January 15th, 2003 Volume 2, Issue 1

In This Issue

IBasic Monthly -Volume 2, Issue 1
Editor – Tony Jones – editor@ibasicmonthly.net
Assistant Editor – Rick Lett – mailto:rickl@ibasicmonthly.net

Submission Guidelines
If you would like to have your work considered for publication in IBasic Monthly, please forward your submission in RTF, DOC, HTML or TXT format to: submissions@ibasicmonthly.net. IBasic Monthly is published on the 15th of every month. All submissions must be in by the 8th of the month in order to be considered for the current issue.

January 15th, 2003 Volume 2, Issue 1

Editors REMarks

Hello and welcome to yet another issue of IBasic Monthly. I hope everyone had a happy and wonderful new year!

We've made it to our third issue and I'm sure many of you wondered if you would see a January issue or any future issues for that matter. And considering that I posted a message on the IBasic forums at one point stating that the publication of IBasic Monthly was suspended indefinitely, that would be a fair assessment.

When I started this magazine it was because I wanted to give something back to the community and allow another venue for you to share your thoughts and ideas with each other. That was my goal from the beginning as it is now.

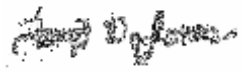
As IBasic Monthly has increased in popularity so has the time and work that it takes to bring it all together. Several forum members and I used to joke how it could become a full-time job and soon I realized that that was exactly what was happening, but I was *enjoying* it nonetheless.

I decided to post my thoughts on the issue on the forum and received lots of comments and feedback. I was entertaining thoughts of offering IBasic Monthly on a subscription basic but soon realized that was not an option. In the end, I decided that the magazine would remain free for everyone to enjoy and that the production of the magazine would be supported by user donations. I felt that this was the best option that would allow the magazine to remain available to all, while allowing those that wished and were able to support the production of IBasic Monthly.

In closing I would just like to say thank you to everyone for your comments, constructive criticism and continuing support of IBasic Monthly. I hope you enjoy reading the January issue as much as I enjoyed putting it together for you and I look forward to many, many more issues throughout the new year.

Sincerely,

Tony D. Jones



Editor – IBasic Monthly

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Developers Notes

By

Paul Turley

You may have noticed I have been pretty quiet lately. That's normal when I am busy coding away

The new IDE has many options now to make it easier to configure your perfect coding environment. Many of the changes will be incorporated into IBasic Standard as well.

The tabbed interface can be turned on and off now. Scrollbars are now standard in the IDE itself. A docking output window shows any errors in compilation and allows double clicking to bring up the file with the offending error.

Recently I received a strange, vulgar email complaining about the fact that whenever IBasic is run it adds an icon to the 'New->' context menu in explorer for creating ibasic programs. While this is part of the standard shell interface I have changed the IDE to only add the icon once. Personally I never noticed it.

UDT's in IBasic Pro are handled a little differently then in standard. strings and arrays are passed by value to functions by default now. A BYREF keyword has been incorporated to change this behaviour. Which effectively changes the element of the UDT to a pointer. This change will not require any changes to current source code if your using the UDT within your program and not sending it to an API...Otherwise if the DLL/API function expects a pointer to a string just simply add the BYREF keyword.

This change also means nested UDT's are now handled in a more compatible manner and there will be no need to read/write the udt to allocated memory first. Also since UDT's are handled in the exact same manner as C does it is now possible use API/DLL functions that return a pointer to a UDT (structure) directly.

Reguarding numeric constants Pro is very flexible. There will be some changes necessary due to the flexibility and in interest of generating fast and efficient machine code. IBasic Standard uses DOUBLE values internally for all calculations. Reguadless if the operators were both integers, UINT's what have you. While this was very easy to impliment it also results in code that is larger than necessary, and slower as well since all operands end up getting shifted to the math coprocessor.

So what does this all mean? In IBasic Standard if you had a double variable named 'd' and used this statement:

d = 1/2

d would be equal to .5 Which is OK but not technically correct since both operands are really an integer value. Consider it as int (1/2). Which equals 0. IBasic Pro introduces elevation rules and strict data representation. For example in Pro to get the same result one or both operands should be a floting point type. This is accomplished in one of two ways. By using one of the type modifiers (f, d) or by specifying a decimal 0.

d = 1/2f
d = 1/2d
d = 1.0/2
d = 1/2.0
Etcetera.

In the first example the '2' is elevated to a floating-point value. The machine code generated then returns a float number, which is converted by the coprocessor to a double.

Notice that the only time a type modifier is needed is if both operands are specified as an integer number. Generally most other math operations (*, +, -) would not need the type modifiers. However if you're in a time critical loop the compiler will generate more efficient code if all types are matched.

You will be happy to know that FOR/NEXT loops can use any integer variable type as the counter variable in Pro. Including INT, UINT, WORD, CHAR, INT64 and UINT64. Currently IBasic Standard (1.99d) can only use INT or UINT types. Some of you may not even have noticed. You can also use an array element as a counter in Pro which is a feature requested by someone.

Another change to FOR/NEXT functionality is the syntax of specifying the step value. Many of you may not even know that IBasic has a STEP keyword, it always had. The user's guide isn't real clear on this and I see most programmers use the '#' shortcut.

The dilemma arose when I couldn't convince the new parser that '#' was a shortcut to STEP and not the beginning of derferencing a pointer. So currently its been changed to '##' until I figure out some other method. Or use a sledgehammer, whichever comes first. Anyway here are the tenetive syntaxes:

IBasic Standard (current):
FOR x = 1 to 100 STEP 2
...
NEXT x

FOR x = 1 to 100 #2
...
NEXT x

IBasic Pro:
FOR x = 1 to 100 STEP 2
...
NEXT x

FOR x = 1 to 100 ## 2
...
NEXT x

It's not set in stone yet and I am still working it out.

Well thats enough for now. If I get time later tonight I may put up a screen shot of the Pro interface. Depends on how long it takes me to shovel the 3 feet of snow that fell last night

Paul Turley

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

My Adventures With IBasic

(Or how to frustrate yourself for fun!!)

(Part 3)

Strings and things

Hi gang! Guess what? I'm still here, and time to fool the editor again into thinking I know something. (Don't tell ok?).

Well here it is January and time for another article on learning IBasic. Now of course I hope your not waiting for me to tell you everything and that you have been reading the users guide and forum and getting tips from others! There are a lot of great sources to learn from and live help on the forum so use those resources!!! (And always make time to check out IBasicPowers web site, link is at Pyxia web site or check out last months issue.)

Before getting into strings and this months program I thought I would talk about something very important when programming, **commenting your code.** Thats right people you need to comment your code, because memory isn't perfect. And 6 months from now when you want to come back and make changes how are you going to know what you were thinking?

Now, IBasic allows comments in two ways, there is the old reliable " **rem** " and using the " **'** " character. Here's an example of both:

```
rem ------------------------------------------------------------------------------
rem
rem Comment you code here!!!
rem
rem ------------------------------------------------------------------------------
```

And

```
' ------------------------------------------------------------------------------
'
' Here's a nice little block of commented code here
'
'------------------------------------------------------------------------------
```

Also by using this "**:** " and this" **'** " you can comment on the same line as your code like this:

locate 5,5 : ' locates where I want to print on console screen
print " example" : ' prints the word example

Of course look at the example programs to get a feel on how others comment their code and then you can create a style of your own.

That was the thing!

Now on to strings!

First a little background on this little routine, a while back Tony Jones (yeah the editor of IBasicMonthly, I know him really!! I'm not just name-dropping) Put together a drag and drop program to move bitmaps from a file to a window to view, and I thought it was pretty neat. And, by the way, big kudos to Fidcal and his args routine which Tony used to create his program. Anyway I thought just for grins and giggles I would find a way to drag and drop any graphics file into the window for viewing and bingo! This is what I came up with. (You can find the whole thing, drag and drop and Fidcals arg routine together here: http://www.pyxia.com/community/viewtopic.php?t=4869&highlight=) just copy and paste to your browser and give a look see.

So what I decided to do is come up with a routine to select different file extensions to tell the program what image type I wanted to look at. Simple huh?

So here it is!!

```
'-------------------------------------------------------------------------
'- File select routine
'- Rick_Lett for use in drag and drop program
'- console test application
'- NOV. 2002
'-------------------------------------------------------------------------
'                                                          Line numbers
def text$:string                                            :' 1
openconsole                                                 :' 2
input "Enter File Name: ",text$                             :' 3
locate 5,10:Print "The file type is: "                      :' 4
tex =  len(text$) - instr(text$,".")                        :' 5
text$ = right$(text$,tex)                                   :' 6
select UCASE$(text$)                                        :' 7
  case "BMP"                                                :' 8
```

```
    locate 5,28:print "bitmap":'use @IMGBITMAP                    :  9
  case "JPG"                                                      :' 10
    locate 5,28:print "scalable":'use @IMGSCALABLE                : ' 11
  case "GIF"                                                      :' 12
    locate 5,28:print "scalable":'use @IMGSCALABLE                : '13
  case "ICO"                                                      :' 14
    locate 5,28:print "icon":'use @IMGICON                        :  15
  case "CUR"                                                      :' 16
    locate 5,28:print "cursor":'use @IMGCURSOR                    :  17
  default                                                         :' 18
    color 12,0:locate 5,28:print"**No file or wrong file choice was made**"  :' 19
endselect                                                         :' 20
locate 15,5
print "press any key to close"
do:until inkey$<>""                          :'Standard press any key to close
closeconsole                                 :' part of console program!
end
```

Notice that I put numbers next to the commands so that you will be able to follow it better, clever huh? And another way to use commenting, huh? huh?!
OK now the hard part, explaining how this puppy works. First of all we want to grab the image file some way,and since the name of the file is a **string**, or characters instead of numbers or **INT**egers,we have to have a way for our program to recognize them.And that is where **strings** come in.

Now keep in mind that this is only an experimental routine and will require some tweaking to actually work in a program.(see link provided above) But sometimes console apps are the best way to try out ideas and such before using in an application.
So here we are, now after the comment section you'll see at line 1 where we defined our string variable **text$** as a string then at line 2 opened the console. Next we have to have something to work with to see how our program works so we use the **input** command, since this is the first time we've seen it lets talk about it.
Input is a command we use to give the user a way to interact with a console application, and to use it is very simple, it's **input** then enter instructions or comments like a **print** command, within quotes, then", " then the variable name that the user will supply. When the program is run then

the message will be displayed and then wait for the user to type in the requested information. In this case we are looking for an image file name with extension so the program can perform its magic on it.

