How to use External Stored Procedures with .NET Applications

by Craig Pelkie

"Book learning" is certainly valuable, but when it collides with experience, it usually folds pretty quickly. For example, I've been using SQL stored procedures in some applications: code SQL statements, surround them by the requisite procedure statements and create the stored procedures. These types of stored procedures run SQL SELECT, INSERT, UPDATE and DELETE statements and can be coded and tested using tools in the iSeries Navigator. It is easy to create .NET code that uses SQL stored procedures, as there are many examples that show how to get started.

There is another type of stored procedure on the iSeries called an external stored procedure. It is essentially a thin wrapper around an iSeries program object. After creating the external stored procedure, you can call it from a client application, passing parameters to it and receiving values back through the parameter list. This provides a mechanism for client programs to call iSeries programs when there is no other readily available technique. For example, in the IBM Toolbox for Java and the IBM OLE DB Provider there is support for directly calling iSeries programs. However, the .NET Provider from IBM does not include program call support and there has been no clear statement from IBM that such support is forthcoming. However, because it is possible to create an external stored procedure over an iSeries program, .NET applications can call iSeries programs through the stored procedure call mechanism that is used in the .NET Provider.

Until trying to use external stored procedures, I didn't imagine there would be any particular mysteries or difficulties. The iSeries Navigator includes a tool to create an external stored procedure; it simply asks for the name of the program and its parameters. However, there is an option on the tool to indicate the "parameter passing style", which was the root cause of my recent experience. In the tool, there are four options for the parameter style:

- SQL
- Simple, allow null values
- Simple, no null values
- Java

The SQL option is a superset of the Simple, allow null values option. Java is used if you are wrapping a Java class. For my tests, I wanted to wrapper RPG programs, so the two options I tried were the "Simple" options.

The basic question to be answered, before you start creating external stored procedures, is do you want to allow nulls to be passed as parameter values? It turns out that the decision has tremendous implications for both the client program (the program that calls the external stored procedure) and the program that is wrappered.

The Easy Case

We can look at the easy case first: this is an example of how you might wrapper an existing RPG program as an external stored procedure. An example of such a

program is SPGEN, shown in Figure 1. The header comments show the CREATE PROCEDURE command used to create the external stored procedure; note particularly the last line, PARAMETER STYLE GENERAL. This corresponds to the "Simple, no null values" parameter style option that is presented on the iSeries Navigator tool. As an aside, you need to use the CRTBNDRPG command shown in the header comments to first create the SPGEN program object; you then go into an SQL environment like STRSQL or the iSeries Navigator tools and run the CREATE PROCEDURE command.

This program is typical of most RPG programs that you probably have on your iSeries. That is, the programs, when written, were not concerned with the possibility that you might someday be calling them through the database manager as an external stored procedure, so you wouldn't have given any thought at all to the idea of accommodating null values in the parameter list.

The reason why the "Simple, no null values" option is appealing is because you can use the program as-is. In fact, you can simply call the program from an OS/400 command line to test it (it simply adds two numbers, returns the result and a status code and prints its results). To ensure that you can see what's happening with this program when it is called as an external stored procedure, I've included the anachronistic DUMP opcode to produce an RPG formatted dump. As you'll see, DUMP can be valuable when trying to discover exactly what is being passed to an external stored procedure. (I'm well aware that it is possible, somehow, to debug server programs, but for my purposes, it is much more useful to have the dump.)

The corresponding .NET client program is shown in Figures 2C and 2V (the C# version and the Visual Basic version). Both versions of the program perform exactly the same functions.

In Section A, you are prompted to enter your iSeries hostname and are connected to the iSeries via the .NET Provider. If you can't get a connection (for example, you mistype the hostname or enter an invalid user ID or password if prompted), the code in HandleError tells you about the problem.

Section B sets up the CALL statement as a string. The CALL statement is used to call the external stored procedure. You can see that there are four parameters passed on the call, corresponding to the four parameters shown in the RPG program SPGEN (Figure 1).

As shown in Section C, an iDB2Command object is created on the connection, using the string that will call the external stored procedure. Following IBM's recommendation, the command type is set to Text and the DeriveParameters method is used to get information about the parameters from the external stored procedure back into the client program. You should note this section well, as it is contrary to how stored procedures are used with other database providers. Microsoft generally recommends that you explicitly define and code your stored procedure parameters; by doing that, you avoid a trip to the server just to find out what the parameters are. IBM's documentation states that their provider will make the trip to the server anyway, so it is unnecessary to define the parameters in the client program. Given that there will be a trip to the server, it is likely that you would want to code your client applications so that DeriveParameters is done only once.

In Sections D and E the program prompts for integer values for the first and second parameters. The if conditions test the parameter lengths and set default values for the parameters if no value is entered (when prompted, you simply press Enter). If you're the adventurous type, after you get these test programs working, go back and

remove the if test and the assignment in the if; leave the assignment statement in the else clause (in other words, simply assign whatever is entered on the console to the parameter). Run the program and do not enter a value for the parameter, just press Enter. You'll find that your call to the external stored procedure is rejected; the HandleError code will provide details for you.

The code in Section F is used to initialize the third and fourth parameters (result and state). You may find this curious. If you look in the header comments in Figure 1, the third and fourth parameters are defined as OUT parameters. Their values are returned from the external stored procedure to the client program. However, if you were to comment out the two assignment statements in Section F, the call to the external stored procedure fails. You might think that if you defined parameters 3 and 4 as INOUT parameters you could avoid the assignment statements: but no, you still need to put something into the parameters before calling the external stored procedure. In this case, it is harmless to put in default initialization values as shown (a value of zero for the integer, an empty string for the string), as the values to be returned back to you will be provided by SPGEN upon returning from it.

Section G calls the external stored procedure. The actual call occurs in the ExecuteNonQuery statement, which is bracketed by calls to the PrintParameters routine. That routine prints out all available information about each of the four parameters, before and after the ExecuteNonQuery. Coupled with the dump in the RPG program, you have a complete view of what's happening before the call, during the call and after the call.

Upon returning from the call and assuming no exceptions, the values of the third and fourth parameters are displayed. As you'll see, you've added two numbers together.

The SQLSTATE is a "pseudo" SQLSTATE code. That is, although I've called it SQLSTATE, setting its value does not affect the database manager. I'll describe this in more detail in the next section.

Section H is normal end-of-program processing.

