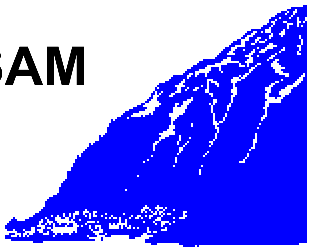
File Percentages with ISAM

(or any other file system)



With Listviews, there is just about no way to avoid using percentages to tell the listview just how far into the file the current record sits. This is what Microsoft Listviews use to adjust the size and position of the thumb, the slider in the scroll bar.

The problem is the ISAM files or relative record files do not provide percentages. Some database packages do, and some don't. Those that do may not have the flexibility you desire.

This paper describes a general purpose method of obtaining File Percentages with ISAM files for the COBOL world, without having to read the entire file to count re-cords.

Basic Concept

The theory is this:

By obtaining the key of the first record in the file (or subset of records-of-interest), and obtaining the key of the last such record, you can compute a Range of Keys.

The Range of Keys allows you to calculate where, within this range, any given re-cord resides.

This Range of Keys, and the the position of any given record, can be normalized over the range of zero to one hundred with a degree of precision sufficient for the needs of a listview control.

Method of Operation

The basic method has to do with COBOL Data Types, and redefinition of data elements.

If you define a large COMPUTATIONAL data element subordinate to a ALPHANUMERIC element you can obtain a **numeric** representation of **alphanumeric** data.

Lets suppose we have a file with 8 byte alphanumeric keys. Lets call them account numbers. They might range from 00000000 through ZZZZZZZ and can contain letters or numbers in any position.

If the **first** key on file was 0873RT45, and the **last** key was Z273AB12, it would be difficult to determine just what percentage record 5FG78343 would be within this range.

However if we were able to convert each to a number, we could calculate the range, and the percentage within that range via simple math.

Obtaining Numbers from Letters.

There might be elaborate methods of assigning each letter a value, and scanning the keys, converting and adding as you go. This would be slow and inefficient.

Our method relies on data representation in the COBOL world, and the fact that the same data can be seen using different PICTURE clauses and the conversion costs nothing. Its instantaneous, and requires no computational logic to do the conversion.

Let us assume we have the following WORKING STORAGE data structure defined:

```
01  CONVERT-KEY-X.
    05 CONVERT-KEY                      PIC 9(3)V9(15) COMP-4.
```

The Comp-4 numeric definition of CONVERT-KEY defines 8 bytes. Therefore, the size of CONVERT-KEY-X is also 8 bytes. (For a discussion on why we chose that particular picture clause see Keys Size and Distribution on page 9).

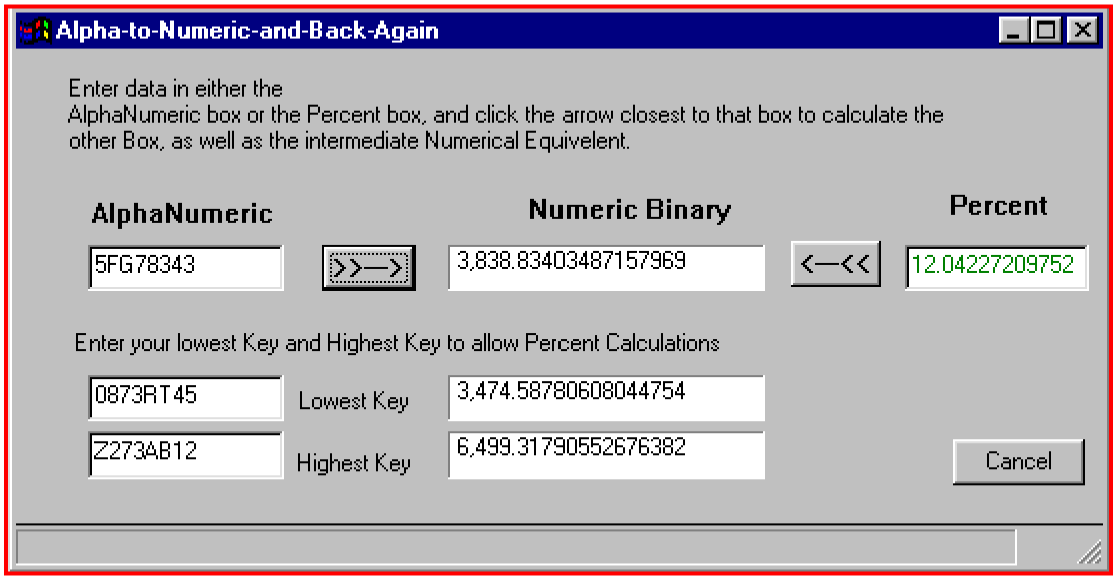
If you move 8 alphanumeric bytes into CONVERT-KEY-X, the numeric value of CONVERT-KEY is instantly changed. This change is dictated by the encoding sequence used by the computer[1].

1 We assume ASCII in this document. Of course this is also the normal Collating Sequence for ISAM

Thus, the value in in CONVERT-KEY is a numeric value that represents the key. A key near the beginning of the file will have a lower value, a key near the end of the file will have a higher value, based strictly on the collating sequence.