Line 4 is our standard desciption of our output, and now to line 5 where we talk about some string functions.

The first thing we need to do is to find a way to look at our file extension to see what it is so we can select the proper command, so to isolate it we use the **LEN** and **INSTR** function. **LEN** returns the total length of the string, like if we asked for **LEN(like) = x** then x would be 4, then I used the command **INSTR** to get the length of the string up to the ". " (Period) in our string. **INSTR** returns the **INTeger** length of the string up to character ". " from right to left.Subtracting these two amounts gives us the variable **tex** which is an **INT**eger.Whew, this gives us the location of where our extension exists in our string **text$**. Now we have to use this number to isolate our extension so we can compare it and **Select** the proper extension to display in our output. So what we want to do now is get rid of everything to the right of the ". " in our file name,so to do that we use the command **RIGHT$**. The syntax of this command is **result$ = right$(string, count)** but since we are trying to isolate the extension of the **string text$** we substitute **result$** with **text$** then we look at **text$** with the **INT**eger **tex**. So this **text$ = right$(text$,tex)** returns the string of the extension no matter what its length because it removes everything in the **string** from the right up to the value of **tex,** confused yet?(to get an idea how it works insert **print text$** under line six so that it will print out what the results of all this is then you'll get a better idea of what's going on)

Now that we got all those gymnastics out of the way we need to do something with the result, so first we want to make it case insensitive so that you can type capital letters or lowercase, to do that we use the command **UCASE$.** This converts all the letters in the **string** into capital letters and then just uses those characters to compare too in our **select/case** commands where we select the extension and then print out the correct file type.

Hey try it, run it then type in at the prompt **Bob.ico** or **larryA.bmp** or **fletchie.jpg** or **Rick.cur.**

Pretty sharp huh?

Well hope I confused you enough this month, so until February. Remember if your programming and not having fun then you aren't using **IBasic!!!!!**

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Linked Lists Made Easy

Part III - By Bizzy

In Part II - Creating a Single Linked List - designed and built the software to do just that - build a Single Linked List. We also Add UDT by Insert and Append Modes, Printed the Linked List into a Listbox and Deleted the Linked List.

In this article we will Delete an Item from the List, Sort the List in two different ways and Save to a Binary File. The Delete and Sort functions have their own set of Rules just as the Add to Linked List has its rules.

To bring the articles up to date I will list the Button Controls and what their function is:-

CREATE LIST - The Create List Button when clicked reads into the linked list a list of Records from within the program.

PRINT LIST - The Print List Button when clicked iterates through the linked list and formats the records and then adds them to the List Box Control.

DELETE LIST - The Delete List Button when clicked will iterate through the Linked List and delete all the records in the linked list. The linked list is then empty!

DELETE ITEM - The Delete Item Button is used to delete a single record from the linked list. The record deleted will be whichever one is selected in the List Box.

SORT LAST NAME - This button will sort the linked list alphabetically on the Last Name. Records are read from the BFile that has been saved and each record is inserted into the linked list in its correct position.

SORT AGE - This button works the same as the Sort Last Name button by reading each record from the BFile and placing the record in its correct position in the linked list. Sort Age not only sorts by the Age but performs a secondary sort on the Last Name when two records with the same Age are found.

SAVE FILE - The Save File button opens a BFile and then iterates through the linked list and saves it to the BFile.

USING SAVE FILE

Before you save a file the linked list must have records in it so Click Create List Button first if the linked list is empty. You can test whether its empty by clicking the Print List Button.

```
SUB SaveMainFile
        IF(OPENFILE(NameFile,GETSTARTPATH + "NAMES.DAT","W") = 0)
        node = start.nxt
        WHILE node
                WRITE NameFile,#node
                 node = #node.nxt
        ENDWHILE
       CLOSEFILE NameFile
        ENDIF
RETURN
```

1. **OPENFILE** with **'w'** to create a new file.
2. Set **node** to equal **start.nxt**.
3. Commence a **WHILE** loop to iterate through the linked list - saving each record by **WRITE** file method.
4. Set **node** to equal the next node in the linked list.
5. **CLOSEFILE**.

USING DELETE ITEM

To delete a single record from the linked list click on the record to be deleted in the List Box before clicking the Delete Item Button.

```
SUB DeleteSingleItem

DEF ref, previous:pointer
DEF SelNum:int
DEF ChosenItem, ListItem:String

' Obtain the Selected Item from the List box
' that user has chosen to delete from Linked List
SelNum = GETSELECTED (Main, 5)
IF(SelNum > -1)
        ChosenItem = GETSTRING (Main, 5, SelNum)
        ' Set ref pointer to first Address in Linked List & previous to the start address
        previous = start
        ref = start.Nxt
        IF(ref)   : ' check if Linked List is empty
                WHILE ref > 0
                        ' format the Item in linked list to compare
                        ListItem = APPEND$(#ref.FName," ",#ref.LName,", Age ",STR$(#ref.Age),", Area
Code ",STR$(#ref.AreaCode)))
                        IF(ChosenItem = ListItem)
                                #previous.Nxt = #ref.Nxt
                                DELETE ref
                                ref = 0
                        ELSE
                                ref = #ref.Nxt
                                previous = #previous.Nxt
                        ENDIF
                ENDWHILE
        ENDIF
        ' print List to Listbox
        PrintLinkedList
ENDIF
RETURN
```

1. Get the Selected Number of the Record in the List Box by using **GETSELECTED** (Main, 5)
2. **IF**(SelNum > -1) Check for -1 which says NO ITEM SELECTED IN LIST BOX
3. Obtain the text in the selected item in the List Box with **GETSTRING** (Main, 5, SelNum) into **ChosenItem**
4. We have **DEF** two pointers - **ref** and **previous** - which we will use to keep track of the iteration through the linked list.

   Set **previous** to equal **start** and set **ref** to equal the first record in the linked list - which is pointed to in **start.Nxt**.

5. **IF** (**ref**), checks to see that the linked list is NOT empty!
6. Enter a **WHILE** Loop to iterate through the linked list so we can find the record that has been selected in the List Box.
7. Format into **ListItem** the contents of the current linked list record. Use the same format that was used in the **Print List** procedure
8. Test **IF** the current record equals the selected record by comparing **ListItem** to **ChosenItem**. If not equal the **ELSE** clause is used to go to the next record in the linked list.
9. When the selected item in the variable **ChosenItem** equals the record read into **ListItem** we can then change the Links.
10. Set **#previous.Nxt** to equal **#ref.Nxt**. **Previous.Nxt** originally pointed to the current record but now points to the next record after the current record. That address was contained in **#ref.Nxt.**
11. Now the found record is deleted by **DELETE ref**
12. When the Delete has been done the **SUB** then goes to **SUB PrintLinkedList** to clear the List Box and replace its contents without the deleted record.

USING SORT LAST NAME

The Sort By Last Name Subroutine will clear the Linked List, Open the BFile on disc and read the file record by record, placing each record in to the linked list in alphabetical order according to the Last Name.

A Variable is declared ListIn of type list to store the record that is read in from the file. Two pointers are also declard, newnode (to place the new record into from the file) and previous (which keeps track of the prior node in the linked list). The previous pointer always points to ONE record prior to the node pointer. Node pointer is used to iterate through the linked list.

```
SUB SortByLName

DEF ListIn:list
DEF newnode,previous:POINTER
DEF EOLL:INT

IF(OPENFILE(NameFile,GETSTARTPATH + "NAMES.DAT","R") = 0)
        ' empty Node List
        DeleteLinkedList
        start.nxt = 0  :'Empty List
        WHILE(READ(NameFile,ListIn) = 0)
                ' Find previous record
                previous = start
                ' Set start of node list to address in start.Nxt
                node = start.Nxt
                EOLL = 1
```

```
                ' get data from first record in file
                WHILE EOLL > 0
                        IF(node > 0)
                                IF(ListIn.LName > #node.LName)
                                        node = #node.nxt
                                        previous = #previous.nxt
                                ELSE
                                        EOLL = 0
                                ENDIF
                        ELSE
                                EOLL = 0
                        ENDIF
                ENDWHILE
                ' create a newnode to contain the record
                newnode = new(list,1)
                #newnode.nxt = node
                #previous.nxt = newnode
                #newnode.FName = ListIn.FName
                #newnode.LName = ListIn.LName
                #newnode.Age = ListIn.Age
                #newnode.AreaCode = ListIn.AreaCode
                #newnode.UniqueID = ListIn.UniqueID
         ENDWHILE
         CLOSEFILE NameFile
        ' print List to Listbox
        PrintLinkedList
ENDIF
RETURN
```

1. **OPENFILE**
2. Empty the Linked List
3. Set the **start** variable to equal **0** (indicating the linked list is empty)
4. Set up a **WHILE** Loop to iterate through the file reading.
5. Set the **previous** pointer to the address at **start** and the **node** pointer to the first address in start (**start.Nxt**)
6. Set another **WHILE** Loop to iterate through the linked list to find the correct place for each record.
7. Check for the **node** pointing to a valid record in the linked list (will equal zero if empty or at end of linked list)
8. The **ListIn** variable contains the record just read from the file so it is compared to the record pointed to in the **node** pointer. **IF**(ListIn.LName > #node.LName)

Suppose the record read in by the file contained the Last Name "Schumacher". It is compared to the record in the linked list which say contains "Andretti". Schumacher is greater than Andretti so the following code is processed - which moves the **node** pointer to the next **node** in the linked list which say contains "Villeneuve" in the Last Name field. Now Schumacher

is NOT greater than Villeneuve so the program will go into the **ELSE** code and set the variable **EOLL** to equal 0 which in turn allows the **ENDWHILE** to finish.

9. After coming out of the **WHILE** loop the code then creates a new record **newnode = new(list,1)** which is pointed to by the pointer **newnode**.
10. **#newnode.Nxt** is then given the Address of the last **node** we read in the iteration through the linked list (which contained the Last Name "Villeneuve").
11. The **#previous.Nxt** points to the address of the **newnode**. Remember that **previous** points to one record before the record pointed to by **node**. And in this example that was a record containing the last name "Andretti".
12. Before this new **node** was inserted into the linked list the **previous.Nxt** pointed to the record containing "Villeneuve". It now points to the **newnode** record containing "Schumacher" which now points to "Villeneuve".
13. After all records are read in and processed the code the goes on to **CLOSEFILE**.
14. Finally the **SUB** Calls the Subroutine **PrintLinkedList** to print the linked list into the list box.

