“Gain incredible flexibility with dynamic SQL statements using DESCRIPTORS”

-- OR --

“SQL DESCRIPTORS – the “prepare” statement on steroids!”

Pre-requisites

- IBM web site where my program was derived
  - https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/sqlp/rbafyallocexample.htm
    - (Many of the comments in my program are directly from the IBM example on their web site)

Program Structure

- Parameters
- Data Definitions
- Main Procedure
  - Del_File()
  - Crt_File()
  - GetFileData()
  - Closing

  (more on this later)

Parameters

```
dcl-pr MAIN extPgm('SQLTOIFS');
  Complete_IFS_File_Name          varchar(80);
  SQL_Statement_for_Extract_File varchar(4096);
  Include_Header_YN               char(1) OPTIONS(*NOPASS);
  Column_Separator                char(1) OPTIONS(*NOPASS);
end-Pr;

dcl-pi MAIN   ;
  FileName       varchar(80);
  SQLStatement  varchar(4096);
  pInclude_Header char(1) OPTIONS(*NOPASS);
  pColumn_Separator char(1) OPTIONS(*NOPASS);
end-Pi;
```

Complete_IFS_File_Name varchar(80)

- Complete file name
  - IFA path
  - file name
  - Extension

For example:

/home/derrja/myFoldername/MyFileName.txt

SQL_Statement_for_Extract_File varchar(4096)

Let the fun begin!

Use any SQL Service to develop your SQL statement

```
4 SELECT      *
5 FROM        Employee;
```

SELECT * FROM Employee

File Edit View

EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-0
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-1
000030	SALLY	A	KWAN	C01	4738	1975-04-0

Done: 42 rows retrieved.

Any SQL statement

```
 4 SELECT       *
 5 FROM         Employee;
 6
 7 SELECT       EmpNo, CASE WHEN MidInit = ''
 8                          THEN TRIM(FirstNme) || ' ' || TRIM(LastName)
 9                          ELSE TRIM(FirstNme) || ' ' || MidInit || TRIM(LastName)
10                          END  AS CustName,
11              HireDate, Job
12 FROM         Employee
13 WHERE        BirthDate > '1950-01-01'
14 ORDER BY     BirthDate DESC;
```

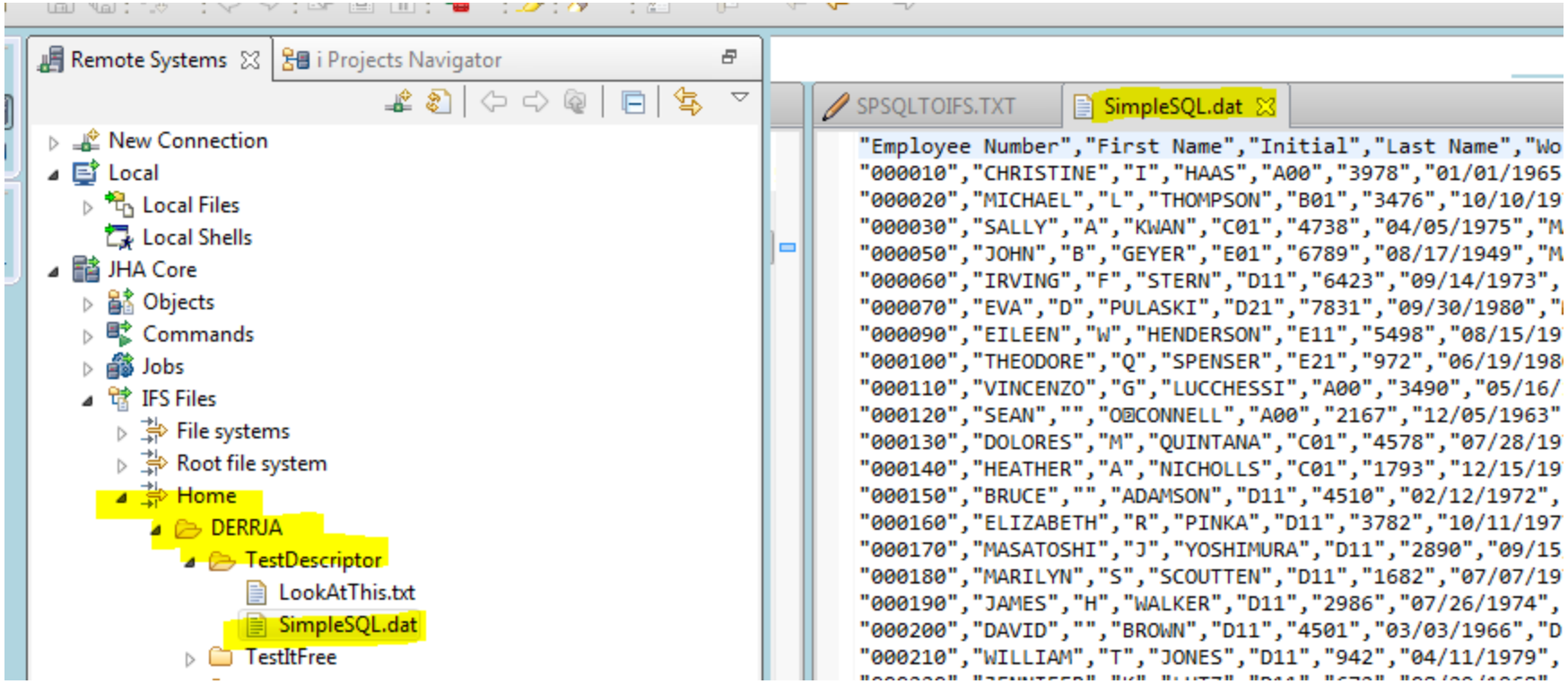
SELECT EmpNo, CASE WHEN MidInit = '' THEN TRIM(FirstNme) || ' ' || TRIM(LastName) ELSE TRIM(FirstNme...

