RESOURCE AND PATIENT MANAGEMENT SYSTEM

BMXNet20
ADO.NET Data Access and Connectivity Utilities for RPMS
(BMX)

User Manual

Version 2.0
March 2007

Office of Information Technology (OIT)
Division of Information Resource Management
Albuquerque, New Mexico

PREFACE

This manual provides information regarding only the release and installation of the BMX Version 2.0 package. BMXNet.dll supports .Net 1.0 framework. This manual has been updated to support BMXNet20.dll which is a rewrite of BMX supporting the .Net 2.0 framework.

TABLE OF CONTENTS

1.0 Introduction

BMXNet was released previously to support Windows applications written using the *.Net 1.0* and *1.1* framework. *BMXNet20.dll* represents the conversion of BMXNet to the .Net 2.0 framework.

BMXNet20 is a set of software utilities designed to enable Windows applications written using the *.NET* framework to exchange data with RPMS using standard ADO.NET tables. Specifically, BMXNet20 is an ADO.NET Data Provider for RPMS.

BMXNet20 is intended to be used by Windows software developers who are writing .NET applications based on data contained in the RPMS clinical repository. BMXNet20 may be used by any kind of .NET application, including rich client, web services or ASP.NET.

ADO.NET defines a set of classes that enable .NET programmers to use data access services. ADO.NET is the product of the evolution of earlier legacy technologies such as ADO, OLEDB and ODBC.

Note: All BMXNet Classes are contained in the .NET namespace *IndianHealthService.BMXNet.*

Related Information
Detailed information about .NET and ADO.NET can be found in the references listed in Section 7.0 of this document.

2.0 Orientation

The following conventions are used throughout this manual:

Code Text
Text specific to actual code is offset by a gray background with a black border, as shown in *Figure 2-1*.

```
    ci.LoadConnectInfo(“127.0.0.1”, 9200);
```

Figure 2-1: Code text convention

Syntax Text
Text specific to SQL syntax includes an example with callouts. A table following each example explains the associated callouts. Components that make up a specific argument are indicated with a callout including the entire argument, as well as callouts for each component. For instance, in *Figure 2-2*, *Patient.Name*, *VA_Patient.DOB*, *‘Date of Birth’, and Patient.Age* are all components of the collective select_list argument. Therefore, B refers to the select_list argument, while C, D, and E refer to the components of that argument.

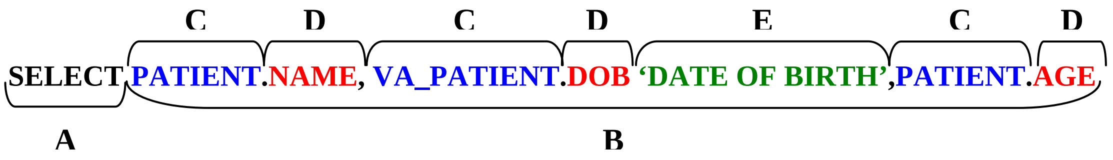

C
D
C
D
E
C
D
SELECT PATIENT.NAME, VA_PATIENT.DOB ‘DATE OF BIRTH’,PATIENT.AGE
A
B

Figure 2-2: Syntax text convention

Section	Syntax	Description
A	SELECT	Specifies that the system should return the values by the following select_list statement.
B	select_list	Specifies which fields (columns) the system should select for the returned set. The select_list a series of expressions separated by commas.
C	table_name	Specifies from which files (tables) the FROM clause should return fields.
D	field_name	Specifies from which field (column) the FROM clause should return. The field_name is associated with the preceding table_name by using a dot (.) between the two.
E	column_alias	Specifies an alternative name to replace the column name in the query result set. For example, an alias such as “Quantity”, “Quantity to Date”, or “Qty” can be specified for a column named **quantity**

Table 2-1: Syntax component convention

3.0 Managing BMXNet Monitors

Before data can be exchanged using BMXNet, the BMXNet Monitor software must be running on the RPMS server. Management of BMXNet Monitors is accomplished via BMXNet Menu options in RPMS as shown in the following example:

```
   EDIT   Add/Edit BMXNet Monitor Entries
   STRT   Start All BMXNet Monitors
   STOP   Stop All BMXNet Monitors
```

Figure 3-1: Sample of BMXNet menu options

Details about each BMXNet Monitor process are stored in the BMXNET MONITOR file which may be edited by selecting the EDIT option in the BMXNet Menu as shown in the following example:

```
Select BMXNET MONITOR PORT: 10501
PORT: 10501//
ENABLED: YES//
INTEGRATED SECURITY: YES//
SESSION NAMESPACE:
```

Figure 3-2: Sample of BMXNet edit option

The PORT determines the TCP/IP port on which the BMXNet monitor will listen for client connections. Be sure that other TCP/IP listeners, such as those in the XWB, CIA and BGU namespaces are not set up to run on the same port as any of the BMXNet Monitor entries.

The ENABLED field can be used to switch a monitor entry on or off. The switch will not take effect until BMXNet monitors are restarted.

INTEGRATED SECURITY designates whether users will be authenticated using Windows NT security. If this field is YES, then users connecting on this monitor will be prompted only once for an Access and Verify code. Thereafter, BMXNet will automatically log the user in to RPMS based on the user’s Windows NT login. If INTEGRATED SECURITY is NO, then users will be prompted for Access and Verify code at each connection.

SESSION NAMESPACE designates the namespace in which the client sessions will execute. If this field is null, client sessions will execute in the same namespace as the BMXNet Monitor is running.

The option BMX MONITOR START should be scheduled with TaskMan to run at system startup. When this option is scheduled, then all enabled BMXNet Monitors will be automatically started when TaskMan starts.

3.1 Initial Monitor Setup

Monitor Setup in Multi Name-spaced Environments
A new feature of BMXNet20.dll supports the use of just one BMXMON listener running in a multi name-spaced environment. When an application using the BMXNet20.dll connects, the connection and access/verify process will attempt to job a BMX handler in the server namespace defined. If the Server Namespace field is empty, the current namespace is implied. The handler will be jobbed in the same namespace in which the BMXMON listener is running. If the Server Namespace has a value, a login will be attempted based on the settings of the BMXMON monitor settings in the current namespace. That is, if the primary BMXMON supports INTEGRATED SECURITY, the jobbed namespace will try and validate the user, using the WINID instead of the access/verify process *in the namespace* defined by the Client Server Namespace field.

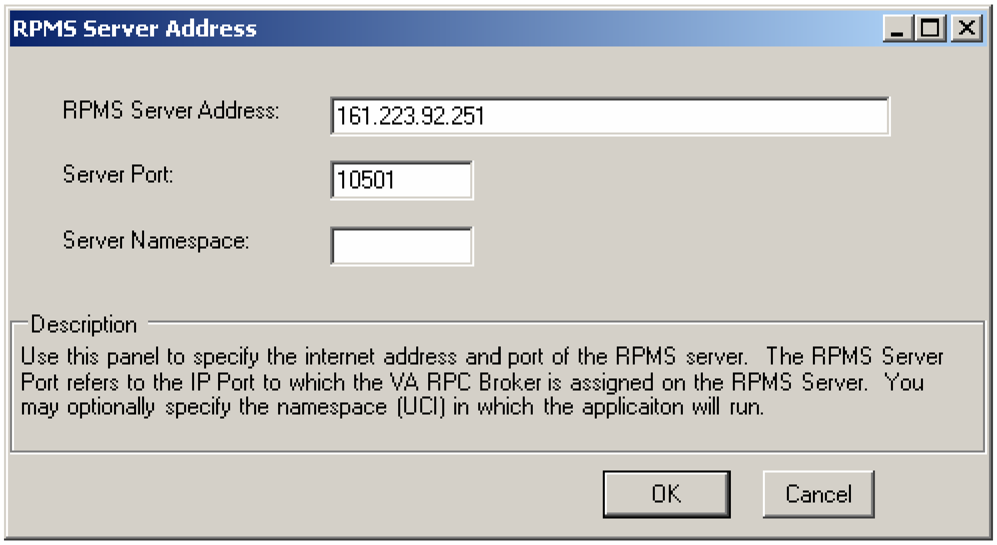
The site manager must install the BMX_0200.k KIDS software in each namespace to be accessed. The site manager should then start up one listener running in a primary namespace. All of the other namespaces *must not* have listeners using the same port address assigned and running in their BMXNET MONITOR file. If multiple listeners are started using the same port address, the RPMS system will not accept connections, and a complete reboot of the system will be required to correct the problem. The site manager should then assign the proper Server Namespace to the client (listener) running the application.

Multiple BMXMON listeners can be assigned to the same namespace or namespaces as long as there is no port number conflict. Each BMXNET MONITOR can be defined for *integrated security*, *session namespace*, and *if enabled*.

When restarting the BMXNET MONITOR, it is not necessary to stop monitors before restarting. If monitors are running and the option BMX MONITOR START IS SELECTED, the option will stop and restart the monitors.

Note: If you have changed the existing settings (e.g. port or namespace), without first stopping the monitors, old monitors may continue to run.

The following screen capture shows where the client server namespace is entered.



RPMS Server Address: 161.223.92.251

Server Port: 10501

Server Namespace:

Description

Use this panel to specify the internet address and port of the RPMS server. The RPMS Server Port refers to the IP Port to which the VA RPC Broker is assigned on the RPMS Server. You may optionally specify the namespace (UCI) in which the applicaiton will run.

Figure 3-3: Example of screen used for entering a client server namespace

4.0 Connecting to RPMS Using BMXNet

In addition to data access, BMXNet also provides RPMS connectivity and user log on services. Developers can choose to have BMXNet manage the entire connection and log on process automatically using the *BMXNetConnectInfo* class, or they can choose to use the low-level connectivity APIs that BMXNet exposes in the *BMXNetLib* class.

Note: Developers who wish to display their own dialogs to capture server and password information should use the Open Connection() method exposed by the BMXNetLib class.

4.1 BMXNetConnectInfo Class

BMXNetConnectInfo.LoadConnectInfo simplifies the RPMS connection process by encapsulating the functionality of the lower-level *BMXNetLib.OpenConnection* method. The *LoadConnectInfo* method fully supports automatic log on capability using securely cached information.

Note: LoadConnectInfo will automatically prompt for any cached log on information that it cannot locate in its cache. If LoadConnectInfo is unable to connect, it will raise a BMXNetException.

4.1.1 Using BMXNetConnectInfo.LoadConnectInfo Parameters

BMXNetConnectInfo.LoadConnectInfo constructor accepts four parameters in three overload combinations.

```
LoadConnectInfo(string MServerAddress, int Port, string AccessCode, string
VerifyCode)
```

Figure 4-1: Sample of parameters

When called with no parameters, *LoadConnectInfo* will connect to the most recently connected RPMS server using the most recent RPMS user login. *Figur*2 demonstrates an RPMS connection procedure with no parameters. BMXNet will display dialog boxes to collect information it requires to connect.

```
   private void cmdConnect_Click(object sender, System.EventArgs e)
   {
      BMXNetConnectInfo ci = new BMXNetConnectInfo();
      try
      {
         ci.LoadConnectInfo();
      }
      catch (Exception ex)
      {
         MessageBox.Show(ex.Message, "BMXNetTest");
      }
   }
```

Figure 4-2: Connecting to RPMS

Developers may also call overloads of *LoadConnectInfo* to log on using specific combinations of values for server address, port, access, and verify codes. *Figur*3 the overload with specific parameters for server and port. In this case, BMXNet will attempt to connect to the specified server as the most recent RPMS User. If the most recent user information is invalid or unavailable, BMXNet will display a dialog box to collect access and verify codes.

```
      ci.LoadConnectInfo(“127.0.0.1”, 9200);
```

Figure 4-3: Connecting to a specific server and port

To force *LoadConnectInfo* to prompt for access, and verify codes, use the overload shown in *Figu*4 specifying null string values for the access and verify code parameters. In this case, BMXNet will collect the A/V codes and then attempt to connect to the most recently used server. If the most recent server information is invalid or unavailable, BMXNet will display a dialog box to collect it.

```
      ci.LoadConnectInfo(“”, “”);
```

Figure 4-4: Force LoadConnectInfo to prompt for Access and Verify codes

To force *LoadConnectInfo* to prompt for all connection information, use the overload shown in *Figu*5 specifying null string values for the access code, verify code and server address and 0 for the port parameter. In this case, BMXNet will display dialogs to collect both the A/V codes and server address and port.

```
      ci.LoadConnectInfo(“”, 0, “”, “”);
```

Figure 4-5: Force LoadConnectInfo to prompt for all connection information

4.1.2 Using *BMXNetConnectInfo* Properties

After connecting to RPMS, developers may set the Application Context using the *BMXNetConnectInfo.AppContext* property. The Application Context is an entry in the OPTION file in RPMS. Users must have access to this a particular application context before they may access any remote procedures contained in the context’s RPC Multiple.