Figures 3 and 4 show the RPG dump output when calling the SPGEN external stored procedure. In Figure 3, you can see that valid values were entered for PARM1 and PARM2. Because the dump is taken right after the *ENTRY PLIST, the RESULT and STATE values are the defaults provided in Section F of the client program. The STATE field was initialized using the .NET String.Empty value, yielding a character field of all hex 0's.

Figure 4 shows what happens when I pressed Enter in response to the prompt for parameter 2. In this case, the if condition in Section E sets the parameter value to a default value of zero. When passed to the SPGEN program, the effect is the same as if you had actually entered the value zero.

To summarize the GENERAL or "Simple, no null values" parameter style, you need to pay attention to the following:

- Provide valid values or suitable defaults for the input or input/output parameters.
- For output parameters, provide a default "placeholder" value. The value you provide must be valid for the data type of the parameter.

The More Difficult Case

Because stored procedures, even the external kind, are at heart database components, there must be an accommodation for null values. It may be difficult to see why it is useful to call a program and pass null parameters, if you think of calling programs with input from a user interface. After all, you can either pester the user to really enter a value or simply provide a default and move along. However, you might want to call an external stored procedure and pass parameter values that originate from database fields, where the value might truly be stored as null, or you might want to call an external stored procedure that will update columns in the database with null values (when and why to allow nulls in database columns is another issue).

To handle null parameters, you create the external stored procedure with the parameter style specified as GENERAL WITH NULLS. You can see the CREATE PROCEDURE statement used to create the external stored procedure SPGENNULL in the header comments of Figure 5. If you compare the rest of the CREATE PROCEDURE in Figure 5 with the statement in Figure 1, you'll see that the rest of the command is the same (other than the name of the procedure and the program). Note especially that the parameter list specified on the CREATE PROCEDURE command in Figure 5 is the same as that shown in Figure 1.

When you use the GENERAL WITH NULLS parameter style (or, the iSeries Navigator Simple, allow null values option), you need to add another parameter to the *ENTRY PLIST to contain an array of "indicators" that is used to let you know if a null value was passed on a parameter. You can see the fifth parameter defined in RPG program SPGENNULL (Figure 5). The name of the parameter is nullind.

The nullind parameter is defined as a data structure name with four subfields. In external stored procedures, a null indicator is not a traditional RPG indicator. Instead, the indicator is passed in a 16-bit value, which can be defined as a Binary(2,0) field or an Integer(5,0) field, as shown in the figure. Either of those data type definitions in RPG yields a two-byte long integer numeric field, which satisfies the requirements for a null indicator.

You need to provide a null indicator for each parameter list field, hence the four indicator subfields defined for the nullind data structure. The null indicator fields must be located in contiguous memory, so the null indicators are usually defined as a data structure or as an array, with as many elements as there are parameters. Note that you do not define a null indicator for the null indicator parameter; that is, looking at Figure 5, you are responsible for providing a null indicator area for four parameters (your part of the *ENTRY PLIST), not for five parameters as shown in the figure.

When you call an external stored procedure using this parameter style, the database manager sets the value of each null indicator to identify parameters that were passed as null values. If the parameter value is null, the corresponding null indicator is set to a negative value (-1). If the parameter value is not null, its null indicator is non-negative (0). For example, Figure 7 shows the DUMP output of a call to the SPGENNULL external stored procedure. In the dump, you can see that the first two null indicators are set to zero, meaning that non-null values were passed in the first two parameters. The last two null indicators are negative; those null indicators correspond to the OUT parameters. As you'll see, OUT parameters can in fact be passed as null values when the external stored procedure is called.

In the external stored procedure you test the value of a parameter's null indicator before using the value in the parameter. You can see this in the SPGENNUL RPG

program, where there are if conditions for both of the IN parameters. If either parameter's null indicator value is set, the parameter is set to a default value of 0 and the state variable (used to represent the SQLSTATE) is set to a user-defined value of 01HNU. When a null indicator is set for a parameter, that is a signal for you that you may need to take action to initialize the parameter to a valid value.

You should read the comments in Figure 5 that describe how the SQLSTATE is used. In the two examples shown (SPGEN and SPGENNULL), the value being set for SQLSTATE is simply a parameter value that is returned to the calling program. In either of these programs, you could set the value for SQLSTATE to anything you want. The database manager ignores whatever value you set.

If you use either of the parameter styles SQL or DB2SQL, the value you assign to SQLSTATE is seen by the database manager when you return from the external stored procedure. That is, if you set the value to other than "00000" (the SQLSTATE value for successful execution), the database manager will, in a .NET environment, raise an exception that can be caught in the client program. IBM documentation suggests that you use a value of 01Hxx (where you set a value for "xx") for warning conditions and a value of 38yxx (where you set values for "y" and "xx") for error conditions. If you use parameter style SQL or DB2SQL there are additional parameters passed to the external stored procedure by the database manager that you must add to the *ENTRY PLIST. Examples of those types of stored procedures are shown in the IBM documentation. At this point, it is sufficient to say that parameter style SQL is a superset of the GENERAL WITH NULLS style, and DB2SQL is a superset of the SQL style.

Another issue with the GENERAL WITH NULLS style concerns the OUT parameters. As shown in the dump (Figure 7), the third and fourth null indicators are set for the OUT parameters (result and state). To return values to the client application through those parameters, you need to set the values for the parameters and you need to change the null indicators for the parameters to the non-null value. The statements near the end of the SPGENNULL program show the nullind3 and nullind4 null indicators being set to a value of 0, which means that the corresponding parameter values no longer contain a null value. The null indicator values are used by both the external stored procedure to determine if input values were passed and also by the calling client program to determine if output values are being returned. Note that the null indicators for output parameters are not automatically set when you assign a value to the parameters: you need to explicitly set the null indicator values yourself before returning from the external stored procedure.

Figures 6C and 6V show the C# and Visual Basic client programs that call the SPGENNULL external stored procedure. The code in these versions is substantially the same as the code shown in Figures 2C and 2V; even the comment section numbers are the same.

Look first at how the external stored procedure call statement is coded at Section B in Figure 6. You'll see that it is still coded with four parameters, even though the SPGENNULL RPG program (Figure 5) is coded with five parameters. When you call the SPGENNULL external stored procedure, the database manager adds the null indicator parameter at run-time. You do not code the null indicator parameter on the stored procedure CALL statement.