USING SORT AGE

The Sort Age Subroutine is much the same as the Sort Last Name Subroutine which is covered in the text above. There is a small difference in the code and that is because we will sort by Age primarily and then by Last Name where the Age values are equal. The code listing below is only the main area that is changed from the Sort Last Name Code.

```
SUB SortByAge
. . .
. . .
WHILE EOLL > 0
        IF(node > 0)   : ' Is linked list empty
                IF(ListIn.Age >= #node.Age)      : ' CHANGE - check if equal or greater
                        IF(ListIn.Age = #node.Age)
                                IF(ListIn.LName > #node.LName) :' If Age equal compare Last Names
                                        node = #node.nxt
                                        previous = #previous.nxt
                                ELSE
                                        EOLL = 0
                                ENDIF
                        ELSE
                                node = #node.nxt
                                previous = #previous.nxt
                        ENDIF
                ELSE
                        EOLL = 0
                ENDIF
        ELSE
```

```
                    EOLL = 0
            ENDIF
ENDWHILE
' Create newnode to contain the record
newnode = new(list,1)
. . .
. . .
RETURN
```

1. The first change is to compare the **Age** for **Greater Than** and also for **EQUAL**.
2. If the **Age** is found to be **Greater Than or Equal** another test is then made on **Age** to see if it is **EQUAL**.
3. If **Age** is found to have **EQUAL** Ages then the **Last Names** are compared - same as in the **Sort Last Name Sub**.
4. If **Age** is NOT found to be **EQUAL** then it must be **Greater Than** and the code in the **ELSE** is used, which updates the **previous** and **node** pointers to their next records respectively.
5. Once a test on **Age** or **Last Name** is found to be true then the variable **EOLL** is set to equal **0** to cause the **WHILE** loop to end.
6. The code then follows on as in the **Sort Last Name Subroutine** and creates a **newnode** and adds the record to the newnode.
7. The **Nxt** pointers are updated as well to make the linked list point to each link according to **Age** and **Last Name**.

Once a Linked List has been Sorted it can be Saved to File. The Linked List will be saved in the order that it has been sorted by. The File can be sorted again and re-saved again as often as you wish.

This article concludes now having shown the code for Deleting a Record, Sorting by Last Name, Sorting by Age and Last Name and Saving the Linked List to a BFile.

FURTHERMORE . . .

Other code you could work with for example is to have two Edit Boxes to type in the Area Code and have the code retrieve only certain records from the Linked List for display that have Area Code numbers between the two numbers in the Edit Boxes. Also another file could be opened and chosen records saved to that different file.

Furthermore another Linked List could be made to keep details about each of the Records we now have. The new Linked List could contain the **UniqueID** of the first Linked List Record as a reference. This would give you Linked List and BFile with a **One to Many Relationship** as

found in Databases. The Linked List we have now would be the **Master File** and the new Linked List with the Details would be the **Detail File**.

Even the **Detail File** could have a **UniqueID** of its own that could be used to create another linked list and BFile that would be a **Sub Detail File**. In this case the **Master File** for the **Sub Detail File** would be the **Detail File**. This would give the **Detail File** and **Sub Detail File** a **One to Many Relationship**.

---------- ***Bizzy***

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Opinion

Programming Guru's And Eclectic Thoughts

By

Rick Lett

Hi everyone and welcome to another year and another issue of IBasic Monthly!! With the new year and of course a new century, I felt it was time to express some thoughts that I have had lately, especially after reading some things on the forum and else where.

One of the things that has surprised me the most I think about reading peoples opinions about the net and web pages and stuff is the almost fanatical way that likes and dislikes are thrown about and the rampant message of "well I'm right and everyone else is wrong" kind of message that's out there. And this is from people who use various programming platforms and such and who should be on the cutting edge of the various offending type of displays and formats that exist for the commercial exploitation of the net. I mean if anyone can benefit from it, it should be the very people who can use and or abuse these things for profit.

But it seems just the opposite really and some attitudes remind me of the Luddites of a couple of centuries ago who claimed everything that could be known is and that there is no reason to continue with science any longer.

At the same time that I've been reading these opinions (in various places I should add), I've been also reading about the exponential increase in computer capability and knowledge explosion that is happening faster and faster everyday almost. I kind of find these things contradictory from what I would think would actually be the case.

It seems to me that bemoaning the fact that Applet ads and things like flash and sounds and various other annoying visual and aural displays are things that are going to be with us from now on. The net is no longer the playground of the techie crowd, it is and will continue to be an economic and commercial engine with more and more users, sophisticated or not, and these various forms of attention getting are what is going to make this engine run. People generally go for the flash and pizzazz, A boring web site is exactly that.

With out giving people a reason to stay, they won't, and they will just move on to the next web site that dares debase itself with these annoying forms of shallow entertainment, where they

will linger long enough to get the message and after being amused are more likely to buy something, or give something or learn something. And of course the person who writes these things (the traitor!!!) will charge huge Bucks to do so and clutter the web more, while making a *really* decent living.

So gee, there is every reason in the world that this evil pernicious form of web creation will go on, and here's the rub folk, **YOU CAN"T CHANGE IT ONE BIT!!!**

Now this doesn't mean that you have to embrace it either, I mean you can get ad blockers, adjust your browser, and turn off the sound. And then go your merry way. But that is it people, no amount of complaining or whining or letters to the editor will change the face of the web or the nature of the net. If I am annoyed by a site I don't go to it. I may miss out but that's my choice, kind of like watching T.V.

As for me instead of complaining about what I don't like I would just as soon do something more productive with my time, HOW ABOUT YOU.

Hope everyone has a great Year!!

The views and ideas expressed in this article are those of the author and may not represent the views of IBasic Monthly. If you have any comments or suggestions you may email the author rickl@IBasicmonthly.com. By submitting a response you give IBasic Monthly permission to publish in a letters to the editor format in a subsequent issue and to edit for space and content. IBasic Monthly is intended for G type viewing so please make all comments constructive.

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Re-using Records In A Random Access File

By

Entiretech

In the December issue of the magazine I submitted a way of indexing for random access files. One problem that could (probably would at some time) come up is that as the index is kept in memory you would lose it: windows crash, improper shutdown etc. The solution would be to read the data file one record at a time, load the key and record number into a list box for sorting and save the new index.
When using a random access file you will often have some or a lot of deletions. Usually the records are marked as deleted, Dbase and similar programs use this method and the data files can grow quite large, access slows down as “deleted” records are checked and rejected for retrieval. The file needs to be “compacted’, this is simple reading through the file, rejecting the files marked as deleted and saving the records in use to a new file. This is similar to rebuilding the index; it can be quite time consuming.
Another way of handling deletions is to record the records available and re-use them instead.

Concept
Record number one in the data file is reserved as a "control record".
One field in the record is used to track deletions if you are sure that the value of an existing field will be greater than the possible number of records, otherwise an integer field is required.
In this demo The control record starts with a value of 1 for record number, itself.
Records in use contain a zero.
The idea is quite simple and becomes obvious if you step through it.
Before entering any records the control record field points to itself (1).
A few records are added, all contain zero in the control field, an impossible record number indicating that they are in use.
When the first deletion is made, the record number of the “deleted” record is placed in the control field of record 1, the record number from the control field in record 1 is placed in the control field of the record just deleted.
With just the one deleted record, the control record (record 1) holds the address of the next available record, the next available record holds the address of the next record that will become available when it is used (in this case it will point back to the control record).
As more records are deleted a list of available records is built up. Each new deletion will contain the address (record number) of the previously deleted record; the control record will contain the address of the next available record. The diagrams should make it clear.

Record Number	Control Field
1	1
2	0
3	0
4	0
5	0
6	0
7	0
8	0

Fig.1

Several records Added, no Deletions.

Record Number	Control Field
1	5
2	0
3	0
4	0
5	1
6	0
7	0
8	0

Fig 2

Record 5, deleted. The control field now shows that record 5 is the next available record for re-use.

Record 5 shows that it is the only record available and points back to record 1.

Record Number	Control Field
1	7
2	0
3	0
4	0
5	1
6	0
7	5
8	0

Fig 3.

Record 7, deleted. The control field now shows that record 7 is the next available record for re-use.

Record 7 shows that record 5 is the next available record for re-use.

Record 5 still point back to one

We simple swap the values in the control field in record 1 (the control record) and the control field in the record that is to be deleted. So that deleting record 2 and then 4 would result in.

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Record Number	Control Field
1	2
2	7
3	0
4	0
5	1
6	0
7	5
8	0

Fig 4

And

Record Number	Control Field
1	4
2	7
3	0
4	2
5	1
6	0
7	5
8	0

Fig 5

Records have been deleted in the sequence 5,7,2,4.
The available records are “stacked” and re-use takes the form of last deleted first re-used.

When a new record is to be added we get the next available from the control record (1). In fig 5 this is 4. We use 4 to “insert” a record in the existing file. When we write the record to 4 we take the control field value in 4 (this is 2) and put it in the control field of the control record. This will bring us back to the situation in Fig 4.
Next reusable would then be 2, swapping would show that 7 would be the next available as in fig 3.
And so on until there were no more records available as in Fig 1. Adding a record at this time would mean a simple append to the end of the file.

When I posted this demo program in tips n tricks I said that I could not remember where I found the concept. While in that strange state between being awake and going to sleep (for me this is usually the period between breakfast and bedtime) the name brosco came to mind. If AAW is the same person that runs a restaurant specializing in deep fried DLLs and code soup, I am sure she will recall the name as well.