Note: Other properties of the BMXNetConnectInfo class return information about the current connection, including UserName, DUZ, and MserverAddress.

4.2 BMXNetLib Class

The BMXNetLib class primarily contains low-level methods which are used internally by BMXNet.

BMXNet is designed to return data in ADO.NET DataTable form using standard ADO.NET procedures. Occasionally, however, developers may wish to directly execute remote procedure calls (RPCs) against RPMS. The *BMXNetLib.TransmitRPC* method executes the named RPC along with any parameters and returns a string. *Figure*6 contains sample code to illustrate connecting to RPMS and using *BMXNetLib.TransmitRPC* to execute an RPC.

```
private void cmdTransmit_Click(object sender, System.EventArgs e)
   {
      try
      {
         BMXNetLib bnLib;
         bnLib = new BMXNetLib();
         BMXNetConnectInfo ci = new BMXNetConnectInfo();
         ci.LoadConnectInfo();
         ci.AppContext = "BMXRPC";
         bnLib.OpenConnection(ci);
         string sDUZ = bnLib.DUZ;
         string sUser = bnLib.TransmitRPC("BMX USER", sDUZ);
         MessageBox.Show("Connection to RPMS succeeded.  User ame = " + sUser,
"BMXNetLib Test");
         bnLib.CloseConnection();
      }
      catch (Exception ex)
      {
         MessageBox.Show(ex.Message, "BMXNetTest Error");
      }
   }
```

Figure 4-6: Connecting to RPMS and directly executing an RPC

5.0 BMXNet and Exchanging Data with RPMS

Using BMXNet to Exchange Data with RPMS
BMXNet returns data from the Cache environment as an ADO.NET *DataTable*. An ADO.NET *DataTable* is a fundamental structure for exchanging and manipulating data in .NET applications. For example, ADO.NET *DataTables* can be used to populate data controls in .NET and as the basis for building Crystal Reports.

There are three ways to call BMXNet to retrieve a DataTable containing M data:

- By calling a **custom-written remote procedure (RPC)** registered in VA FileMan’s REMOTE PROCEDURE file. You must design your remote procedure’s M routine to return its data in the specific format described later in this document.
- By using a **Structured Query Language (SQL) SELECT** statement to retrieve data from VA FileMan files. BMXNet supports a subset of SQL-like statements that are useful for simple data retrieval tasks.
- By using the **BMX ADO SS** remote procedure provided with the BMXNet package to generate fully-updateable ADO.NET DataTables.

Each of these three options will be discussed in detail in the following pages.

Use a **custom remote procedure (RPC)** when your data retrieval task is complex or when the data to be retrieved is not stored in FileMan’s files. Remote procedures can be used to update and delete FileMan data and generally to do anything that can be done in an M routine.

Use BMXNet’s **SQL-like capability** to quickly retrieve a read-only DataTable without having to write an RPC call or create any other supporting file entries.

Use **BMX ADO SS** when you want to retrieve an updateable DataTable. **BMX ADO SS** is a generalized Remote Procedure Call which generates an updateable DataTable. **BMX ADO SS** requires that an entry be made in the BMX ADO SCHEMA file describing the DataTable.

Complete source code for two sample applications, BMXNetTest and BMXSchemaBuilder, are distributed with BMXNet. Examine these sample applications to see different methods for connecting to RPMS, retrieving ADO.NET DataTables and populating windows controls with data retrieved from RPMS. Additionally, the sample applications can be used to create and test your own RPCs, SQL queries and schemas.

Note: The details of using ADO.NET in your Windows or Web application are beyond the scope of this document; however, there are many reference books and articles available from the Microsoft Developer's Network (MSDN) and other commercial sources that describe how to incorporate ADO.NET into applications.

5.1 Designing RPCs and M Routines for BMXNet

You can create M routines that return data via remote procedure calls (RPCs) to use with BMXNet. These M routines must return data in a specific format in order for BMXNet to process the records into ADO.NET *DataTable* rows.

Generally, a BMXNet RPC must create an array which describes the data fields and contains the data itself. The first record in the array contains schema information about the data fields. This first record is called the *schema record.* BMXNet uses two kinds of schema records, the *Minimum Schema Record* and the *Full Schema Record.* Use the Full Schema Record when retrieving tables that will be updated using BMX ADO SS. Use the Minimum Schema when retrieving static tables.

Even if no data records are returned, the RPC must always return the schema record containing metadata which describes the Data Table columns. Subsequent records contain data organized according to the information in the schema record.

5.1.1 Minimum Schema Record

At a minimum, the schema record must contain the elements described in *Table 5-* for each data field. The *Minimum Schema Record* contains enough information to create a read-only Data Table. (To create an updateable Data Table using the Full Schema Record, see Section 5.3.1.)

$$\overbrace{\textbf{T}}^{A}\overbrace{\textbf{00030}}^{B}\overbrace{\textbf{COLUMNNAME}}^{C}$$

Figure 5-1: Minimal Schema illustration

The following table details structure information of the *Minimum Schema Record.*

Section	Use	Description
A	Data Type	Specifies the data type of the column. Available data types are **T**ext, **N**umeric, **I**nteger and **D**ate.
B	Column Length	Specifies the maximum length, in characters, of the column.
C	Column Name	Specifies the text name of the column.

Table 5-1: Table of Minimal Schema Record Structure

5.1.2 BMXNet Record Structure

The record returned from a BMXNet RPC must adhere to these rules:

- BMXNet fields are delimited by the ^ character.
- BMXNet records are delimited by the ASCII character 30, the End-Of-Record marker.
- The entire set of records in BMXNet is terminated by the ASCII character 31, the End-Of-File marker.

5.1.3 Sample BMXNet RPC

Figure 5-2 shows the M code for a simple remote procedure call (BMX PATIENT DEMO) with two parameters to return a list of patients. See the following example. Note the inclusion of an error trap in the routine; BMXNet RPC routines must contain a functional error trap.

```
PDEMO(BMXY,BMXPAT,BMXCOUNT)   ;EP
   ;This simple RPC demonstrates how to format data
   ;for the BMXNet ADO.NET data provider
   ;
   ;Returns a maximum of BMXCOUNT records from the
   ;VA PATIENT file whose names begin with BMXPAT
   ;
   N BMXI,BMXD,BMXC,BMXNODE,BMXDOB
   ;
   ; BMXY is passed in by reference.  Set it to
   ;the value of the variable in which we will return our data:
   S BMXY="^TMP(BMX,"_$J_")"
   ;
   ;The first subnode of the data global contains the column header
information
   ;in the form "txxxxxCOLUMN1NAME^txxxxxCOLUMN2NAME"_$C(30)
   ;where t is the column data type:
   ;  T for text
   ;  I for integer
   ;  N for floating point number
   ;  D for date/time.
   ;xxxxx is the length of the column in characters.
   ;
   S BMXI=0,BMXC=0
   S ^ TMP(BMX,$J,BMXI)="T00030NAME^T00010SEX^D00020DOB"_$C(30)
   ;
   ;You MUST set an error trap:
   S X="PDERR^BMXRPC6",@^%ZOSF("TRAP")
   ;
   ;Strip CR, LF, TAB from BMXCOUNT parameter
   S BMXCOUNT=$TR(BMXCOUNT,$C(13),"")
   S BMXCOUNT=$TR(BMXCOUNT,$C(10),"")
   S BMXCOUNT=$TR(BMXCOUNT,$C(9),"")
   ;
   ;Iterate through the global and set the data nodes:
   S:BMXPAT="" BMXPAT="A"
   S BMXPAT=$O(^DPT("B",BMXPAT),-1)
   S BMXD=0
   F  S BMXPAT=$O(^DPT("B",BMXPAT)) Q:BMXPAT=""  S BMXD=$O(^DPT("B",BMXPAT,0))
I +BMXD  S BMXC=BMXC+1 Q:(BMXCOUNT)&(BMXC>BMXCOUNT)  D
   . Q:'$D(^DPT(BMXD,0))
   . S BMXI=BMXI+1
   . S BMXNODE=^DPT(BMXD,0)
   . ;Convert the DOB from FM date to external form
   . S Y=$P(BMXNODE,U,3)
   . I +Y X ^DD("DD")
   . S BMXDOB=Y
   . ;The data node fields are in the same order as the column header, i.e.
NAME^SEX^DOB
   . ;and terminated with a $C(30)
   . S ^ TMP(BMX,$J,BMXI)=$P(BMXNODE,U)_U_$P(BMXNODE,U,2)_U_BMXDOB_$C(30)
   ;
   ;After all the data nodes have been set, set the final node to $C(31) to
indicate
   ;the end of the recordset
   S BMXI=BMXI+1
   S ^ TMP(BMX,$J,BMXI)=$C(31)
   Q
   ;
PDERR ;Error trap for PDEMO
   ;
   S ^ TMP(BMX,$J,BMXI+1)=$C(31)
   Q
```

Figure 5-2: Remote procedure call example

Figure 5-3 displays the correct format for the entry in the REMOTE PROCEDURE FILE.

```
AME: BMX DEMO                          TAG: PDEMO
  ROUTINE: BMXRPC6                     RETURN VALUE TYPE: GLOBAL ARRAY
```

Figure 5-3: Remote procedure file entry example

Note: Remote Procedures and the routines which they invoke must be name-spaced according to RPMS Standards and Conventions.

5.1.4 Invoking Remote Procedures from .NET code

Figure 5-4 demonstrates one of many ways to invoke a remote procedure in a .NET application to return a *Data Table*. You may display the data-table in a data grid by setting the grid's Data Source property to the data-table returned from RPMS. The grid is displayed in *Tabl*.

```
private void cmdTest1_Click_1(object sender, System.EventArgs e)
{
   try
   {
      BMXNetConnectInfo ci = new BMXNetConnectInfo();
      ci.LoadConnectInfo();
      ci.AppContext = "BMXRPC";
      BMXNetConnection conn = new BMXNetConnection(ci);
      conn.Open();
      if (conn.State != ConnectionState.Open)
      {
         throw new BMXNetException("Unable to connect to RPMS.");
      }
      BMXNetCommand cmd = (BMXNetCommand) conn.CreateCommand();
      cmd.CommandText = "BMX DEMO^S^10”
      BMXNetDataAdapter da = new BMXNetDataAdapter();
      da.SelectCommand = cmd;
      DataSet ds = new DataSet();
      da.Fill(ds,"BMXNetTable");
      dataGrid2.DataSource = ds.Tables["BMXNetTable"];
      conn.Close();
   }
   catch (Exception ex)
   {
      MessageBox.Show(ex.Message);
      return;
   }
}
```

Figure 5-4: Invoking a remote procedure in a .NET application

NAME	SEX	DOB
SAEL,ROBERTA	F	12/19/1954
SAEL,ROBERTA	F	12/19/1954
SELIARS,ARDEEN RENEE	F	8/18/1983
SELIARS,BILLY LEE	M	8/27/1953
SELIARS,DANIEL MICHAEL	M	5/6/1980
SELIARS,JOHN DELBERT	M	12/1/1978
SEM,ALLISON RAE	F	2/23/1966
SEM,JOHN DELBERT	M	6/28/1985
SEM,JOHNNIE	M	10/31/1927
SEM,MICHAEL JIM	M	9/16/1963

NAME	SEX	DOB
SAEL,ROBERTA	F	12/19/1954

Table 5-2: RPC results displayed in a DataGrid

In order for any RPC to be available to a user, four requirements must be met:

- An entry in the OPTION File (#19) which has the TYPE of “Broker (Client/Server)” must exist.
- The RPC must be entered in the RPC multiple of the OPTION.
- The user must have access to the OPTION via the RPMS menu tree either as secondary menu item.
- The AppContext property of the BMXNetConnectInfo object must be set to the name of the OPTION.

5.2 Using SQL-like Queries to Retrieve FileMan Data

BMXNet supports a subset of SQL-like keywords to provide read access to VA FileMan files. Note the following items when creating SQL statements in BMXNet.

- Substitute the underscore character (”_”) for all spaces in VA FileMan field, file, and subfile names in a BMXNet SELECT statement.

  Example: VA PATIENT must be expressed as **VA_PATIENT**

- The BMXNet provider recognizes the field BMXIEN as the internal entry number field of a VA FileMan file. This special field corresponds to the FileMan **NUMBER** field.
- Use the INTERNAL[] function to return the internally stored value of a VA FileMan field. This function corresponds to the VA FileMan INTERNAL() function.

5.2.1 SQL SELECT Syntax

The basic syntax of a SQL SELECT statement in BMXNet is indicated in *Figur*5. Each component of this statement is individually addressed within sections 5.2.1.1 - 5.2.1.5.

```
SELECT select_list
FROM table_source
[WHERE search_condition [INDEX:i]]
[MAXRECORDS:nnn]
[SHOWPLAN]
```

Figure 5-5: SQL Select Syntax

5.2.1.1 SELECT Clause

The SELECT clause specifies the columns that the system should return from the query. *Figure*6 shows a SELECT clause example and *Table 5-2* describes each component within the clause.