The code in Sections D and E shows how to pass a null value for a parameter. In the .NET environment, you set the parameter value to System.DBNull.Value (because the System namespace is defined with a using or imports statement, you can refer to the class as DBNull.Value). When you set a parameter to that value, the

corresponding null indicator is set to a negative value when the external stored procedure is called. For example, see Figure 8 which is a sample of the DUMP output when a null parameter is passed. In Figure 8, the second parameter (PARM2) was passed as a null value so the null indicator NULLIND2 is set. Even though the client code in Section E explicitly set the PARM2 value to null, you can see in the dump that the apparent value for PARM2 is a valid packed field with the value zero. I surmise that the database manager puts the valid packed field value into the parameter when the external stored procedure is called, to satisfy RPG's requirement to have valid numeric data. As you can see from this code example, the only sure way to tell if a parameter value is indeed null is to check its null indicator value.

Read the note in Section F. When you use the GENERAL WITH NULLS parameter style you do not need to provide initial values for OUT parameter fields before calling the external stored procedure (see the corresponding Section F in Figure 2). The null parameter indicator for an OUT field is set when the external stored procedure is called (see Figures 7 and 8).

The rest of the code in Figure 6 is the same as in Figure 2. If there are no apparenet values for the OUT parameters upon returning from the external stored procedure (Section G), verify that you have reset the null indicators for the OUT parameters in the external stored procedure before returning from it.

Calling CL Programs

You can also call CL programs as external stored procedures. However, you can only use the GENERAL parameter style with CL programs. The database manager does not pass null indicators to CL programs.

Summary

The external stored procedure capability of OS/400 is a powerful tool that can help you access native iSeries code from .NET applications. Although in the end it is not that much more involved to use the GENERAL WITH NULLS parameter style, most native iSeries applications will use the GENERAL parameter style. If you create new code for an external stored procedure, you should carefully consider what data will be passed to the external stored procedure. If you decide to use the GENERAL WITH NULLS parameter style, you should document your reasoning and the need for the feature in the program code for later reference.

References

DB2 Universal Database for iSeries SQL Programming, Version 5 Release 3

DB2 Universal Database for iSeries SQL Reference, Version 5 Release 3

Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries (Redbook: SG24-6503, http://www.redbooks.ibm.com)

Figure 1: RPGLE program SPGEN. This is used for the external stored procedure with parameter style GENERAL.

```
*****************************************************************
*  SPGEN - parameter style: GENERAL
*
*  Compile this program as a traditional RPGLE program:
*
*      CRTBNDRPG PGM(SPROCLIB/SPGEN)
*                SRCFILE(SPROCLIB/QRPGLESRC)
*
*  Use the following SQL command to create an external stored
*  procedure based on this program:
*
*      CREATE PROCEDURE SPROCLIB.SPGEN (
*          IN  PARM1  DECIMAL(5, 0) ,
*          IN  PARM2  DECIMAL(5, 0) ,
*          OUT RESULT DECIMAL(6, 0) ,
*          OUT STATE  CHAR(6))
*          LANGUAGE RPGLE
*          SPECIFIC SPROCLIB.SPGEN
*          NOT DETERMINISTIC
*          NO SQL
*          CALLED ON NULL INPUT
*          EXTERNAL NAME 'SPROCLIB/SPGEN'
*          PARAMETER STYLE GENERAL ;
*****************************************************************

H debug(*YES)

Fqsysprt   o    f  132        printer

C     *entry        plist
C                   parm                    parm1             5 0
C                   parm                    parm2             5 0
C                   parm                    result            6 0
C                   parm                    state             5

C                   dump

C                   eval      result = parm1 + parm2
C                   eval      state  = '00000'

C                   except    ex01

C                   seton                                        lr

Oqsysprt   e            ex01              1
O                                              'SPGEN  '
O                                              '  parm1: '
O                       parm1
O                                              '  parm2: '
O                       parm2
O                                              '  result: '
O                       result
O                                              '  state: '
O                       state
```

