ss")
  Write #outputFileNum, "FILE NAME", "USER NAME", "FILE DATE"

  '** traverse the ID directory
  idFileName = Dir$(idDir & slash & "*.ID")
  Do Until (idFileName = "")
      idFileNameWhole = idDir & slash & idFileName
      '** find the user name in the file, with the user-defined
      '** function SearchFile
      searchResults = SearchFile(idFileNameWhole, "CN=", 2)

      '** get the name so we can use it
      idFileNum = Freefile()
      Open idFileNameWhole For Input As idFileNum
      Seek #idFileNum, Cint( Mid$(searchResults(1), Instr(searchResults(1), ";") + 1) )
      nameString = Input$(100, #idFileNum)
      nameString = Trim$(Left$(nameString, Instr(nameString, Chr(0)) - 1))

      '** print ID file information to our output file
      Write #outputFileNum, idFileName, nameString, Filedatetime(idFileNameWhole)

      '** get the next ID file
      idFileName = Dir$()
  Loop

  Close #outputFileNum

  ListIDFileInfo = True
  Exit Function

processError:
  If fileOpened Then
      Print #outputFileNum, "Error: " & Error$
  End If
  Reset
  ListIDFileInfo = False
  Exit Function

End Function
```

The comments within the code should help you understand what's going on in the function. Notice how all the calls to the Dir$ function after the first call don't include any arguments. That's because the Dir$ function automatically remembers the last set of arguments you used, and it gets the next file based on those arguments.

Binary File Manipulation: Getting Information about an MP3 File

Here's a fun example that will give you a little taste of reading binary file. It's a function that will read information about all the MP3 files in a given directory and output the information into a comma-delimited file.

Script – Getting information about MP3 files (binary read)

```
Function OutputMP3Info (mp3Dir As String, outputFileName As String) As Integer
  '** send information about MP3 files in a directory to a
  '** comma-delimited file
  On Error Goto processError
```

```
 Dim outputFileNum As Integer
 Dim fileOpened As Integer
 Dim mp3FileNum As Integer
 Dim mp3FileName As String
 Dim mp3FileNameWhole As String
 Dim tagInfo As String * 64
 Dim song As String, artist As String, album As String
 Dim songYear As String, comment As String, genre As Integer
 Dim slash as String

 slash = GetFileSeparator()

 '** open the output file
 outputFileNum = Freefile()
 Open outputFileName For Output As outputFileNum
 fileOpened = True
 Print #outputFileNum, "MP3 File Information for " & mp3Dir & "; created " &
Format(Now(), "m-d-yy h:mm:ss")
 Write #outputFileNum, "FILE NAME", "SONG NAME", "ARTIST", "ALBUM", "YEAR", "COMMENT",
"GENRE NUMBER"

 '** traverse the MP3 directory
 mp3FileName = Dir$(mp3Dir & slash & "*.MP3")
 Do Until (mp3FileName = "")
     mp3FileNameWhole = mp3Dir & slash & mp3FileName

     '** open the MP3 file so we can get the tag information, which is
     '** the last 128 bytes of the file
     mp3FileNum = Freefile()
     Open mp3FileNameWhole For Binary Access Read Shared As #mp3FileNum

     Seek #mp3FileNum, Filelen(mp3FileNameWhole) - 127
     Get #mp3FileNum, , tagInfo

     '** translate the tagInfo to a LotusScript string, so we can
          '** get the substrings
     tagInfoString = ConvertBinaryText(tagInfo, " ")

     song = Trim$(Mid$(tagInfoString, 4, 30))
     artist = Trim$(Mid$(tagInfoString, 34, 30))
     album = Trim$(Mid$(tagInfoString, 64, 30))
     songYear = Trim$(Mid$(tagInfoString, 94, 4))
     comment = Trim$(Mid$(tagInfoString, 98, 30))
          genre = Asc(Mid$(tagInfoString, 128, 1))

     Close #mp3FileNum

     '** print MP3 file information to our output file
     Write #outputFileNum, mp3FileName, song, artist, album, songYear, comment, genre

     '** get the next file
     mp3FileName = Dir$()
 Loop

 Close #outputFileNum

 OutputMP3Info = True
 Exit Function

processError:
 If fileOpened Then
     Print #outputFileNum, "Error: " & Error$
 End If
```

```
    Reset
    OutputMP3Info = False
    Exit Function

  End Function
```

Because we' re reading the MP3 files in Binary mode, we' ll also need this helper function to read the tag information as a string:

Script – Translate binary string information to LotusScript format

```
  Function ConvertBinaryText (binString As String, convertChrZero As String) As String
       Dim newLen As Long
       Dim returnString As String
       Dim i As Integer
       Dim nextChar As String

       newLen = Len(binString) * 2
       returnString = Space$(newLen)
       returnString = ""