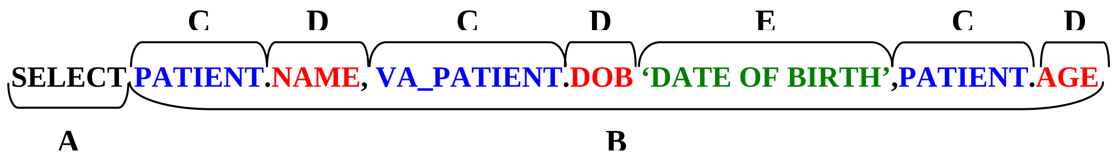

C
D
C
D
E
C
D
SELECT PATIENT.NAME, VA_PATIENT.DOB 'DATE OF BIRTH',PATIENT.AGE
A
B

Figure 5-6: Graphic example of a SELECT Clause

Section	Syntax	Description
A	SELECT	Specifies that the system should return the values by the following select_list statement.
B	select_list	Specifies which fields (columns) the system should select for the returned set. The select_list a series of expressions separated by commas.
C	table_name	Specifies from which files (tables) the FROM clause should return fields.
D	field_name	Specifies from which field (column) the FROM clause should return. The field_name is associated with the preceding table_name by using a dot (.) between the two.
E	column_alias	Specifies an alternative name to replace the column name in the query result set. For example, an alias such as “Quantity”, “Quantity to Date”, or “Qty” can be specified for a column named **quantity**

Table 5-2: SELECT clause components

5.2.1.2 FROM Clause

The FROM clause specifies the table(s) from which the system should retrieve rows. The FROM clause is required and specifies the primary table, or FileMan file, and related tables for the SELECT clause.

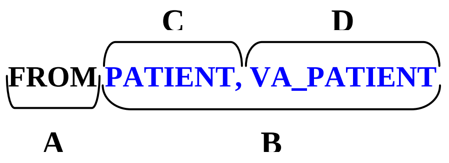


Figure 5-7: Graphic example of a FROM clause

Section	Syntax	Description
A	FROM	Specifies that the system should return the values from the following table_list statement.
B	table_source	Specifies from which files (tables) the FROM clause should return fields.
C	primary_table_nam e	The file from which the system should first retrieve data.
D	related_table_name	The file from which the system should receive data that matches the primary table.

Table 5-3: FROM clause components

5.2.1.3 WHERE Clause

The WHERE clause specifies search condition(s) to restrict the rows returned and to join tables on a common field.

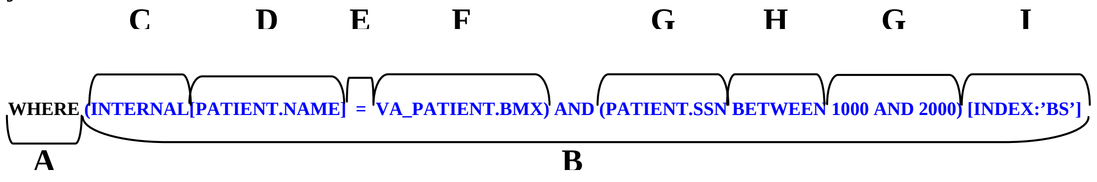

C
D
E
F
G
H
G
I
WHERE (INTERNAL[PATIENT.NAME] = VA_PATIENT.BMX) AND (PATIENT.SSN BETWEEN 1000 AND 2000) [INDEX:'BS']
A
B

Figure 5-8: Sample of a WHERE clause

Section	Syntax	Description
A	WHERE	Specifies a search condition to restrict the rows returned.
B	search_condition	Restricts the rows returned in the result set by using predicates. There is no limit to the number of predicates that can be included in a search condition.
C		

Section	Syntax	Description
D	primary_table_name	The file from which the system should first retrieve data.
E	join_type	Specifies a join using nonstandard syntax and the WHERE clause. The =* operator is used to specify a one-to-many (OTM) join. Use the OTM join to express relationship between tables A and B such that a record in table A can be referenced by a FileMan pointer field in one or more records in table B.
F	related_table_name	The file from which the system should receive data that matches the primary table.
G	Operator Arguments	A value or expression dictating the information upon which the related operator acts.
H	Operator	A symbol that specifies which operation the systems should perform relative to the indicated operator arguments.
I	INDEX	Identifies a specific FileMan cross-reference to use when retrieving data. Always use the SHOWPLAN keyword in conjunction with the INDEX keyword to ensure that you achieve the intended result. [NOT SUPPORTED IN BMXNET VERSION 1.0]

Table 5-4: WHERE clause components

Operators
Table 5-5 describes the WHERE clause operators that BMXNet recognizes.

Operator	Description
=	Equals
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to
=*	One-to-many Table Join
LIKE	Pattern match. Use the % wildcard character to specify a string of zero or more characters. BMXNet currently supports use of one % character on the right side of the comparison value, e.g. LIKE ‘SMITH%’ will return all records beginning with the string SMITH.

Operator	Description
BETWEEN	Specifies an inclusive range to test. BETWEEN ‘value1’ AND ‘value2’ will return all records with column values between value1 and value2, inclusive. Comparison values can be string, date, or numeric.
TEMPLATE	Records in the specified FileMan search template will be returned. The syntax is: TEMPLATE[*filename*] = ‘*Search Template Name*’. By using the TEMPLATE keyword, complex searches can be conducted using VA FileMan or other M-based tools and the results can be displayed using common desktop applications capable of processing ADO DataTables.

Table 5-5: Available WHERE operators

Example

The following example specifies a One-to-many join, in which all the rows from **Tab2** in which **Tab2.id = Tab1.id** are included in the result set.

```
SELECT Tab1.name, Tab2.id
FROM Tab1, Tab2
WHERE Tab1.id =* Tab2.id
```

Figure 5-9: Sample of a one-to-many join

5.2.1.4 MAXRECORDS Keyword

The MAXRECORDS clause specifies the maximum number of records that the system should return. Use MAXRECORDS when designing and testing BMXNet SQL statements which could return a large number of records. *Figure 5*-10 represents a MAXRECORDS clause example and *Table 5-6* describes each component within the clause.

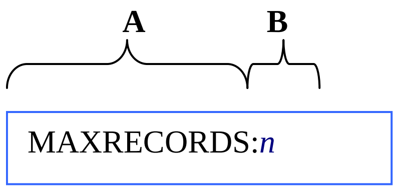


Figure 5-10: MAXRECORDS clause graphic example

Section	Syntax	Description
A	MAXRECORDS	Specifies the maximum number of records to return
B	*n*	Integer number indicating the maximum records to return.

Table 5-6: MAXRECORDS clause components

5.2.1.5 SHOWPLAN Keyword

The SHOWPLAN keyword returns the query plan, including the M code that will be executed to iterate through and retrieve the records. You can use SHOWPLAN to confirm that your query statement is using a particular index. The query plan report contains at least two rows, including the *Index* row and the *Screen* row(s).

- The Index row includes the M code created by BMXNet that will execute on the RPMS server and iterate through the FileMan file.
- The Screen rows include the M code that BMXNet will execute to filter the results. The "^" character in the query plan is replaced by the "~" character.

The following example displays a SHOWPLAN used with a WHERE clause that includes a table join and two screening filters. *Table 5-7* shows the index and screening that BMXNet will generate to iterate through and select the records.

```
SELECT VA_PATIENT.NAME, VA_PATIENT.SSN, PATIENT.CURRENT_COMMUNITY
FROM VA_PATIENT, PATIENT
WHERE VA_PATIENT.BMXIEN =* INTERNAL[PATIENT.NAME] AND VA_PATIENT.NAME LIKE
'DEMO%' AND VA_PATIENT.AGE BETWEEN 15 AND 24 SHOWPLAN
```

Figure 5-11: SHOWPLAN keyword example

Item	Result
INDEX(1)	S BMXV="DELVA,STEVEN O." F S BMXV=$O(~DPT("B",BMXV)) Q:BMXV="" Q:BMXV'?1"DEMO".E Q:BMXM>BMXXMAX S D0="" F S D0=$O(~DPT("B",BMXV,D0)) Q:D0="" Q:BMXM>BMXXMAX
SCREEN	X BMXSCR("C") I 1&1&((BMXSCR("X",1)'<15)&(BMXSCR("X",1)'>24))
SCREEN(1)	X ~DD(2,.033,9.5) S X=$E(Y(2,.033,15),Y(2,.033,16),X) S Y=X,X=Y(2,.033,13),X=X<Y,Y=X,X=Y(2,.033,8),X=X-Y

Table 5-7: SHOWPLAN keyword components

Analysis of the result indicates that the query will use the "B" cross references of ^DPT and to iterate through the patient files. The screening code will execute the

FileMan computed field code stored in ^DD for the AGE field and then apply the screening logic.

5.2.2 Sample BMXNet SQL Queries:

Sections 5.2.2.1 - 5.2.2.4 illustrate the use of BMXNet to retrieve FileMan data using SQL SELECT statements.

5.2.2.1 Simple Select Statement Example

The following example depicts a simple Select statement.

```
SELECT NAME 'PATIENT', DOB
FROM VA_PATIENT
WHERE NAME LIKE 'DEMO%'
```

Figure 5-12: Simple SELECT statement example

Patient	DOB
DEMO,JANICE	12/1/56
DEMO,JILL	9/1/23
DEMO,JOE	6/1/33
DEMO,JOE	3/22/54
DEMO,JOHN	2/23/44
DEMO,ONE	3/2/34
DEMO,PATIENT	4/1/56
DEMO,TEST	8/23/33
DEMO,TWO	2/1/33

Table 5-8: Results of Simple SELECT statement example

5.2.2.2 Retrieving Subfield Data

When retrieving subfield data from VA FileMan, the MAXRECORDS keyword applies to the count of parent records, not to the subfields. In this example, data for two patients is returned, although one patient has more than one health record in the FACILITY multiple.

```
SELECT PATIENT.NAME 'PATIENT', PATIENT.DOB,
PATIENT.HEALTH_RECORD_NO..HEALTH_RECORD_FAC FACILITY,
PATIENT.HEALTH_RECORD_NO..HEALTH_RECORD_NO. CHART
FROM PATIENT
WHERE NAME LIKE 'WABAR,J%' MAXRECORDS:2
```

Figure 5-13: Retrieving subfield data example

Patient	Patient DOB	Facility	Chart
WABAR,JESSE Z	6/23/1923	PORTEGE EHRTEST	101875
WABAR,JESSE Z	6/23/1923	MAKE U WELL FACILITY	101876
WABAR,JUDITH Y	3/1/1982	PORTEGE EHRTEST	104047

Table 5-9: Results of retrieving subfield data example

5.2.2.3 Joining FileMan Files

The following example shows how to use a join to connect two FileMan files and access data from each file.

```
SELECT VA_PATIENT.NAME 'NAME', VA_PATIENT.STATE 'STATE', STATE.ABBREVIATION,
VA_PATIENT.AGE 'AGE'
FROM VA_PATIENT, STATE
WHERE INTERNAL[VA_PATIENT.STATE] = STATE.BMXIEN AND VA_PATIENT.NAME LIKE 'WH%'
MAXRECORDS:5
```

Figure 5-14: Joining FileMan files

Patient	State	State Abbreviation	Age
WHAALAR,ALLEN	NEW MEXICO	NM	50
WHAALAR,BEVERLY A	ARIZONA	AZ	61
WHAALAR,BONNIE M	MINNESOTA	MN	44
WHAALAR,CHRISTINE M	MINNESOTA	MN	34
WHAALAR,DUSTIN B	MINNESOTA	MN	27

Table 5-10: Results of joining FileMan files

5.2.2.4 One-to-Many Join

The following example shows how to use a one-to-many join to retrieve data from VISIT and VISIT-related files.

```
SELECT VISIT.BMXIEN,  VISIT.PATIENT_NAME, VISIT.CLINIC, V_DENTAL.BMXIEN,
V_DENTAL.SERVICE_CODE
FROM VISIT, V_DENTAL
WHERE VISIT.PATIENT_NAME LIKE 'GEGN%'  AND VISIT.CLINIC = 'DENTAL' AND
VISIT.BMXIEN =* INTERNAL[V_DENTAL.VISIT]
```

Figure 5-15: Sample of a one-to-many join retrieval

VISIT.BMXIEN	VISIT.PATIENT_NAME	V_DENTAL.SERVICE_CODE
5219	GEGNUN,BETTE A	9320
5219	GEGNUN,BETTE A	0190
5979	GEGNUN,ROBERTA J	7111
5979	GEGNUN,ROBERTA J	0000
5979	GEGNUN,ROBERTA J	0140
5979	GEGNUN,ROBERTA J	0270
5979	GEGNUN,ROBERTA J	0270
5979	GEGNUN,ROBERTA J	0270
5219	GEGNUN,BETTE A	9320
5219	GEGNUN,BETTE A	0190
5979	GEGNUN,ROBERTA J	7111

Table 5-11: Results of one-to-many join retrieval

5.3 Creating Updateable Data Tables with BMXNet

BMXNet provides a way to generate **updateable** ADO.NET DataTables from FileMan data. By using an updateable DataTable, the BMXNet developer is able to use standard *.NET* techniques in his Windows/Web application to access and edit FileMan data and can avoid having to write custom RPCs for these tasks.