Demo Code

```
’ A method of re-using deleted records in a random access file.
’ we use a "virtual stack" to store the next available record.
’ the first record is used for control,
‘ the first record always points to the top of the "stack" of available records.
' the first deleted record leaves its record number in the control record.
' the next deleted record swaps it's own record number
‘ with the record number stored in the control record.
' and so on, the control record contains the next free record number.
' the free record holds the address of the previously deleted record or 1 if there are no others free.
' Adding records we first check the control record to find a free record, if none simply add to end of file
```

```
' If a free record exists we use that record, the record number stored in that record points to the next
' free record or to 1, in either case we copy the stored record number to the control record.
autodefine "off"
def main as window
def run as int
def retval as int
' variables used to track record for add/delete
def notify,sel as int
def sel$ as string
' Use setid to give name to controls
setid "ListInUse",1
setid "ListAvailable",2
setid "EditAdd",3
setid "EditDelete",4
setid "btnAdd",5
setid "btnCancelAdd",6
setid "btnDelete",7
setid "btnCancelDelete",8
setid "btnExit",9
setid "True",1
setid "False",0
' Define data file
def datafile as bfile
def fileLoc as string
def temp as string
fileloc = getstartpath
def DatName as string
DatName = "rec_dat.txt"
type datastruct
        def avail as int
        def info as string
endtype
def records as datastruct

window main,0,0,455,380,0x80C80080,0,"Re-Using Record Numbers",mainevents
setwindowcolor main,RGB(128,0,0)
control main,"L,ListBox1,26,25,199,201,0x50800140|@ctlistnotify,@ListInUse"
control main,"L,ListBox2,227,25,199,201,0x50800140,@ListAvailable"
control main,"E,Edit1,26,259,199,22,0x50800000,@EditAdd"
control main,"E,Edit2,227,259,199,22,0x50800000,@EditDelete"
control main,"B,Add,25,291,70,20,0x50000000,@btnAdd"
control main,"B,Cancel,95,291,70,20,0x50000000,@btnCancelAdd"
control main,"B,Delete,228,291,70,20,0x50000000,@btnDelete"
control main,"B,Cancel,299,291,70,20,0x50000000,@btnCancelDelete"
control main,"B,Exit,25,320,404,20,0x50000000,@btnExit"
control main,"T,Datafile - current  records in use,26,1,196,20,0x5000010B,100"
control main,"T,record Numbers available,232,2,195,19,0x5000010B,101"
control main,"T,Record to add,26,237,193,17,0x5000010B,102"
```

```
control main,"T,Record to be deleted,228,237,194,17,0x5000010B,103"
setcontrolcolor main,101,RGB(255,255,128),RGB(128,0,0)
setcontrolcolor main,102,RGB(255,255,128),RGB(128,0,0)
setcontrolcolor main,103,RGB(255,255,128),RGB(128,0,0)
setcontrolcolor main,100,RGB(255,255,128),RGB(128,0,0)
setfont main,"ms sans serif",10,400,0,100
setfont main,"ms sans serif",10,400,0,101
setfont main,"ms sans serif",10,400,0,102
setfont main,"ms sans serif",10,400,0,103
setfont main,"ms sans serif",10,400,0,@ListInUse
setfont main,"ms sans serif",10,400,0,@ListAvailable
setfont main,"ms sans serif",10,400,0,@EditAdd
setfont main,"ms sans serif",10,400,0,@EditDelete
setfont main,"ms sans serif",10,400,0,@btnAdd
setfont main,"ms sans serif",10,400,0,@btnCancelAdd
setfont main,"ms sans serif",10,400,0,@btnDelete
setfont main,"ms sans serif",10,400,0,@btnCancelDelete
setfont main,"ms sans serif",10,400,0,@btnExit
' definitions end - program starts

testfirstrun
LoadData
run = 1
do
wait
until run = 0
' belt and braces - the file should always be closed as soon as possible by the routine that openend it.
closefile datafile
closewindow main
end

sub mainevents
def txt as string
        select @CLASS
                case @IDCONTROL
                        select @CONTROLID
                                case   @listinuse
                                        ' notification codes in the upper two bytes of @CODE.
                                        ' Dividing retrieves the value
                                        ' the item clicked in the list is slected for deletion
                                        notify = @code / 0xFFFF
                                        if notify = @True :' singleclick
                                                sel = getselected (main,@listInUse)
                                                sel$ = getstring (main,@ListInUse,sel)
                                                setcontroltext main, @editDelete, sel$
                                        endif
                                case @btnDelete
                                        ' call the delete record routine
```

```
                                        txt = getcontroltext(main,@editdelete)
                                        if len(txt) > 0
                                                deleterecord
                                        else
                                messagebox main,"Please select something to delete","Delete record error"
                                        endif
                                        ' reset the control to empty string
                                        setcontroltext main,@editdelete,""
                                        ' update the list
                                        refreshdata
                                case @btnAdd
                                        ' call the add record routine
                                        txt = getcontroltext(main,@editadd)
                                        if len(txt) > 0
                                                addrecord
                                        else
                                        messagebox main,"Sorry nothing to add","Add record error"
                                        endif
                                        ' reset the control to empty string
                                        setcontroltext main,@editadd,""
                                        ' update the list
                                        refreshdata
                                case @btncanceldelete
                                        setcontroltext main,@editdelete,""
                                case @btnCancelAdd
                                        setcontroltext main,@editadd,""
                                case @btnExit
                                        run = 0
                        endselect
                case @IDCLOSEWINDOW
                        run = 0
                case @IDCREATE
                        centerwindow main
        endselect
        return

sub TestFirstRun
        ' Check to see if main data file exists, if not create and 'put' some dummy data
        if(openfile(datafile,fileloc+datname,"a") = 0)
        ' messagebox main,"Using existing file","Data File Exists"
        else
                if(openfile(datafile,fileloc+datname,"w") = 0)
                        'messagebox main, "Using new data file","Data File Created"
                        records.avail = 1
                        records.info = "Control Record"
                        put datafile,1,Records
                        records.info ="                    "
                        records.avail = 0
```

```
                        records.info = "Peter"
                        put datafile,2,Records
                        records.avail = 0
                        records.info = "Paul"
                        put datafile,3,Records
                        records.avail = 0
                        records.info = "Mary"
                        put datafile,4,Records
                        records.avail = 0
                        records.info = "Fred"
                        put datafile,5,Records
                else
                        messagebox main, "Unable to open data file","First run"
                endif
        endif
        closefile datafile
        return

sub loadData
' read the datafile and fill appropriate list boxes
def numrecs as int
def display as string
        ' open file for reading
        if(openfile(datafile,fileloc+datname,"r") = 0)
                ' using a for loop is quicker than testing for EOF
                numrecs =len(datafile)/len(records)
                if numrecs > 0
                        for i = 2 to numrecs
                        get datafile,i,records
                        display = "  "+records.info+"  ("+str$(records.avail)+")"
                        if (records.avail = 0) then addstring main,@ListInUse,str$(i)+display
                        if (records.avail > 0) then addstring main,@ListAvailable,str$(i)+display
                        next i
                endif
        else
                messagebox main,"unable to open index file","Loading data"
        endif
        closefile datafile
        return

sub addnew
def numrecs as int
' simply increment the number of records and "put" the new data
        if(openfile(datafile,fileloc+datname,"a") = 0)
                numrecs =len(datafile)/len(records)
                records.avail = 0
                records.info = ltrim$(getcontroltext (main,@editAdd))
                put datafile,numrecs+1,records
```

```
                ' change list to reflect the new record
                addstring main,@ListInUse,str$(numrecs+1)+"     "+records.info
        else
                messagebox main,"unable to open file","File read error"
        endif
        closefile datafile
        return

sub AddRecord
def nextrecord as int :' points to the next available "free" record
def temp as int : ' used for swaps
        if(openfile(datafile,fileloc+datname,"a") = 0)
                get datafile,1,records
                nextrecord = records.avail
                ' if pointer to next available record is 1 simply append
                if nextrecord = 1
                        closefile datafile
                        addnew
                        return
                else
                        ' we have an available record
                        get datafile,nextrecord,records
                        temp = records.avail
                        records.avail = 0
                        records.info =  ltrim$(getcontroltext (main,@editAdd))
                        put datafile,nextrecord,records
                        records.avail = temp
                        records.info = "Control Record"
                        put datafile,1,records
                        endif
        else
                messagebox main,"unable to open file","Read error"
        endif
        closefile datafile
        return

sub DeleteRecord
def nextrecord as int :' points to the next available "free" record
def currentdelete as int :' record number to be deleted, received from main program
def temp as int : ' used for swaps
        ' currentdelete is the record to delete, get value from the listbox selection
        currentdelete = val(left$(sel$,8))
        if(openfile(datafile,fileloc+datname,"a") = 0)
                ' check control record (1) for a free record
                get datafile,1,records
                nextrecord = records.avail
                temp = nextrecord
                records.avail = currentdelete
```

```
                records.info = "Control Record"
                put datafile,1,records
                records.avail = temp
                records.info = "Deleted"
                put datafile,currentdelete,records
        else
                messagebox main,"unable to open file","File open errorl"
        endif
        closefile datafile
        return

sub refreshdata
' q&d clear listboxes and update data, obviously not a good solution, but this is only a demo routine.
def NumLines, i as int
        NumLines = getstringcount(main,@listinuse)
        i = 0
        while i <= numLines
                deletestring main,@listinuse,0
                i = i + 1
        endwhile
        NumLines = getstringcount(main,@listavailable)
        i = 0
                while i <= numLines
                        deletestring main,@listavailable,0
                        i = i + 1
                endwhile
        loaddata
        return
```

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Using Arrays For Action And Option Validation

By

John P. 'Tuddypat' Sylvester

Is there an alternative to IF/ENDIF or SELECT/UNSELECT constructs?

Given that a program offers the user a set of options, it is imperative that the choice of option by the user is validated to ensure correct operation and flow of a program.

For example, when a certain state exists only certain actions are allowed, the user is not permitted to perform any action other than those permitted. An example would be of a driver of a car sat in their car in the driveway with the engine off, they are allowed to get out or start the engine. They cannot drive and for simplicity, they put cannot put it in gear.

So having started the engine, the options that the driver has are to stop the engine or drive, the engine cannot be started as it is already running, and again for simplicity, the driver cannot get out while the engine is running. Of course one may wish to allow the driver the options that were not permitted in the simplistic example as long as they followed some set of logical rules. For instance one could allow the option for the driver to exit the vehicle while in motion, but ordinarily and logically this would not be permitted.

In a program, to validate the options available one must know the state and option selected. If the option is invalid for the given state the user must either be informed or prevented from choosing an invalid option.

So before we can code, we must know what states exist and what actions are permitted for each state. To continue with the example above, Table 1 shows what States and Actions may exist for a driver. This table is by no means definitive or exhaustive.

	Get In	Get Out	Start Engine	Stop Engine	Put in Gear	Take Out of Gear	Start Moving	Stop
Car Empty								
Car Not Empty	0	1	1	0	0	0	0	0
Engine On								
Engine Off								
In Gear								
Out Of Gear								
Moving								
Stationary								

Table 1

In Table 1 above, the state is described in the left column and the actions possible are in the top row.
Using our earlier example I have included more states and actions.