File Edit View

EMPNO	CUSTNAME	HIREDATE	JOB
000100	THEODORE QSPENSER	1980-06-19	MANAGER
000160	ELIZABETH RPINKA	1977-10-11	DESIGNER
000240	SALVATORE MMARINO	1979-12-05	CLERK
200240	ROBERT MMONTEVERDE	1979-12-05	CLERK
000070	EVA DPULASKI	1980-09-30	MANAGER
000270	MARIA LPEREZ	1980-09-30	CLERK
000210	WILLIAM TJONES	1979-04-11	DESIGNER
000190	JAMES HWALKER	1974-07-26	DESIGNER
000170	MASATOSHI JYOSHIMURA	1978-09-15	DESIGNER
200170	KIYOSHI YAMAMOTO	1978-09-15	DESIGNER

Simple example

```
call spSQLtoIFS('/home/derrja/TestDescriptor/SimpleSQL.dat','SELECT * FROM Employee');
```



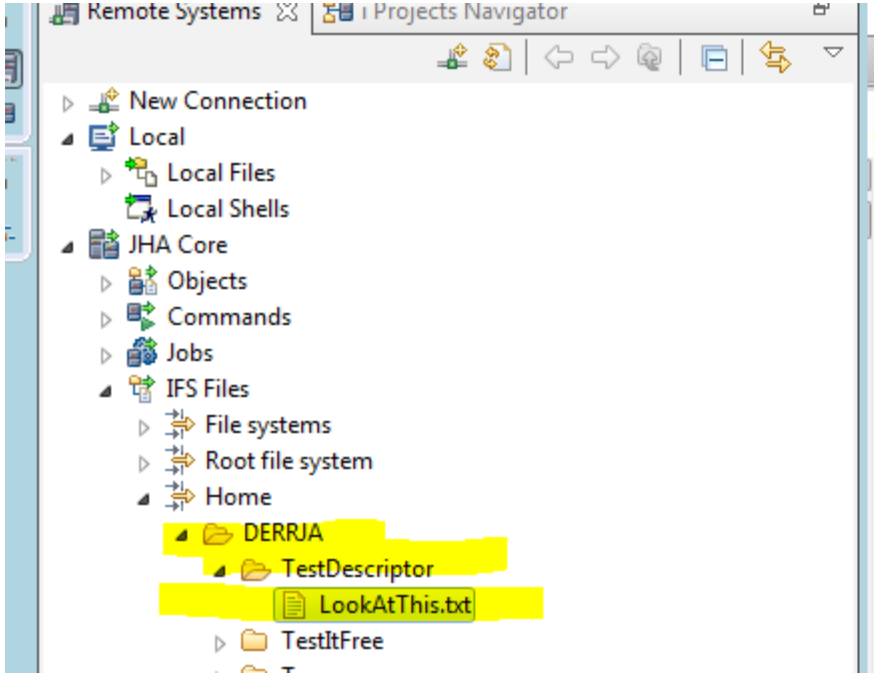
Deal with the ticks

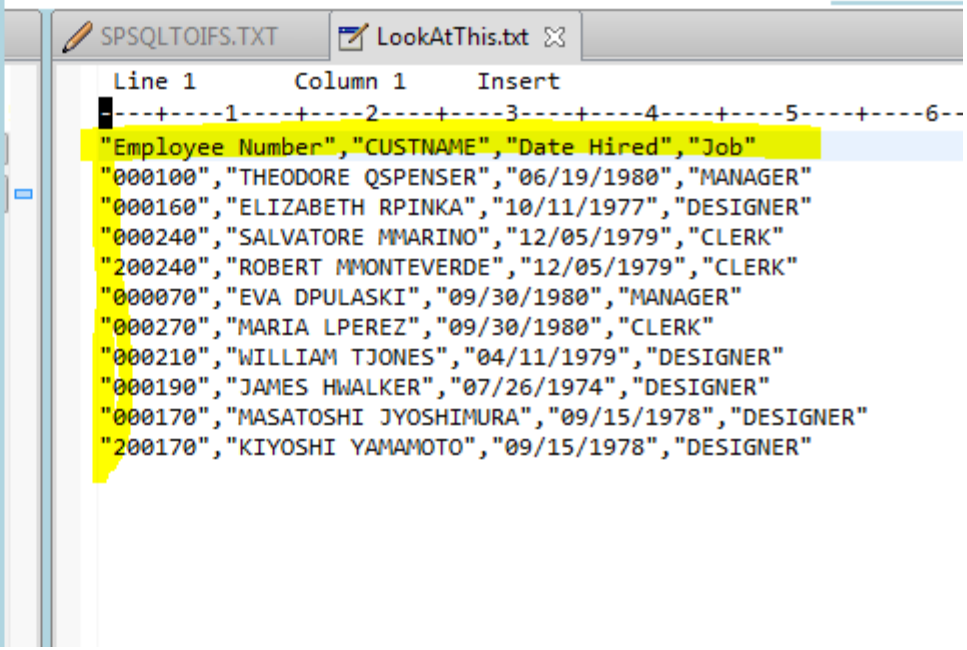
```
SELECT      *
FROM        Employee;

SELECT      EmpNo, CASE WHEN MidInit = '' ''
                    THEN TRIM(FirstNme) || '' '' || TRIM(LastName)
                    ELSE TRIM(FirstNme) || '' '' || MidInit || TRIM(LastName)
                    END  AS CustName,
            HireDate, Job
FROM        Employee
WHERE       BirthDate > ''1950-01-01''
ORDER BY    BirthDate DESC;
```

Put it together

```
call spSQLtoIFS('/home/derrja/TestDescriptor/LookAtThis.txt' ,
                'SELECT      EmpNo, CASE WHEN MidInit = '' ''
                                         THEN TRIM(FirstNme) || '' '' || TRIM(LastName)
                                         ELSE TRIM(FirstNme) || '' '' || MidInit || TRIM(LastName)
                                         END  AS CustName,
                                    HireDate, Job
                 FROM        Employee
                 WHERE       BirthDate > ''1950-01-01''
                 ORDER BY    BirthDate DESC');
```





DESCRIPTORS - Let's ignore the common stuff

Delete the file on the IFS,
Create the same file,
Open the file,
Close the file,
Close the cursor,
bla, bla, bla

Populate the SQL statement string from the second parameter.

```
dString = %trim(SQLStatement);
```

```
// The statement is assigned to a host variable. The host variable, in this case named DSTRING,
//  is then processed by using the PREPARE statement as shown:
EXEC SQL
  PREPARE s1 FROM :dstring;


// Next, you need to determine the number of result columns and their data types. To do this,
//  you need to allocate the largest number of entries for an SQL descriptor that you think
//  you will need. Assume that no more than 20 columns are ever expected to be accessed by a
//  single SELECT statement.
EXEC SQL
  ALLOCATE DESCRIPTOR 'myDescriptor' WITH MAX 20;
```

