Writing Microsoft SQL Server Extended Stored Procedures

by Berend de Boer

Microsoft SQL Server has the powerful capability to make functions in DLLs available as stored procedures. Microsoft calls them *Extended Stored Procedures*.

In this article I'll describe how to write Extended Stored Procedures and present a nice object-oriented framework which makes the task extremely easy. I'll describe some example programs created with this framework, and finally I'll tell you how to install Extended Stored Procedures in SQL Server.

What Are Extended Stored Procedures?

Extended Stored Procedures (I'll call them *XPs* from now on) are part of Microsoft's Open Data Services (ODS) for SQL Server. With ODS you can do three things.

Firstly, you can make routines in a DLL available as stored procedures to any SQL Server user. Technically this DLL is part of SQL Server, therefore programmer errors may corrupt your SQL Server. Writing such XPs is the subject of this article.

Secondly, you can write procedure server applications. These are similar to XPs, but run as separate network server applications, they could even be running on a different machine (for a 3-tier architecture). I won't discuss them in this article.

Lastly, you can write gateways to non-SQL Server based environments. I won't discuss these either.

Making parts of your application available on the server has some advantages. Some things are easy to write in Delphi, but difficult, or even impossible, using SQL Server's Transact SQL. Delphi routines can run much faster than Transact SQL, which can be important for complex numerical calculations, for example. You can also make interfaces with other programs, databases and so on. For example, you could write an XP that accepts the name of a Paradox table and returns the contents of the table as a SQL Server result set.

1	Check that the caller of the procedure has provided all of the required parameters and that each parameter is of the appropriate datatype. Return an appropriate message if not.
2	Define the columns for returning a result set.
3	Create each record for returning to the caller.
4	Set up any output parameters and return status data used by the procedure.
5	When finished returning results, send the results completion message using srv_senddone with the SRV_DONE_MORE status flag.
6	Return from the procedure with the desired Transact-SQL return status.

➤ *Table 1: Tasks for Extended Stored Procedures*

Writing Extended Stored Procedures

XPs live in DLLs and can therefore be written in any language which can produce DLLs, which of course includes Delphi. Before going into detail about how to write XPs, first let's consider some examples from a user's point of view.

Let's assume we have an XP called `xp_incbyone1` which increments a given number by one. We can call `xp_incbyone1` as follows:

```
declare @mynumber integer
select @mynumber = 1
exec master..xp_incbyone1 @mynumber
  output
select @mynumber
```

The `declare` statement declares a variable `@mynumber` of type `integer`. Next we set it to one, pass it to the XP and allow the XP to modify it by appending `output` to the parameter. Finally we display the number with a `select` statement, to see if it has been updated. The result should be 2 of course.

In the following example we have an XP which returns an output parameter. XPs can also return a result set. The example `xp_incbyone2` returns the number as a result set. The code to call it would be:

```
declare @mynumber integer
select @mynumber = 1
exec master..xp_incbyone2 @mynumber
```

`xp_incbyone2` will return a table of just one column and one row containing the value 1. Both `xp_incbyone1` and `xp_incbyone2` are described in detail in the next section, where I present the Delphi framework.

Each implementation of an XP needs to do the same things: see Table 1 above, which is taken from the *Microsoft SQL Server Programmers' Toolkit*. Step 1 is necessary because, unless you are using normal SQL stored procedures, it is up to the programmer to validate any user-specified parameters for XPs. Steps 2 and 3 are optional and are applicable only if you return a result set. Step 4 is also optional and applies only if you return output parameters.

So, now let's see how to make the whole thing a lot easier...

The Framework

The C programmer who wants to develop XPs needs to install the Microsoft BackOffice resource kit, which contains all the required header files and demonstration programs. Unfortunately, Borland has not supplied a translation of these header files with Delphi. Therefore I had to translate the most important parts by hand into Pascal. This means that the BackOffice resource kit is not needed if you use my framework to write your own XPs. If you want to add more pieces, though, you will need the resource kit. Or you could always ask me nicely and if I've time I might expand the framework to cover the missing pieces!

Look again at Table 1 for the tasks an XP should do. The framework makes Steps 1 through 4 quite a lot easier by providing Delphi specific type conversions, Steps 5 and 6 are automatically done by the framework.

To use the framework, first create an object of class `TSQLXProc` and implement its `Execute` method. Then write a procedure that allocates this object, calls its `Run` method and frees the object. The name of this procedure should be the same as the name of your extended stored procedure.