So as stated before, when the driver is inside the car, the only valid actions they have are Get Out or Start Engine. So for this state we can mark the valid options. Rather than write 'VALID' we will assign a true value as '1' and instead of 'INVALID' we will use a false value as '0'
Now we can continue and complete the table.

	Get In	Get Out	Start Engine	Stop Engine	Put in Gear	Take Out of Gear	Start Moving	Stop
Car Empty	1	0	0	0	0	0	0	0
Car Not Empty	0	1	1	0	0	0	0	0
Engine On	0	0	0	1	1	0	0	0
Engine Off	0	1	1	0	0	0	0	0
In Gear	0	0	0	0	0	1	1	0
Out Of Gear	0	0	0	1	1	0	0	0
Moving	0	0	0	0	0	0	0	1
Stationary	0	1	0	1	0	1	1	0

Table 2

We shall not be getting very deep with this model. As it is, one can see there are certain ambiguities, such as 'Stationary' –

Is that before starting engine?
Is it after stopping?
Is it after starting engine?

And so on, these are the issues that need to be resolved during the planning stage.

Once we have completed the table and it looks like Table 2 above, we now have to store it in a computer friendly manner. The array type is ideal.

Array Design

The array design resembles a logic truth table, but rather than the relationships be defined by logic they are defined by the developer. We can make the array as easy or as complicated as we like. There are three basic methods each method has some drawbacks.

Method 1

This is the easy method. Each cell in the above table can be stored as an integer in a single cell of the array. This allows the state (y) and the action (x) addressing to be carried out using an integer. The drawback with this method is the vast amount of memory it will need as the matrix

grows. As the table becomes larger, the memory required obviously becomes larger and memory wastage is enormous. The wastage arises from the fact that we are only using 1 bit in every thirty-two bits. For small models this wastage is acceptable. If each cell is represented by an integer then the above table would be represented by an array defined as:

```
DEF Model[8,8]

Model[0]  = 1,0,0,0,0,0,0,0
Model[8]  = 0,1,0,1,0,0,0,1
Model[16] = 0,1,0,1,0,0,0,0
Model[24] = 0,0,1,0,0,1,0,1
Model[32] = 0,0,1,0,0,1,0,0
Model[40] = 0,0,0,0,1,0,0,1
Model[48] = 0,0,0,0,1,0,0,1
Model[56] = 0,0,0,0,0,0,1,0
```

Each action can be checked by looking at the relevant cell, if the value is '1', the action is valid. The array consists of unsigned integers. This table will require 64 integers to hold the table; each integer has 4 bytes so that is 256 bytes to hold the above array.

You will notice that the order of the initialising integers above is different to the table, the reason being that in IBASIC the array is stored in Column Order and so the assignment order needs to be made to comply with that. If you assign each element individually in the format Model[0,0] then you may follow the table row by row.

Method 2

This method is slightly harder to implement. A single bit represents each cell in the table. As each integer consists of 4 bytes (32 bits), by using an array of single integers we can reduce the amount of memory to only eight integers. This requires only thirty-two bytes to store the array. By using bits we can use an integer to represent each row of 32 bits and use only those bits required. Since we require the use of the full 32 bits in the integer we now have to ensure our integers are unsigned. The array defined would be defined as:

```
DEF  Model[8]:UINT

Model[0]=0x80000000
Model[1]=0x60000000
Model[2]=0x18000000
Model[3]=0x60000000
```

Model[4]=0x06000000
Model[5]=0x19000000
Model[6]=0x01000000
Model[7]=0x56000000

As each row exceeds the integer size we would extend the array by an integer, so if we had 33 options for each state instead of the eight we have now, we would use two integers to store each row. Therefore we would have an array of [8,2] thus allowing up to 64 options before requiring an extension.

Although we have decreased the amount of memory required to hold the matrix, we now require some mathematical decoding to locate any required cell.

Method 3

This method is the hardest of all, but the most economical in terms of memory usage.
In this method we use an array but only as much as we need to be able to fit the action array into. So to represent the table above we require a minimum of sixty-four bits, that's two integers.

DEF Model[2]:INT

With this method, memory use is at a minimum, for instance with a matrix of 1000 states and 1000 actions it would only use 125kb or an array of [31250] where the easy method would use 4Mb, an array [1000,1000]

This method, like method 2, requires some mathematical decoding, but it is a little more complex than method 2.

As an illustration, I have developed a demonstration program, the main code is in Listing 1 which is being used to help design a Stock Control program. Each item has a state and therefore that state limits what can be done with that item. The table to illustrate this matrix is shown in Table 3.

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

	Transfer	Use	Sell	Loan	Fix	Test
Serviceable	1					
Unserviceable		1	1			
Unknown				1	1	

Table 3

The states are Serviceable, Unserviceable and Unknown.

The Actions possible are Transfer, Use, Sell, Loan, Fix and Test

The logical actions for each state are as follows:

Serviceable

If the part is Serviceable then it can be transferred, used, sold or loaned, there is no sense in fixing or testing a Serviceable part.

Unserviceable

If the part is unserviceable then it can be transferred or fixed. I'll use the logic that if its unserviceable then there is no reason to test it since it is known to be unserviceable. Nor can it be used, sold or loaned.

Unknown

If the condition of the part is unknown then it can be transferred or tested.

If we were to use conventional coding to solve this problem then we would use IF or SELECT constructs. Listings 2 and 3 shows how this would normally be coded using these methods. As you see, if we used a large State/Action table there would be a myriad of IF or SELECT constructs and for the programmer it is easy to introduce bugs into such code.

By using an array for action validation, the code is reduced to a third as can be seen in Listing 4 and has less chance of a bug being introduced, however you have a price to pay. While the IF construct can be read and maybe more easily understood, the array version although simpler in its construction is harder to understand and requires more planning. However as the action validation table gets larger these problems become less problematical since the versions of code in Listing 2 and 3 must by their nature become larger. The array versions in Listing 4, 5 and 6

only increase in the size of the array and of course the initialisation code. The initialisation string obviously becomes larger but since it is a numeric representation it is by far more easily checked than a logical progression through an IF or a SELECT construct.

Decoding

We shall look at the decoding sequence for each Method.
Listings 4, 5 and 6 demonstrate each method respectively.
In each subroutine there are variables 'ma' and 'ms', these are values used to store the maximum allowed values of the Action and State, i.e. the size of the array. These values are defined globally by @MAX_ACTION and @MAX_STATE.

The array is defined and initialised with in the subroutine.

Method 1

Listing 4.

The decoding of the matrix is a simple zero based index, accessing rows (State) and columns (Action) with integer values. Relating the Action and State values, directly to the indexing of the array facilitates the indexing. In this instance the Action and State values are based on 1 and so to access the array, the value of each is reduced by 1. The resultant values are then used to access the array cell. If the value is one the action is valid and may be used. If zero the option is invalid.

The array is define as:

DEF DecisionMatrix[ms,ma]:INT

which is an array of [3,6] and initialised with the data derived from Table 3.

So the first row corresponds to the valid actions of state 'Serviceable' i.e.:

Row 1 1,1,1,1,0,0
Row 2 1,0,0,0,1,0
Row 3 1,0,0,0,0,1

So a value can be accessed by using array[state-1,action-1].

As mentioned earlier from the listing you will see that the order of 1s and 0s initialising the array are in a different order to that above, as explained IBASIC stores the array in column order and so to initialise the array by assignment the elements must be specified in column order.

In Listing 7, the array is initialised element by element in row order, this achieves the same effect.

Method 2

Listing 5.

This method uses as many bits as it can within an integer, if the number of actions increases above the maximum of 32 with in an integer then the matrix is extended by another 'column'.

Our example only requires 6 bits, which are stored at the higher end of the highest byte, the routine has been written to allow any number of actions, only limited by amount of memory or the maximum value of an integer, either of which is unlikely to occur

As in the decode for method 1, 'ms' defines our number of rows, [3] and 'ma' defines the number of columns required [6] Since each column is being represented by a bit within an integer, we divide 'ma' by 32 and add 1, this will give our minimum array column index, in this instance the answer is 1. So in this instance the array is being defined as:

```
DEF DecisionMatrix[3,1]:UINT
```

Having defined the array we now initialise it with the required values.

Looking at Table 3 we can see that the binary values required are:

```
Row 1 111100
Row 2 100010
Row 3 100001
```

Putting each of these into a 32 bit number starting at the MSB we get

```
1111 0000 0000 0000 0000 0000 0000 0000
1000 1000 0000 0000 0000 0000 0000 0000
1000 0100 0000 0000 0000 0000 0000 0000
```

which translates to:

0xF0000000
0x88000000
0x84000000

So these are the values to which DecisionMatrix is set.

The problem now is given a value of 'state' and 'action' we need to decode these values to check the cell value. The 'state' is easy since we subtract 1 and this points to the correct row of the matrix. The real problem is to translate the 'action' value into a bit number value, that is 0 for the LSB and 31 for the MSB.

Since we may be dealing with a row of more than 1 integer we need to divide the action value by 32 and ensure the result is an int. In this case it will be 0. We now need to locate the bit number.

We do this by subtracting the value of Action modulo 32, from 32.

We now generate a mask by raising 2 to the power of the bit number and 'anding' with the array value, if the value is 1 then it is valid!

Method 3

Listing 6.

Like method 2 we must calculate the size of our array. So 'ma' multiplied by 'ms' and divided by 32. The result is 0 so our array will fit into a single integer. For ease the result is stored into a variable 'StateIndex' to which one is added and that is used to define our array of integers:

```
DEF DecisionMatrix[StateIndex+1]:UINT
```

Which translated is

```
DEF DecisionMatrix[1]:UINT
```

Again we must use unsigned integers. To initialise our array we look again at Table 3 and generate binary values as we did for method 2.

Row 1 111100
Row 2 100010

Row 3 100001

Putting each of these into a 32 bit number starting at the MSB we get

1111 0000 0000 0000 0000 0000 0000 0000
1000 1000 0000 0000 0000 0000 0000 0000
1000 0100 0000 0000 0000 0000 0000 0000

However we need to compact these numbers and so we start at the MSB and enter the binary bits:

1111 0010 0010 1000 0100 0000 0000 0000

and set the unused bits to zero.

Translating this into a hex number:

0xF2284000

This is the value we set our DecisionMatrix to.

The decoding is similar to, but a little more complex than, method 2.

'StateIndex' is used to index the correct integer and BitIndex the correct bit.