       For i = 1 To newLen
            newChar = Midb$(binString, i, 1)
            If (newChar = Chr(0)) Then
                 returnString = convertChrZero
            Else
                 returnString = returnString & nextChar
            End If
       Next

       ConvertBinaryText = returnString

  End Function
```

The reason why we have to "translate" the tag data is because a string character in LotusScript is 2 bytes long, while non-Unicode characters in the binary file are all one byte long. The translation routine simply takes the string of characters in the single byte representation and changes it to the double-byte representation that we' re used to. This is also why we defined the tagInfo variable as a string of length 64, despite the fact that we' re reading in 128 characters. Please see the String chapter for more information about string representations.

Of course, if we know that the information we want from the file is all string-based, we can also opt to read the file as a sequential file instead of a binary one – even though it really is a binary file. By doing this, LotusScript will automatically do the string conversions for us.

Script – Getting information about MP3 files (sequential read)

```
  Function OutputMP3Info2 (mp3Dir As String, outputFileName As String) As Integer
    '** send information about MP3 files in a directory to a
    '** comma-delimited file
    On Error Goto processError

    Dim outputFileNum As Integer
    Dim fileOpened As Integer
    Dim mp3FileNum As Integer
    Dim mp3FileName As String
    Dim mp3FileNameWhole As String
    Dim tagInfo As String
    Dim tagInfoString As String * 128
    Dim song As String, artist As String, album As String
```

```
 Dim songYear As String, comment As String, genre As Integer

 '** open the output file
 outputFileNum = Freefile()
 Open outputFileName For Output As outputFileNum
 fileOpened = True
 Print #outputFileNum, "MP3 File Information for " & mp3Dir & "; created " &
Format(Now(), "m-d-yy h:mm:ss")
 Write #outputFileNum, "FILE NAME", "SONG NAME", "ARTIST", "ALBUM", "YEAR", "COMMENT",
"GENRE NUMBER"

 '** traverse the MP3 directory
 mp3FileName = Dir$(mp3Dir & "\*.MP3")
 Do Until (mp3FileName = "")
     mp3FileNameWhole = mp3Dir & "\" & mp3FileName

     '** open the MP3 file so we can get the tag information, which is
     '** the last 128 bytes of the file
     mp3FileNum = Freefile()
     Open mp3FileNameWhole For Input Access Read Shared As #mp3FileNum

     Seek #mp3FileNum, Filelen(mp3FileNameWhole) - 127
     tagInfoString = Input$(128, #mp3FileNum)

     song = Char0Trim(Mid$(tagInfoString, 4, 30))
     artist = Char0Trim(Mid$(tagInfoString, 34, 30))
     album = Char0Trim(Mid$(tagInfoString, 64, 30))
     songYear = Char0Trim(Mid$(tagInfoString, 94, 4))
     comment = Char0Trim(Mid$(tagInfoString, 98, 30))
     genre = Asc(Mid$(tagInfoString, 128, 1))

     Close #mp3FileNum

     '** print MP3 file information to our output file
     Write #outputFileNum, mp3FileName, song, artist, album, songYear, comment, genre

     '** get the next file
     mp3FileName = Dir$()
 Loop

 Close #outputFileNum

 OutputMP3Info2 = True
 Exit Function

processError:
 If fileOpened Then
     Print #outputFileNum, "Error: " & Error$
 End If
 Reset
 OutputMP3Info2 = False
 Exit Function

End Function
```

This also has a helper function, which trims off the Chr(0) characters from the end of the substrings:

Script – Trim the Chr(0) characters from a string

```
Function Char0Trim (theString As String) As String
 '** get everything to the left of the first Chr(0) in the string
 Dim pos As Integer
 pos = Instr(theString, Chr(0))
```

```
    If (pos > 0) Then
        Char0Trim = Trim$(Left$(theString, pos - 1))
    Else
        Char0Trim = Trim$(theString)
    End If

  End Function
```

The trick with reading binary files is having knowledge about the format of the information in the file. Unlike a structured text file, which is generally easy to translate just by looking at the text in the file, the structure of data within a binary file is usually pretty cryptic.

For more information about the structure of MP3 files, a good reference is the Programming section of http://www.mp3-tech.org .

Searching for Files or Directories

This function will allow you to search for a file or directory recursively, so that the search includes subdirectories. You can use any wildcards supported by the Dir function.

Script – Searching for Files or Directories

```
  Function FindFiles (Byval startPath As String, searchString As String,
  includeDirectories As Integer) As String
    '** recursively find files, searching all subdirectories, and return
    '** all file names in a semi-colon-delimited string
    On Error Goto processError

    Dim tempString As String
    Dim resultString As String
    Dim fileName As String
    Dim dirName As String
    Dim dirNameList List As String
    Dim slash as String

    slash = GetFileSeparator()

    If (Right$(startPath, 1) = slash) Then
        startPath = Left$(startPath, Len(startPath) - 1)
    End If