We have created a calculation program[2] that can be used to experiment with these calculations, and we use it here to show the example values mentioned above.

In the lower two rows, you see the Lowest Key (0873RT45) and the Highest Key (Z273AB12), and the numerical equivalent of each.



Alpha-to-Numeric-and-Back-Again

Enter data in either the
AlphaNumeric box or the Percent box, and click the arrow closest to that box to calculate the other Box, as well as the intermediate Numerical Equivelent.

AlphaNumeric		Numeric Binary		Percent
5FG78343	>>—>	3,838.83403487157969	<—<<	12.04227209752

Enter your lowest Key and Highest Key to allow Percent Calculations

0873RT45	Lowest Key	3,474.58780608044754
Z273AB12	Highest Key	6,499.31790552676382

In the upper row, you see a key from the file (5FG78343), and its numerical equivalent, and the Percentage location within the file where this key would appear. 12.04%.

An astute reader might notice that the Numeric values have larger integer values than would fit in the Picture Clause of PIC 9(3)V9(15) mentioned above. This is explained on page 5 under "Calculation Caveats.".

Putting it in Practice

The method of utilization of this Alpha to Numeric conversion, therefore, is to first find

files. Unless of course you have an ISAM package that allows you to change the collating sequence. If this is the case you should stop reading now. This method will probably not work for you.

2 You can download this program in source or executable format. See Download Samples on page 20.

the Range of all possible keys on file[3] so that we can compute a percentage position for any key.

First, you must fetch the <u>first record</u> in the file (or from the subset of records of interest) and move its key to CONVERT-KEY-X. You then have a numeric value in CONVERT-KEY that represents the lowest possible key in the file.

Save this numeric value as the *Lowest-key* value in the file.

Then obtain numerical equivalent of the <u>last record</u> key in the file.

> **NOTE**: In COBOL, you move HIGH-VALUES into the record buffer, and START file-name KEY NOT LESS THAN record-key. Of course this will fail, because there is (in all probability) no record on file with a key of high-values. Your FILE-STATUS value will be the End-of-File value. No matter! The record pointer is now set to the end of file. If you then simply READ PRIOR RECORD you will have the last record in the file[4].

Then move the key of this last record into CONVERT-KEY-X, and you will find the numeric value of the last key in CONVERT-KEY.

Save this value as the *Highest key*.

Subtract the numerical value of the *Lowest key* from the numerical value of the *Highest key*, and you have the numerical representation of the **<u>Range of keys</u>** in the file[5]. (At this point you really don't need the Highest key any further. It was the Range we were after.) Save this value in *Key-Range*.

Now, for any given key in the file, you can compute it's percentage.

Move the key of the record in question to CONVERT-KEY-X, and move the value in CONFERT-KEY to a *Record key* data element.

Next you need to compute what percent of *Key-Range* that this *Record key* represents.

3 We use the entire range of keys in the file in this example. However, this method works equally well if we wanted to select a subset of records from the file, such as all the “Smiths” in or database. In this case we would fetch the first Smith, and the last Smith, and use these records for our calculations.

4 If File Status was not end-of-file you simply READ file-name NEXT RECORD. This would only happen if you actually had a High-Value key in the file.

5 To be perfectly pedantic, you would expect to add ONE to the range derived in this way. However the size of the numbers we are dealing with make this unnecessary, and undesirable.

But first you have to subtract our *Lowest key* from *Record key*. This is necessary to normalize the *Record-key* to the Range.

Then you divide the *Record key* by the *Key-Range*, which yields a value ranging from zero to 1. Multiply that quotient by 100 and you have the <u>percentage position</u> within the file.

If you are in the practice of putting all File IO in a callable subroutine[6], you can embed this percentage calculation into the subroutine and never have to worry about it again.

- When you OPEN the file, DELETE a record, or ADD a record, you fetch the First record, Last record, and compute Range. You save that Range in the working storage of the File IO subroutine.

- Each READ against the file (or READ NEXT, READ PRIOR, etc) calculates the percentage, and returns it to the calling program.

If you are doing this in a callable file-IO subroutine this logic is completely hidden from your main programs. The Percentage is passed back in one of the arguments.

Calculation Caveats.

There are a few things to be aware of when using CONVERT-KEY-X to convert text to numbers.

1. You must use COMP-4, because BINARY is compiler dependent, and COMP-5 has low order and high order bytes inverted.

2. You do not need and do not want Signed Picture Clauses.

3. You have to be cognizant of size equivalences of COMP fields. You can use two bytes, four bytes or 8 byte for the text keys, as these equate to standard increments of COMP fields. Some Compilers support quad precision 16 byte COMP numbers.

4. You must move your data out of the COMP-4 field to a larger field, because COMP-4 fields can hold values larger than their picture clause as implemented in most compilers. Example: 9(4) COMP-4 can actually hold a number as large as 32k, (32768). Some alphabetic sequence in CONVERT-KEY-X will likely equate to a number larger than 9999. Therefore, you need one extra digit ahead

6 If not, why not? It makes maintenance SO much easier.

of the decimal point for your target field.