To calculate the value of 'SatetIndex' we must calculate the number of bits in all rows up to our selected row and then add the number of bits in that row upto our action.
This figure is then divided by 32, ensuring the result is a proper integer, to find our integer index.

E.g state 2 action 3

((2-1) * 6) + 3 = 12
12 / 32 = 0

We are looking in integer [0] or DecisionMatrix[0].

If say we had 10 states and 6 actions possible
Our array would be [2], so if state 10 and action 4 occurred then

((10-1) * 6) + 4 = 58
58/32 = 1

So the bit to check would be in integer[1].

We now need to calculate which bit we need to check in integer[0].

So taking modulo 32 of the total number of bits and subtracting from 32 gives our required bit number. As in method 2 to we raise 2 to the power of this number and we then 'and' that with the integer under test to see if the bit is 1 and if it is the action is valid.

Although it is in some ways, more complex, the saving in time and money in tracing any bugs is reduced. Other uses are windows control validation that allows control enabling for each defined situation in a window. There is scope for a dynamic options validation where the array contents may change, however, this use presents its own problems with debugging as the true values within the array would change and it would be difficult to assess what value existed or should exist at any point. In fact any situation where you are faced with a nested or complex IF or SELECT construct, maybe simplified using this idea.

I hope this maybe of some use, or at least some interest.

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Using The Rich Edit Control

By

Paul Love

One of the IBasic projects I've spent some time developing is NoteWorx, an "information manager" which makes considerable use of the rich edit control. Since I've been forced to learn a few things about using this gadget, I thought I would show some examples of my rich edit routines (not great coding, but they serve the purpose).

First of all, you have to define a rich edit control:

```
CONTROL d1,"RE,,420,60,200,290,@CTEDITMULTI|@VSCROLL|@HSCROLL,35"
```

Then you may want to set a few properties for this control (in the @IDINITDIALOG section of the dialog handler in this case, since I'm using a dialog for these examples):

```
CONTROLCMD d1,35,@RTSETLIMITTEXT,128000
CONTROLCMD d1,35,@RTSETDEFAULTFONT,"Arial",9,0,0
CONTROLCMD d1,35,@RTSETDEFAULTCOLOR,RGB(0,0,0)
```

This will set the limit for the number of characters that can be keyed or loaded into the rich edit control, as well as establishing the default font (Arial 9 point) and default text color (black).

If you want to be able to check for mouse clicks or specific key presses inside the rich edit control, you'll also need to define some user constants and a user defined type to catch those events:

```
'Trap keyboard and mouse events
SETID "ENMKEYEVENTS",0x10000
SETID "ENMOUSEEVENTS",0x20000
SETID "ENMSGFILTER",0x700



TYPE MSGFILTER
```

```
    def hwndFrom:INT
    def idFrom:INT
    def code:INT
    def msg:INT
    def wparam:INT
    def lparam:INT
  ENDTYPE
  DEF mf:MSGFILTER
  DEF mem1:MEMORY
```

You'll also need to add a couple of statements to the @IDINITDIALOG section to tell IBasic to watch for keyboard and mouse events in the rich edit control:

```
CONTROLCMD d1,35,@RTSETEVENTMASK,@ENMKEYEVENTS
CONTROLCMD d1,35,@RTSETEVENTMASK,@ENMMOUSEEVENTS
```

Then in the dialog handler you can check @NOTIFYCODE to see if you got an @ENMSGFILTER event and if so, execute whatever code you want to use to respond to that particular mouse click or key press. In the example below, a right click with the mouse (within the rich edit control) will pop up a context menu, allowing the user to "select all", "copy" or "paste":

```
case @IDCONTROL
   select @CONTROLID
     case 35: ' RE control
       if @NOTIFYCODE = @ENMSGFILTER
         mem1 = @QUAL
         READMEM mem1,1,mf
         select mf.msg
           case @IDRBUTTONUP
             mx = mf.lparam&0xffff
             my = mf.lparam/0x10000
             CONTEXTMENU d1,mx+420,my+60,"I,Select All,0,21","I,Copy,0,22","I,Paste,0,23"
         endselect
       endif
```

Of course the first step in using a rich edit control is usually to load some text (plain or rich format) into the control.

Here's an example of loading a text file, which can be either plain or rich text:

```
SUB loadrich
```

```
    rtflag = 0
    setfocus d1,35
    setcontroltext d1,35,""
    if(openfile(refile,filename$,"R") = 0)
        ret = CONTROLCMD (d1,35,@RTLOAD,refile,1)
        if ret = 0 then rtflag = 1
        closefile refile
    endif
    if ret <> 0
        if(openfile(refile,filename$,"R") = 0)
          ret = CONTROLCMD (d1,35,@RTLOAD,refile,0)
          closefile refile
        endif
    endif
    if ret <> 0 then messagebox d1,"Error loading file",""
RETURN
```

In the routine above, the focus is set to the rich edit control, and then the control itself is cleared. Next the file containing the text to be loaded is opened and read as a rich text file (since the last parameter in the CONTROLCMD is a "1").

If the value returned is zero (the file loaded OK), then rtflag is set to 1 to indicate that the file is in rich format. If ret is not zero, then the file failed to load, so we open it again and try reading it as a plain text file (note that the last parameter in the CONTROLCMD is a "0"). If the result is still not zero, then an error message is displayed.

Saving the contents of the control to a file involves a similar operation:

```
if rtflag = 1
    if(openfile(refile,filename$,"W") = 0)
      ret = CONTROLCMD (d1,35,@RTSAVE,refile,1)
      closefile refile
    endif
  if ret <> 0 then messagebox d1,"Error saving file",""
endif
```

You may also want to add the ability to do a "find" and "find next" function, so the user can search the contents of the rich edit control for specific text. Below is a "simple" example showing one way to go about adding a search function:

```
case 24: ' find
    answer3 = domodal d3
    if answer3 = @IDOK
      first1 = 0
      gosub findtext
    endif
```

```
      case 25: ' find next
         gosub findtext
```

Assume case 24 and 25 are menu selections, and dialog d3 is a simple dialog that allows the user to enter a text string (ftext$) to search for. The "findtext" routine would look something like this:

```
SUB findtext
' find search text
     setfocus d1,35
     fpos2 = CONTROLCMD (d1,35,@RTFINDTEXT,ftext$,first1,1)
     if fpos2 <> -1
        match = match + 1
        first1 = fpos2 + 1
        CONTROLCMD d1,35,@RTSETSELECTION,fpos2,fpos2+len(ftext$)
        fline = CONTROLCMD (d1,35,@RTGETFIRSTLINE)
        sline = CONTROLCMD (d1,35,@RTLINEFROMCHAR,fpos2)
        lines = sline - fline
        if lines < 3 then lines = 3
       CONTROLCMD d1,35,@RTSCROLL,lines-3,0
     else
        messagebox d1,"No match found for "+ftext$,""
     endif
RETURN
```

First, the focus is set to the rich edit control, then we do an @RTFINDTEXT command to look for the string ftext$ (note that the search is not case sensitive in this example since the last parameter of the CONTROLCMD is a "1"). If the result, fpos2, is not -1 then a match was found and we highlight the matching text and scroll down to it -- otherwise, an error message is displayed. Notice that the @RTFINDTEXT started from position "first1" in the rich edit control, and that first1 is set to zero to start with so the search starts at position zero -- if a match is found, first1 is set to one position past the matching text, so the next search starts at that point.

If you're a glutton for punishment, there are a lot of other things you can do with the rich edit control. For example, you can get a (very close) approximation of the word count for the text loaded in the rich edit control by using a routine like this:

```
Def wordcnt,txtlen,i,wcount:INT
Def prevchar$:STRING
Def tbuffer[32766]:ISTRING

SUB wordcnt
  txtlen  = CONTROLCMD (d1,35,@RTGETTEXTLENGTH)
  if txtlen < 32766
```

```
        setfocus d1,35
        CONTROLCMD d1,35,@RTSETSELECTION,0,-1:  ' select all text in the rich edit control
        tbuffer = CONTROLCMD (d1,35,@RTGETSELTEXT):  ' store the selected text in tbuffer
        CONTROLCMD d1,35,@RTHIDESEL,1:  '  hide the selection
        for i = 1 to txtlen
          if (mid$(tbuffer,i,1) = " ") & (prevchar$ <> " ")
            wcount = wcount + 1
          endif
         prevchar$ = mid$(tbuffer,i,1)
       next i
      wcount = wcount + 1
    endif
    CONTROLCMD d1,35,@RTSETSELECTION,-1,-1
    CONTROLCMD d1,35,@RTHIDESEL,0
    messagebox d1,"Word count = "+str$(wcount),""
  RETURN
```

That's probably more than you wanted to know about the rich edit control, so we'll stop here.... there's a lot of other information available on the forum (especially tips on loading and saving from and to an ISTRING variable).

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

SQL Structured Query Language

By

EntireTech

SQL was designed for manipulating relational databases. It is a relational Data Manipulation Language (DML). There are other approaches Quel which is very similar and used in a Database Management System (DBMS) called Ingres, Query By Example used by Borland Paradox, MS ACCESS also uses a form of QBE, Access is worth looking at particularly because it uses the easy to understand graphical QBE and translates this into SQL, so that looking at some of their standard databases and the tables involved you can see how the queries are made graphically and then examine the SQL code produced.
SQL is a transform-oriented language; it uses relations to transform inputs into a desired output. SQL is more than DML it also has the ability to define data, to describe the structure of the database; it is a Data Definition Language (DDL).

Databases
Before looking at SQL, usually pronounced sequel, it is useful to look at databases in general. It is likely that database management began with the APOLLO project in the 1960's. Up until then data management with computers was essentially a set of flat files, lists of information held by different departments in a company, for example, Payroll would have a list; Personnel would have a list; Production would have a list; Sales would have list and so on. There was a lot of information but no way of correlating it. APOLLO needed a way of coordinating the huge amounts of data involved in the mission. IBM developed the system, noted that it had a much wider use and developed it further into "Information Management System" (IMS) that is still used today. The early forms where viewed as a network of information or a hierarchy of information. The methods involve an explicit definition of the relationships between different record types, asking for information about Fred would involve following a set of defined pointers to get the complete information from the set of record types. The systems were very quick but not very versatile, requirements needed to be complete and clearly defined.

Relational Databases
In 1970 Dr. E. F. Codd proposed a new, radically different approach to data management; the relational model. In this model:

1. The logical and physical characteristics are separated.
2. The model is a more easily understood, no complex path to follow; it appears more "natural".