    '** get all the matching files in this directory
    fileName = Dir$(startPath & slash & searchString, 0+2+4+16)
    Do Until (fileName = "")
        If (Getfileattr(startPath & slash & fileName) And 16) Then
            '** if we got a subdirectory that matched, only include it
            '** in the list if includeDirectories = True
            If (includeDirectories = True) Then
                tempString = tempString & ";" & startPath & slash & fileName & "[DIR]"
            End If
        Else
            '** otherwise, it's a file, and should be added to the list
            tempString = tempString & ";" & startPath & slash & fileName
        End If

        fileName = Dir$()
    Loop

    '** now get all the subdirectories under startPath, so we can
    '** check them too
```

```
 dirName = Dir$(startPath & slash & "*.*", 16)
 Do Until (dirName = "")
     If (Getfileattr(startPath & slash & dirName) And 16) And (dirName <> "..") And
(dirName <> ".") Then
         dirNameList( dirName ) = startPath & slash & dirName
     End If
     dirName = Dir$()
 Loop

 '** now recursively search all the subdirectories
 Forall subdir In dirNameList
     resultString = FindFiles(subdir, searchString, includeDirectories)
     If Not (resultString = "") Then
         tempString = tempString & ";" & resultString
     End If
 End Forall

 If (Left$(tempString, 1) = ";") Then
     tempString = Mid$(tempString, 2)
 End If

 Erase dirNameList
 FindFiles = tempString
 Exit Function

processError:
 FindFiles = Error$
 Erase dirNameList
 Exit Function

End Function
```

This function uses recursion to search through subdirectories. You can use a script like the following to see what it does:

```
Dim result$, lastPos%, pos%
result$ = FindFiles("C:\", "notes*.*", True)

lastPos% = 1
pos% = Instr(lastPos, result$, ";")
Do Until (pos = 0)
    Print Mid$(result$, lastPos, pos - lastPos)
    lastPos = pos + 1
    pos = Instr(lastPos, result$, ";")
Loop
```

One interesting point: the reason why you need to create a list of the subdirectories before you search through them is because of the way the Dir function works. When you call the Dir function with no arguments, it simply looks for the next file, based on the last filespec you gave to it. However, if you're looping through a directory using Dir and then you call another function that also calls Dir, then the next time you call Dir in the original function, it will use the filespec from the second function, because that was the last one it used. So in this function we get all the subdirectory names before we recurse, to make sure that doesn't happen.

LotusScript Version of the DOS Deltree Command

Here's a variation on some of the previous scripts: a LotusScript implementation of the Deltree command in DOS, which deletes a directory plus all files and subdirectories under it. LotusScript has a built-in RmDir function that will delete directories, but it will only delete empty ones. This function takes care of that.

Script – LotusScript version of Deltree

```
Function Deltree (Byval dirName As String, shouldDeleteDir As Integer) As Integer
  '** LotusScript version of the Deltree DOS command. The User optionally
  '** has the ability to specify whether the base directory should be
    '** deleted (when shouldDeleteDir is True), or if we just want to clean
    '** out all the files and subdirectories in it (False).
  On Error Goto processError

  Dim dirNameList List As String
  Dim fileName As String
  Dim fullFileName As String

  '** trim off any trailing "\" on the directory name
  Do While (Right$(dirName, 1) = "\")
      dirName = Left$(dirName, Len(dirName) - 1)
  Loop

  '** exit early if there is no directory of this name
  If (Dir$(dirName, 16) = "") Then
      Deltree = False
      Exit Function
  End If

  '** first, delete all the files in this directory
  fileName = Dir$(dirName & "\*.*", 2+4+16)
  Do Until (fileName = "")
      fullFileName = dirName & "\" & fileName

      If (Getfileattr(fullFileName) And 16) Then
          '** it's a directory, so add it to the dirNameList,
          '** unless it's one of the "." or ".." directories
          If (fileName <> ".") And (fileName <> "..") Then
              dirNameList(fileName) = fullFileName
          End If
      Else
          '** it's a file, so delete it
          '** (4.6 doesn't like to delete Read-Only files, so...)
          Setfileattr fullFileName, 0
          Kill fullFileName
      End If

      fileName = Dir$
  Loop

  '** due to a limitation in some versions of 4.6, we also need to
  '** look for subdirectories like this in a separate loop.
  '** Be aware that some versions of 4.6 can never see Hidden
  '** directories with the Dir function.
  fileName = Dir$(dirName & "\*.*", 16)
  Do Until (fileName = "")
      fullFileName = dirName & "\" & fileName

      If (Getfileattr(fullFileName) And 16) Then
          '** it's a directory, so add it to the dirNameList,
          '** unless it's one of the "." or ".." directories
          If (fileName <> ".") And (fileName <> "..") Then
              dirNameList(fileName) = fullFileName
          End If
      End If
```