5. You need to be cognizant of COBOL's handling of intermediate results so as not to lose precision in the decimal portion. Luckily, the degree of precision needed for Listviews is not so great that this can't be handled with 6 to 8 places behind the decimal.

Finally, there are some compilers which do not handle all values, especially very small decimal values in COMP fields. Some compilers also impose high-order truncation when moving a large number like 32k from a PIC 9(4) source even if the target is PIC 9(5). (There are usually tricks to get around this.) These lead to weird problems that are costly to debug. Therefore its best to move the COMP-4 values to regular Display Values prior to any other computation. Display handling seems more reliable.

Calculation Data Elements

For the discussion at hand, we would use the following data elements:

Our Conversion Field:

```
      01  CONVERT-KEY-X.
          05  CONVERT-KEY              PIC 9(3)V9(15) COMP-4.
```

Our Storage Fields and calculation fields. (All are Display Numeric).

```
      01  HIGHEST-KEY                  PIC 9(4)V9(14).
      01  LOWEST-KEY                   PIC 9(4)V9(14).
      01  KEY-RANGE                    PIC 9(4)V9(14).
      01  FILE-PERCENT                 PIC 9(3)V9(12).
```

And its also convenient to have a Working computational field:

```
      01  WORK-CALC                    PIC 9(4)V9(14).
```

Once a value is run through CONVERT-KEY and rendered to a number it is immediately placed in one of the Display Numeric fields. Then, and only then, are any math operations performed.

You may use simple COBOL arithmetic verbs, ADD, SUBTRACT, DIVIDE, MULTIPLY, or you may use COMPUTE statements. Compute statements are often slower, and more prone to loss of precision in intermediate results. For portability reasons, we therefore prefer to avoid COMPUTE verbs. Picture Clauses are the key to proper precision, but check your COBOL manual regarding intermediate results.

Calculation Code

As mentioned above, EVERY time you OPEN the file, DELETE a record, or ADD a record you have to recalculate the Highest and Lowest keys. This costs two extra read operations against the file. If done in a FILE-IO subroutine you will never be aware of them. If your system does massive numbers of consecutive of Key-High record ADD operations, you might want to delay this recalculation until you are finished adding. Further, you could forgo recalculating the range all together if you knew you were doing mid-file insertions or deletions. The speed with which you can do math is far greater than the speed you can do file IO, so you might want to pass each key for ADD or DELETE operations through CONVERT-KEY-X and determine if CONVERT-KEY was higher than HIGHEST-KEY or lower than LOWEST-KEY and **only then** re-fetch HIGHEST-KEY and LOWEST-KEY **<u>after</u>** the file operation completes. For most transactional data files where small numbers of records are added this may prove unnecessary.

Code for OPEN, ADD, DELETE:

Typical code for OPEN, ADD or DELETE operations appears below. The point here is to re-fetch the Lowest, Highest, and Range and keep these in memory for subsequent read operations. We prefer to put this in a stand alone paragraph and perform that paragraph upon any of the above operations.

```
GET-HIGHEST-LOWEST-PARAGRAPH.
    MOVE ZERO TO KEY-RANGE.
    MOVE LOW-VALUES TO my-FILE-RECORD.
    START my-FILE KEY IS GREATER THAN my-FILE-KEY

    READ my-FILE NEXT RECORD
    IF my-FILE-STATUS = '00'
      MOVE my-FILE-KEY TO CONVERT-KEY-X
      MOVE CONVERT-KEY TO LOWEST-KEY

      MOVE HIGH-VALUES TO my-FILE-RECORD
      START my-FILE KEY IS GREATER THAN my-FILE-KEY

      READ my-FILE PRIOR RECORD
      IF my-FILE-STATUS = '00'
        MOVE my-FILE-KEY TO CONVERT-KEY-X
        MOVE CONVERT-KEY TO HIGHEST-KEY
        SUBTRACT LOWEST-KEY FROM HIGHEST-KEY
            GIVING KEY-RANGE.
```

The above will yield a value in KEY-RANGE which will be positive if all file operations succeeded, and zero otherwise.

Code for READ (any variety):

Upon any read operation, we perform a paragraph which calculates the percentage. We do the calculations in simple arithmetic because it often is faster, and its always easier to debug.

```
      CALC-PERCENT-PARAGRAPH.
          MOVE ZERO TO FILE-PERCENT.
          IF KEY-RANGE > ZERO
            MOVE my-FILE-KEY TO CONVERT-KEY-X
            MOVE CONVERT-KEY TO WORK-CALC
            SUBTRACT LOWEST-KEY FROM WORK-CALC
            DIVIDE WORK-CALC BY KEY-RANGE GIVING WORK-CALK
            MULTIPLY 100.00000000 BY WORK-CALC GIVING FILE-PERCENT.
```

Code for START-AT-PERCENT: (Yes, a new extension for COBOL).

Listviews frequently ask you to start loading records from some percentage point in the file, usually because the user has dragged the listview thumb down the scroll-bar some arbitrary amount. The listview will ask you to start returning records beginning with, as an example, %75.3764 into the file. (Absolute precision is not important to the listview in most instances.)