5.3.1 The Full Schema Record

Updateable DataTables require use of the **full schema** record rather than the minimal schema record described in Section 5.1.1. While the minimal schema record contains only data type, column length and column name information, the full schema record contains sufficient metadata for BMXNet to generate fully-updateable ADO.NET DataTables.

The full schema is also able to express relational joins between multiple tables. BMXNet can use this information to build ADO.NET DataRelationship objects between tables in a client ADO.NET DataSet that mirror the FileMan File relationships on the RPMS Server.

It is possible to write custom RPCs which use full schema instead of the minimal schema. The data nodes of the returned M array take the same form regardless of whether the full or minimal schemas are used. See Section 5.1 for information on writing custom RPCs with BMXNet.

Because the full schema is complex, BMXNet provides software utilities to simplify using the full schema. Rather than writing a custom RPC, developers may define and store full schemas in the **BMX ADO SCHEMA** file. Developers may then access these full schemas using the **BMX ADO SS** remote procedure call as described in Section 5.1.4.

5.3.2 Structure of BMXNet Data

Developers who use BMXNet's utilities may never need to examine records returned by BMXNet utilities in the Cache environment. However, when troubleshooting and developing it is important to understand the structure of the data returned by BMXNet's M utilities.

The BMXNet M components return data in a string delimited by ASCII 30 (EOR) characters. The end of the entire set of records is marked by the ASCII 31 (EOF) character:

```
SchemaRecord EOR DataRecord1 EOR DataRecord2 EOR…DataRecordN EOR EOF
```

Figure 5-16: Sample of return data in a string delimited by ASCII 30 (EOR) characters

The Schema Record may be in either Full or Minimum format. (See Section 5.1.1 for structure of the minimum schema record.) The Full Schema Record consists of an **introductory section** followed by one or more **field definition sections**. Like all fields in BMXNet records, each of these sections is delimited by the up-arrow character "^". This is a printable character: ASCI 94 [$C(94)]

```
IntroductorySection^FieldDef1^FieldDef2^…^FieldDefinitionN EOR
```

Figure 5-17: Sample of a BMXNet record

Each major section of the schema string is further subdivided into basic elements by the vertical bar character "|": ASCI 124 [$C(124)].

5.3.3 The Introductory Section of the Schema Record

The introductory section of the Schema String is contained within the first "^" piece of the schema string. This section is further subdivided by the "|" character.

Consider the following sample Introductory Section of a Schema String:

```
"@@@meta@@@BMXIEN|2160010.224||^…”
```

Figure 5-18: Sample of the introductory section of a schema string

First “|” Piece: The “@@@meta@@@BMXIEN” text is a **lead tag** that tells BMXNet that this schema is a full schema rather than a minimal schema. If the return data contains multiple schema/data pairs to define a relational join, the lead tag delimits the boundaries of each pair.

Second “|” Piece: The FileMan file or sub-file number. This is the primary file/sub-file that contains the data, and it is also the starting point for inter-file navigation. In the example shown above, the second piece references a sub-file.

Third “|” Piece: This piece usually is empty. It only contains a value if (1) the record set contains data records along with a schema record, (2) data is being returned to a calling application in chunks; e.g. 50 records at a time and (3) there is more data to be passed in the next chunk. If all three of these conditions are met, then this piece contains the “seed” value that tells the retrieving routine where to start the next chunk. In other words, the seed initiates the *iteration* that generates the rows of the data table. Management of chunks and the use of seeds will be discussed in detail later.

Fourth “|” Piece: This piece is usually empty. It may contain a numeric value if the schema defines a relational join of two or more data tables. Management of relational joins will be discussed in detail later.

5.3.4 The Field Definition Section of the Schema Record

The field definition is contained in “^” pieces 2 through *n* of the schema record. Each “^” piece defines a specific field or column in the schema. There is no technical limit to the number of “^” pieces/field definitions that can be included in this record. Within each “^ piece, there are eight sub-pieces delimited by the “|” character. A field definition section contains the components shown in the following example.

```
“…^FILE#|FIELD#|DATATYPE|LENGTH|FIELDNAME|READONLY|KEYFIELD|NULLOK^”
```

Figure 5-19: Sample of components in a field definition section

Following is an example of a field definition section.

```
“…^2160010.224|.03|D|12|Birthdate|FALSE|FALSE|FALSE^…”
```

Figure 5-20: Sample of a field definition section

First “|” Piece
This is the FileMan file or sub-file number. Typically, this number will be identical for every data element in the schema.

Second "|" Piece
This is the FileMan field number. This number is different for every data element in the schema. The schema string contains at least one *virtual field number*, as described in section 5.3.5.

Third "|" Piece
The Third "|" Piece is the ADO Data Type. See the example below:

D = Date, I = Integer, N = floating point Number, T = Text

Fourth "|" Piece
The Fourth "|" Piece is the maximum length of the field in characters.

Fifth "|" Piece
This is the ADO DataTable Column Header. Often this is the same as the FileMan field name.

Sixth "|" Piece
This is a "Read Only" flag. "TRUE" if data cannot be edited. "FALSE" if data can be updated.

Seventh "|" Piece
The Seventh "|" Piece is the "Key Field" flag. "TRUE" if this field is the key field. "FALSE" if this is not the key field. Unless we are dealing with the *.001* field (the IEN), this value will be "FALSE"

Eighth "|" Piece
This is the "Null OK" flag. "TRUE" if value can be null. "FALSE" if data is mandatory. This piece is important because it is a potential determinant of transaction "rollback". If the value is "FALSE" and no data value is supplied for this field, then the entire transaction will be rolled back. For example a VISIT file entry will be rolled back if the VISIT LOCATION is not specified. If the FileMan field is "required", this piece should be set to "FALSE".

Like all BMXNet records, the schema record terminates with the non-printable EOR character, ASCII 30.

5.3.5 Virtual Fields

The schema may contain so-called "virtual fields." Virtual fields do not exist in FileMan - they only exist in the ANR – but they are derived from FileMan data and data definitions. Virtual fields are always "read only," and they are always created on an *ad hoc* basis when the ANR is generated. Some virtual fields *must* be linked to a specific, "real" field in the schema, and all share certain naming conventions and properties. There are four types of virtual fields:

- PRIMARY KEY
- PARENT KEY. POINTER VALUE
- IDENTIFIER
- TRIGGERED

5.3.5.1 Primary Key Virtual Fields

The primary key of any FileMan file is its internal entry number (IEN). The internal entry number of the file is stored in the .001 field of the ANR.

Note: The *.001* field is *always* defined in the schema.

5.3.5.2 Parent Key Virtual Fields

Sub-File representation and management will be discussed in detail later in this document. For now, it is sufficient to know that if the ANR defines a FileMan sub-file, the IEN of the *parent* file will be stored in the .0001 field of the sub-file's ANR. If it exists, the .0001 field will always be the first field in the schema. It is followed by the .001 field.

5.3.5.3 IEN Virtual Fields

In FileMan, many data elements are stored as "pointer" values. A pointer value is an internal entry number – a primary key of another file. Even though the data elements may be stored in internal format (IEN's), they are displayed in their external format. For example, the V MEASUREMENT file stores the patient IEN, but when measurement results are displayed, the patient's full name is shown.

Typically, when data is passed to the ANR, it is in external format. To avoid potential ambiguity, it may be useful to include the IEN along with the external value in the ANR data. A field that contains the IEN can be added to the ANRS at the last moment.

The virtual field follows a naming convention: xxIEN where xx is the field number. For example, ".04IEN" contains the internal value (IEN) of field .04.

Identifier Virtual Fields

In FileMan, certain fields are embellished with identifiers – additional pieces of information that are included whenever the key field is displayed. For example whenever a patient name is displayed, the DOB, SSN and chart number identifiers are automatically displayed as well. The EBCU is capable of automatically generating identifier fields for the ANR - *even if those fields are not included in the FileMan data definition.* The identifiers are included in a text string that is the value of a virtual field. The naming convention for an identifier field is: **xxIDy** where xx is the number of the field that is "identified" and y is an instance identification number. The instance identification number is necessary because there may be multiple identifier strings associated with a single field. For example "**.01ID2**" is the name of the second identifier field associated with the primary field .01.

Triggered Virtual Fields

The EBCU is capable of automatically transforming the value of one field and displaying the transformed value within another (virtual) field. For example, a date of birth could trigger an age value that is displayed in a virtual field. This functionality is similar to – but independent of – FileMan's "triggered" and computed fields. In fact FileMan's triggered/computed fields can be displayed along side the EBCU's triggered fields in the ANR. The naming convention for a triggered virtual field is xxTRIGGERy where xx is the number of the triggering field and y is the instance identification number. The instance identification number is necessary because there may be multiple triggers associated with a single field. For example "**.01TRIGGER2**" is the name of the second *triggered* field associated with *triggering* field .01.

The following schema is used to display patient measurements. It contains virtual fields (shown in bold text) that are automatically added to the schema by the EBCU. These virtual fields do not exist in FileMan.

```
@@@meta@@@BMXIEN|9000010.01|^
9000010.01|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^
9000010.01|.01|T|30|TYPE|FALSE|FALSE|FALSE^
9000010.01|.01IEN|I|00009|TYPE_IEN|FALSE|FALSE|FALSE^
9000010.01|.02|T|30|PATIENT NAME|FALSE|FALSE|FALSE^
9000010.01|.02IEN|I|00009|PATIENT
NAME_IEN|FALSE|FALSE|FALSE^
9000010.01|.02TRIGGER1|I|00003|PATIENT AGE|TRUE|FALSE|TRUE^
9000010.01|.02ID1|T|00080|PATIENT
IDENTIFIERS|TRUE|FALSE|TRUE^
9000010.01|.03|T|30|VISIT|FALSE|FALSE|FALSE^
9000010.01|.03IEN|I|00009|VISIT_IEN|FALSE|FALSE|FALSE^
9000010.01|.04|T|100|VALUE|FALSE|FALSE|TRUE
```

Figure 5-21: Sample of schema displaying patient measurements

5.3.6 The BMX ADO SCHEMA File

The specification for a BMXNet schema can be stored in the **BMX ADO SCHEMA** File. Storing a schema in a standard FileMan file helps to ensure the data integrity of the schema. BMXNet contains both command-line and Windows-based utilities for creating and updating schemas in the BMX ADO SCHEMA file.

```
.01        SCHEMA NAME (RF), [0;1]
.02        FILE OR SUBFILE NUMBER (NJ22,9), [0;2]
.03        DATASET IS READ ONLY
1          FIELD NUMBER (Multiple-90093.991), [1;0]
           .01  FIELD NUMBER (MNJ22,9), [0;1]
           .02  DATA TYPE (S), [0;2]
           .03  FIELD LENGTH (F), [0;3]
           .04  COLUMN HEADER (F), [0;4]
           .05  READ ONLY (S), [0;5]
           .06  KEY FIELD (S), [0;6]
           .07  NULL ALLOWED (S), [0;7]
           .08  IEN AUTOMATICALLY INCLUDED
           .09  ALWAYS GET INTERNAL VALUE
              1  AUTO IDENTIFIER EXTR FUNCT
              2  SPECIAL UPDATE EP
```

Figure 5-22: BMX ADO SCHEMA file structure

SCHEMA NAME
The SCHEMA NAME is a brief descriptive name that tells the purpose of the schema; e.g., “VISIT FILE - ADD NEW ENTRY”. Schema entries should be namespaced according to RPMS Standards and Conventions to avoid naming collisions between packages.

FILE OR SUBFILE NUMBER
The FileMan FILE OR SUBFILE NUMBER is described in the previous section.

DATASET IS READ ONLY
The DATASET IS READ ONLY has a value of “YES” or “NO.” If the value is “YES,” then *none* of the fields in the schema can be edited. This is the equivalent of the setting the READ ONLY property of every field to “TRUE” (see below). DATASET IS READ ONLY overrides all related properties.

FIELD
Field is a multiple containing all the fields represented in the schema.