Figure 2C: C# class SPGEN. This is the client program used to invoke the SPGEN external stored procedure.

```
using IBM.Data.DB2.iSeries;
using System;

/// <summary>
/// Tester for calling external stored procedure SPGEN
/// Parameter style GENERAL
/// </summary>
class SPGEN  {

    static void Main(string[] args)     {

        //*************************************************************
        // A: get connection to server
        //*************************************************************
        Console.WriteLine("SPGEN(CS) - Parameter style GENERAL");
        Console.Write("Enter your iSeries host name: ");
        String hostName = Console.ReadLine();

        iDB2Connection cn = new iDB2Connection("DataSource=" + hostName);

        try {
            cn.Open();
        }
        catch (iDB2Exception ex) {
            HandleError("An error occurred on cn.Open()", ex);
            return;
        }

        //*************************************************************
        // B: define stored procedure - GENERAL parameter style
        //*************************************************************
        String sp = "call sproclib.spgen(@parm1, @parm2, @result, @sqlstate)";

        iDB2Command cmd;

        try {

            //*************************************************************
            // C: set up command, derive parameters, set input parm values
            //*************************************************************
            cmd = new iDB2Command(sp, cn);
            cmd.CommandType = System.Data.CommandType.Text;
            cmd.DeriveParameters();

            //*************************************************************
            // D: parm1 - get value or set to default
            //*************************************************************
            Console.Write("Enter integer value for numeric parameter 1: ");
            String p1 = Console.ReadLine();

            if (p1.Length == 0) {
                cmd.Parameters["@parm1"].Value = 0;
            }
            else {
                cmd.Parameters["@parm1"].Value = p1;
            }

            //*************************************************************
            // E: parm2 - get value or set to default
            //*************************************************************
            Console.Write("Enter integer value for numeric parameter 2: ");
            String p2 = Console.ReadLine();

            if (p2.Length == 0) {
                cmd.Parameters["@parm2"].Value = 0;
            }
            else {
                cmd.Parameters["@parm2"].Value = p2;
```

```
            }

            //*************************************************************
            // F: The result and sqlstate parameters are defined on the
            // CREATE STORED PROCEDURE statement as OUT parameters.
            //
            // However, since the SPGEN stored procedure is created
            // as GENERAL, initial values for the parameters need to
            // be set before calling the stored procedure.
            //
            // If the initial values were not set, you would get an
            // error, stating that you tried to pass null parameters
            // to the stored procedure, which is not allowed for GENERAL
            //*************************************************************
            cmd.Parameters["@result"  ].Value = 0;
            cmd.Parameters["@sqlstate"].Value = String.Empty;

            //*************************************************************
            // G: run stored procedure, display output value
            //*************************************************************
            PrintParameters("Parameters - before call", cmd);

            cmd.ExecuteNonQuery();

            PrintParameters("Parameters - after call", cmd);

            Console.WriteLine("Return value: " + cmd.Parameters["@result"
].Value);
            Console.WriteLine("SQLSTATE: "     +
cmd.Parameters["@sqlstate"].Value);
        }

        catch (iDB2Exception ex) {
            HandleError("Error on Stored Procedure call", ex);
            return;
        }

        //*************************************************************
        // H: normal end-of-program processing
        //*************************************************************
        Console.WriteLine("Press ENTER to end");
        Console.ReadLine();

        cmd.Dispose();
        cn.Close();
    }

    //*****************************************************************
    // Handle an error during program execution
    //*****************************************************************
    static void HandleError(String errorOccurred,
                            iDB2Exception ex) {

        Console.WriteLine(errorOccurred);
        Console.WriteLine("Source:         " + ex.Source);
        Console.WriteLine("SQLState:       " + ex.SqlState);
        Console.WriteLine("Message:        " + ex.Message);
        Console.WriteLine("MessageCode:    " + ex.MessageCode);
        Console.WriteLine("MessageDetails: " + ex.MessageDetails);
        Console.ReadLine();
    }

    //*****************************************************************
    // Print parameters collection data
    //*****************************************************************
    static void PrintParameters(String      message,
                                iDB2Command cmd) {

        Console.WriteLine("********************************");
        Console.WriteLine(message);
        Console.WriteLine("********************************");
```

```
        foreach(iDB2Parameter p in cmd.Parameters) {
            Console.WriteLine("Name:          " + p.ParameterName);
            Console.WriteLine("DbType:        " + p.DbType);
            Console.WriteLine("Direction:     " + p.Direction);
            Console.WriteLine("iDB2DbType:    " + p.iDB2DbType);
            Console.WriteLine("Precision:     " + p.Precision);
            Console.WriteLine("Scale:         " + p.Scale);
            Console.WriteLine("Size:          " + p.Size);
            Console.WriteLine("SourceColumn:  " + p.SourceColumn);
            Console.WriteLine("SourceVersion: " + p.SourceVersion);
            Console.WriteLine("ToString:      " + p.ToString());

            if (p.Value == DBNull.Value) {
                Console.WriteLine("Value:         DBNull.Value");
            }
            else {
                Console.WriteLine("Value:         " + p.Value);
            }
            Console.WriteLine("------------------------------");
        }
    }
}
```

Figure 2V: Visual Basic module SPGEN. This is the client program used to invoke the SPGEN external stored procedure.

```
imports IBM.Data.DB2.iSeries

''' Tester for calling external stored procedure SPGEN
''' Parameter style GENERAL
Module SPGEN

    Sub Main()

        '-------------------------------------------------------------
        ' A: get connection to server
        '-------------------------------------------------------------
        Console.WriteLine("SPGEN(VB) - Parameter style GENERAL")
        Console.Write("Enter your iSeries host name: ")

        Dim hostName As String = Console.ReadLine()

        Dim cn As iDB2Connection = New iDB2Connection("DataSource=" & hostName)

        Try
            cn.Open()

        Catch ex As iDB2Exception
            HandleError("An error occurred on cn.Open()", ex)
            Return
        End Try

        '-------------------------------------------------------------
        ' B: define stored procedure - GENERAL parameter style
        '-------------------------------------------------------------
        Dim sp As String = "call sproclib.spgen(@parm1, @parm2, @result,
@sqlstate)"

        Dim cmd As iDB2Command

        Try
            '---------------------------------------------------------
            ' C: set up command, derive parameters
            '---------------------------------------------------------
            cmd = New iDB2Command(sp, cn)
            cmd.CommandType = System.Data.CommandType.Text
            cmd.DeriveParameters()

            '---------------------------------------------------------
            ' D: parm1 - get value or set to default
            '---------------------------------------------------------
```

```
            Console.Write("Enter integer value for numeric parameter 1: ")
            Dim p1 As String = Console.ReadLine()

            If (p1.Length = 0) Then
                cmd.Parameters("@parm1").Value = 0
            Else
                cmd.Parameters("@parm1").Value = p1
            End If

            '-------------------------------------------------------
            ' E: parm2 - get value or set to default
            '-------------------------------------------------------
            Console.Write("Enter integer value for numeric parameter 2: ")
            Dim p2 As String = Console.ReadLine()

            If (p2.Length = 0) Then
                cmd.Parameters("@parm2").Value = 0
            Else
                cmd.Parameters("@parm2").Value = p2
            End If

            '-------------------------------------------------------
            ' F: The result and sqlstate parameters are defined on the
            ' CREATE STORED PROCEDURE statement as OUT parameters.
            '
            ' However, since the SPGEN stored procedure is created
            ' as GENERAL, initial values for the parameters need to
            ' be set before calling the stored procedure.
            '
            ' If the initial values were not set, you would get an
            ' error, stating that you tried to pass null parameters
            ' to the stored procedure, which is not allowed for GENERAL
            '-------------------------------------------------------
            cmd.Parameters("@result"  ).Value = 0
            cmd.Parameters("@sqlstate").Value = String.Empty

            '-------------------------------------------------------
            ' G: run stored procedure, display output value
            '-------------------------------------------------------
            PrintParameters("Parameters - before call", cmd)

            cmd.ExecuteNonQuery()

            PrintParameters("Parameters - after call", cmd)

            Console.WriteLine("Return value: " & cmd.Parameters("@result"
).Value)
            Console.WriteLine("SQLSTATE:     " &
cmd.Parameters("@sqlstate").Value)

        Catch ex As iDB2Exception
            HandleError("Error on Stored Procedure call", ex)
            Return
        End Try

        '-------------------------------------------------------------
        ' H: normal end-of-program processing
        '-------------------------------------------------------------
        Console.WriteLine("Press ENTER to end")
        Console.ReadLine()

        cmd.Dispose()
        cn.Close()

    End Sub

    '-------------------------------------------------------------
    ' Handle an error during program execution
    '-------------------------------------------------------------
    Sub HandleError(errorOccurred As String, _
                    ex            As iDB2Exception)
```

```
        Console.WriteLine(errorOccurred)
        Console.WriteLine("Source:         " & ex.Source)
        Console.WriteLine("SQLState:       " & ex.SqlState)
        Console.WriteLine("Message:        " & ex.Message)
        Console.WriteLine("MessageCode:    " & ex.MessageCode)
        Console.WriteLine("MessageDetails: " & ex.MessageDetails)
        Console.ReadLine()

    End Sub

    '------------------------------------------------------------
    ' Print parameters collection data
    '------------------------------------------------------------
    Sub PrintParameters(message As String, _
                        cmd     As iDB2Command)

        Console.WriteLine("****************************************")
        Console.WriteLine(message)
        Console.WriteLine("****************************************")

        For Each p As iDB2Parameter In cmd.Parameters

            Console.WriteLine("Name:          " & p.ParameterName)
            Console.WriteLine("DbType:        " & p.DbType)
            Console.WriteLine("Direction:     " & p.Direction)
            Console.WriteLine("iDB2DbType:    " & p.iDB2DbType)
            Console.WriteLine("Precision:     " & p.Precision)
            Console.WriteLine("Scale:         " & p.Scale)
            Console.WriteLine("Size:          " & p.Size)
            Console.WriteLine("SourceColumn:  " & p.SourceColumn)
            Console.WriteLine("SourceVersion: " & p.SourceVersion)
            Console.WriteLine("ToString:      " & p.ToString())

            If (p.Value Is DBNull.Value) Then
                Console.WriteLine("Value:           DBNull.Value")
            Else
                Console.WriteLine("Value:           " & p.Value.ToString())
            End If

            Console.WriteLine("----------------------------------------")
        Next
    End Sub

End Module
```