We treat START-AT-PERCENT as if it were a valid COBOL start condition. That is, we position the read pointer such that the next read will produce the record at the desired percentage point in the key space. This is in keeping with other COBOL start statements. We do not actually return the record until the user does the READ NEXT or READ PRIOR.

We calculate the key that would be closest to the actual percentage point in the file. We don't expect this to be a valid key. We don't care. We will tell COBOL to START NOT LESS THAN this calculated key, which will leave the record pointer ready to read the next record.

Again, it is shown here as a performable paragraph:

```
      START-AT-PERCENT-PARAGRAPH.

        COMPUTE WORK-CALC =
          ((FILE-PERCENT / 100.000000) * KEY-RANGE) + LOWEST-KEY
        MOVE WORK-CALC TO CONVERT-KEY
        MOVE CONVERT-KEY-X TO my-FILE-KEY
        START my-FILE KEY IS NOT LESS THAN my-FILE-KEY
        MOVE my-FILE-STATUS TO FILE-STATUS-RETURN.
```

Note that FILE-PERCENT is expected to be loaded prior to this paragraph being invoked. We used the COMPUTE verb here because this is a seldom performed operation and efficiency is not too important.

Keys Size and Distribution

As described herein, this method is sufficient to handle keys up to 8 characters.

Most compilers have a quad precision COMP-4: PIC 9(18)V9(18) which can handle up to 16 byte Alphanumeric keys. By changing the picture clause of CONVERT-KEY you would be able to convert the code here in to accommodate these larger keys. You would have to adjust all other work fields accordingly.

However, problems can result when converting to Numeric Display due to the limitation of 18 digits for numeric display items imposed by COBOL. Low order truncation may result, but the precision should be close enough for Listview usage.

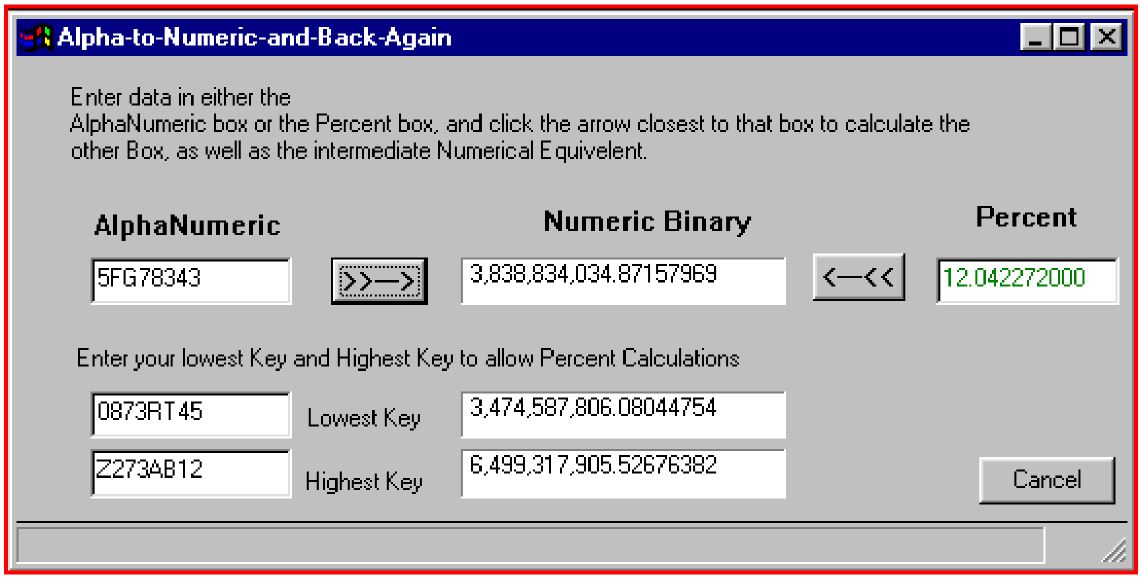
While the position of the decimal point is somewhat arbitrary in these examples. You need to coordinate the decimal positions in the CONVERT-KEY with your working fields such that the working fields have an integer portion <u>larger</u> by one position.

Making the integer portion too large results in shedding precision when the result is moved to the calculated percent field due to its three digit integer position. Three digits of integer is all that is needed in a percentage value used to indicate position within a file. The math involved will necessarily shed precision by dropping two positions of the decimal portion of the number during the calculation process as the quotient is multiplied by 100 to deliver a percent value. Losing two our of 14 positions of precision is better than losing 2 out of 8 or 2 out of 6 positions.

However, key-spaces with keys that tend to vary only in the rightmost least significant positions require as much precision as can be mustered.

> An example of such least-significant variation is phone numbers, where there are many with the same area code and exchange portion, and variability only in the last four positions.

If our Alpha-to-Numeric calculator is modified to calculate based on CONVERT-KEY being defined with PIC 9(9)V9(9), we only obtain 8 decimal digits after the move to working fields. This results in a calculated percentage value that has shed precision from the low end. We have, at best, with 6 decimal digits of precision.



Enter data in either the
AlphaNumeric box or the Percent box, and click the arrow closest to that box to calculate the other Box, as well as the intermediate Numerical Equivelent.