FIELD NUMBER
FIELD NUMBER is the FileMan field number. Every file has a .001 field. This is a *virtual* field (one that may not actually exist in FileMan) that stores the internal entry number (AKA IEN or unique primary key) of the record. *The .001 field is*

automatically inserted in the schema string by the schema generator (see next section). If the schema describes a FileMan sub-file, a .0001 field will also be automatically added to the schema. This field contains the IEN of the entry in the parent file.

DATA TYPE
DATA TYPE contains the data type: NUMBER, INTEGER, TEXT, DATE. Note that if the FileMan data type for the field is a pointer- and the external value is to be displayed - then the data type will be automatically converted from integer (i.e. the IEN/actual pointer value) to whatever data type is appropriate for the pointed-to field.

FIELD LENGTH
FIELD LENGTH is a number. This value is used by ADO.Net to establish its internal data definitions.

COLUMN HEADER
COLUMN HEADER is a brief descriptive name – typically identical to the FileMan field name. When BMXNet creates the ADO.NET DataTable, this value will be used for the ColumnName.

KEY FIELD
KEY FIELD has a value of ‘YES’ or ‘NO’. The .001 field is always the primary key field. No other field can be the primary key field.

NULL ALLOWED
NULL ALLOWED has a value of “YES” or “NO.” If “NO,” the field is mandatory; i.e., the record will *not* be updated unless this filed has a non-null value.

IEN AUTOMATICALLY INCLUDED
IEN AUTOMATICALLY INCLUDED has a value of “YES” or “NO.” If the value is “YES,” the schema will automatically include an additional field to store the IEN of a pointer-type field. The data type of the IEN field is a number.

ALWAYS GET INTERNAL VALUE
ALWAYS GET INTERNAL VALUE has a value of “YES” or “NO.” If “YES,” the internal FileMan value will always be used along with the corresponding data type and field length for the internal value.

AUTO IDENTIFIER EXTR FUNCT
AUTO IDENTIFIER EXTR FUNCT specifies an extrinsic function entry point in the format "TAG^LINE." If an entry point is specified, BMXNet will use this extrinsic function to automatically create an identifier field and include it in the schema. This is another virtual field not found in FileMan.

SPECIAL UPDATE EP
SPECIAL UPDATE EP specifies an entry point in the format "TAG^LINE." If an entry point is specified, BMXNet will use the custom code in this call to file the data.

EXTR FUNCT FOR TRIGGERED VALUE
EXTR FUNCT FOR TRIGGERED VALUE specifies an entry point for creating another virtual field. The value of the triggering field is passed to this entry point and transformed. The transformed value is then passed to the virtual field. Multiple triggered fields can be produced from a single source field.

5.3.7 BMXNet Data Records

BMXNet Data Records – if present, follow the BMXNet Schema Record. Like all BMXNet records, the data record terminates with the non-printable EOR character, ASCII 30. Each data record consists of ^-delimited field data. The ^-position of the field determines which schema field it is associated with; the *n*th ^-piece of a data record corresponds to the *n+1*th ^-piece of the schema record.

The last data record is terminated by the ASCII 31 EOF character.

A sample BMXNet record set, including schema and data records, is shown in *Figure* 23. Note that the "^"-pieces of the schema (shown in black text) correspond to the "^"-pieces of each data record (shown in red text). In this example, there are seven pieces in the schema record and seven pieces in each data record. Each ^-piece corresponds to a field/column.

```
@@@meta@@@BMXIEN|9000010^9000010|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^9000010|.01|
D|21|TIMESTAMP|FALSE|FALSE|FALSE^9000010|.03|T|3|TYPE|FALSE|FALSE|FALSE^900001
0|05|T|30|PATIENT|FALSE|FALSE|FALSE^9000010|.06|T|30|FACILITY|FALSE|FALSE|FALS
E^9000010|.07|T|10|CATEGORY|FALSE|FALSE|FALSE^9000010|.08|T|30|CLINIC|FALSE|FA
LSE|TRUEEOR
71140^3040419.09^I^1^4585^A^1EOR
71144^3040419.1^I^1^4585^A^1EOR
71143^3040419.1^I^1^4585^A^1EOR
71142^3040419.1^I^1^4585^A^1EOR
71141^3040419.1^I^1^4585^A^1EOR
71145^3040419.11^I^1^4585^A^1EOR
71146^3040419.12^I^1^4585^A^1EOR
71147^3040419.15^I^1^4585^A^1EOR EOF
```

Figure 5-23: Sample of BMXNet Records

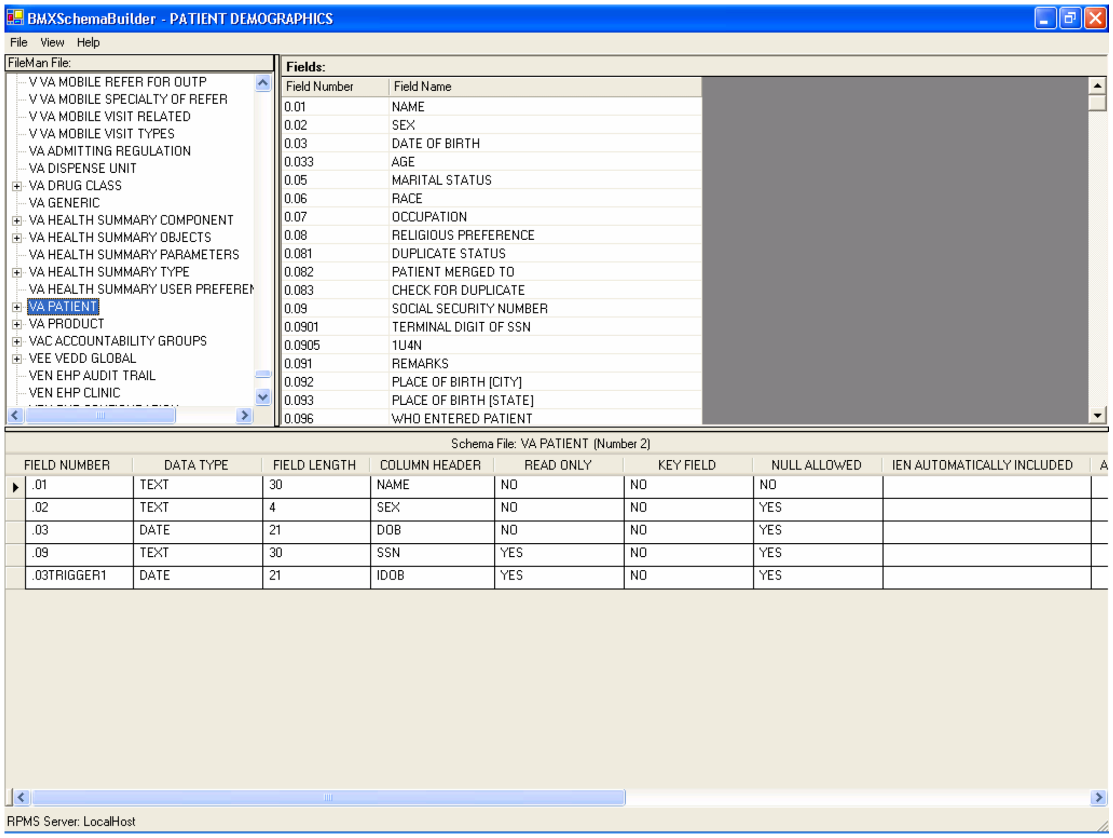
5.3.8 Schema Management Utilities

BMXNet includes two utilities for creating and editing schemas. The command-line based Schema File Updater utility is accessed at the entry point *SS^BMXADOS* on the RPMS Server. A sample session is shown in *Figure*24.

A Windows-based utility, *BMXSchemaBuilder*, is also available and a sample screen is shown in *Figure*25. The C# source code for BMXSchemaBuilder is included in the BMXNet package. BMXSchemaBuilder makes extensive use of BMXNet-generated ADO.NET data and is thus useful as a tutorial. A view of the BMXSchemaBuilder screen is displayed in *Figure*25.

```
>D ^BMXADOS

Enter schema name: PATIENT DEMOGRAPHICS
  Are you adding 'PATIENT DEMOGRAPHICS' as
    a new BMX ADO SCHEMA (the 11TH)? y  (Yes)
Select FILE:  VA PATIENT

The BMX PATIENT file contains the following sub-files

1 ADMISSION DATE  (2.95)
2 HEMODIALYSIS DATE  (2.951)
3 TRANSFER DATE  (2.96)
4 TREATING SPECIALTY  (2.9501)

Is the schema linked to a sub-file in this list? No// N  (No)

Select a field from this file :
1  NAME  [.01]
2  SEX  [.02]
3  DOB  [.03]
4  PRIMARY CLINIC  [.04]

Select a number from the list (1-4) or press <ENTER> to continue: 1
Column header: NAME//
Is null allowed? Yes// N  (No)
Want to display identifiers with this field? No// Y  (Yes)
Entry Point to generate Identifiers: DEMOGR^BMXADOX
The identifier field has been created

Select a number from the list (1-14) or press <ENTER> to continue: 2
Column header: SEX//
Is null allowed? Yes//   (Yes)

Select a number from the list (1-4) or press <ENTER> to continue: 3
Column header: DOB//
Is null allowed? Yes//   (Yes)

1  NAME  [.01]
2  SEX  [.02]
3  DOB  [.03]
4  PRIMARY CLINIC  [.04]

Select a field from the list:  (1-4): 4
Column header: LOCAL FACILITY//
Is null allowed? Yes//   (Yes)
This field is a pointer value (IEN).
Want to automatically insert the lookup value in the schema? No// Y  (Yes)
Select a number from the list (1-4) or press <ENTER> to continue:
OK. A lookup field named ‘.04IEN’ has been added to this schema!
Done!
```

Figure 5-24: Sample Schema Generator command-line dialog



BMXSchemaBuilder - PATIENT DEMOGRAPHICS

FileMan File:

- V VA MOBILE REFER FOR OUTP
- V VA MOBILE SPECIALTY OF REFER
- V VA MOBILE VISIT RELATED
- V VA MOBILE VISIT TYPES
- VA ADMITTING REGULATION
- VA DISPENSE UNIT
- VA DRUG CLASS
- VA GENERIC
- VA HEALTH SUMMARY COMPONENT
- VA HEALTH SUMMARY OBJECTS
- VA HEALTH SUMMARY PARAMETERS
- VA HEALTH SUMMARY TYPE
- VA HEALTH SUMMARY USER PREFEREN
- VA PATIENT
- VA PRODUCT
- VAC ACCOUNTABILITY GROUPS
- VEE VEDD GLOBAL
- VEN EHP AUDIT TRAIL
- VEN EHP CLINIC

Fields:

Field Number	Field Name
0.01	NAME
0.02	SEX
0.03	DATE OF BIRTH
0.033	AGE
0.05	MARITAL STATUS
0.06	RACE
0.07	OCCUPATION
0.08	RELIGIOUS PREFERENCE
0.081	DUPLICATE STATUS
0.082	PATIENT MERGED TO
0.083	CHECK FOR DUPLICATE
0.09	SOCIAL SECURITY NUMBER
0.0901	TERMINAL DIGIT OF SSN
0.0905	1U4N
0.091	REMARKS
0.092	PLACE OF BIRTH [CITY]
0.093	PLACE OF BIRTH [STATE]
0.096	WHO ENTERED PATIENT

Schema File: VA PATIENT (Number 2)

FIELD NUMBER	DATA TYPE	FIELD LENGTH	COLUMN HEADER	READ ONLY	KEY FIELD	NULL ALLOWED	IEN AUTOMATICALLY INCLUDED	A
.01	TEXT	30	NAME	NO	NO	NO		
.02	TEXT	4	SEX	NO	NO	YES		
.03	DATE	21	DOB	NO	NO	YES		
.09	TEXT	30	SSN	YES	NO	YES		
.03TRIGGER1	DATE	21	IDOB	YES	NO	YES		

RPMS Server: LocalHost

Figure 5-25: BMXSchemaBuilder Utility

5.3.9 Using BMX ADO SS to Retrieve Updateable FileMan DataTables

BMXNet supplies the BMX ADO SS remote procedure call to allow developers to retrieve updateable DataTables using schemas stored in the BMX ADO SCHEMA file. *BMX ADO SS* takes up to three parameters and is invoked from *.NET* in the same fashion as other custom RPCs as described in Section 5.1.4 using syntax as demonstrated in the following example.

```
BMX ADO SS^SIEN^DAS^VSTG
```

Figure 5-26: BMX ADO SS Syntax

Note: Internally, the BMX ADO SS remote procedure invokes M routine SS^BMXADO. It is useful when developing and testing schemas to call SS^BMXADO directly from the M command line and then inspect the resulting output array. In this case, an additional parameter, the OUT parameter, is used as the first parameter. This parameter should be passed by reference using the “dot syntax.” OUT does not have to have a value when the call is made. When the call is completed, OUT will contain a text string that is a closed global reference to the location where the output array is stored.