3. There are powerful operators available, complex operations can be accomplished with brief commands.
4. The model gives a solid foundation for database design.

Terminology.
A relation is a two dimensional table with these properties:

- All entries have a single value
- Each field has a distinct name, the attribute name.
- All values in a field belong to the same attribute.
- The ordering of fields is immaterial.
- Each record is distinct.
- The order of the rows is immaterial.

The formal terms are: relation, tuple, attribute.
Common alternatives are table, row, column, or, file, record, field. The last three are probably closest to the names we use mostly in IBasic.

A relational database is a collection of relations.

I will use the expressions table, record and field.

Each record has a field that is the key to the record, if we are certain that each value in the field will be unique this field can be the key, if not an additional filed will be required that will ensure that "each record is distinct" (property 5). This key is called the Primary Key.

Another key field of extreme importance is the Foreign Key. The foreign key is a field that has the same value as the primary key in another table, it is used to match the primary key in another table and establish the relationship between them. This allows a query such as – get me all the details about Fred (whose unique identifier is 999 in table A) from tables B and C where the foreign key is 999 and put them in a new (temporary) table called FredDetails.

A database holds information about many different types of record, and the relationships between them. The two most common relationships are *one-to-many* and *many-to-many*. The most common example used (probably in 99% of all schools) is the Invoice/Purchase order.

An order is allocated a unique number, that order may have several lines each for an individual item.

The relationship between the Order and the Order-lines is *one-to-many,* that is, an Order may have many lines, but each Order-line refers to only one order. The order-line table

will have a foreign key that defines its relationship to the order number (primary key) of the order table.

Each item will have details stored in a separate table, say, Products, possibly with a primary key of ItemNumber.

The relationship between Order and Products is *many-to-many.* Each item may be included in many orders and each order may contain many items.

One-to-many

Suppose a company has a rule that only one sales person represents each customer but each salesperson represents many customers. We would have two tables one with salesperson details and another with customer's details.

Sales Person Table

SP Number	SP Name
3	Fred
6	Joe
7	Mary
12	June
13	James

Primary Key is SP Number

Customer Table

Customer Number	Customer Name	SP Number
321	Mr. Smith	3
444	Ms. Jones	3
345	Ms. Maria	7
535	Mr. Clark	12
536	Mr. Smith	12

Foreign Key is SP Number.

Each Customer appears in only one record of the Customer Table, this record contains a single sales person number, each customer is associated with exactly one sales person.

Note that the sales person table contains no customer number field, the only way to find the customer(s) for a given sales person is to find those records in the customer table that contain the sales persons number.

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

A pseudo code query would be: find all Fred's customers. This would return Mr. Smith and Ms. Jones.

Note: The use of Customer Number clearly distinguishes between the two Mr. Smiths. It is usual to underline the primary key.

Many-to-many

Suppose that each order contains lines, and each item is found on lines in many different orders. We need a table for the Orders and a table for the Items (Products). The many-to-many relationship shows up in the third table. In this table the Primary Key is the combination of Order Number and Item. This is called a concatenated key.

Order Table

Order Number	Order date
12567	20020103
12568	20020103
12569	20020104
12570	20020105
12571	20020106

Primary Key is Order Number

Product Table

Item	Description
Ab201	Steel widget
Ab202	Brass widget
Cc001	Wooden thing
De031	Copper doodad
Xz001	Steel gadget

Primary Key is Item

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Order-line Table
~~~~~~~~~~~~~~~~~~~~

Order Number	Item	Number Ordered	Price
12567	Ab201	10	10.05
12569	Ab201	10	10.05
12569	Ab202	5	12.00
12569	De031	12	9.85
12570	De031	6	9.85

Primary Key is the concatenation of Order Number and Item

To find the details for a given order number we need to take the order number and the Item as parameters for a query which could look something like:
Find me details for order number 12569, this would return 3 records from the Order-line table to complete the query you would then retrieve the details for the Items. This would return a (temporary) table that would look something like:

12569	20020103	10	Steel widget	$10.05
12569	20020103	5	Brass widget	$12.00
12569	20020103	12	Copper doodad	$9.85

Depending on the formatting and other parameters to the query, this could be sorted by Item or price.

Another important use for primary and foreign keys is for maintaining the integrity of the database.
If a salesperson were to leave the company you should not simply delete his details while he still had customers allocated to him/her.
You cannot add an item to the order unless it is already entered in the Product file; similarly you should not delete an item from the Product file while there were still order-lines referencing that item.
As part of your program you will need to allow for these and similar potential anomalies by implementing what is known as a “Cascading delete” or a method of ensuring “referential integrity”.

Normal forms
You need to structure your data before you can make a structured query. The method used is called normalization. First looking at the definition of a table (relation)
~~~~~~~~~~~~~~~~~~~~

- All entries have a single value
  While it is possible to join several bits of data together and store it in one field it is not permitted in this tight definition. Not only that, it makes searching incredibly complicated.
- Each field has a distinct name, the attribute name.
  Fairly obvious, we have to have some way of identifying the area we are going to search. It is usually possible to use either the name or the field order. Obviously names must be unique to the table, it is often the case that people will use the same name in multiple tables where these fields are primary key fields and foreign key fields. This is not required but can make the logic easier to follow. To distinguish between these common names dot notation is usually used as for a UDT.
  Say, main_table.userID, and to keep it clear, detail_table.userID
- All values in a field belong to the same attribute.
  If a string field is called 'LastName' it would break the rules and commonsense to put, say, a company name in that field .
- The ordering of fields is immaterial.
  This is generally true, but some implementations expect that the first field is the key field.
- Each record is distinct.
  Duplicate records are not permitted; this is often one of the functions of the primary key, as each key is unique, each record in the table will different from every other record.
- The order of the records is immaterial.
  It is the job of the query engine to find each record, field combination as needed. This is implied by the first point in the description of the relational model:
  "The logical and physical characteristics are separated. "
  That is it should not matter to the programmer or the user if the indexing and searching methods are changed, isam, btree, hashing all return the records and fields asked for by the query.

Designing tables.

A critical concept is designing a relational database is Functional Dependence. This is complicated name for a fairly simple concept. If the sales person file had an extra field called pay class, and the commission received on sales was calculated according the pay class of the sales person, it could be said that – the remuneration is a function of the percentage commission rate, or, in plain English, how much he gets depends on his commission rate. A formal definition is:

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

> An attribute (field), B, is functionally dependent on another attribute, A (possibly a collection of attributes), if a value for A determines a single value for B at any one time.

Given a value for A, do we know that we can find a single value for B ? if so, then B is functionally dependent on A (often written as A→B). In this case we can also say that A functionally determines B.
In the Customer table the customer Name is functionally dependent on Customer Number, Customer number 321 will gives us Mr.Smith. is the SP Number functionally dependent on Customer Name – no – because Mr.Smith would return two values. The same would apply if the table snippet would more realistic and included addresses, phone/fax numbers etc.

Customer Table

Customer Number	Customer Name	SP Number
321	Mr. Smith	3
444	Ms. Jones	3
345	Ms. Maria	7
535	Mr. Clark	12
536	Mr. Smith	12

Customer Name is functionally dependent on Customer number.

SP Number is not dependent on Customer Name.

Functional dependence is also determined by the policy or business rules applied to the database as a whole. Here it is true that Customer Number → SP Number because the Company policy is that each customer will be serviced by one and only one sales person, if a customer was serviced by whoever was available it would not be true, because there is no way of telling from the customer number who the sales person was.

The second concept underlying the relational model is the Primary Key. It builds on functional dependence, and completes the framework required to understand Normalization.

> Attribute (field) A (or a collection of fields – a concatenated key) is the Primary Key for a table T, if:

1. All fields in T are functionally dependent on A.
2. No subcollection of the fields in A (assuming A is a concatenated key and not just a single field) also has property 1.
3.

Customer name is the primary key for the customer table, because all fields are functionally dependent on customer name.
In the order-line table neither order number nor item can be the primary key, because neither can completely define the information required. Only a combination of the two will do this for us.

It is possible that there is more than one possible primary key, these are called a candidate keys. From all possible keys we need to select the most suitable for the job in hand as the primary key.

Normalization gives a method of analyzing a relational database and check for potential problems, called update anomalies. Like all programming it is an iterative process, start with a broad outline and work down to the details (we all do that don't we? at least every book I have read says we should).

First normal form

A table is in 1st normal form if it does not contain any repeating groups.

Orders (OrdNumb, OrdDate, PartNumb, NumbOrd)

This notation shows a table called Orders, with a primary key OrdNumb underlined and a field OrdDate, with repeating fields PartNum and NumbOrd. The repeating group would be indicated with a line over the group.

To convert this to 1st normal form (1NF) the repeating group is removed giving

Orders (OrdNumb, OrdDate, PartNumb, NumbOrd)

This could look something like this:

Orders

OrdNumb	OrdDate	PartNumb	NumbOrd
1499	20020102	AB123	12
1500	20020103	AB123	6
1500	20020103	AX321	8
1500	20020103	BD222	45
1501	20020103	AB123	6
1501	20020103	XA654	9
1502	20020104	AN324	10

The primary key in un-normalized form was simply OrdNumb which would have had 1500 as the OrdNumb but three items each stored in PartNumb and NumbOrd.

1st normal form gives this table with three distinct lines: primary key OrdNumb and PartNumb as the key to the repeating group. Giving a combined key

Second normal form
If the record had full details required for each line in the order it would look like this:

Orders (OrdNumb, OrdDate, PartNumb, PartDesc, NumbOrd, Price)

This design contains update anomalies; there are four types of anomaly.

1. Update. A change in PartDescription requires many changes, each row with a given PartNumb must be changed. The update procedure becomes cumbersome, complicated and time consuming.
2. Inconsistent Data. Nothing prevents a PartNumb from having more than one description.
3. Additions. This creates real problems. The primary key is both OrdNumb and PartNumb. Values for both are needed. We cannot add a part unless there is already an order for it, the only solution would be to make a dummy order, add the part and then replace the dummy with a real order number when you had one – not really acceptable.
4. Deletions. If there were only one order with a given part, details of that part will be lost.

These problems arise from the fact that we have fields that are dependent on partial keys and not the complete combined key. 2NF improves on 1NF by removing these anomalies.

A field is a non-key field if it is not part of the primary key.

A table is in second normal form (2NF) if it is in 1NF and no non-key field is dependent on only a part of the primary key.