AlphaNumeric	Numeric Binary	Percent
5FG78343	3,838,834,034.87157969	12.042272000

Enter your lowest Key and Highest Key to allow Percent Calculations

0873RT45	Lowest Key	3,474,587,806.08044754
Z273AB12	Highest Key	6,499,317,905.52676382

Therefore, the optimum decimal position in our CONVERT-KEY is indicated by picture clause of 9(3)V9(15). Due to fact that COMP-4 can hold one more digit than the picture clause shows, we need a picture clause of 9(4)V9(14) in our working fields.

Also, if you know your keys are all numeric, you can dispense with the conversion from alphanumeric text fields and simply use the keys you have. It is important to be aware of decimal places, and you may want to use a redefinition of your numeric key to make sure you have adequate decimal places in your working fields so that calculations do not shed precision.

Files with really long keys are probably not suitable for this method. Nor are files with long keys which only vary in the right most position, as such keys tend to yield all the same percentage values.

There is no guarantee that the keys in any file are Uniformly Distributed over the key-space. Therefore, using this technique, there is no guarantee that dragging the thumb of a listview to the 50% range will start in the second half of the file. But this percentage calculation method is close enough that it can be used to satisfy the percentage based positioning of Listviews with reasonable accuracy, and more importantly, repeatability and consistently across all file operations on the given file.

Because this method is based on Key Values, it is usable for almost any file type. It is not limited to ISAM.

The File-IO Subroutine.

We long ago realized that it is far better to put all file IO into a single subroutine, or one such subroutine per file. That way we do not have Select statements, File sections, and all the responsibility of proper open/close and status checking scattered in dozens of programs comprising the system. Further, if we change a record layout, add an Alternate Index, convert from ISAM to Btrieve or SQL, none of the main line programs need to be changed.

All of the Percentage Calculation functions described in this paper can and should be embedded in your File-IO subroutine. That way they are “fire and forget”. You will have a percentage value available on any read, at little or no computational cost.

A typical File-IO subroutine will be called with two arguments.

One, of course is the file record itself. The other is what we call the File-Request argument structure.

This File-Request argument provides your calling programs with all of the information and capabilities they would have if the file was embedded in that program. It carries instructions to the File-IO subroutine, and carries results back to your program.

This File-Request structure needs an area to pass IN the desired Operation. Read, write, open, close, rewrite, delete, etc.

It also needs an area to pass OUT the File Status code resulting from the operation.

It is best to render File Status Codes to a standardized set so that you are not tied to one particular file type. Further, it is best to encode these in COBOL 88 names so you don't have to change your programs if you switch compilers, or switch file systems. You simply map the new compiler's status to the standardized set of 88 values.

You will need codes for OK, Not Found, EOF, Locked Record, Duplicate Record, and perhaps a few more.

Finally, for the subject at hand, you would want a Percentage field.

Other optional features in the File Request argument might include provision for a Alternate Index number if your file supports operations based on different keys, Cursor Numbers if your file system supports multiple Cursors, Relative Record number if your file has no internal keying structures, and a “file specification” field so that the file-IO routine is not hard coded to using a specific file location, and you have the flexibility of telling it where the files are located for OPEN operations.

Linking File-IO Subroutines

File-IO subroutines can be linked statically or dynamically. For a discussion of the differences see: http://screenio.com/gui_screenio/gs_htmlhelp_subweb/techtips/calls.htm

The short version of the above document is that Static calls embed the File-IO routine into your EXE, and the Dynamic calls require that you compile and link it as a standalone DLL.

There are also difference in the Call statement:

Static call:

```
      CALL “FILEIO” USING FILE-REQUEST FILE-RECORD.
```

Dynamic call:

```
      01   MY-DATA-FILE-HANDLER      PIC X(8) VALUE “FILEIO”.
      .
      .
      CALL MY-DATA-FILE-HANDLER USING FILE-REQUEST FILE-RECORD.
```

We STRONGLY recommend dynamic calls when your application consists of more than one EXE, or one EXE and several DLLs. If you don't use Dynamic calls when file-io must be performed in the various DLLs of your application you risk file corruption, and record locking problems.

If your application is one single EXE, and will never grow beyond that then static linking works fine.

Converting is laborious, so if there is any chance your application will grow beyond a single EXE, take the time to make all file-io subroutine calls dynamic. Its easier than going through every program later to change them, and dynamic calling is actually desirable in most cases, as you can fix the file-io subroutine without recompiling the entire application again.

An Example File-IO Subroutine.

We provide the following example of a file-IO subroutine. This routine was designed to work with GUI ScreenIO programs where Listview controls were employed to display file data. It is designed for large files, with many thousands of records where it would not be practical to “count records” or use other contrivances to arrive at a percentage figure to satisfy the listview's need for a percents.

The File-Request copybook.