5.3.9.1 BMX ADO SS Parameter 1: SIEN

This is the IEN (or, alternatively, an unambiguous name) of an entry in the BMX ADO SCHEMA file. The schema that is referenced here determines the scope and format (column definitions) of the resulting record set. Of the three parameters that can be passed to BMX ADO SS, this is the only one that is required to have a value. *If DAS and VSTG are undefined, then the resulting record set will contain only the schema string – no data.*

5.3.9.2 BMX ADO SS Parameter 2: DAS

DAS stands for “DA String.” FileMan references internal entry numbers with the local variable “DA”. The value of DAS is null unless the record set includes data. Even then, the DAS will be null unless (1) the record set is a continuation of a previous request and/or (2) the record set is connected to a sub-file.

If the schema is attached to a top level file (i.e., not a sub-file) the DAS will be a single integer corresponding to the FileMan internal entry number (IEN) – specifically, the *DA value of the last record in the previous record set in a continuation process*. In this instance, the DAS is a “seed.” It initiates the iteration, but the record associated with IEN is not actually included in the continuation record set.

Here is an example of a continuation request: the client asks for a list of all patients and the list is to be delivered 50 names at a time by IEN. Assume there are 185 patients with contiguous IENs. Starting with IEN = 1, it will take 4 calls to SS^BMXADO to deliver the entire list. Since iteration is to start at the beginning, the initial value of the seed (DAS) should be null. For the second call, the seed will have a value of 50 because record #50 was the last IEN accessed during the first pass. The seed of the third call will be 100. The seed of the fourth and last call will be 150. On the last call, only 35 names will be returned.

If the schema is attached to a sub-file, the DAS will be comma delimited string of integers corresponding to the DA array. The first integer is the IEN of the top-most file; the second integer is the IEN of the first level descendant file and so on. The last integer is the IEN of an entry in the sub-file. The last IEN will be null (or have a value of zero) unless the record set is part of a continuation process. Thus, the DAS "6,2,7" corresponds to "DA(2),DA(1),DA" where DA(2)=6, DA(1)=2, and DA=7. Do not confuse the DAS with the IENS string utilized by FileMan client server calls. The IENS string is in ascending order (child -> parent) while the DAS is in descending order (parent -> child).

Example of a continuation DAS (i.e., a "seed"): "7"

Example of a DAS for a sub-file: "6,2," (Note the terminal comma)

Example of a DAS for a sub-file seed: "6,2,7"

5.3.9.3 BMX ADO SS Parameter 3: VSTG

VSTG stands for "View String." This parameter is null unless the result set includes data (i.e. a "view"). The VSTG is a "~" (tilde) delimited string with nine pieces. *Within* some of these pieces, there may be additional delimitated data using the " |" or "," characters. *The purpose of the VSTG is to drive the Iterator:* the MUMPS code that populates the rows of the result set. From a functional perspective, the *Iterator* is an IEN generator. It identifies a series of IENs for the home file (or sub-file) specified by the schema.

Note: In general, the Iterator controls the searching and sorting, and the schema controls the formatting.

In most cases, the Iterator will use an index specified in the VSTG. The VSTG may also specify where the iteration is to *start* and *stop* as well as the *maximum* number, if records are to be delivered in one chunk (as part of a continuation process). The VSTG also has a flag to determine is the values in the dataset are in internal or external format. In some cases, non-standard iteration is required or complex filters must be applied to the iteration process. To accommodate this requirement, the VSTG can reference *special entry points* and parameters to be passed to custom coded iteration subroutines. *Finally, the BMX ADO SS is capable of generating* ***multiple record sets in one pass*** *- if the records are connected by a* ***relational join***. The VSTG can contain information to initiate the join(s).

The general structure of the VSTG is shown in *Figur*27.

```
INDEX~START~STOP~MAX~FORMAT~TAG~ROUTINE~PARAM~JOIN
```

Figure 5-27: VSTG Structure

Remember, VSTG is a parameter that is passed to BMX ADO SS, so it may *not* contain the "^" character.

VSTG Piece 1: INDEX

This is the FileMan index that cycles the iterator. If null, the iterator will cycle by IEN. BMX ADO SS understands all common indices used in PCC including "B," "AA," "AC," "AD," etc.

In most cases the data type of START and STOP must correspond to the data type of the field associated with the index. For example, if we are dealing with the "B" index of the PATIENT file, the data type for START and STOP must be text. There is one important exception. If a DINUM relationship exists between the home file and another (reference) file, the data type can correspond to the data type of the *.01* field of the reference file. For example, suppose you specify the "B" index of the PATIENT FILE (9000001). The .01 field is a number, not a name – actually it is a pointer value or IEN. *However*, because the pointer value is DINUM'd to the VA PATIENT file, you *can* use text values for START and STOP. BMX ADO SS will automatically determine it should use the "B" index of the VA PATIENT file to drive the iteration, so text values are acceptable in this instance.

If the "AA" index is used, the START and STOP parameters must be dates in external format (e.g., "1/25/04") AND the PARAM parameter must conform to the following conventions. Determine if the home file has a two level or a three level "AA" index. The VISIT file has a two level "AA" index: patient IEN, and inverse date. The V MEASUREMENT file has a three level "AA" index: patient IEN, measurement type IEN, and inverse date.

For a two level index PARAM will contain "PATIENT IEN|SORT ORDER." SORT ORDER can have a value of "C" for chronological or "R" for reverse chronological. For example: "4591|R" means "iterate chronologically through a date range for patient # 4591."

For a three level "AA" index, PARAM will contain "PATIENT IEN|ENTITY TYPE IEN|SORT ORDER" Example: "4591|2|R" means do reverse chronological iteration for patient #4591 and entity type #2. Typically the entity type is specified by its internal entry number. You may place an accent grave before this number (e.g., "4591|`2|R") – but this is not required. The second piece can also be a text string so long as this string is an unambiguous lookup value in the file that stores the type. For example, 4591|**WT**|R would specify that WEIGHTS are to be returned in the record set.

VSTG Piece 2: START

This is the first lookup value to be used by the Iterator; i.e., the first value that can be used with specified index. START is not a "seed" value because it may actually be included in the iteration. (The seed is contained in the DAS parameter.) START is used only once - to initiate the iteration. If the developer defines MAX, subsequent fetches will use the last IEN delivered as the seed for the next iteration. The value of

START must always precede the value of STOP or the VSTG will be rejected by BMX ADO SS as invalid.

VSTG Piece 3: STOP

This is the last lookup value allowed for the Iterator (the last value used in specified index). The value of STOP may be included in the data set, but no values that follow STOP will be included.

VSTG Piece 4: MAX

This is the maximum number of records returned in the result set in a single chunk. It is independent of START and STOP. If MAX is not specified, it defaults to a value of 100. In certain instances, MAX is ignored. If multiple record sets are returned in a single transaction because the VSTG specifies a join, MAX will be applied to the primary record set but not to any of the related record sets. BMX ADO SS will return as many related record sets as necessary to complete the joins for the primary file. MAX is also ignored when a custom-coded iterator is used.

VSTG Piece 5: FORMAT

If this parameter has a value of “I” – for Internal. All values in the result set will be returned in internal FileMan format.

VSTG Piece 6: TAG

If a custom coded subroutine is to be used for iteration, TAG will contain the name of the entry point *line*.

VSTG Piece 7: ROUTINE

If a custom coded subroutine is to be used for iteration, ROUTINE will contain the name of the entry point *routine*.

VSTG Piece 8: PARAM

When Tag and Routine are not null, PARAM may contain a string that is passed to TAG^ROUTINE to invoke the custom iterator. *Custom iterators can be used whenever there are complex or unusual sorting/filtering requirements for the data rows*. For example, if TAG=”EN”, ROUTINE = “ABC”, BMX ADO SS translates this to D EN^ABC(PARAM).

PARAM can also contain information for use with the “AA” index (see discussion above).

VSTG Piece 8: JOIN

In the context of relational tables, a "join" means that a data value in one table can be used to unambiguously look up an entry in another table. In this way, data elements from rows in two different tables can be joined together – thus creating a "virtual record" that can be viewed and edited. For example, it may be useful to join information from a patient measurements file and a patient registration file so that clinical and demographic information can be viewed together. In the FileMan world, joins are accomplished via pointers and sub-files.

ADO.Net follows a **Master File – Details File** model for managing joins. The master file is a core "reference" or "foundation" file like the PATIENT file or VISIT file. A details file contains information that extends or embellishes the contents of the master file; e.g. a patient problem list file or a patient insurance coverage file. In FileMan, at least one field in details file must point to its master file. From here on, we will use the Master/Details terminology to refer to the primary file and the secondary file that is joined to it. In the ADO environment, the flow of a join is always from master table to details table. Consequently, the BMX ADO SS is limited to "master to detail" joins. Obviously, other types of joins are possible, but these are not supported by BMX ADO SS because of the limitations of ADO. More information on join limitations will be presented later.

In BMX ADO SS, a join is defined by a four piece, comma-delimited string. The *first piece* is the IEN of the details schema that is associated with the details file.

The *second piece* is a specific field identifier in the *master* schema that is used to initiate the join. (The master schema has already been defined by the SIEN parameter.) The data value of this field can be used to do an unambiguous lookup in the details file. In the case of a "sub-file join" - where a parent file is "joined" to its sub-file (described in detail below) - the second piece will contain the text string "SUB."

The *third piece* contains the field identifier in the details schema that is used to complete the join. In the case of a sub-file join, this piece is null.

The *fourth piece* contains special iterator instructions for generating data rows. In fact, the fourth piece is a nested view string (a VSTG within a VSTG). The first two comma-delimited pieces of the join string are mandatory. The third piece is mandatory if the join is standard (i.e., not a sub-file join). The fourth piece (nested VSTG) is optional.

```
schema_ien, master_field_number, details_field_number, view_string
```

Figure 5-28: Join Syntax

Here are a few examples:

```
…|4,.05,.02,
```

Figure 5-29: Example standard join

This is an example of a **standard join**. In the simplest case, only the first three pieces of the join instruction string are specified. Here "4" is the IEN of the schema attached to the details file, ".05" is the field number of the master file that is used to complete the join and ".02" is the field in the details file that completes the join. No special iterator is needed, so the fourth piece is null. Two result sets will be returned in this transaction. The first record set contains the schema and data records from the master file. The second record set contains the schema and "joined" data records from the details file.

The next example shows the join instruction string for a **sub-file join**. Schema 5 is associated with a sub-file of the master file. The third piece is null because this is a sub-file. No special iterator is used to view the sub-file, so the fourth piece is null.

```
…|5,SUB,,
```

Figure 5-30: Example sub-file join

The next example demonstrates the use of a *nested VSTG.*

```
…|6,.001,.02IEN,AA~1/1/1988~12/31/1988~~~~~|WT|R
```

Figure 5-31: Example nested VSTG

Schema 6 is associated with a details file, e.g., V MEASUREMENTS. The join is made via a link between the .001 field (IEN) of the master file (e.g., the PATIENT file) and the ".02IEN" field of the details file. The fourth piece specifies that the iterator for generating the detail data rows will use the "AA" index, and it will limit the output to weights ("WT") entered in 1988. The detail data rows will be sorted in reverse ("R") chronological order.

It is possible to define more than one join in a VSTG; i.e. *the master file can be joined to multiple details files in the same transaction.* This is called a **compound join**. To accomplish this, the join segment is subdivided into multiple pieces by conforming to the following convention: each 3-piece set of join instructions must be separated by the text string "@JOIN@". Standard joins and sub-file joins can be mixed in any order within a compound join. *Figure*2, which follows, shows an example of a compound join string.

```
…|6,.001,.02IEN,AA~1/1/1988~12/31/1988~~~~~|WT|R@JOIN@7,.001,.02IEN,AA~1/1/196
0~6/30/2004~~~~~|C
```

Figure 5-32: Example compound join

In this case, *three* result sets will be returned in the transaction: the first for the master schema and its associated data records, the second for the details schema #6 and its associated data records, and the third for details schema #7 and its associated data records.

