## uspGetBillOfMaterials

The stored procedure `uspGetBillOfMaterials` is designed to generate a multi-level Bill of Materials (BOM) for a given product. The BOM is a comprehensive list of raw materials, components, and assemblies required to construct, manufacture or repair a product or service.   
  
The procedure takes two parameters: `@StartProductID` and `@CheckDate`. The `@StartProductID` is the ID of the product for which the BOM is to be generated. The `@CheckDate` is used to filter out any components that are no longer used in the product on this date.  
  
The procedure uses a recursive Common Table Expression (CTE) named `BOM\_cte` to generate the BOM. The CTE starts with the initial list of components for the product assembly (`@StartProductID`) where the `@CheckDate` is between the start and end dates of the component's usage. This is the anchor member of the CTE.  
  
The recursive member of the CTE then joins the `BillOfMaterials` table with the `BOM\_cte` on the `ProductAssemblyID` and `ComponentID` fields, effectively getting the next level of components for each component in the previous level. This continues until all levels of components have been retrieved or the maximum recursion level (25) is reached.  
  
The final SELECT statement from the CTE groups the components by their ID, description, product assembly ID, BOM level, recursion level, standard cost, and list price, and calculates the total quantity of each component. The results are then ordered by BOM level, product assembly ID, and component ID.  
  
In summary, this stored procedure provides a detailed breakdown of all the components and their quantities required to build a specific product, taking into account the date of check to ensure only relevant components are included.

## uspGetEmployeeManagers

The stored procedure `uspGetEmployeeManagers` is designed to retrieve a list of managers for a specific employee in a hierarchical organization structure. The procedure takes a single parameter, `@BusinessEntityID`, which is the unique identifier of an employee.  
  
The procedure uses a recursive Common Table Expression (CTE) named `EMP\_cte` to traverse the organization hierarchy. The CTE starts with the employee specified by `@BusinessEntityID` and then recursively joins to the `Employee` table to find each successive manager in the hierarchy. The `OrganizationNode` field, which represents the position of an employee in the organization hierarchy, is used to find the manager of each employee (the manager is the immediate ancestor of the employee in the hierarchy).  
  
The CTE includes the following fields: `BusinessEntityID`, `OrganizationNode`, `FirstName`, `LastName`, `JobTitle`, and `RecursionLevel`. The `RecursionLevel` field is used to track the level of recursion, with 0 representing the initial employee.  
  
The final SELECT statement joins the CTE back to the `Employee` and `Person` tables to return the list of managers for the specified employee. The result set includes the recursion level, the business entity ID, first name, and last name of each manager, and the organization node of each manager.  
  
The `OPTION (MAXRECURSION 25)` statement limits the maximum recursion level to 25 to prevent infinite loops in case of a circular reference in the organization hierarchy.  
  
In terms of business logic, this procedure could be used in a human resources application to display the management chain for an employee, or to determine the reporting structure within the organization.

## uspGetManagerEmployees

The stored procedure `uspGetManagerEmployees` is designed to retrieve a list of employees who report directly or indirectly to a specific manager. The manager is identified by their `BusinessEntityID`, which is passed as a parameter to the procedure.  
  
The procedure uses a recursive Common Table Expression (CTE) named `EMP\_cte` to traverse the organization hierarchy. The CTE starts with the manager (the anchor member) and then recursively includes all employees who report to the manager or to any of the manager's subordinates (the recursive member). The `RecursionLevel` column in the CTE indicates the level of each employee in the hierarchy, with 0 being the manager.  
  
The procedure then joins the CTE with the `Employee` and `Person` tables to return the following information for each employee: `RecursionLevel`, `OrganizationNode`, `ManagerFirstName`, `ManagerLastName`, `BusinessEntityID`, `FirstName`, and `LastName`. The `OrganizationNode` column represents the position of the employee in the organization hierarchy.  
  
The results are ordered by `RecursionLevel` and `OrganizationNode`, and the maximum recursion level is set to 25 to prevent infinite loops in case of a circular hierarchy.  
  
In terms of business logic, this procedure could be used to generate an organization chart or to determine the reporting structure within the company.

## uspGetWhereUsedProductID

The stored procedure `uspGetWhereUsedProductID` is designed to retrieve a multi-level Bill of Materials (BOM) for a given product ID and a specific date.   
  