The file-request copybook is, as mentioned above designed to specify the FILE-REQUEST argument structure. Its fairly simple in this example.

```
      *                 :--------------------------------------
      *-----------------: This is a simple file request copybook
      *                 : used with a file-io subroutine.
      *                 : It provides all basic io functionality
      *                 : commonly found with ISAM files.
      *                 :--------------------------------------
       01  FILE-REQUEST.
         05  FILE-OPERATION             PIC 99.
           88  FILE-OPEN-INPUT          VALUE 01.
           88  FILE-OPEN-OUTPUT         VALUE 02.
           88  FILE-OPEN-IO             VALUE 03.
      *
           88  FILE-CLOSE               VALUE 04.
           88  FILE-UNLOCK              VALUE 05.
      *
           88  FILE-READ                VALUE 06.
           88  FILE-READ-NEXT           VALUE 07.
           88  FILE-READ-PRIOR          VALUE 08.
           88  FILE-READ-LOCK           VALUE 09.
      *
           88  FILE-WRITE               VALUE 10.
           88  FILE-REWRITE             VALUE 11.
           88  FILE-DELETE              VALUE 12.
      *
           88  FILE-START-EQUAL         VALUE 14.
           88  FILE-START-NOT-LESS      VALUE 15.
           88  FILE-START-GREATER       VALUE 16.
           88  FILE-START-AT-PERCENT          VALUE 17.
      *
      *                 : Adjust these file status values to
      *                 : your compiler.  Always use 88 names.
         05  FILE-STATUS-RETURN         PIC XX.
           88  FILE-STATUS-OK           VALUE '00'.
           88  FILE-STATUS-LOCKED-RECORD  VALUE '9L'.
           88  FILE-STATUS-EOF          VALUE '10'.
      *
      *                 : This is only needed on OPEN
      *                 : statements.
      *                 : This specifies the PATH only, the
      *                 : individual file names are dictated
      *                 : by the io module.
         05  FILE-PATH-SPEC            PIC X(161).
      *
      *                 : Used as input to a start-at-percent
      *                 : request and output for any read.
         05  FILE-PERCENT              PIC 9(3)V9(12).
      *                 :
```