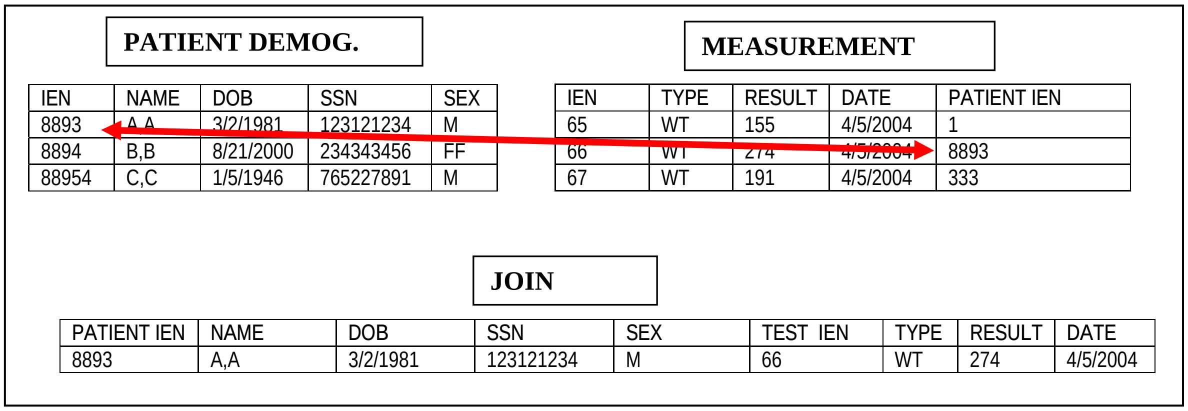
5.3.10 Detailed BMX ADO SS Examples

5.3.10.1 Standard Join

In a standard join, the value of a data element in the primary file is the IEN (.001 field) of an unambiguous IEN (primary key) in a second file. When BMX ADO SS is instructed to perform a standard join, it generates multiple record sets in a *single transaction*. The output contains the record set for the primary file as well as the record sets of any secondary files that are joined to it. The record sets of the secondary files contain only those records that are required to complete the join to the data rows the primary file.

Note: Extended joins (i.e., joins that require simultaneous traversal of *three* or more tables) are not supported in this version of BMXNet.

Suppose you have two schemas: a primary schema called PATIENTS and a secondary schema called MEASUREMENTS. The PATIENTS schema is associated with a file (table) called PATIENT DEMOGRAPHICS and the MEASUREMENTS schema is associated with a file (table) called MEASUREMENTS. Assume that there is a PATIENT IEN field in the MEASUREMENTS file. This field always contains a valid IEN of a record in the PATIENT file. The IEN is also contained in the *.001* field (the key field) of the PATIENT file. The IEN value can be used to join the two schemas/files together. See the following examples.



IEN	NAME	DOB	SSN	SEX
8893	A,A	3/2/1981	123121234	M
8894	B,B	8/21/2000	234343456	FF
88954	C,C	1/5/1946	765227891	M

IEN	TYPE	RESULT	DATE	PATIENT IEN
65	WT	155	4/5/2004	1
66	WT	274	4/5/2004	8893
67	WT	191	4/5/2004	333

PATIENT IEN	NAME	DOB	SSN	SEX	TEST IEN	TYPE	RESULT	DATE
8893	A,A	3/2/1981	123121234	M	66	WT	274	4/5/2004

Figure 5-33: Graphic example

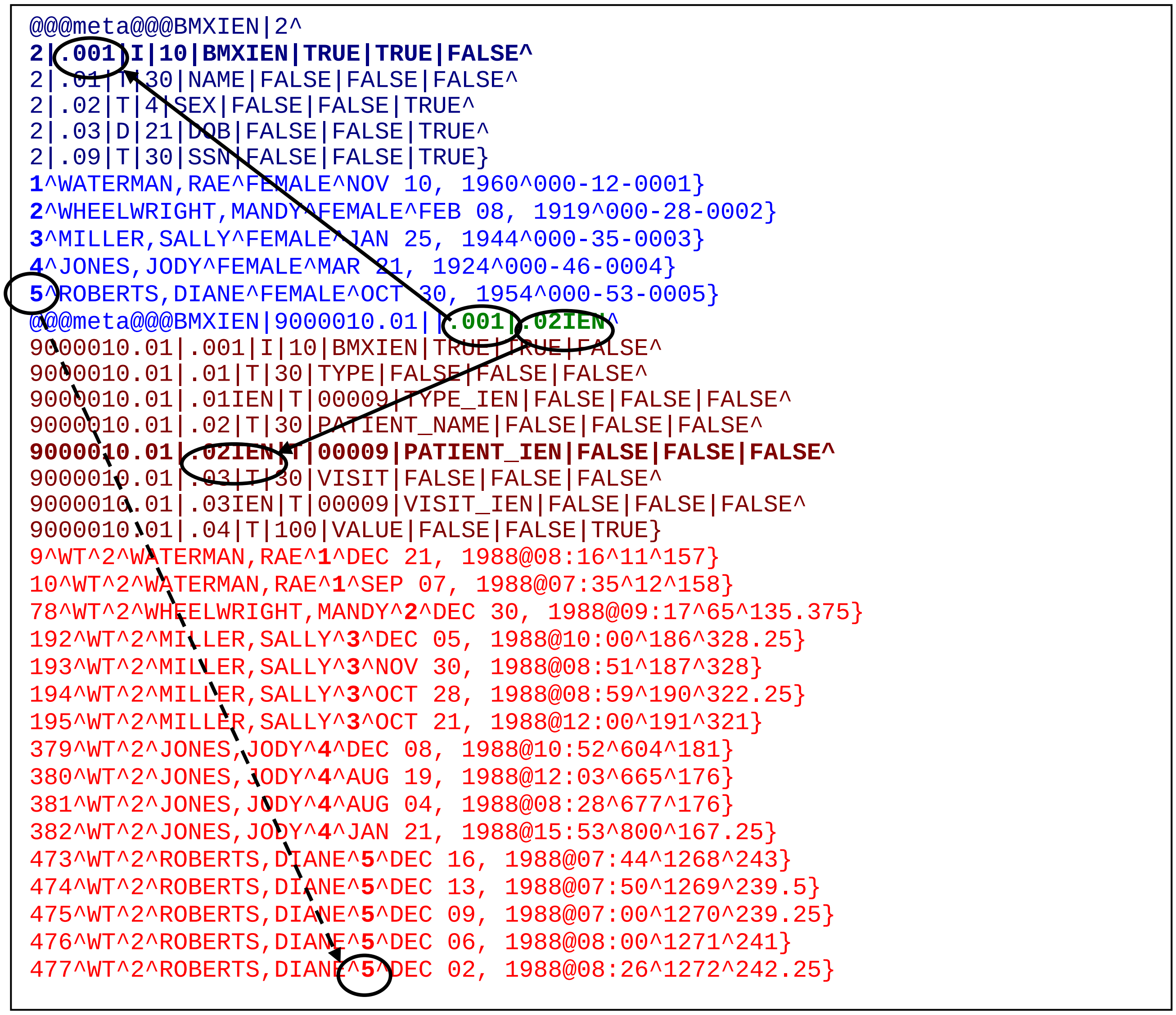
In the next example, we present M code that produces a standard join as well as the result set.

```
S SIEN1=1 ; SIEN1 = THE IEN OF THE MASTER SCHEMA (PATIENT)
S SIEN2=2 ; SIEN2 = THE IEN OF THE DETAILS SCHEMA (MEASUREMENTS)
S VSTG="~1~5~~~~~~"_SIEN2_",.001,AA~1/1/1988~12/31/1988~~~~~|WT|R
D SS^BMXADO(.OUT,SIEN1,"",VSTG) ; GENERATE THE JOINED RECORD
Q
```

Figure 5-34: M Code Sample Standard Join

BMX ADO SS (called internally here from the M command line using SS^BMXADO) is instructed to join two files: PATIENT and MEASUREMENTS. The iterator for the master (PATIENT) file restricts the data to the first five IENs. The iterator for the details (MEASUREMENTS) file restricts data to weights entered during the year 1988. The details data rows are to be sorted in reverse chronological order.

The result set produced by this code is shown in the following example.



```
@@@meta@@@BMXIEN|2^
2|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^
2|.01|T|30|NAME|FALSE|FALSE|FALSE^
2|.02|T|4|SEX|FALSE|FALSE|TRUE^
2|.03|D|21|DOB|FALSE|FALSE|TRUE^
2|.09|T|30|SSN|FALSE|FALSE|TRUE}
1^WATERMAN,RAE^FEMALE^NOV 10, 1960^000-12-0001}
2^WHEELWRIGHT,MANDY^FEMALE^FEB 08, 1919^000-28-0002}
3^MILLER,SALLY^FEMALE^JAN 25, 1944^000-35-0003}
4^JONES,JODY^FEMALE^MAR 21, 1924^000-46-0004}
5^ROBERTS,DIANE^FEMALE^OCT 30, 1954^000-53-0005}
@@@meta@@@BMXIEN|9000010.01||.001|.02IEN^
9000010.01|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^
9000010.01|.01|T|30|TYPE|FALSE|FALSE|FALSE^
9000010.01|.01IEN|T|00009|TYPE_IEN|FALSE|FALSE|FALSE^
9000010.01|.02|T|30|PATIENT_NAME|FALSE|FALSE|FALSE^
9000010.01|.02IEN|T|00009|PATIENT_IEN|FALSE|FALSE|FALSE^
9000010.01|.03|T|30|VISIT|FALSE|FALSE|FALSE^
9000010.01|.03IEN|T|00009|VISIT_IEN|FALSE|FALSE|FALSE^
9000010.01|.04|T|100|VALUE|FALSE|FALSE|TRUE}
9^WT^2^WATERMAN,RAE^1^DEC 21, 1988@08:16^11^157}
10^WT^2^WATERMAN,RAE^1^SEP 07, 1988@07:35^12^158}
78^WT^2^WHEELWRIGHT,MANDY^2^DEC 30, 1988@09:17^65^135.375}
192^WT^2^MILLER,SALLY^3^DEC 05, 1988@10:00^186^328.25}
193^WT^2^MILLER,SALLY^3^NOV 30, 1988@08:51^187^328}
194^WT^2^MILLER,SALLY^3^OCT 28, 1988@08:59^190^322.25}
195^WT^2^MILLER,SALLY^3^OCT 21, 1988@12:00^191^321}
379^WT^2^JONES,JODY^4^DEC 08, 1988@10:52^604^181}
380^WT^2^JONES,JODY^4^AUG 19, 1988@12:03^665^176}
381^WT^2^JONES,JODY^4^AUG 04, 1988@08:28^677^176}
382^WT^2^JONES,JODY^4^JAN 21, 1988@15:53^800^167.25}
473^WT^2^ROBERTS,DIANE^5^DEC 16, 1988@07:44^1268^243}
474^WT^2^ROBERTS,DIANE^5^DEC 13, 1988@07:50^1269^239.5}
475^WT^2^ROBERTS,DIANE^5^DEC 09, 1988@07:00^1270^239.25}
476^WT^2^ROBERTS,DIANE^5^DEC 06, 1988@08:00^1271^241}
477^WT^2^ROBERTS,DIANE^5^DEC 02, 1988@08:26^1272^242.25}
```

Figure 5-35: Result set from standard join

In *Figure*35, the schema record of the primary table (PATIENT) is in dark blue and the data records are in blue. The schema record of the secondary table (MEASUREMENTS) is in dark red and the data records are in red. The End of Record (EOR) character is represented by "}." Five PATIENT records are joined to sixteen MEASUREMENTS records. The join is based on the value of the .001 field in the PATIENT file that is linked to the .02IEN (PATIENT_IEN) field of the MEASUREMENTS file. The values (IENs) used to complete the join are in bold print. One of the joins is highlighted by a dashed arrow. Note that the second record set contains only those records that are required to complete the join. There is a one-to-many relationship between master and details records.

The details record set contains special information to indicate that it is joined to the master record set. We know that the .001 field of the master file and the .02IEN field of the details file are responsible for the join. Consequently, in the Introductory Segment of the schema record of the details record set, ".001" is the value of the 4th piece and .02IEN is of the 5th piece (both pieces are shown in dark green).

5.3.10.2 Sub-file Join

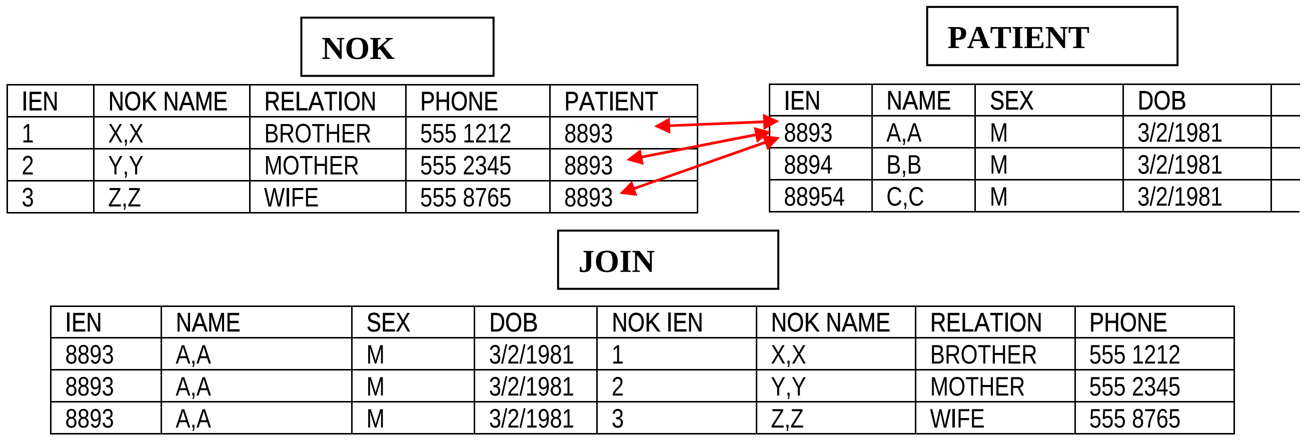
FileMan supports a hierarchical file structure through the use of sub-files (also known as "multiples"). By convention, relational tables are *never* structurally hierarchical; i.e., they never contain sub-tables or nested tables. In the relational world, hierarchical relationships are supported via joins. BMX ADO SS is capable of creating a record set that makes a sub–file and its parent *appear* to be two *joined* tables that can be shared with the rest of the (relational) world. In this way, FileMan files and sub-files become compatible with all other ADO.NET-compliant applications.