(don't worry – we're dynamic!)

```
// Now that the descriptor is allocated, the DESCRIBE statement can be done to get the
//  column information.
EXEC SQL
  DESCRIBE s1 USING DESCRIPTOR 'myDescriptor';

// When the DESCRIBE statement is run, SQL places values that provide information about the
//  statement's select-list into the SQL descriptor area defined by 'myDescriptor'.

// If the DESCRIBE determines that NOT ENOUGH ENTRIES were allocated in the descriptor,
//  SQLCODE +239 is issued. As part of this diagnostic, the second replacement text value
//  indicates the number of entries that are needed. The following code sample shows how
//  this condition can be detected and shows the descriptor allocated with the larger size.
```

```
// Determine the returned SQLCODE from the DESCRIBE statement */
EXEC SQL
  GET DIAGNOSTICS CONDITION 1: returned_sqlcode = DB2_RETURNED_SQLCODE;

// Get the second token for the SQLCODE that indicated
//   not enough entries were allocated
if returned_sqlcode = 239;
  EXEC SQL
    GET DIAGNOSTICS CONDITION 1: token = db2_ordinal_token_2;
  //* Move the token variable from a character host variable into an integer host variable */
  EXEC SQL
    SET :var1 = :token;
  //* Deallocate the descriptor that is too small */
  EXEC SQL
    DEALLOCATE DESCRIPTOR 'myDescriptor';
  //* Allocate the new descriptor to be the size indicated by the retrieved token */
  EXEC SQL
    ALLOCATE DESCRIPTOR 'myDescriptor' WITH MAX :var1;
  //* Perform the describe with the larger descriptor */
  EXEC SQL
    DESCRIBE s1 USING DESCRIPTOR 'myDescriptor';
endif;
```

```
// At this point, the descriptor contains the information about the SELECT statement.
//  Now you are ready to retrieve the SELECT statement results. For dynamic SQL, the
//  SELECT INTO statement is not allowed. You must use a cursor.
EXEC SQL
  DECLARE c1 CURSOR FOR s1;


// You will notice that the prepared statement name is used in the cursor declaration instead
//  of the complete SELECT statement. Now you can loop through the selected rows, processing
//  them as you read them. The following code sample shows how this is done.
EXEC SQL
  OPEN c1;

dou sqlcode = 100;        // while not at end of data;

  EXEC SQL
    FETCH c1 INTO SQL DESCRIPTOR 'myDescriptor';
  // SQLSTATE '01534' simply means that a date, time or timestamp is present in the data.
  if sqlcode = 100 or (sqlcode <> 0 AND sqlcode <> 100 AND sqlstate <>'01534');
    leave;
  endIf;

  //* process current data returned  (see below for discussion of doing this) */
```

COUNT - How many fields did I get?

```
// The cursor is opened. The result rows from the SELECT statement are then returned one at a time
//  using a FETCH statement. On the FETCH statement, there is no list of output host variables.
//  Instead, the FETCH statement tells SQL to return results into the descriptor area.

// After the FETCH has been processed, you can use the GET DESCRIPTOR statement to read the values
//  First, you must read the header value that indicates how many descriptor entries were used.
EXEC SQL
  GET DESCRIPTOR 'myDescriptor' :icount = COUNT;
iRow += 1;


// Next you can read information about each of the descriptor entries. After you determine the dat
//  type of the result column, you can do another GET DESCRIPTOR to return the actual value. To
//  get the value of the indicator, specify the INDICATOR item. If the value of the INDICATOR item
//  is negative, the value of the DATA item is not defined. Until another FETCH is done, the
//  descriptor items will maintain their values.
//  (i = the number of fields that were returned).
for i = 1 to iCount;
```

```
for i = 1 to iCount;
   Ds1(i).Data  = *blank;
   Ds1(i).Name  = *blank;
   Ds1(i).Label = *blank;

   EXEC SQL
     -- set entry number to get */
     -- get the data type */
     -- length value */
     //      :iscale = SCALE,
     //      :iprecision = PRECISION,
     GET DESCRIPTOR 'myDescriptor'
       VALUE :i :itype = TYPE,
                :ilength = LENGTH,
                :iresult_ind = INDICATOR,
                :icolumn_name = DB2_COLUMN_NAME,
                :ilabel = DB2_LABEL,
                :iname = DB2_COLUMN_NAME,
                :idatetimetype = DATETIME_INTERVAL_CODE;
       //      :idata = DATA;

   // Set the column headings. If the column lablel is blank, use the column name.
   if iresult_ind >= 0;
     if iLabel = *blank;
         ds1(i).Label = iName;
         ds1(i).Name  = iName;
       else;
         ds1(i).Label = iLabel;
         ds1(i).Name  = iName;
     endif;
```

```
// Parse the columns. Check each of the column type and translate it into a common
// field for output.
SELECT;
```