FILE-IO COBOL Example.

```
000010*$CALL
000020*@S                       SUBROUTINE
000030 IDENTIFICATION DIVISION.
000040*                         : I-O subroutine for ISAM
000050*                         : This is an instructional example.
000060*                         : You are free to use as you see fit
000070*                         : at your own risk.  If something goes
000080*                         : wrong its not our fault, even if you
000090*                         : say it is, because we accept no
000100*                         : responsibility for loss or damages.
000110*                         : It must be something YOU did.
000120 PROGRAM-ID. FILEIO.
000130 DATE-COMPILED.
000140*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
000150 ENVIRONMENT DIVISION.
000160*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
000170 CONFIGURATION SECTION.
000180 SOURCE-COMPUTER. IBM-PC.
000190 OBJECT-COMPUTER. IBM-PC.
000200*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
000210 INPUT-OUTPUT SECTION.
000220*                         : This section may need reconfiguring
000230*                         : for some compilers.
000240 FILE-CONTROL.
000250*
000260     SELECT DATA-FILE ASSIGN TO VARYING DATA-FILE-SPEC
000270         ORGANIZATION IS INDEXED
000280         ACCESS MODE IS DYNAMIC
000290         RECORD KEY IS FD-KEY
000300         LOCK MODE IS MANUAL
000310         FILE STATUS IS FILE-STAT.
```

```
000320*
000330*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
000340 DATA DIVISION.
000350*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
000360 FILE SECTION.
000370*                         : This section may need reconfiguring
000380*                         : for some compilers.
000390 FD  DATA-FILE
000400         LABEL RECORDS STANDARD.
000410* -=====-
000420*                         : This subroutine does not care about
000430*                         : the structure of the file or data
000440*                         : names in it.  All it needs to know
000450*                         : is the record length over all, and
000460*                         : key size and position.
000470*                         : So keep this FD record simple.
000480*
000490*                         : RECORD AND KEY SIZES EXAMPLES ONLY
000500*                         : --Adjust to your needs --
000510 01  FD-REC.
000520     05  FD-KEY                       PIC X(20).
000530     05  FD-FILLER                    PIC X(1480).
000540*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
000550 WORKING-STORAGE SECTION.
000560 01  V--VERSION-LEVEL                 PIC 9(3) VALUE 1.
000570*
000580*                         : THIS IS THE ACTUAL FILE NAME AND
000590*                         : EXTENSION.  Adjust as needed..
000600*                         : DO NOT USE EMBEDDED SPACES IN THIS
000610*                         : NAME.
000620 01  MY-DATA-FILE-NAME                PIC X(20) VALUE
000630                                         'DATAFILE.DAT'.
000640*
000650 01  DATA-FILE-SPEC                   PIC X(161).
000660 01  FILE-STAT                        PIC XX.
000670*
000680 01  1HUNDRED                       PIC 9(4)V9(14) VALUE 100.0.
000690*
000700*                         :-------------------------------------
000710* ------------------------: Note regarding these calc fields.
000720*                         : There is high-order truncation
000730*                         : possible when moving a comp field
000740*                         : to a display field of equal size
000750*                         : because the comp field can hold more
000760*                         : than its picture size shows.
000770*                         : Example: 9(4) COMP-4 can hold 32k.
000780*                         :
000790*                         : Therefore we only use the comp flds
000800*                         : for the conversion from alpha to
```

```
000810*                         : numeric, and then move that to a
000820*                         : display field which is one order of
000830*                         : magnitude larger: 9(4) vs 9(3) etc.
000840*                         : This prevents high order overflow.
000850*                         :
000860*                         : You might be tempted to think you
000870*                         : could use comp all the way thru. But
000880*                         : A compiler bug (or something) also
000890*                         : crops up with very small numbers in
000900*                         : some compilers.
000910*                         : You can not put 0.000002259 into a
000920*                         : PIC 9(3)V9(15) COMP-4  field because
000930*                         : the some compilers lose the decimal
000940*                         : location.
000950*                         :--------------------------------------
000960 01  LOWEST-KEY                       PIC 9(4)V9(14).
000970 01  HIGHEST-KEY                      PIC 9(4)V9(14).
000980 01  KEY-RANGE                        PIC 9(4)V9(14).
000990 01  WRK-TARGET-KEY                   PIC 9(4)V9(14).
001000*
001010 01  CONVERT-KEY-X.
001020     05  CONVERT-KEY                  PIC 9(3)V9(15) COMP-4.
001030*
001040 01  STATUS-BYTE                      PIC X VALUE 'C'.
001050     88  IS-OPEN                      VALUE 'O'.
001060     88  NOT-OPEN                     VALUE 'C'.
001070*
001080*                         :
001090 01  WS-IDX                           PIC S9(4) COMP-5.
001100*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
001110 LINKAGE SECTION.
001120* -=====-
001130         COPY FILEREQU.
001140*
001150*                         : RECORD AND KEY SIZES ARE EXAMPLES
001160*                         : --Adjust to your needs --
001170 01  LINKAGE-DATA-RECORD.
001180     05  LINKAGE-KEY                  PIC X(20).
001190     05  LINKAGE-FILLER               PIC X(1480).
001200*
001210*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
001220 PROCEDURE DIVISION USING FILE-REQUEST LINKAGE-DATA-RECORD.
001230*
001240* -=====-
001250 0001-MAIN-LINE-LOGIC.
001260*
001270     EVALUATE TRUE
001280       WHEN FILE-OPEN-INPUT
001290         IF IS-OPEN
```

```
           CLOSE DATA-FILE
           SET NOT-OPEN TO TRUE