Figure 3: RPG Dump output from SPGEN, no null values passed.

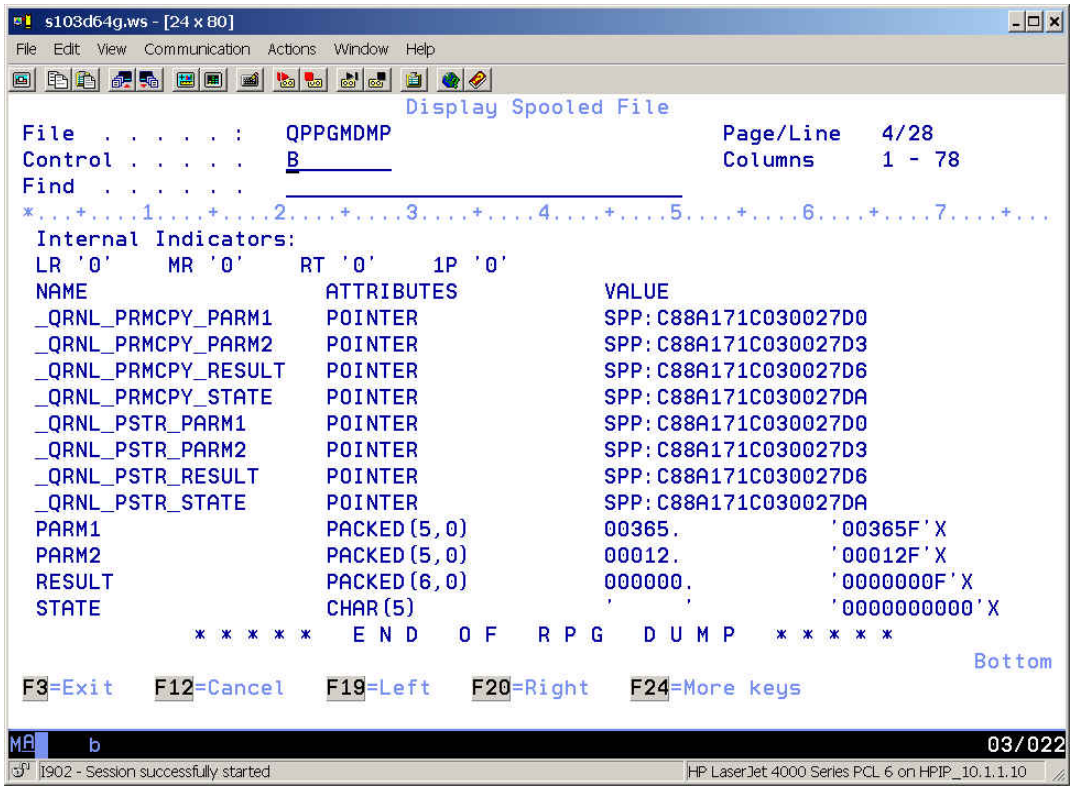

Figure 4: RPG Dump output from SPGEN, null value passed for PARM2.