Now you must check each of the possible data types, with specific logic for each!

```
////////////////////////////////////////////////////////
// DESCRIPTOR - RETURNED COLUMN TYPE
////////////////////////////////////////////////////////
dcl-s tCharacter       int(10) inz( 1);
dcl-s iDate_time_Stamp int(10) inz( 9);
dcl-s tDecimal         int(10) inz( 3);
dcl-s tInteger         int(10) inz( 4);
dcl-s tVarChar         int(10) inz(12);

////////////////////////////////////////////////////////
// DESCRIPTOR - SUB TYPES WHEN TYPE IS 9 (DATE/TIME/TIMESTAMP)
////////////////////////////////////////////////////////
dcl-s iDateType        int(10) inz( 1);
dcl-s iTimeType        int(10) inz( 2);
dcl-s iTimeStampType   int( 3) inz( 3);
```

```
// DATE/TIME/TIMESTAMPS
      when iType = iDate_time_Stamp;
        select;
          when iDateTimeType = iDateType;
           EXEC SQL
             GET DESCRIPTOR 'myDescriptor'
               VALUE :i :idata_date_08 = DATA;
           ds1(i).Data = %char(iData_Date_08:*usa/);
          when iDateTimeType = iTimeType;
           EXEC SQL
             GET DESCRIPTOR 'myDescriptor'
               VALUE :i :idata_time_08 = DATA;
           ds1(i).Data = %char(iData_Time_08:*usa:);
          when iDateTimeType = iTimeStampType;
           EXEC SQL
             GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_stamp = DATA;
           ds1(i).Data = %char(iData_Stamp);
        endSl;
```

These types are easy – they are self-defined

```
// CHARACTERS
      when iType = tCharacter;
       EXEC SQL
         GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_char = DATA;
       ds1(i).Data = %trim(iData_Char);

// VARYING CHARACTERS
      when iType = tVarChar;
       EXEC SQL
         GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_varchar = DATA;
       ds1(i).Data = %trim(iData_VarChar);

// INTEGERS
      when iType = tInteger;
       EXEC SQL
         GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_integer = DATA;
       ds1(i).Data = %editc(iData_Integer : 'Z');
```

There may be a better way – I'm still looking for it!

```
// DECIMALS (KNOWN LENGTHS AND PRECISIONS ONLY)
      when iType = tDecimal;
        ExtractDecimalData();

    endsl; //* continue checking and processing for all data types that might be returned */
 endif;
endfor;
```

```
//////////////////////////////////////
//*=====================Extract Decimal Data ========================
// ---------------------------------------------
// Procedure name:  ExtractNumericData
// Purpose:         Identifies the returned column's decimal length and precision.
// Returns:
// Author:          John Derr
// Created:         02/14/2018
//--------------------
DCL-PROC ExtractDecimalData  ;

DCL-PI *N   ;
END-PI ;

  // The following permutations of length and scale were derived from running a
  // "distinct" query of all data on our main data library.
  dcl-s iData_Dec01_00   packed(01:00);
  dcl-s iData_Dec02_00   packed(02:00);
  dcl-s iData_Dec02_01   packed(02:01);
  dcl-s iData_Dec02_02   packed(02:02);
  dcl-s iData_Dec03_00   packed(03:00);
  dcl-s iData_Dec03_01   packed(03:01);
  dcl-s iData_Dec03_02   packed(03:02);
  dcl-s iData_Dec03_03   packed(03:03);
  dcl-s iData_Dec04_00   packed(04:00);
  dcl-s iData_Dec04_01   packed(04:01);
  dcl-s iData_Dec04_02   packed(04:02);
  dcl-s iData_Dec04_03   packed(04:03);
  dcl-s iData_Dec04_04   packed(04:04);
  dcl-s iData_Dec05_00   packed(05:00);
```

(and many more)

```
EXEC SQL
  GET DESCRIPTOR 'myDescriptor'
    VALUE :i :iprecision = PRECISION,
             :iscale = SCALE;

// The following permutations of length and scale were derived from running a
// "distinct" query of all data on the main data library. As new conditions are discovered,
// simply add them to this process.
select;
  when                                    iPrecision = 01 AND
                                              iScale = 00;
    EXEC SQL
      GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_dec01_00 = DATA;
    ds1(i).Data =                 %trim(%editc(iData_Dec01_00 : 'N'));

  when                                    iPrecision = 02 AND
                                              iScale = 00;
    EXEC SQL
      GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_dec02_00 = DATA;
    ds1(i).Data =                 %trim(%editc(iData_Dec02_00 : 'N'));

  when                                    iPrecision = 02 AND
                                              iScale = 01;
    EXEC SQL
      GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_dec02_01 = DATA;
    ds1(i).Data =                 %trim(%editc(iData_Dec02_01 : 'N'));

  when                                    iPrecision = 02 AND
                                              iScale = 02;
    EXEC SQL
      GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_dec02_02 = DATA;
    ds1(i).Data =                 %trim(%editc(iData_Dec02_02 : 'N'));

  when                                    iPrecision = 03 AND
                                              iScale = 00;
    EXEC SQL
      GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_dec03_00 = DATA;
    ds1(i).Data =                 %trim(%editc(iData_Dec03_00 : 'N'));
```