         END-IF
         PERFORM 0002-BUILD-FILE-SPEC
         OPEN INPUT DATA-FILE
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         IF FILE-STATUS-OK
           SET IS-OPEN TO TRUE
           PERFORM 0003-GET-HIGHEST-LOWEST
         END-IF
*
       WHEN FILE-OPEN-OUTPUT
         IF IS-OPEN
           CLOSE DATA-FILE
           SET NOT-OPEN TO TRUE
         END-IF
         PERFORM 0002-BUILD-FILE-SPEC
         OPEN OUTPUT DATA-FILE
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         IF FILE-STATUS-OK
           SET IS-OPEN TO TRUE
         END-IF
*
       WHEN FILE-OPEN-IO
         IF IS-OPEN
           CLOSE DATA-FILE
           SET NOT-OPEN TO TRUE
         END-IF
         PERFORM 0002-BUILD-FILE-SPEC
         OPEN I-O DATA-FILE
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         IF FILE-STATUS-OK
           SET IS-OPEN TO TRUE
           PERFORM 0003-GET-HIGHEST-LOWEST
         END-IF
*
       WHEN FILE-CLOSE
         IF IS-OPEN
           CLOSE DATA-FILE
           SET NOT-OPEN TO TRUE
         END-IF
*
       WHEN FILE-READ
         MOVE LINKAGE-KEY TO FD-KEY
         READ DATA-FILE KEY IS FD-KEY
         MOVE FD-REC TO LINKAGE-DATA-RECORD
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         PERFORM 0004-CALCULATE-PERCENT
*
```

```
       WHEN FILE-READ-LOCK
         MOVE LINKAGE-KEY TO FD-KEY
         READ DATA-FILE WITH LOCK
             KEY IS FD-KEY
         MOVE FD-REC TO LINKAGE-DATA-RECORD
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         PERFORM 0004-CALCULATE-PERCENT
*
       WHEN FILE-UNLOCK
         MOVE LINKAGE-KEY TO FD-KEY
         UNLOCK DATA-FILE
*
       WHEN FILE-READ-NEXT
         READ DATA-FILE
         NEXT RECORD
         MOVE FD-REC TO LINKAGE-DATA-RECORD
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         PERFORM 0004-CALCULATE-PERCENT
*
       WHEN FILE-READ-PRIOR
         READ DATA-FILE PRIOR RECORD
         MOVE FD-REC TO LINKAGE-DATA-RECORD
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         PERFORM 0004-CALCULATE-PERCENT
*
       WHEN FILE-WRITE
         MOVE LINKAGE-DATA-RECORD TO FD-REC
         WRITE FD-REC
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         PERFORM 0003-GET-HIGHEST-LOWEST
*
       WHEN FILE-REWRITE
         MOVE LINKAGE-DATA-RECORD TO FD-REC
         REWRITE FD-REC
         MOVE FILE-STAT TO FILE-STATUS-RETURN
*
       WHEN FILE-DELETE
         MOVE LINKAGE-DATA-RECORD TO FD-REC
         DELETE DATA-FILE RECORD
         MOVE FILE-STAT TO FILE-STATUS-RETURN
         PERFORM 0003-GET-HIGHEST-LOWEST
*
       WHEN FILE-START-EQUAL
         MOVE LINKAGE-DATA-RECORD TO FD-REC
         START DATA-FILE KEY IS EQUAL TO FD-KEY
         MOVE FILE-STAT TO FILE-STATUS-RETURN
*
       WHEN FILE-START-NOT-LESS
         MOVE LINKAGE-DATA-RECORD TO FD-REC
         START DATA-FILE KEY IS NOT LESS THAN FD-KEY
         MOVE FILE-STAT TO FILE-STATUS-RETURN
*
       WHEN FILE-START-GREATER
         MOVE LINKAGE-DATA-RECORD TO FD-REC
         START DATA-FILE KEY IS GREATER THAN FD-KEY
         MOVE FILE-STAT TO FILE-STATUS-RETURN
*
       WHEN FILE-START-AT-PERCENT
*                         : Calculate the NEEDED key...
*                         : its x percent of the range plus
*                         : the lowest key.
         COMPUTE CONVERT-KEY =
         ( (FILE-PERCENT / 1HUNDRED) * KEY-RANGE) + LOWEST-KEY
*
         MOVE CONVERT-KEY-X TO FD-KEY
         START DATA-FILE KEY IS NOT LESS THAN FD-KEY
         MOVE FILE-STAT TO FILE-STATUS-RETURN
*
     END-EVALUATE
     GOBACK.
*
 0002-BUILD-FILE-SPEC.
*                         : This method of supplying a variable
*                         : file spec is CA-Realia Specific.
*                         : You may need to change this code.
     MOVE FILE-PATH-SPEC TO DATA-FILE-SPEC
     PERFORM
           VARYING WS-IDX FROM LENGTH OF DATA-FILE-SPEC BY -1
           UNTIL WS-IDX NOT > 0
           OR DATA-FILE-SPEC (WS-IDX:1) > SPACE
       MOVE LOW-VALUE TO DATA-FILE-SPEC (WS-IDX:1)
     END-PERFORM.
*                         : Now point to first low-value
     ADD 1 TO WS-IDX
     STRING MY-DATA-FILE-NAME DELIMITED SPACE
         '[X:B4:D2:I4]' LOW-VALUE DELIMITED SIZE
          INTO DATA-FILE-SPEC WITH POINTER WS-IDX.
*
 0003-GET-HIGHEST-LOWEST.
     MOVE ZERO TO KEY-RANGE.
*                         : First get the lowest actual key
     MOVE LOW-VALUES TO FD-REC
     START DATA-FILE KEY IS GREATER THAN FD-KEY
*
     READ DATA-FILE
     NEXT RECORD
     IF FILE-STAT = '00'
       MOVE FD-KEY TO CONVERT-KEY-X
```

```
002770       MOVE CONVERT-KEY TO LOWEST-KEY
002780*                        : now get the highest key
002790       MOVE HIGH-VALUES TO FD-REC
002800       START DATA-FILE KEY IS GREATER THAN FD-KEY
002810*                        : We know the above will yield eof
002820       READ DATA-FILE PRIOR RECORD
002830       IF FILE-STAT = '00'
002840         MOVE FD-KEY TO CONVERT-KEY-X
002850         MOVE CONVERT-KEY TO HIGHEST-KEY
002860         SUBTRACT LOWEST-KEY FROM HIGHEST-KEY
002870             GIVING KEY-RANGE.
002880*
002890 0004-CALCULATE-PERCENT.
002900*                        : This calcs the % value of the key
002910*                        : for the record just read in
002920     MOVE ZERO TO FILE-PERCENT.
002930     IF KEY-RANGE > ZERO
002940       MOVE FD-KEY TO CONVERT-KEY-X
002950       MOVE CONVERT-KEY TO WRK-TARGET-KEY
002960       SUBTRACT LOWEST-KEY FROM WRK-TARGET-KEY
002970       DIVIDE WRK-TARGET-KEY BY KEY-RANGE GIVING WRK-TARGET-KEY
002980       MULTIPLY 1HUNDRED BY WRK-TARGET-KEY GIVING FILE-PERCENT.
002990*
```

Download Samples

We provide our Alpha to Numeric calculator both in source code (you will require GUI ScreenIO to compile this) and also in binary mode which you can simply unzip into a directory and click on the A2N.EXE file to play with the numbers.

We also provide a sample File-IO subroutine (above) which you may use as a pattern for ISAM files from which you want to load Listviews in GUI ScreenIO, or anywhere else you need file Percentages. The subroutine was designed to require the minimum of customization so that it can be cloned easily.

Both are available here:
http://www.screenio.com/gui_screenio/gs_htmlhelp_subweb/download/downloads.htm