```
        fileName = Dir$
    Loop

    '** do this recursively for all the subdirectories
    Forall subDir In dirNameList
        Call Deltree(subDir, True)
    End Forall

    '** delete the main directory itself, if we're supposed to
    If shouldDeleteDir Then
        Rmdir dirName
    End If

    Erase dirNameList
    Exit Function


 processError:
    '** if we got an error, we probably didn't have rights to delete
    '** a file, or it was in use or something, but go ahead and try
    '** to clean out the rest of the directory
    Dim errMsg As String
    errMsg = "Error " & Err & ": " & Error$
    Deltree = False
    Resume Next

 End Function
```

The inline comments in the code have some information about how some of the functions in Notes 4.6 are a little quirky. Otherwise, this script should be pretty easy to follow: get all the files in the directory and delete them one by one, and then do the same for any subdirectories by making a recursive call.

The one little addition to this function is the shouldDeleteDir option, which is a Boolean value that allows you to specify whether you want to do a full Deltree, or if you just want to clear out the directory you specified. You could easily modify this function so that it only deletes certain files and directories in a directory tree, based on a given search string (like MS*.DLL or something), or it deletes files but not directories, or whatever else might be useful to you.

Miscellaneous Code

The code in this chapter is a collection of functions and routines that didn't seem to fit anywhere else in the book, but that might be useful in general. Information is presented in no particular order

Running an Agent on the Last Day of the Month

Using the agent scheduling features in Notes (up to at least R5), you can't write an agent that runs on the last day of the month. The monthly scheduling option only allows you to enter a number in the day-of-month field, but not all months have the same amount of days, so if you schedule an agent for day 28, it won't really run on the last day of most months, and if you schedule it for day 31, then it won't run at all on some months.

One answer is to run the agent daily, and add this little bit of script to the beginning of the Initialize sub:

```
Dim tomorrow As New NotesDateTime(Today)
tomorrow.AdjustDay(1)
If (Month(tomorrow.LSLocalTime) = Month(Today)) Then
    '** if tomorrow is the same month as today, we're
    '** not on the last day of the month
    Exit Sub
End If
```

The script merely checks to see if tomorrow's date is the same month as today's date (if it's the end of the month, tomorrow will be a different month), and if it is, it immediately exits the script. This way, the agent will start to run every day, but it will exit early if it's not the end of the month.

Get Server Time

Here's an easy way to get the time on the server that a database is on (you have to have at least Author rights to the database that you put this code in).

```
 Dim session As New NotesSession
 Dim db As NotesDatabase
 Dim ServerName As NotesName
 Dim doc As NotesDocument

 Set db = session.CurrentDatabase
 Set ServerName = New NotesName(db.Server)
 Set doc = db.CreateDocument

 Messagebox "The current time on " & ServerName.Common & " is " & Cstr(doc.Created) &
Chr(10) & _
 "The time on this workstation is " & Cstr(Now), 0, "Time"
```

Run Agent on Server

This is an agent that allows you to run other agents on the server.

```
Dim session As New NotesSession
Dim db As NotesDatabase
Set db = session.CurrentDatabase

'** get a list of all the agents in this database
i% = 0
```

```
Forall ag In db.Agents
    Redim Preserve agentArray(i%) As String
    agentArray(i%) = ag.Name
    i% = i% + 1
End Forall

'** sort the list of agents to make it easier to find things
'** (this is a fairly slow sort, but it'll do for small arrays;
'** it'll probably take much more time to get the list of agents
'** than it will to sort them)
For j% = 0 To (i% - 2)
    For k% = (j% + 1) To (i% - 1)
        If (agentArray(k%) < agentArray(j%)) Then
            swap$ = agentArray(j%)
            agentArray(j%) = agentArray(k%)
            agentArray(k%) = swap$
        End If
    Next
Next

'** ask the user which agent to run
sVariant = ws.Prompt(PROMPT_OKCANCELLIST, "Choose An Agent", _
"Which agent would you like to run?", "", agentArray)

'** exit if the user hits cancel or doesn't choose anything
If (Cstr(sVariant) = "") Then
    Exit Sub
Else
    agentName$ = Cstr(sVariant)
End If

'** make sure the user really wants to do this
prompt$ = "Are you sure you want to run """ & agentName$ & """ on the server?"
shouldContinue% = Messagebox(prompt$, 4+32, "Continue?")
If (shouldContinue% = 7) Then
    Exit Sub
End If

'** run this agent on the server
Print "Attempting to run agent. This may take a while, depending on the agent."
Set agent = db.GetAgent(agentName$)
res% = agent.RunOnServer

If (res% = 0) Then
    Messagebox """" & agentName$ & """ ran successfully.", 0, "Success"
Else
    Messagebox """" & agentName$ & """ did not run successfully.", 0+48, "Failure"
End If
```

Toggle Scheduled Agent Status

This is an agent that allows you to toggle the status of a scheduled agent in a database, without having to go to the agent list in Designer.