The procedure takes two parameters: `@StartProductID` and `@CheckDate`. The `@StartProductID` is the ID of the product for which the BOM is to be generated. The `@CheckDate` is the date for which the BOM is to be generated.   
  
The procedure uses a recursive Common Table Expression (CTE) named `BOM\_cte` to generate the BOM. The CTE starts by selecting the initial list of components for the product assembly from the `Production.BillOfMaterials` and `Production.Product` tables where the `ComponentID` equals the `@StartProductID` and the `@CheckDate` is within the `StartDate` and `EndDate` of the component.   
  
The CTE then recursively joins the `Production.BillOfMaterials` and `Production.Product` tables on the `ProductAssemblyID` and `ComponentID` to get the next level of components. The recursion continues until all levels of components have been retrieved or the maximum recursion level of 25 is reached.  
  
The final result of the procedure is a list of components for the product assembly, grouped by `ComponentID`, `ComponentDesc`, `ProductAssemblyID`, `BOMLevel`, `RecursionLevel`, `StandardCost`, and `ListPrice`. The list is ordered by `BOMLevel`, `ProductAssemblyID`, and `ComponentID`.  
  
The business logic of the procedure is to provide a detailed breakdown of the components that make up a product assembly, including the quantity, standard cost, and list price of each component. This information can be used for inventory management, cost calculation, and production planning.

## uspUpdateEmployeeHireInfo

The stored procedure "uspUpdateEmployeeHireInfo" is used to update an employee's hire information and log their pay history in the HumanResources database.  
  
The procedure takes seven parameters: BusinessEntityID, JobTitle, HireDate, RateChangeDate, Rate, PayFrequency, and CurrentFlag.   
  
The BusinessEntityID is used to identify the specific employee in the database. The JobTitle, HireDate, and CurrentFlag are used to update the corresponding fields in the Employee table for the specified employee.  
  
The procedure then inserts a new record into the EmployeePayHistory table with the BusinessEntityID, RateChangeDate, Rate, and PayFrequency. This effectively logs the pay history for the employee.  
  
The procedure is wrapped in a transaction to ensure data integrity. If any part of the transaction fails (either the update or the insert), the entire transaction is rolled back to prevent partial updates. This is done in the catch block where it also calls another stored procedure "uspLogError" to log the error.  
  
The "SET NOCOUNT ON;" statement at the beginning of the procedure is used to stop the message indicating the number of rows affected by a SQL statement from being returned as part of the result set.  
  
The "WITH EXECUTE AS CALLER" clause specifies that the procedure will be executed in the context of the caller, meaning it will have the same permissions as the user who calls it.

## uspUpdateEmployeeLogin

The stored procedure "uspUpdateEmployeeLogin" in the "HumanResources" schema is used to update an employee's information in the "Employee" table. The procedure takes six parameters: BusinessEntityID, OrganizationNode, LoginID, JobTitle, HireDate, and CurrentFlag.  
  
The BusinessEntityID is used to identify the specific employee record to be updated. The other parameters (OrganizationNode, LoginID, JobTitle, HireDate, and CurrentFlag) are the new values that will replace the existing values in the employee's record.  
  
The procedure is executed with the privileges of the caller, not the owner of the procedure. The SET NOCOUNT ON command is used to stop the message that shows the count of rows affected by a Transact-SQL statement or stored procedure from being returned as part of the result set.  
  
The procedure uses a TRY...CATCH construct to handle errors that occur during the execution of the procedure. If an error occurs, the procedure "uspLogError" in the "dbo" schema is executed to log the error.  
  
The business logic is straightforward: it updates an employee's information based on the provided parameters. The CurrentFlag parameter might be used to indicate whether the employee is currently employed or not, but without more context, this is just a guess.

## uspPrintError

The stored procedure "uspPrintError" is designed to print out detailed information about any errors that occur during the execution of a SQL Server database operation. It is intended to be called from within the CATCH block of a TRY...CATCH construct, which is a common way to handle errors in SQL Server.  
  
The procedure begins by setting NOCOUNT to ON, which stops the message that shows the count of the number of rows affected by a Transact-SQL statement or stored procedure from being returned as part of the result set.  
  