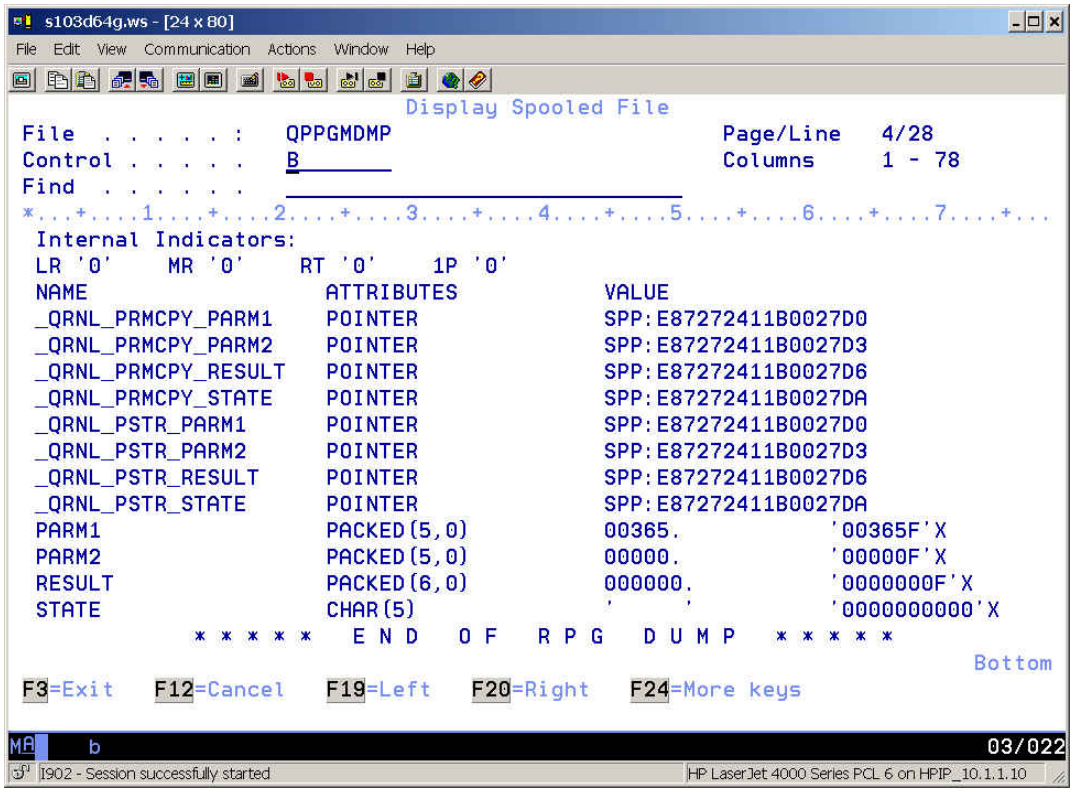

Figure 5: RPGLE program SPGENNULL. This is used for the external stored procedure with parameter style GENERAL WITH NULLS.

```
 ****************************************************************
 *  SPGENNULL - parameter style: GENERAL WITH NULLS
 *
 *  Compile this program as a traditional RPGLE program:
 *
 *      CRTBNDRPG PGM(SPROCLIB/SPGENNULL)
 *                SRCFILE(SPROCLIB/QRPGLESRC)
 *
 *  Use the following SQL command to create an external stored
 *  procedure based on this program:
 *
 *      CREATE PROCEDURE SPROCLIB.SPGENNULL (
 *          IN  PARM1  DECIMAL(5, 0) ,
 *          IN  PARM2  DECIMAL(5, 0) ,
 *          OUT RESULT DECIMAL(6, 0) ,
 *          OUT STATE  CHAR(6))
 *          LANGUAGE RPGLE
 *          SPECIFIC SPROCLIB.SPGENNULL
 *          NOT DETERMINISTIC
 *          NO SQL
 *          CALLED ON NULL INPUT
 *          EXTERNAL NAME 'SPROCLIB/SPGENNULL'
 *          PARAMETER STYLE GENERAL WITH NULLS;
 ****************************************************************

H debug(*YES)

Fqsysprt   o    f  132        printer

 ****************************************************************
 * Data Structure nullind
 *
 * Used to contain the "null indicators". Note, although these
 * are called "inidcators" they are not indicators in the
 * traditional RPG sense.
 *
 * When the program is called, the value of a null indicator
 * will be negative (-1) if the corresponding parameter value
 * passed to the program is null.
 *
 * When returning from the program, you need to set the value
 * of the corresponding null indicator to a non-negative value
 * (typically 0) to pass the parameter value back. If you don't
 * set the null indicator value to non-negative, the value
 * returned in the corresponding parameter is Null.
 ****************************************************************
D nullind         ds
D    nullind1                    5i 0
D    nullind2                    5i 0
D    nullind3                    5i 0
D    nullind4                    5i 0

 ****************************************************************
 * The *ENTRY PLIST includes the four parameters that are
 * defined in the Stored Procedure CREATE statement. The
 * implicit nullind parameter must be added to the *ENTRY
 * PLIST. The stored procedure client program (the caller)
 * does not directly work with the nullind parameter. Instead,
 * it works with the other values in the PLIST, either setting
 * the values as Null before the call or working with the
 * values or the Null value upon return from the stored proc.
 ****************************************************************

C     *entry        plist
C                   parm                    parm1             5 0
C                   parm                    parm2             5 0
C                   parm                    result            6 0
C                   parm                    state             5
```

```
C                   parm                    nullind

C                   dump

C                   eval      state = '00000'

 ****************************************************************
 * If you're going to allow nulls to be passed, you need to
 * test each parameter's null indicator.
 *
 * In this test, if a Null value is passed for the first
 * parameter, the value is set to a default of 0.
 *
 * Note the setting of the STATE field, which is meant to
 * represent a user-defined SQL STATE value.
 *
 * In the GENERAL or GENERAL WITH NULLS parameter styles,
 * the value set for STATE is a simple parameter (it is up
 * to the client program to handle the value).
 *
 * In the SQL or DB2SQL parameter styles, the value set
 * for SQL STATE is known to the database manager. If set,
 * an actual SQL warning or error is returned to the client
 * program.
 *
 * IBM documentation suggests the following values for
 * program-set SQL STATE:
 *
 *      00000: Successful execution, no errors
 *
 *      01Hxx: Warning. The "xx" can be any two digits or
 *             uppercase letters. Setting this as the
 *             SQL STATE results in SQLCODE 462.
 *
 *      38yxx: Error. "y" can be any letter or number.
 *             "xx" is any two digits or uppercase letters.
 *              Setting this as the SQL STATE results in
 *              SQLCODE -443.
 ****************************************************************

C                   if        nullind1 < 0
C                   eval      parm1 = 0
C                   eval      state = '01HNU'
C                   endif

 ****************************************************************
 * Same as above, set default for parm2 if Null was passed.
 ****************************************************************

C                   if        nullind2 < 0
C                   eval      parm2 = 0
C                   eval      state = '01HNU'
C                   endif

C                   eval      result = parm1 + parm2
C                   except    ex01

 ****************************************************************
 * When this program is called as a stored procedure, the values
 * for nullind3 and nullind4 are -1, indicating that the third
 * and fourth parameters values were Null when called.
 *
 * During the course of the program, values have been assigned
 * to the third and fourth parameters (result and state).
 *
 * To make those values available to the stored procedure client,
 * you need to set the corresponding null indicators to a non-
 * negative value. If you don't reset the null indicators,
 * the client would receive back a Null value for the parameters.
 ****************************************************************

C                   eval      nullind3 = 0
```

```
     C                   eval      nullind4 = 0

     C                   seton                                            lr

     Oqsysprt   e            ex01              1
     O                                              'SPGENNULL  '
     O                                              '  parm1: '
     O                       parm1
     O                                              '  parm2: '
     O                       parm2
     O                                              '  result: '
     O                       result
     O                                              '  state: '
     O                       state
```