To make this process more concrete, let's implement the `xp_incbyone1` stored procedure. The first step is to create a new object based on `TSQLXProc` and implement its `Execute` method. Its header looks like this:

```
type
  TXXPIncByOne1 = class(TSQLXProc)
    function Execute: Boolean;
      override;
  End;
```

The `Execute` method looks like this:

```
function TXPIncByOne1.Execute:
  Boolean;
begin
  Params[1] := Params[1] + 1;
  Result := True;
End;
```

The second step is to write a procedure that calls this object. This is the procedure that SQL Server is actually calling. For `xp_incbyone1` it looks like Listing 1. It's that easy!

```
function xp_incbyone1(srvproc: PSRV_PROC): SRVRETCODE;
const
  ExpectedParams = 1;
var
  xp: TSQLXProc;
begin
  xp := TXPIncByOne1.Create(srvproc, ExpectedParams);
  Result := xp.Run;
  xp.Free;
end;
```

➤ *Listing 1*

ODS Constant	Transact SQL Datatype(s)	Delphi Datatype(s)
SRVVARCHAR	varchar	string
SRVCHAR	char	string
SRVINTN	Tinyint, smallint, int	shortint,smallint,integer
SRVBIT	bit	Boolean
SRVDECIMAL	numeric/decimal	n/a (string)
SRVNUMERIC	numeric/decimal	n/a (string)
SRVFLTN	Real, float	Single, double
SRVMONEYN	Smallmoney, money	n/a (integer, DBMONEY)
SRVDATETIMN	smalldatetime, datetime	TDateTime

➤ *Table 2: Supported types for use with DescribeColumn*

Let's look in more detail at the first step. The only thing you'll ever need to do is implement the `Execute` method. This function returns `True` or `False`. If `False` is returned, an error is returned to the calling application or user. Exceptions are caught by the code that calls your `Execute` method and a similar error is also returned to the calling application or user.

You can get access to the parameters of a stored procedure by using the `Params` property. Parameters are numbered from one upwards. As noted earlier, SQL Server does no type checking on XP parameters. The framework returns parameters as variants, so it's a bit more robust against different parameters, but variant conversion errors may occur if a parameter type does not match. You might want to use the ODS API call `srv_paramtype` to explicitly retrieve and check parameter types, but so far I've not found a need for this. Another solution for checking parameter types is to use the standard `VarType` function.

See Table 2 for a list of Transact SQL datatypes and corresponding Delphi datatypes.

If a parameter is `Null`, the `Params` property returns the variant type `Null`. Equally, if you want to return Null, set the corresponding parameter in `Params` to `Null`.

Let's now look in more detail at the second step. This will probably always be the same, except for the value of the `ExpectedParams` constant and the particular object to instantiate. This procedure is called by SQL Server with one parameter: `srvproc`. We pass this parameter to the instantiated object, along with the number of parameters which it should expect. If the actual number of parameters is different to this an error message will be sent back to the calling application or user. Pass zero if you don't want to check for the number of parameters, for example to support a variable number of parameters.

```
function TXPIncByOne2.Execute: Boolean;
var
  myint: integer;
begin
  DescribeColumn('my column name', SRVINT4, 4, SRVINT4, 4, @myint);
  Myint := Params[1] + 1;
  SendRow;
  Result := True;
end;
```

➤ *Listing 2*

```
function xp_incbyone2(srvproc: PSRV_PROC): SRVRETCODE;
const
  ExpectedParams = 1;
var
  xp: TSQLXProc;
begin
  xp := TXPIncByOne2.Create(srvproc, ExpectedParams);
  Result := xp.Run;
  xp.Free;
end;
```

➤ *Listing 3*

```
function TXPDiskList.Execute: Boolean;
var
  drivename: char;
  space_remaining: Int32;
  drivenums: Int32;
  rootname: string;
  SectorsPerCluster, BytesPerSector,
    NumberOfFreeClusters, TotalNumberOfClusters: dword;
  function IsDrive(drive: char): Boolean;
  begin
    IsDrive := (drivenums and (1 shl (Ord(drive) - Ord('A')))) <> 0;
  end;
begin
  DescribeColumn('drive', SRVCHAR, 1, SRVCHAR, 1, @drivename);
  DescribeColumn('bytes free', SRVINT4, 4, SRVINT4, 4, @space_remaining);
  drivenums := GetLogicalDrives;
  for drivename := 'C' to 'Z' do  begin
    if IsDrive(drivename) then begin
      rootname := drivename + ':\';
      GetDiskFreeSpace(PChar(rootname), SectorsPerCluster, BytesPerSector,
        NumberOfFreeClusters, TotalNumberOfClusters);
      space_remaining :=
        SectorsPerCluster * NumberOfFreeClusters * BytesPerSector;
      SendRow;
    end;
  end;
  Result := True;
end;
```

➤ *Listing 4*

Next we call the `Run` method of the instantiated object, which in turn will call our `Execute` method (surrounded by, among other things, a `try..except` block). Finally we free the object.