Then, it uses the PRINT statement to output a string that includes the following information about the error:  
  
- Error number: This is the unique identifier for the error. It is converted to a string using the CONVERT function.  
- Severity: This indicates the type or level of the error. It is also converted to a string.  
- State: This provides detailed information about the error. It is also converted to a string.  
- Procedure: This is the name of the stored procedure or trigger where the error occurred. If this information is not available, a dash ("-") is printed instead.  
- Line: This is the line number in the SQL batch or stored procedure where the error occurred. It is converted to a string.  
  
Finally, it prints the error message associated with the error.  
  
There is no specific business logic in this procedure; it is a general utility for error handling in SQL Server.

## uspUpdateEmployeePersonalInfo

The stored procedure "uspUpdateEmployeePersonalInfo" is designed to update an employee's personal information in the "Employee" table of the "HumanResources" database.   
  
The procedure takes five parameters:   
  
1. @BusinessEntityID: This is an integer that uniquely identifies an employee in the database.  
2. @NationalIDNumber: This is a string of up to 15 characters that represents the employee's national ID number.  
3. @BirthDate: This is a datetime value that represents the employee's birth date.  
4. @MaritalStatus: This is a single character that represents the employee's marital status.  
5. @Gender: This is a single character that represents the employee's gender.  
  
The procedure executes with the permissions of the caller, and it turns off the count of rows affected by the SQL statements within the procedure.  
  
The procedure tries to update the employee's national ID number, birth date, marital status, and gender in the "Employee" table where the BusinessEntityID matches the @BusinessEntityID parameter.  
  
If there is an error during the execution of the update statement, the procedure catches the error and calls another stored procedure "uspLogError" in the "dbo" schema to log the error.   
  
There is no explicit business logic in this procedure, it simply updates the database with the provided information. The business logic would be implemented in the code that calls this procedure, which would determine when and how to update the employee's information.

## uspLogError

The stored procedure `uspLogError` is designed to log error information in the `ErrorLog` table. This procedure is intended to be executed within a CATCH block of a TRY...CATCH construct, as it logs the error that caused the execution to jump to the CATCH block.  
  
The procedure takes an output parameter `@ErrorLogID` which will contain the ID of the row inserted into the `ErrorLog` table. If no error information is logged, the output parameter value will be 0.  
  
The procedure first checks if there is any error information to log. If there is no error, it returns without doing anything.   
  
Next, it checks the transaction state. If the transaction is in an uncommittable state, it prints a message and returns without logging the error. This is because data insertion/modification is not allowed when a transaction is in an uncommittable state.  
  
If there is an error and the transaction state is committable, it inserts a new row into the `ErrorLog` table. The row contains the username, error number, error severity, error state, error procedure, error line, and error message. The username is obtained by converting the current user to sysname. The error information is obtained from the system functions `ERROR\_NUMBER()`, `ERROR\_SEVERITY()`, `ERROR\_STATE()`, `ERROR\_PROCEDURE()`, `ERROR\_LINE()`, and `ERROR\_MESSAGE()`.  
  
After inserting the row, it sets the `@ErrorLogID` output parameter to the ID of the inserted row.  
  
If an error occurs while executing the procedure, it catches the error, prints a message, executes the `uspPrintError` procedure to print the error information, and returns -1.

## uspSearchCandidateResumes

The stored procedure `uspSearchCandidateResumes` is designed to perform a full-text search on the `JobCandidate` table in the `HumanResources` schema of a database. The procedure takes four parameters:  
  
1. `@searchString`: The text string to search for in the resumes.  
2. `@useInflectional`: A boolean flag indicating whether to use inflectional forms of the search string (e.g., run, runs, running).  
3. `@useThesaurus`: A boolean flag indicating whether to use thesaurus expansions of the search string (e.g., synonyms).  
4. `@language`: The language code identifier (LCID) for the language to use in the search. If this parameter is not provided or is zero, the procedure uses the default language of the SQL Server instance.  
  
The procedure constructs a search query based on the provided parameters and executes it against the `JobCandidate` table. The query uses either the `FREETEXTTABLE` or `CONTAINSTABLE` function, depending on the parameters, to return a table of `JobCandidateID` and `RANK` values. The `RANK` value indicates the relevance of each result to the search string, with higher values indicating greater relevance.  
  
The business logic of the procedure is to provide a flexible and powerful search capability for candidate resumes, allowing for variations in language use and synonyms. This can help HR personnel or recruiters to find suitable candidates based on specific keywords or phrases in their resumes.