Figure 6C: C# class SPGENNULL. This is the client program used to invoke the SPGENNULL external stored procedure.

```
using IBM.Data.DB2.iSeries;
using System;

/// <summary>
/// Tester for calling external stored procedure SPGENNULL
/// Parameter style GENERAL WITH NULLS
/// </summary>
class SPGENNULL     {

    static void Main(string[] args)     {

        //*************************************************************
        // A: get connection to server
        //*************************************************************
        Console.WriteLine("SPGENNULL(CS) - Parameter style GENERAL WITH
NULLS");
        Console.Write("Enter your iSeries host name: ");
        String hostName = Console.ReadLine();

        iDB2Connection cn = new iDB2Connection("DataSource=" + hostName);

        try {
            cn.Open();
        }
        catch (iDB2Exception ex) {
            HandleError("An error occurred on cn.Open()", ex);
            return;
        }

        //*************************************************************
        // B: define stored procedure - GENERAL WITH NULLS parameter style
        //*************************************************************
        String sp = "call sproclib.spgennull(@parm1, @parm2, @result,
@sqlstate)";

        iDB2Command cmd;

        try {

            //*********************************************************
            // C: set up command, derive parameters
            //*********************************************************
            cmd = new iDB2Command(sp, cn);
            cmd.CommandType = System.Data.CommandType.Text;
            cmd.DeriveParameters();

            //*********************************************************
            // D: parm1 - get value or set to null
            //*********************************************************
            Console.Write("Enter integer value for numeric parameter 1: ");
            String p1 = Console.ReadLine();

            if (p1.Length == 0) {
```

```
                cmd.Parameters["@parm1"].Value = DBNull.Value;
            }
            else {
                cmd.Parameters["@parm1"].Value = p1;
            }

            //*************************************************************
            // E: parm2 - get value or set to null
            //*************************************************************
            Console.Write("Enter integer value for numeric parameter 2: ");
            String p2 = Console.ReadLine();

            if (p2.Length == 0) {
                cmd.Parameters["@parm2"].Value = DBNull.Value;
            }
            else {
                cmd.Parameters["@parm2"].Value = p2;
            }

            //*************************************************************
            // F: no need to set values for the OUT parameters when
            // using GENERAL WITH NULLS. The null indicators for the
            // values is set on the call. The external stored procedure
            // program must reset the null indicators for the
            // OUT parameters to make their values available to
            // the client program.
            //*************************************************************

            //*************************************************************
            // G: run stored procedure, display output value
            //*************************************************************
            PrintParameters("Parameters - before call", cmd);

            cmd.ExecuteNonQuery();

            PrintParameters("Parameters - after call", cmd);

            Console.WriteLine("Return value: " + cmd.Parameters["@result"
].Value);
            Console.WriteLine("SQLSTATE: "     +
cmd.Parameters["@sqlstate"].Value);
        }

        catch (iDB2Exception ex) {
            HandleError("Error on Stored Procedure call", ex);
            return;
        }

        //*************************************************************
        // H: normal end-of-program processing
        //*************************************************************
        Console.WriteLine("Press ENTER to end");
        Console.ReadLine();

        cmd.Dispose();
        cn.Close();
    }

    //*************************************************************
    // Handle an error during program execution
    //*************************************************************
    static void HandleError(String        errorOccurred,
                            iDB2Exception ex) {

        Console.WriteLine(errorOccurred);
        Console.WriteLine("Source:         " + ex.Source);
        Console.WriteLine("SQLState:       " + ex.SqlState);
        Console.WriteLine("Message:        " + ex.Message);
        Console.WriteLine("MessageCode:    " + ex.MessageCode);
        Console.WriteLine("MessageDetails: " + ex.MessageDetails);
        Console.ReadLine();
    }
```

```
    //*************************************************************
    // Print parameters collection data
    //*************************************************************
    static void PrintParameters(String      message,
                                iDB2Command cmd) {

        Console.WriteLine("*********************************");
        Console.WriteLine(message);
        Console.WriteLine("*********************************");

        foreach(iDB2Parameter p in cmd.Parameters) {
            Console.WriteLine("Name:          " + p.ParameterName);
            Console.WriteLine("DbType:        " + p.DbType);
            Console.WriteLine("Direction:     " + p.Direction);
            Console.WriteLine("iDB2DbType:    " + p.iDB2DbType);
            Console.WriteLine("Precision:     " + p.Precision);
            Console.WriteLine("Scale:         " + p.Scale);
            Console.WriteLine("Size:          " + p.Size);
            Console.WriteLine("SourceColumn:  " + p.SourceColumn);
            Console.WriteLine("SourceVersion: " + p.SourceVersion);
            Console.WriteLine("ToString:      " + p.ToString());

            if (p.Value == DBNull.Value) {
                Console.WriteLine("Value:         DBNull.Value");
            }
            else {
                Console.WriteLine("Value:         " + p.Value);
            }

            Console.WriteLine("-------------------------------");
        }
    }
}
```