```
Dim session As New NotesSession
Dim db As NotesDatabase
Dim agent As NotesAgent
Set db = session.CurrentDatabase

'** get a list of all the scheduled agents in this database
Redim agentArray(0) As String
```

```
    i% = 0
    Forall ag In db.Agents
        If (ag.Trigger = 1) Then
            '** this is a scheduled agent; add it to the list
            Redim Preserve agentArray(i%) As String
            agentArray(i%) = ag.Name
            i% = i% + 1
        End If
    End Forall

    '** sort the list of agents to make it easier to find things
    '** (this is a fairly slow sort, but it'll do for small arrays;
    '** it'll probably take much more time to get the list of agents
    '** than it will to sort them)
    For j% = 0 To (i% - 2)
        For k% = (j% + 1) To (i% - 1)
            If (agentArray(k%) < agentArray(j%)) Then
                swap$ = agentArray(j%)
                agentArray(j%) = agentArray(k%)
                agentArray(k%) = swap$
            End If
        Next
    Next

    '** present a list of the scheduled agents, and let the user pick one
    Dim ws As New NotesUIWorkspace
    sVariant = ws.Prompt(PROMPT_OKCANCELLIST, "Choose An Agent", _
    "Please choose an agent to enable/disable from the list below:", "", agentArray)

    '** exit if the user hits cancel or doesn't choose anything
    If (Cstr(sVariant) = "") Then
        Exit Sub
    Else
        agentName$ = Cstr(sVariant)
    End If

    '** get the status of the agent the user picked, and ask if they
    '** really want to do this
    Set agent = db.GetAgent(agentName$)
    If (agent.IsEnabled) Then
        agentStatus$ = "Enabled"
        agentNewStatus$ = "Disabled"
    Else
        agentStatus$ = "Disabled"
        agentNewStatus$ = "Enabled"
    End If

    prompt$ = "The scheduled agent " & agentName$ & " is currently " & _
    agentStatus$ & ". Would you like to toggle the status of this agent?"
    shouldContinue% = Messagebox(prompt$, 4+32, "Continue?")
    If (shouldContinue% = 7) Then
        Exit Sub
    End If

    '** If we're here, toggle the agent status
    agent.IsEnabled = Not agent.IsEnabled
    Call agent.Save

    Messagebox "Done. The Agent " & agentName$ & " is now " & agentNewStatus$, 0,
  "Success"
```

Get User Groups

This is an example of a recursive function that gets all the groups that a user is in, including nested groups. I've included an example Initialize sub from an agent to demonstrate the use.

```
Sub Initialize
 '** Determine the groups that the current user is in, including
 '** all nested groups
 On Error Goto processError

 Dim session As New NotesSession
 Dim db As NotesDatabase
 Dim view As NotesView
 Dim serverName As String
 Dim userName As String
 Dim groupList As Variant

 serverName = "MyNABServer"
 userName = session.UserName

 Set db = session.GetDatabase( serverName, "names.nsf" )
 Set view = db.GetView( "Groups" )

 '** create a text file for output
 fileNum% = Freefile()
 fileName$ = "C:\UserGroupInfo.txt"
 Open fileName$ For Output As fileNum%
 Print #fileNum%, "User Group Info for " & userName & " on " & serverName
 Print #fileNum%, ""

 '** get the group information
 Call GetGroups( userName, view, "", 0, fileNum% )

 '** close the file and exit
 Close fileNum%

 Exit Sub

processError:
 Messagebox "Error " & Cstr(Err) & ": " & Error$
 Reset
 Exit Sub

End Sub

Function GetGroups (lookupName As String, groupView As NotesView, alreadyUsed As
String, _
indentLevel As Integer, fileNum As Integer)
 '** This sub will recursively iterate through all the groups in the NAB,
 '** figuring out which ones the given user or group is in.
 On Error Goto processError

 Dim doc As NotesDocument
 Dim memberItem As NotesItem
 Dim groupName As String
 Dim tabString As String

 '** use tabString to indent the entry, indicating that a group is a
 '** member of the group below it
 For i% = 1 To indentLevel
     tabString = tabString & Chr(9)
 Next
```

```
  '** step through the group documents in the NAB that we're looking at
  Set doc = groupView.GetFirstDocument

  Do While Not (doc Is Nothing)
      Set memberItem = doc.GetFirstItem( "Members" )
      groupName = doc.ListName(0)

      '** Check for direct inclusion in a group. If the lookup name is
      '** in the Members text list and we haven't already used the group
      '** (if we did, it will be in the alreadyUsed string, and would
      '** represent a circular reference), output the group name to our
      '** file and recurse
      If (memberItem.Contains( lookupName )) And (Instr(1, alreadyUsed, "~" & groupName
& "~", 5) < 1) Then
          Print #fileNum%, tabString & groupName
          '** recursion will find other groups that this group is
          '** a member of
          Call GetGroups( groupName, groupView, alreadyUsed & "~" & groupName & "~",
indentLevel + 1, fileNum )
      End If

      Set doc = groupView.GetNextDocument( doc )
  Loop

  Exit Function

processError:
  Print #fileNum%, "Error " & Cstr(Err) & ": " & Error$
  Exit Function

End Function
```

Get Database Sizes (even if you don't have access to the databases)

This script lists the sizes of all the databases on a given server, even if you don’t have access to the databases themselves. It’s handy for comparing mail file sizes on a mail server.