The method for removing these anomalies uses the concept of functional dependency.
The table is 1NF with the following functional dependencies

OrdNumb → OrdDate
PartNumb → PartDesc
OrdNumb,PartNumb → NumbOrd,Price

First, for each subset of the set of fields that make up the primary key, begin a table with this subset, as it's primary key.

(OrdNumb,
(PartNumb,
(OrdNumb, PartNumb,

Next, add each of the appropriate fields to it's corresponding key, that is, place each one with the minimal collection on which it depends, and give each table a suitable name, giving:

Orders(OrdNumb,OrdDate)

Parts(PartNumb,PartDesc)
OrdLine(OrdNumb, PartNumb, NumbOrd, Price)

This is the same as the description of a many-to-many relationship, using a formal notation. Update anomalies have been removed:

- Changing the description is only done once.
- A description appears once, no redundancy.
- A new part and its description can be added to the parts table without an order being placed.
- Deleting an order does not lose the description of the part(s) in the order.

Third normal form.
There can still be problems with tables in 2NF. Looking at the complete information for customers and the salesperson.

Customer(CustNumb, CustName, CustAddr, SalsRepNumb, SalesRepName)

With the functional dependencies:
CustNumb → CustName, CustAddr, SalsRepNumb, SalesRepName
SalesRepNumb → SalesRepName

CustNumb determines all the other fields, additionally, SalesRepNumb determines SalesRepName.

If the primary key is in a single field no field will depend on a part of the key, there is no combined key so that the Customer table is in 2NF.
Similar update anomalies exist, even though the table is in 2NF:

1. Update. A change in the name of a sales rep requires several changes.
2. Inconsistent Data. There is nothing to prevent the Sales rep from having two or more names.
3. Additions. You cannot add a new sales rep unless you have a customer for him/her. As before you would need to add a dummy customer, again, not really acceptable.
4. Deletions. If all customers for sales rep 6 all information for sales rep 6 would also be deleted.

> The update anomalies come from the fact that SalesRepNumb determines for SalesRepName, but it is not the primary key.

2NF is an improvement on 1NF but can bring it's own problems. Third normal form (3NF) gives a method of eliminating the anomalies. A new term needs to be introduced, a name given to the field that determines another field. Any field (or collection of fields) that determines another field is called a determinant.

The primary key will be a determinant, and any candidate key will also be a determinant (a candidate key is one that could have chosen as the primary key). SalesRepNumb is a determinant but it is not the primary key, and that is the problem.

> A table is in third normal form (3NF) if it is in 2NF and if the only determinants it contains are candidate keys.

Note: this is not the original definition, but is a later, preferable, definition, called Boyce-Codd normal form (BCNF).

The method for resolution of this problem is:
First - for each determinant that is not a candidate key, remove the fields from the table that depend on this determinant.

Next – create a new table with all the fields from the original table that depend on this determinant.

Last – make the determinant the primary key of the new table.

So, that:

Customers(CustNumb, CustName, CustAddr, SalsRepNumb, SalesRepName)

Becomes:

Customers(CustNumb, CustName, CustAddr, SalesRepNumb
SalesReps(SalesRepNumb, SalesRepName)

This is the same as the description of a one-to-many relationship, using a formal notation.
Update anomalies have been removed:

- Changing the SalesRep is only done once.
- A SalesRep appears once, no redundancy.
- A new SalesRep can be added to the without being first allocated a Customer.
- Deleting customers will not cause any loss of SalesRep information.

There are higher normal forms, but for most work this should cover the requirements.

To complete the simple orders database, all that is needed is to relate the customer to the order that is to be processed. As in:

Orders(OrdNumb,OrdDate, CustNumb)

From this we can retrieve all the information we need using the tables:

Customers (CustNumb, CustName, CustAddr, SalesRepNumb)
SalesReps (SalesRepNumb, SalesRepName)
Orders (OrdNumb, OrdDate, CustNumb)
OrdLine (OrdNumb, PartNumb, NumbOrd, Price)
Parts (PartNumb, PartDesc)

To query orders by customer:
For a given customer retrieve all from *Orders* with a matching CustNumb. We could (sort the order by date, get the orders before or after a given date or between dates)

For each row retrieved, retrieve all from *OrdLine* with a matching OrdNumb.
Add other details- who sold the order from SalesRepNumb - description of the part ordered from PartNumb

In a real application there would be a lot more information stored, but the principal would remain the same. For example the price would likely be derived from the parts table, you may have a discount for some customers: in either case you would only need to change the information in one place.

This proved to be a “bit” longer than intended, but I do think that the elements of relational database design are needed before you can use the full power of SQL. SQL is, after all, the most widely used means of manipulating data in a relation database. What we are going to need next is some kind person to find a grid control so that we can show the results, and, perhaps, with any luck, a report writer.

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Inside The Windows API

How to use the GetUserNameA & GetComputerNameA API call to get the PC Name & User Name.

By

Matt "Lucifer" Cox

STEP 1 - You need to add some variables to the top of your code within the same location as all the rest of your declarations.

```
DEF strUser,strName,a$:STRING
DEF nSize,a,b:INT
DEF pSize:POINTER
```

STEP 2 - You need to declare the **GetUserNameA** and **GetComputerNameA** API calls to the top of your code within the same location as all the rest of your declarations.

```
DECLARE "kernel32",GetComputerNameA(buffer:STRING,size:POINTER),INT
DECLARE "advapi32",GetUserNameA(buffer:STRING,size:POINTER),INT
```

STEP 3 - The pointer **pSize** will be used to send the maximum string length for the return value from either of the API calls used here. The pointer needs to be pointed to something so we'll point it to **nSize**. **nSize** will contain the value for the maximum string length.

```
pSize = nSize
```

STEP 4 - Now to make our first call. Let's call the **GetUserNameA**.

```
nSize = 255
GetUserNameA(strUser,pSize)
a = nSize
```

nSize has been given the value of **255** and that means we can receive a maximum string length of **255** characters.
When we call the **GetUserNameA** the returned string will be put into **strUser** and have a maximum of **255** characters.
Now, you may remember we pointed **pSize** to **nSize**. After we call **GetUserNameA** the number of characters copied to **strUser** is placed in **nSize**. You may notice that this value is one more then the actual number of visible characters in the string. This is caused by a termination

character with the value of **0** (zero) also being copied. If this call fails, the return value will be **0** (zero). You could add:

```
IF GetUserNameA(strUser,pSize) = 0 THEN MESSAGEBOX(0,"Can't get User Name","ERROR").
```

STEP 5 - Now to make our second and final call. Let's call the **GetComputerNameA**.

```
nSize = 255
GetComputerNameA(strName,pSize)
b = nSize
```

nSize has been given the value of **255** and that means we can receive a maximum string length of **255** characters. We have to set this value again because the return value from the G**etUserNameA** affects **nSize**. This would produce an incorrect value should the string length of **GetUserNameA** be less then the string value of **GetComputerNameA.**
When we call the **GetComputerNameA** the returned string will be put into **strName** and have a maximum of **255** characters.
As I said before, we pointed **pSize** to **nSize**. After we call **GetComputerNameA** the number of characters copied to **strName** is placed in **nSize**. You may notice that this value is the actual number of visible characters in the string. This is because the termination character is not copied with this call.
If this call fails, the return value will be 0 (zero). You could add:

```
IF GetComputerNameA(strName,pSize) = 0 THEN MESSAGEBOX(0,"Can't get Computer
Name","ERROR").
```

Lets put a small demo together to get it working.

Example –

```
DEF strUser,strName,a$:STRING
DEF nSize,a,b:INT
DEF pSize:POINTER

DECLARE "kernel32",GetComputerNameA(buffer:STRING,size:POINTER),INT
DECLARE "advapi32",GetUserNameA(buffer:STRING,size:POINTER),INT

pSize = nSize

OPENCONSOLE

nSize = 255
GetUserNameA(strUser,pSize)
```

```
a = nSize

nSize = 255
GetComputerNameA(strName,pSize)
b = nSize

PRINT "COMPUTER NAME AND USER NAME"
PRINT "============================"
PRINT
PRINT "User Name              : "+strUser
PRINT "User Name length is    :"+STR$(a)
PRINT
PRINT "Computer Name          : "+strName
PRINT "Computer Name length is :"+STR$(b)
PRINT
INPUT "Press ENTER to exit",a$
CLOSECONSOLE

END
```

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

Freeware Showcase

Welcome to the latest addition of IBasic Monthly, the Freeware Showcase. Here is where we will feature freeware written in IBasic by you, our readers. So, if you have a piece of shareware written in IBasic and want to get the word out, why not consider advertising it in IBasic Monthly? After all, it's free!

NoteWorx

(Powered by IBasic - www.pyxia.com)

NoteWorx is a tool to build and manage various types of information, ranging from plain and rich text notes to web pages to electronic forms and database files. For free-form data, NoteWorx functions somewhat like an outline where you set up topics, with one or more "items" linked to each topic -- and each item can be linked to a detailed "note". For database information, such as an inventory file or a contact list, NoteWorx provides an integrated "Data Manager" module. And if you're looking for a way to handle electronic forms, you can build HTML "forms" (using NoteWorx's built-in Web Page Editor and browser) to create and maintain collections of documents, such as invoices or sales orders. For additional flexibility, NoteWorx also includes a built-in scripting language, which can be used to do things like creating a "slide show".

(Available for download at: http://www.geocities.com/plworx**)**

```
With NoteWorx you can:

 *  Enter, import, attach or paste notes linked to a particular item
 *  Attach notes in plain, rich text, HTML, Microsoft Word or pdf format
 *  Include special formatting, graphics, video and sound in your notes
 *  View notes in full-screen mode
 *  Print notes or export them to a text file
 *  Place a note on the desktop
 *  Separate different groups of topics into "books".
 *  Search all topic/item records in the current book for specific text
 *  Bookmark a place within a note for instant access
 *  Enter or import HTML code and preview it in the built-in browser
 *  Surf the Internet with NoteWorx's browser
 *  Use HTML forms to enter/edit/print data
 *  Enter/edit/search structured data in database mode
```

IBasic Monthly
Dedicated to everything IBasic!

January 15th, 2003 Volume 2, Issue 1

```
*  Automate certain functions using the built-in scripting language
*  Filter notes ("datamining")
*  Encrypt/Decrypt plain text notes to ensure privacy
*  View and edit other text files on your system
*  Do screen captures
*  Launch other applications such as Microsoft Paint and Media Player
*  Display word counts
*  And more....
```

Screen shot: