Issue Date: iSeries edition
May 2004

iSeries EXTRA: 'Sorting' it All Out

Susan Gantner and Jon Paris

We've recently taught several sessions on RPG capabilities in V5R2. We've often said that we use these features in combination with other techniques to, for example, provide the facility to sort the data in a subfile. Ears perk up when we mention this, as end users often request this capability. Of course it's always been possible, but the task has become easier—one reason being that a DS can now be specified as the source/receiver of the data on an I/O operation. This month we thought we'd share the details of one such demonstration program with you.

Before we begin however, we should note that a full discussion of some of the techniques used here, such as the qsort API, are beyond the scope of this article. We will, however, reference other documents that provide more information.

The program we'll discuss is a simple one. It simply displays a subfile containing all of the records in a database. It uses a "load all" technique to keep the example as simple as possible. Modifying it to use a "page-at-a-time" approach is left as an exercise for you, gentle reader.

As you can see below, the main line of the program is a simple Do loop **(B)** that continues until the user asks to exit by pressing F3.

The program begins by invoking the LoadArray subprocedure **(A)**. As we'll see, this processes the records in the database, adding each one to an array of subfile records. It returns a count of the number of records loaded.

The program then loads the subfile from the array **(C)**, displays it **(D)** and waits for the user to select one of two sort sequences—by Product Code or by Product Description. In our example this is handled by assigning a function key to each sequence, but it would be simple to use a single function key and make the program sensitive to the column in which the cursor is positioned.

Next the program determines which sequence was requested **(E)** and sets up the appropriate procedure pointer. The program then sorts the subfile to the requested sequence **(F)**, and redisplays it to the user.

That's all there is to it.

```
(A) Count = LoadArray();    // Load all data for subfile

(B) DOU *In(Exit);

(C)   LoadSubfile(Count);

(D)      DisplaySubfile();

(E)     SELECT;       // Determine sort option

            WHEN *In(SortPrCode);
             SortRoutine = SortProduct;

            WHEN *In(SortDesc);
             SortRoutine = SortDescr;

          OTHER;      // Default to Product sort
             SortRoutine = SortProduct;
```

```
          ENDSL;

            // Call the sort routine

(F)           SortDS(SubFileData:
                     Count:
                     %Size(SubfileData.SubfileRec):
                     SortRoutine);

          ClearSubfile();

      ENDDO;

    *INLR = *On;
```

Now that we've seen the basic structure, let's take a quick look at some of the detail behind it. First we should mention that for those who find our use of indicators such as *IN(Exit) and *IN(SortPrCode) a little strange, you can click here.

First let's look at the LoadArray routine. The definition of the array looks like this:

```
D SubfileData      DS                  Qualified
D    SubfileRec                           LikeRec(ProdSfl:*Output) Dim(99)
```

The point here is that the array elements are defined as being based on the output fields of the ProdSfl subfile format. By using LikeRec we create a nested DS that contains all of the output fields in the record format. These are then loaded with the corresponding data from the database **(G)**. If you're unfamiliar with the use of qualified names, click here. For the sake of simplicity, we've hard-coded the array with a length of 99 elements. In practice, we would use a much larger number and perhaps use dynamic memory or a User Space to contain the array, but that's outside the scope of this piece.

```
      // Read all records and fill array of subfile records
      DOU %EOF(Product);

           READ ProductR;

           IF Not %Eof(Product);

             n +=1;
< BR>(G)       SubfileData.SubfileRec(n).ProdCD = ProdCd;
           SubfileData.SubfileRec(n).ProdDS = ProdDs;
           SubfileData.SubfileRec(n).CatCod = CatCod;
           SubfileData.SubfileRec(n).StOH   = StOH;
           SubfileData.SubfileRec(n).SellPr = SellPr;
< BR >           ENDIF;

       ENDDO;

      RETURN n;
```

Loading the actual subfile (LoadSubfile) is simplicity itself, because the "hard" work of moving the individual fields is done in LoadArray. Because each of the elements in the array represents a complete subfile record, all that remains is to write them to the subfile **(H)**. This is achieved by specifying the elements of the array as the result field of the WRITE operation. The FOR loop itself is controlled by the record count passed on the call to LoadSubfile **(C)**:

```
      FOR RRN = 1 to RecordCount;

(H)     WRITE ProdSfl SubfileData.SubfileRec(RRN);
```

```
     ENDFOR;
```

The DisplaySubfile routine is straightforward enough to leave to your imagination (though it's worth noting that, with meaningful names like these, you don't need as much imagination as you would if we'd used cryptic six-character names with "$" signs or something equally horrible).

So what's next? The invocation of the sort itself **(F)**. Note that rather than actually code two separate sort invocations, the SELECT clause **(E)** sets up the procedure pointer SortRoutine so it references either the SortProduct or the SortDescription subprocedure. More on this in a moment. We can only briefly describe the qsort API here. For a full understanding, read the Redbook, "Who Knew You Could Do That With RPG?", which includes a fully worked example of qsort and its companion routine, bsearch. The example in question also uses dynamic memory allocation.

The qsort API (prototyped here under the name SortDS) takes four parameters. The first is the structure to be sorted. In most examples you'll see this passed as a pointer (e.g., %Addr(DataToBeSorted) but we're taking advantage of the fact that a field passed by reference is the same as a pointer passed by value (see the Redbook example if you haven't a clue what we mean by this). This allows us to have an "RPG-feel" to the thing. The second parameter is the count of the number of entries to be sorted. The third is the size of each entry. (Note that we use %Size. Never hard-code this value—let the compiler work it out for you. It will always get it right.) The last parameter is the procedure pointer to the sequencing routine. What's that? Well, one of the nice things about qsort is that, unlike SORTA, you as the RPG programmer get to decide what is bigger than which—this adds flexibility. For example, you could upper-case the values before comparing them. We haven't done that here, but you can do so. Let's look at one of the sequencing routines.

Below you see the code for the description sequencing routine. Pretty simple, isn't it? This is known as a call-back routine. What happens is that we call qsort, and qsort "calls-back" into our program for help in sequencing the data. Each time qsort calls, it passes us a pair of entries: Element1 and Element2 to our code **(I)**. Our job is to compare them **(J)** and return to qsort a value that indicates which of the two has the higher value. Qsort calls us repeatedly until it knows that the whole structure is in sequence. At that time it returns to where it was originally called, and the data is in sequence.

```
     P DescrSort       B

     D                 PI            10I 0
(I)  D   Element1                              LikeRec(ProdSfl: *Output)
     D   Element2                              LikeRec(ProdSfl: *Output)

       /FREE

          SELECT;
(J)         WHEN Element1.ProdDs > Element2.ProdDs;
                RETURN High;
              WHEN Element1.ProdDs < Element2.ProdDs;
                RETURN Low;
              OTHER;
                RETURN Equal;
            ENDSL;
```

You may wonder if qsort permits you to perform a multi-key sort. You betcha! For instance, if we had a series of addresses, and wanted to sort (say) City in State, the code might look something like this:

```
       SELECT;
              WHEN Address.State1 > Address.State2;
                   RETURN High;
               WHEN Address.State1 < Address.State2;
                   RETURN Low;
              OTHER;
                   SELECT;
```

```
          WHEN Address.City1 > Address.City2;
            RETURN High;
         WHEN Address.City1 < Address.City2;
            RETURN Low;
         OTHER;
             RETURN Equal;
      ENDSL;
   ENDSL;
```

Not that hard at all, is it? The qsort API offers many possibilities that are difficult if not impossible to achieve through the use of SORTA. Of course you could always write your own sort, but we'll leave that for the masochists among you.

That about wraps things up. We're sure you can think of numerous uses for these techniques. In particular, if you haven't already read it, you may be interested in this article, which describes another possible use for the result field on I/O operations (in this case to trap and destroy those pesky decimal-data errors).

As always, we welcome your comments and suggestions for future articles. In next month's iSeries EXTRA, we hope to bring you a brief overview of what's in store for V5R3. In the meantime, you'll find the full version of the code used in this article together with the display and database file definitions on our Web site.