```
  Dim dbdir As New NotesDbDirectory("MyMailServer")
  Dim db As NotesDatabase

  '** create a text file for output
  fileNum% = Freefile()
  fileName$ = "C:\MailDatabases.csv"
  Open fileName$ For Output As fileNum%

  Set db = dbdir.GetFirstDatabase(DATABASE)
  While Not(db Is Nothing)
      Print #fileNum%, db.Title & "," & db.FilePath & "," & db.Size
      Set db = dbdir.GetNextDatabase
  Wend

  Close fileNum%
```

Largest Subset of Two Strings

This is just a little something I was playing around with for a while. I read about how biological researchers often had to find the largest subset of a string between multiple strings, in order to make genetic comparisons. The trick was, it was the largest subset of characters that proceeded each other, but weren't necessarily next to each other. For example, in the two strings "dogdog" and "digdig", the largest subset between these two strings would be "dgdg".

Anyway, I thought this would be an interesting code project, so here's how I looked at the problem. I have no idea if this is an efficient implementation, or even an entirely correct one.

```
'** global variables
Dim matchList List As String
Dim matchListArray List As Variant
Dim matchListCount As Integer
Const MAX_LIST_SIZE = 10000

Function MatchTwoString (string1 As String, string2 As String, resetList As Integer)
As String
  '** find the first best possible match of sequential characters in two
  '** strings ("first" means that there may be multiple matches that are
  '** the same length, but we just want the first of those matches)
  Dim char1 As String, char2 As String
  Dim pos1 As Integer, pos2 As Integer
  Dim usedLetter1 As String, usedLetter2 As String
  Dim i As Integer, j As Integer
  Dim lastMatch As String, testMatch As String

  '** reset our global variables, if necessary. We use these to
  '** determine if we've already calculated this match before.
  If resetList Then
      Erase matchList
      matchListCount = 0
  End If

  '** optimization: exit early if we've already done this evaluation
  If Iselement(matchList(string1 & "~" & string2)) Then
      MatchTwoString  = matchList(string1 & "~" & string2)
      Exit Function
  End If

  '** optimization: exit early if every single letter in the shorter
  '** or the strings has a match in the longer of the two
  Dim shortString As String
  Dim longString As String
  If (Len(string1) <= Len(string2)) Then
      shortString = string1
      longString = string2
  Else
      shortString = string2
      longString = string1
  End If

  For i = 1 To Len(shortString)
      If (pos1 = Len(longString)) Then
          pos1 = 0
          Exit For
      End If

      pos1 = Instr(pos1 + 1, longString, Mid$(shortString, i, 1))
```

```
        If (pos1 = 0) Then
            Exit For
        End If
    Next

    '** if we never reset pos1 back to zero, then all the characters
    '** in string1 were a match, and our match is string1. Also, since
    '** this is a pretty quick matching technique, don't bother adding
    '** the result to the list, because that will waste memory.
    If (pos1 > 0) Then
        MatchTwoString = shortString
        'Exit Function
        Goto endOfFunction
    End If

    '** if we got here, we should start checking for matches
    '** (NOTE: you can also substitute shortString for string1
    '** and longString for string2 here, but you then lessen your
    '** chances for running into a match you've already checked
    '** for before as you recurse the function)
    For i = 1 To Len(string1)
        char1 = Mid$(string1, i, 1)
        If (Instr(usedLetter1, char1) = 0) Then
            '** for every character we haven't tried before, look for a
            '** string match
            usedLetter1 = usedLetter1 & char1
            pos1 = Instr(string2, char1)

            If (pos1 > 0) Then
                '** try to find a match for the next character after
                '** this one
                If (Len(lastMatch) = 0) Then
                    lastMatch = char1
                End If

                If (i = Len(string1)) Then
                    Exit For
                End If

                usedLetter2 = ""
                For j = i + 1 To Len(string1)
                    char2 = Mid$(string1, j, 1)
                    If (Instr(usedLetter2, char2) = 0) Then
                        '** if we haven't tried looking for
                        '** this character combination
                        '** yet, see what we find
                        usedLetter2 = usedLetter2 & char2
                        pos2 = Instr(pos1 + 1, string2, char2)
                        If (pos2 > 0) Then
                            testMatch = char1 & char2
                            If (j < Len(string1)) And (pos2 < Len(string2)) Then
                                testMatch = testMatch & MatchTwoString(Mid$(string1, j +
1), Mid$(string2, pos2 + 1), False)
                            End If

                            If (Len(testMatch) > Len(lastMatch)) Then
                                lastMatch = testMatch
                            End If
                        End If
                    End If

                Next
            End If
        End If

    Next

    MatchTwoString = lastMatch

endOfFunction:
    '** add to our global match list, if there isn't too much data
    '** in there already
    If (matchListCount < MAX_LIST_SIZE) Then
        matchList(string1 & "~" & string2) = MatchTwoString
        matchListCount = matchListCount + 1
    End If

End Function
```

Export View to CSV

This is a generic routine that will export the current view in comma-delimited format.