(and many more)

```
    when                                    iPrecision = 26 AND
                                                iScale = 09;
      EXEC SQL
        GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_dec26_09 = DATA;
      ds1(i).Data =                 %trim(%editc(iData_Dec26_09 : 'N'));

    when                                    iPrecision = 31 AND
                                                iScale = 00;
      EXEC SQL
        GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_dec31_00 = DATA;
      ds1(i).Data =                 %trim(%editc(iData_Dec31_00 : 'N'));

    when                                    iPrecision = 31 AND
                                                iScale = 02;
      EXEC SQL
        GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_dec31_02 = DATA;
      ds1(i).Data =                 %trim(%editc(iData_Dec31_02 : 'N'));

    other;
      ds1(i).Data = 'Error on row ' + %trim(%editc(iRow : 'Z')) + ', Column ' +
        %trim(iColumn_Name) + ' ';

  endsl;


END-PROC ;
```

```
// DATE/TIME/TIMESTAMPS
      when iType = iDate_time_Stamp;
        select;
        //when iLength = 8;
          when iDateTimeType = iDateType;
           EXEC SQL
             GET DESCRIPTOR 'myDescriptor'
               VALUE :i :idata_date_08 = DATA;
           ds1(i).Data = %char(iData_Date_08:*usa/);
          when iDateTimeType = iTimeType;
           EXEC SQL
             GET DESCRIPTOR 'myDescriptor'
               VALUE :i :idata_time_08 = DATA;
           ds1(i).Data = %char(iData_Time_08:*usa:);
          when iDateTimeType = iTimeStampType;
           EXEC SQL
             GET DESCRIPTOR 'myDescriptor' VALUE :i :idata_stamp = DATA;
           ds1(i).Data = %char(iData_Stamp);
        endSl;
```

Finally – the stored procedure simplifies calling the RPG logic

```
    --*    PROGRAM NAME - spSQLtoIFS
    --*    PROGRAMMER   - John Derr
    --*    DATE         - 02/22/2018
    --*    PURPOSE      - Reads four parameters to run an SQL and write the output to the IFS.
    --*    PARAMETERS   -
    --*      1. Full IFS location of the output file, including the file name and extension.
    --*         Example: /home/derrja/Testdescriptor/testDateStuff.csv
    --*      2. Full SQL statement that you want to execute.
    --*         Example: (in one contiguous string)
    --*                  SELECT CustNo, CAST(SUM(CurBal) AS DEC(11, 2)) AS Current_Balance +
    --*                  FROM   OrderMast +
    --*                  WHERE  status NOT IN (''D'', ''X'') +
    --*                  GROUP BY CustNo +
    --*                  ORDER BY CustNo

CREATE OR REPLACE PROCEDURE BE0598(IN Filelocation VARCHAR(80)
                      ,IN Sqlstatement VARCHAR(4096)
                      ,IN Include_Header_Yn CHAR(1) DEFAULT 'Y'
                      ,IN Column_Separator CHAR(1) DEFAULT '"')
  LANGUAGE RPGLE
  SPECIFIC  SPSQLTOIFS
  NOT DETERMINISTIC
  READS SQL DATA
  CALLED ON NULL INPUT
  EXTERNAL NAME SQLTOIFS
  PARAMETER STYLE SQL;


GRANT ALTER, EXECUTE ON SPECIFIC PROCEDURE SPSQLTOIFS TO PUBLIC;
```

Really, finally – caveats

1. Code is “as is” and is not guaranteed to work.
2. Known error – I saw *very late* that the fourth parameter is called “column separator”, but it is really the string delimiter character! I haven’t had the opportunity to fix it, but how hard could it be?
3. I intend to add the column separator, and will update the code on our web site when I do.

Really, really, finally:

Thank you for listening, I hope you found the presentation interesting, and can find a place for DESCRIPTORS in your shop!

If you have any questions, I can be reached at JohnDerr@charter.net