Figure 6V: Visual Basic module SPGENNULL. This is the client program used to invoke the SPGENNULL external stored procedure.

```
imports IBM.Data.DB2.iSeries

''' Tester for calling external stored procedure SPGENNULL
''' Parameter style GENERAL WITH NULLS
Module SPGENNULL

    Sub Main()

        '-------------------------------------------------------------
        ' A: get connection to server
        '-------------------------------------------------------------
        Console.WriteLine("SPGENNULL(VB) - Parameter style GENERAL WITH NULLS")
        Console.Write("Enter your iSeries host name: ")

        Dim hostName As String = Console.ReadLine()

        Dim cn As iDB2Connection = New iDB2Connection("DataSource=" & hostName)

        Try
            cn.Open()

        Catch ex As iDB2Exception
            HandleError("An error occurred on cn.Open()", ex)
            Return
        End Try

        '-------------------------------------------------------------
        ' B: define stored procedure - GENERAL WITH NULLS parameter style
        '-------------------------------------------------------------
        Dim sp As String = "call sproclib.spgennull(@parm1, @parm2, @result,
@sqlstate)"

        Dim cmd As iDB2Command
```

```
        Try
            '-------------------------------------------------------
            ' C: set up command, derive parameters
            '-------------------------------------------------------
            cmd = New iDB2Command(sp, cn)
            cmd.CommandType = System.Data.CommandType.Text
            cmd.DeriveParameters()

            '-------------------------------------------------------
            ' D: parm1 - get value or set to default
            '-------------------------------------------------------
            Console.Write("Enter integer value for numeric parameter 1: ")
            Dim p1 As String = Console.ReadLine()

            If (p1.Length = 0) Then
                cmd.Parameters("@parm1").Value = DBNull.Value
            Else
                cmd.Parameters("@parm1").Value = p1
            End If

            '-------------------------------------------------------
            ' E: parm2 - get value or set to default
            '-------------------------------------------------------
            Console.Write("Enter integer value for numeric parameter 2: ")
            Dim p2 As String = Console.ReadLine()

            If (p2.Length = 0) Then
                cmd.Parameters("@parm2").Value = DBNull.Value
            Else
                cmd.Parameters("@parm2").Value = p2
            End If

            '-------------------------------------------------------
            ' F: no need to set values for the OUT parameters when
            ' using GENERAL WITH NULLS. The null indicators for the
            ' values is set on the call. The external stored procedure
            ' program must reset the null indicators for the
            ' OUT parameters to make their values available to
            ' the client program.
            '-------------------------------------------------------

            '-------------------------------------------------------
            ' G: run stored procedure, display output value
            '-------------------------------------------------------
            PrintParameters("Parameters - before call", cmd)

            cmd.ExecuteNonQuery()

            PrintParameters("Parameters - after call", cmd)

            Console.WriteLine("Return value: " & cmd.Parameters("@result"
).Value)
            Console.WriteLine("SQLSTATE:     " &
cmd.Parameters("@sqlstate").Value)

        Catch ex As iDB2Exception
            HandleError("Error on Stored Procedure call", ex)
            Return
        End Try

        '-------------------------------------------------------
        ' H: normal end-of-program processing
        '-------------------------------------------------------
        Console.WriteLine("Press ENTER to end")
        Console.ReadLine()

        cmd.Dispose()
        cn.Close()

    End Sub
```

```
    '-------------------------------------------------------------
    ' Handle an error during program execution
    '-------------------------------------------------------------
    Sub HandleError(errorOccurred As String, _
                    ex            As iDB2Exception)

        Console.WriteLine(errorOccurred)
        Console.WriteLine("Source:         " & ex.Source)
        Console.WriteLine("SQLState:       " & ex.SqlState)
        Console.WriteLine("Message:        " & ex.Message)
        Console.WriteLine("MessageCode:    " & ex.MessageCode)
        Console.WriteLine("MessageDetails: " & ex.MessageDetails)
        Console.ReadLine()

    End Sub

    '-------------------------------------------------------------
    ' Print parameters collection data
    '-------------------------------------------------------------
    Sub PrintParameters(message As String, _
                        cmd     As iDB2Command)

        Console.WriteLine("****************************************")
        Console.WriteLine(message)
        Console.WriteLine("****************************************")

        For Each p As iDB2Parameter In cmd.Parameters

            Console.WriteLine("Name:          " & p.ParameterName)
            Console.WriteLine("DbType:        " & p.DbType)
            Console.WriteLine("Direction:     " & p.Direction)
            Console.WriteLine("iDB2DbType:    " & p.iDB2DbType)
            Console.WriteLine("Precision:     " & p.Precision)
            Console.WriteLine("Scale:         " & p.Scale)
            Console.WriteLine("Size:          " & p.Size)
            Console.WriteLine("SourceColumn:  " & p.SourceColumn)
            Console.WriteLine("SourceVersion: " & p.SourceVersion)
            Console.WriteLine("ToString:      " & p.ToString())

            If (p.Value Is DBNull.Value) Then
                Console.WriteLine("Value:           DBNull.Value")
            Else
                Console.WriteLine("Value:           " & p.Value.ToString())
            End If

            Console.WriteLine("---------------------------------------")
        Next
    End Sub

End Module
```

Figure 7: RPG Dump output from SPGENNULL. PARM1 and PARM2 passed with values, RESULT and STATE passed as null values

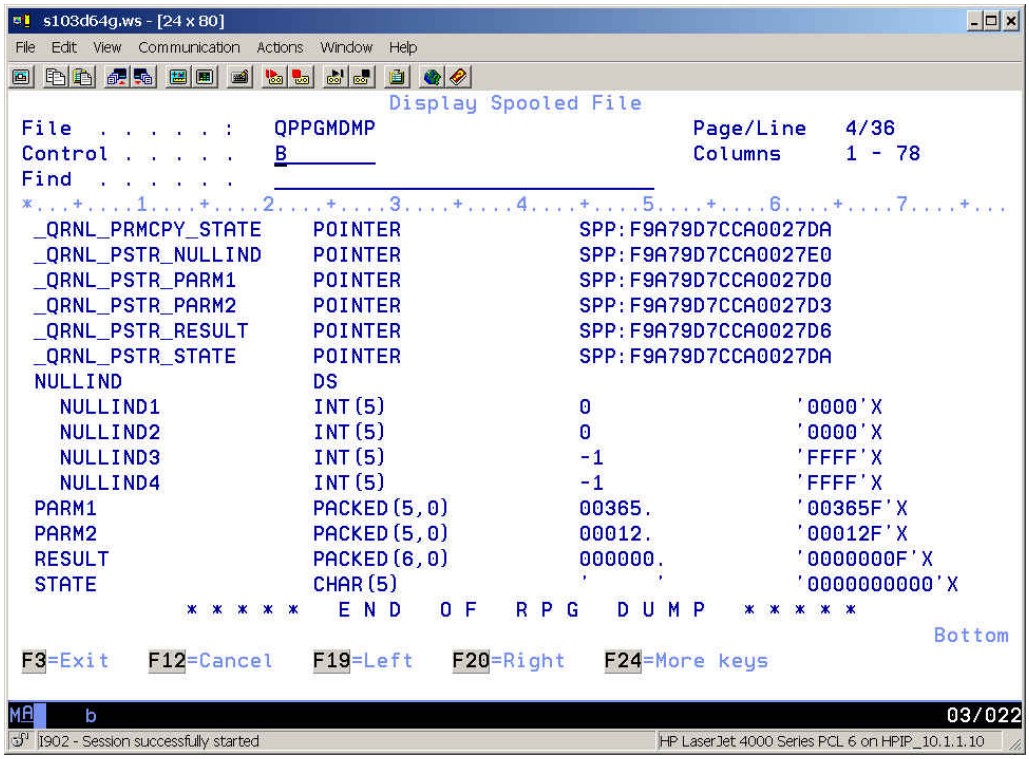

Figure 8: RPG Dump output from SPGENNULL. PARM1 passed with value, null value passed for PARM2, RESULT and STATE passed as null values.