```
    Dim ws As New NotesUIWorkspace
    Dim uiview As NotesUIView
    Dim view As NotesView
    Dim doc As NotesDocument

    Set uiview = ws.currentview
    Set view = uiview.view

    Dim columnList List As Integer
    Dim colcount As Integer
    Forall column In view.Columns
        If (column.IsHidden = False) And (column.IsIcon = False) Then
            columnList(colcount) = colcount
        End If
        colcount = colcount + 1
    End Forall

    Dim fileNum As Integer
    Dim fileName As String
    fileNum = Freefile()
    fileName = Strrightback(view.Name, "\")
    If (fileName = "") Then
        fileName = view.Name
    End If
    fileName = fileName & ".csv"

    Dim var As Variant
    var = ws.SaveFileDialog(False, "File List", "Comma-Delimited Files|*.csv", "c:", 
fileName)
    If (var(0) = "") Then
        Exit Sub
    Else
        fileName = var(0)
    End If

    Open fileName For Output As fileNum

    Dim printString As String
    Dim i As Integer
```

```
  Set doc = view.GetFirstDocument
  Do Until (doc Is Nothing)
      printString = ""
      Forall c In columnList
          If Isscalar(doc.ColumnValues(c)) Then
              printString = printString & """" & ReplaceSubstring(doc.ColumnValues(c),
"""", """""") & ""","
          Else
              printString = printString & """"
              var = doc.ColumnValues(c)
              For i = Lbound(var) To Ubound(var)
                  printString = printString & ReplaceSubstring(var(i), """", """""") &
","
              Next
              printString = printString & ""","
          End If
      End Forall
      Print #fileNum, printString
      Set doc = view.GetNextDocument(doc)
  Loop

  Close fileNum
  Print "Finished exporting view to " & fileName
```

Add Admin to ACLs

This is an agent that will add an “Admin” group as a Manager to the ACL of the specified databases. Normally, you’d want to run this as a scheduled agent on the server.

```
Sub Initialize
  '** This agent will add the ADMIN group to the ACL in all mail databases.
  '** Make sure you run it on the server, while logged in as the server
  On Error Goto processError

  Dim dbACL As NotesACL
  Dim dbACLEntry As NotesACLEntry
  Dim dbdir As New NotesDbDirectory("")
  Dim db As NotesDatabase
  Dim group As String
  Set db = dbdir.GetFirstDatabase(DATABASE)
  group = "Admin"

  Do While Not (db Is Nothing)
      '** if we're in the "mail" directory, check the database
      If (Ucase(Left$(db.FilePath, 4)) = "MAIL") Then
          Print "Checking " & db.FileName

          '** open the database so we can get its elements
          Call db.Open("", "")

          Set dbACL = db.ACL
          Set dbACLEntry = dbACL.GetFirstEntry()
          found% = False

          Do While Not(dbACLEntry Is Nothing)
              '** get the name of this ACL entry
              Set theName = New NotesName(dbACLEntry.Name)
              If (theName.IsHierarchical) Then
                  EntryName$ = theName.Common
              Else
                  EntryName$ = dbACLEntry.Name
```

```
             End If

             '** if this entry is "ADMIN", check the access and modify if necessary
             If (Ucase(EntryName$) = Ucase(group)) Then
                 found% = True

                 If Not (dbACLEntry.Level = ACLLEVEL_MANAGER) Then
                     Print "Updating " & group & " access on " & db.FileName
                     Call db.GrantAccess(dbACLEntry.Name, ACLLEVEL_MANAGER)
                 End If

                 Exit Do
             End If

             '** next entry
             Set dbACLEntry = dbACL.GetNextEntry(dbACLEntry)
         Loop

         '** if ADMIN was not included in this ACL, add it
         If Not found% Then
             Print "Adding " & group & " access on " & db.FileName
             Set dbACLEntry = dbACL.CreateACLEntry (group, ACLLEVEL_MANAGER)
         End If

         '** save the changes
         Call dbACL.Save

     End If

     Set db = dbdir.GetNextDatabase
 Loop

 Print "Finished."
 Exit Sub


processError:
 Print "Error " & Cstr(Err()) & ": " & Error$
 Messagebox "Error " & Cstr(Err()) & ": " & Error$, 0+48, "Error"
 Exit Sub


End Sub
```

Zip Web Server Logs

This agent will zip all the web server logs for the previous day into a zip file. All the logs for a given month are stored in the same file. You would normally schedule this to run sometime early in the morning, after midnight.