Result Sets

Now let's tackle an XP which returns a result set. Its header is:

```
type
  TXPIncByOne2 = class(TSQLXProc)
    function Execute: Boolean;
      override;
End;
```

and its body is shown in Listing 2. The procedure to call this object is shown in Listing 3.

We now have a slightly more complicated `Execute` method. In case we want to return a result set, we need to describe every row in the resulting table: its column name, destination type, destination length, source type, source length and a pointer to the source data. You should call `DescribeColumn` for every column in the result table.

The next step is to fill the source data: that's the assignment to `myint`. The row is now complete, so we can send it to SQL Server using `SendRow`. You should prepare source data and call `SendRow` for every row in the result table. And finally you just return `True` and `Exit`. After that SQL Server will send the entire result table to the client.

The `xp_incbyone2` procedure is still a simple process of calling the object and exiting. In the remaining examples I will omit this part.

More Examples

I implemented two of the sample XPs which Microsoft implemented in their xp.c source file. The first one simply copies the contents of the first parameter to the second parameter. The second example returns the free space from every drive available on the SQL Server computer. To avoid name clashes I called the first XP `xp_delphiecho` instead of `xp_echo`. The second one is called `xp_delphidisklist` instead of `xp_disklist`. I think `xp_echo` especially looks much more elegant than Microsoft's sample program: you really should have a look at xp.c!

The code for `xp_delphiecho` is simply:

```
function TXPEcho.Execute:
  Boolean;
begin
  Params[2] := Params[1];
  Result := True;
End;
```

and the code for `xp_delphidisklist` is shown in Listing 4.

In the first two lines of Listing 4 the description of the result table is given, which consists of two columns: `drive` and `bytes free`. Next, for every drive we fill the variables `drivename` and `space_remaining` and send back the row using `SendRow`.

Compiling The DLL

The DLL containing the Extended Stored Procedures is just a normal DLL, nothing special. So you should start with the `library` keyword and export the procedures which instantiate the framework object (see the sample program XPDELPHI.DPR on the disk).

Installing XPs On SQL Server

All of the material in this section can also be found in the Microsoft *SQL Programmers Toolkit* or in the Microsoft Transact-SQL reference.

When you have compiled your DLL you have to install it in the appropriate directory. Copy the file to the same directory as the standard SQL Server DLL files. Usually this directory is something like C:\MSSQL\BINN, note 'BINN' with two n's not the BIN directory with a single n which also exists!

As with other DLLs, once the Extended Stored Procedure DLL is placed in the appropriate directory and the appropriate paths are set, you can make its functions available to users immediately: it is not necessary to restart the server.

For each function provided in an Extended Stored Procedure DLL, a SQL Server system administrator must run the sp_addextendedproc system procedure, specifying the name of the function and the name of the DLL in which that function resides. For example:

```
Sp_addextendedproc
  'xp_delphiecho', 'xpdelphi.dll'
```

This command registers the function xp_delphiecho, located in the file xpdelphi.dll, as a SQL Server Extended Stored Procedure. You must run sp_addextendedproc in the master database. To drop individual extended stored procedures, a system administrator uses the system procedure sp_dropextendedproc.

SQL Server loads an Extended Stored Procedure DLL as soon as a call is made to one of the DLL's functions. The DLL remains loaded until the server is shut down or until the system administrator uses the DBCC command to unload it. For example:

```
DBCC xpdelphi(FREE)
```

unloads XPDELPHI.DLL, allowing the system administrator to copy in a newer version of this file without shutting down the server. You probably will need this command quite a lot to debug your XPs!

Once a system administrator has added an Extended Stored Procedure, users can find out what new functions are available by using the system procedure sp_helpextendedproc. When used without an argument, sp_helpextendedproc displays all Extended Stored Procedures that are currently registered with the master database. If you specify an Extended Stored Procedure name as an argument, sp_helpextendedproc checks if that function is currently available.

Extended Stored Procedures are subject to the same security mechanisms as regular stored procedures. For example, to give every right to the xp_delphiecho xp, run the following commands in the master database:

```
grant exec on xp_delphiecho
  to public
go
```

Every user would then be able to call xp_delphiecho from every database by prefixing xp_delphiecho with master. For example, to call xp_delphiecho from the pubs database you say:

```
exec master..xp_delphiecho
  @paramin, @paramout output
```

Conclusions

Several source files are provided on this month's disk. XPNUIS.DPR is a sample DLL project containing all the XPs described in this article. XPNUIS.SQL is a script to add all the XPs in XPNUIS.DLL to the master database. Finally, ODSXP.PAS is the unit containing my framework.

Now that writing Microsoft SQL Server Extended Stored Procedures is a breeze I'll look forward to hearing from you about the interesting applications you have developed using this package!

Berend de Boer is President of Nederware, a software engineering firm in the Netherlands, and can be contacted by email at berend@pobox.com