```
Sub Initialize
 On Error Goto processError
 Dim section As String

    '** Get the location of the web server logs, and use that as our working.
    '** directory. So we don't have to hard-code anything, we can look the
 '** directory up in the Server document for this server.
 section = "Get the log directory"
 Dim logdir As String
    ' logdir = "d:\notes\data\domino\logs"
 Dim session As New NotesSession
 Dim db As NotesDatabase
 Dim view As NotesView
 Dim doc As NotesDocument

 Set db = session.GetDatabase("", "names.nsf")
 Set view = db.GetView("($ServersLookup)")
 Set doc = view.GetDocumentByKey(db.Server, True)

 If (doc Is Nothing) Then
     Print db.Server & " server document not found in names.nsf. Zip Web Logs agent
could not run."
     Exit Sub
 Else
     logdir = doc.HTTP_LogDirectory(0)
 End If

 '** If no log directory is specified, the Notes data directory is the default
 If (logdir = "") Then
     logdir = session.GetEnvironmentString("Directory", True)
 End If

 Chdir logdir

 '** This is the name and path of the ZIP executable that we'll be using.
 '** In this case, it’s the program zip.exe from http://www.info-zip.org
 '** This is a nice program because it's free, and it's been ported to all
 '** sorts of different platforms.
 Dim zipExe As String
 zipExe = logdir & "\" & "zip.exe "

 '** Figure out today's date as a string in the format MonthDayYear (like
 '** 11062000). The log files are named xxx-logMMDDYYY.log, so we'll want to
 '** know what today's date string is so we don't archive today's files.
 '** NOTE: the naming convention varies from one version of Notes to the next,
 '** and is dependent on other settings in the logging section of the Server
 '** document. Find out what the server's doing, and adjust as necessary.
 section = "Get today's date"
 Dim todayString As String
 todayString = Format$(Today, "mmddyyyy")

 '** Create a batch file that will do all of the archiving for us. This is
 '** because you can't wait for a Shell statement to end before continuing with
 '** LotusScript statements, so if you try to Shell out all the commands for
 '** zipping and deleting, you'll run into file contention problems.
 section = "Create Batch File"
 batFileNum% = Freefile()
 batFileName$ = logdir & "\" & "LogArc.bat"
 Open batFileName$ For Output As batFileNum%
 Print #batFileNum%, "REM - Web Log Archive Batch File, created " & Format(Now(), "m-
d-yy h:mm:ss")
 Print #batFileNum%, ""

 '** Get the web log directory and start going through all the files
 section = "Step through the log directory"
 Dim zipName As String
 Dim fileName As String
 fileName = Dir$(logdir & "\*.*", 0)

 Do While Not (fileName = "")
     '** Essentially, the logic that we're applying is:
     '** 1.  Check the file name for the presence of today's MMDDYYY date
     '**      string. Skip if it's the same as today's date.
```

```
      '** 2.  Make sure the format of the file name is xxx-logMMDDYYYY.log. Our
      '**      quick check will be to see if the file contains the string
      '**      "-log", which should be enough for the files in this directory.
      '** 3.  If it's something that we want to archive, add it to a ZIP file
      '**      with an appropriate name (MthYear.zip)

      '** If the filename is an acceptable format, archive the file
      section = "Check log file name"
      If (Instr(fileName, "-log") > 0) And (Instr(fileName, todayString) < 1) Then
          section = "Write to the batch file"
          '** Figure out what month this file belongs in. Extrapolate the
          '** information using the knowledge that the file name is
                  '** xxx-logMMDDYYYY.log
          fileDateString$ = Mid$(fileName$, Instr(1, fileName$, "log", 5) + 3, 8)
          zipName$ = Left$(fileDateString$, 2) & "-" & Right$(fileDateString$, 4) &
".zip"
          zipStatement$ = zipExe & " " & logdir & "\" & zipName & " " & logdir & "\" &
fileName

          Print #batFileNum%, "REM - Archive " & fileName
          Print #batFileNum%, zipStatement$
          Print #batFileNum%, "DEL " & logdir & "\" & fileName
          Print #batFileNum%, ""
      End If

      '** Get the next file in the directory and continue
      fileName = Dir$()
  Loop

  '** Now that we're done, we should close the batch file and run it
  section = "Run the batch file"
  Close batFileNum%
  taskId% = Shell(batFileName$, 1)

  Exit Sub


processError:
  thisError$ = "Error " & Cstr(Err) & ": " & Error$ & ". Occurred in section " &
section & "."
  Print thisError$
  Reset
  Exit Sub

End Sub
```