All of this is accomplished through the use of "virtual fields" in the schema. The precedent for a virtual field has already been established. In FileMan, the IEN is contained in the virtual field: *.001*. We extend this convention to sub-files such that the IEN of a sub-file will be stored in its .001 field and *the IEN of its parent file will be stored in the .0001 field.* BMX ADO SS automatically generates a .001 field for all files and a .0001 field if a sub-file is defined. The .0001 field is used to "join" the sub-file to the parent file. In the following example, consider a PATIENT file that contains a NEXT OF KIN multiple:

```
.001  IEN
.01   NAME
.02   DOB
.03   SEX
1  NEXT OF KIN (multiple)
   .01   NOK NAME
   .02   RELATIONSHIP TO PATIENT
   .02   PHONE
```

Figure 5-36: Sample of a patient file containing next of kin multiple

BMX ADO SS translates this hierarchy into two joined files. See the following example.



NOK

IEN	NOK NAME	RELATION	PHONE	PATIENT
1	X,X	BROTHER	555 1212	8893
2	Y,Y	MOTHER	555 2345	8893
3	Z,Z	WIFE	555 8765	8893

PATIENT

IEN	NAME	SEX	DOB
8893	A,A	M	3/2/1981
8894	B,B	M	3/2/1981
88954	C,C	M	3/2/1981

JOIN

IEN	NAME	SEX	DOB	NOK IEN	NOK NAME	RELATION	PHONE
8893	A,A	M	3/2/1981	1	X,X	BROTHER	555 1212
8893	A,A	M	3/2/1981	2	Y,Y	MOTHER	555 2345
8893	A,A	M	3/2/1981	3	Z,Z	WIFE	555 8765

Figure 5-37: Graphic example translated files

The current version of BMXNet does not support traversal of more than one sub-file level. In other words, you can not use sub-sub-files in a join. An analogous restriction applies to standard join where no more than two tables can be traversed to complete the join.

The JOIN syntax for a sub-file is the usual pair except instead of a field number, the word “SUB” is in the second piece:

```
SCHEMA2,SUB; E.G., “9,SUB”.
```

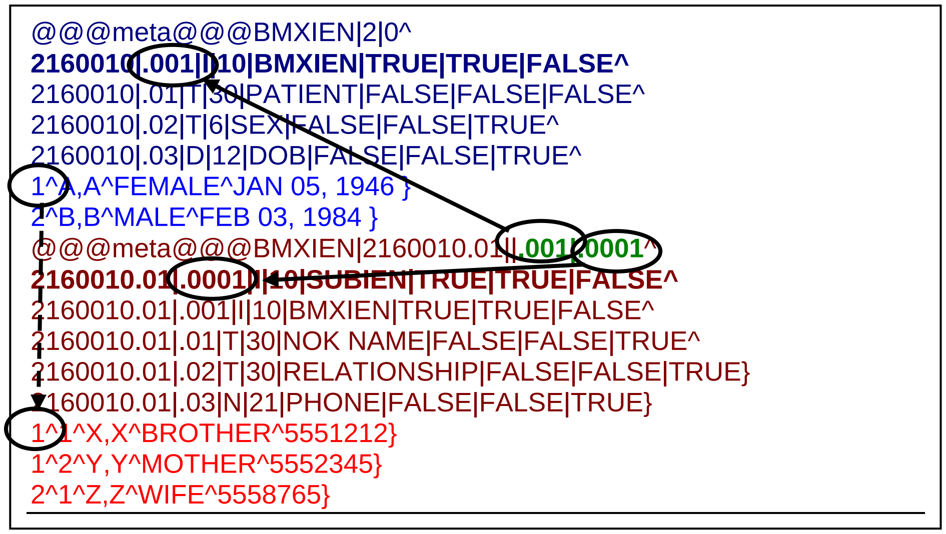
Figure 5-38: Join syntax for a sub-file

In the following example, schema #3 is attached to the master file, and schema #4 is attached to a sub-file.

```
S SIEN1=3 ; SIEN1 = THE IEN OF THE PARENT FILE’S SCHEMA (PATIENT)
S SIEN2=4 ; SIEN2 = THE IEN OF THE SUB-FILE’S SCHEMA (NOK)
S VSTG="~1~2~~~~~~"_SIEN2_”,SUB" ;VSTG RETURNS DATA FOR PTS 1 & 2
D SS^BMXADO(.OUT,SIEN1,"",VSTG) ; GENERATE THE SUB-FILE JOIN
Q
```

Figure 5-39: Sample M Code to generate a sub-file join

BMX ADO SS produces the following array for this kind of join. Note that it contains multiple record sets. Also note that the second record set contains a .0001 field. Also note that the 4th piece of the Introductory Segment of the secondary schema record has a value of ".001". The 5th piece has a value of ".0001" (both shown in dark green). This specifies that the two record sets are related by a "sub-file" join.



```
@@@meta@@@BMXIEN|2|0^
2160010|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^
2160010|.01|T|30|PATIENT|FALSE|FALSE|FALSE^
2160010|.02|T|6|SEX|FALSE|FALSE|TRUE^
2160010|.03|D|12|DOB|FALSE|FALSE|TRUE^
1^A,A^FEMALE^JAN 05, 1946 }
2^B,B^MALE^FEB 03, 1984 }
@@@meta@@@BMXIEN|2160010.01|I|.001|.0001^
2160010.01|.0001|I|10|SUBIEN|TRUE|TRUE|FALSE^
2160010.01|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^
2160010.01|.01|T|30|NOK NAME|FALSE|FALSE|TRUE^
2160010.01|.02|T|30|RELATIONSHIP|FALSE|FALSE|TRUE}
2160010.01|.03|N|21|PHONE|FALSE|FALSE|TRUE}
1^1^X,X^BROTHER^5551212}
1^2^Y,Y^MOTHER^5552345}
2^1^Z,Z^WIFE^5558765}
```

Figure 5-40: Graphic example of array produced for join

5.3.10.3 Compound Joins

In the next example, we see the record sets that result from a compound join. Here the master file, PATIENT, is joined to two details files, V MEASUREMENT and V MED. The result, displayed in *Figure*40, which follows, is a set of three record sets generated in one transaction. Note that all the data rows of the two details record sets are linked to the five IENs in the data rows in the master record set. Also note the join information located in the 4th and 5th pieces of the Introductory Segments of each details schema record (dark green).

```
S SIEN1=1 ; SIEN1 = THE IEN OF THE MASTER SCHEMA (PATIENT)
S SIEN2=2 ; SIEN2 = THE IEN OF THE DETAILS 1ST DETAILS SCHEMA (MEDS)
S SIEN2=3 ; SIEN3 = THE IEN OF THE 2ND DETAILS SCHEMA (MEASUREMENTS)
S VSTG="~1~5~~~~~~" ; MASTER RECORD SET
S VSTG=VSTG_SIEN2_",.001,.02IEN,AA~1/1/1988~12/31/1988~~~~~|WT|R" ; 1ST
DETAILS RECORD SET
S VSTG=VSTG_SIEN3_",.001,.02IEN,AA~1/1/1960~12/31/2003~~~~~|R" ; 2ND DETAILS
RECORD SET
D SS^BMXADO(.OUT,SIEN1,"",VSTG) ; GENERATE THE RECORDS FOR THE JOIN
Q
```

Figure 5-41: Sample M Code to generate a compound join

```
@@@meta@@@BMXIEN|2|^
2|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^
2|.01|T|30|NAME|FALSE|FALSE|FALSE^
2|.02|T|4|SEX|FALSE|FALSE|TRUE^
2|.03|D|21|DOB|FALSE|FALSE|TRUE^
2|.09|T|30|SSN|FALSE|FALSE|TRUE}
1^WATERMAN,RAE^FEMALE^NOV 10, 1960^000-12-0001}
2^WHEELWRIGHT,MANDY^FEMALE^FEB 08, 1919^000-28-0002}
3^MILLER,SALLY^FEMALE^JAN 25, 1944^000-35-0003}
4^JONES,JODY^FEMALE^MAR 21, 1924^000-46-0004}
5^ROBERTS,DIANE^FEMALE^OCT 30, 1954^000-53-0005}
@@@meta@@@BMXIEN|9000010.14||.001|.02IEN^
9000010.14|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^
9000010.14|.01|T|30|MEDICATION|FALSE|FALSE|FALSE^
9000010.14|.01IEN|T|00009|MEDICATION_IEN|FALSE|FALSE|FALSE^
9000010.14|.02|T|30|PATIENT NAME|FALSE|FALSE|FALSE^
9000010.14|.02IEN|T|00009|PATIENT_IEN|FALSE|FALSE|FALSE^
9000010.14|.03|T|30|VISIT|FALSE|FALSE|FALSE^
9000010.14|.03IEN|T|00009|VISIT_IEN|FALSE|FALSE|FALSE^
9000010.14|.05|T|32|SIG|FALSE|FALSE|TRUE^
9000010.14|.06|I|7|QUANTITY|FALSE|FALSE|TRUE}
61^IBUPROFEN 400MG TAB^305^WATERMAN,RAE^1^FEB 14, 2002@11:54^71015^T1T QID^40}
67^IBUPROFEN 400MG TAB 40S^84120^WATERMAN,RAE^1^OCT 30, 2002@17:00^71043^TIT TID^90}
21^IBUPROFEN 400MG TAB^305^WHEELWRIGHT,MANDY^2^JAN 11, 1990@12:00^70165^T1T QID^40}
22^CHLORPROPAMIDE 250MG TAB^954^MILLER,SALLY^3^JAN 01, 1990@12:00^70195^T1T BID^60}
23^DIGOXIN 0.25MG TAB ^172^MILLER,SALLY^3^JAN 01, 1990@12:00^70195^T1T DY FH^30}
24^FUROSEMIDE 40MG TAB^654^MILLER,SALLY^3^JAN 01, 1990@12:00^70195^T1T QD^30}
58^DIGOXIN 0.25MG TAB ^172^JONES,JODY^4^AUG 09, 1990@12:00^70247^T1T DY FH^30}
59^IBUPROFEN 400MG TAB^305^JONES,JODY^4^AUG 09, 1990@12:00^70247^T1T QID^40}
60^IBUPROFEN 400MG TAB^305^JONES,JODY^4^AUG 25, 1990@12:00^70249^T1T QID^40}
52^CHLORPROPAMIDE 250MG TAB^954^ROBERTS,DIANE^5^NOV 01, 1989@08:42^1249^T1T BID^60}
25^IBUPROFEN 400MG TAB^305^ROBERTS,DIANE^5^JAN 05, 1990@12:00^70196^T1T QID^120}
26^CHLORPROPAMIDE 250MG TAB^954^ROBERTS,DIANE^5^JAN 05, 1990@12:00^70196^T2T BID^120}
@@@meta@@@BMXIEN|9000010.01||.001|.02IEN^
9000010.01|.001|I|10|BMXIEN|TRUE|TRUE|FALSE^
9000010.01|.01|T|30|TYPE|FALSE|FALSE|FALSE^
9000010.01|.01IEN|T|00009|TYPE_IEN|FALSE|FALSE|FALSE^
9000010.01|.02|T|30|PATIENT NAME|FALSE|FALSE|FALSE^
9000010.01|.02IEN|T|00009|PATIENT_IEN|FALSE|FALSE|FALSE^
9000010.01|.03|T|30|VISIT|FALSE|FALSE|FALSE^
9000010.01|.03IEN|T|00009|VISIT_IEN|FALSE|FALSE|FALSE^
9000010.01|.04|T|100|VALUE|FALSE|FALSE|TRUE}
9^WT^2^WATERMAN,RAE^1^DEC 21, 1988@08:16^11^157}
10^WT^2^WATERMAN,RAE^1^SEP 07, 1988@07:35^12^158}
85^WT^2^WHEELWRIGHT,MANDY^2^MAR 18, 1988@09:15^78^137.5}
86^WT^2^WHEELWRIGHT,MANDY^2^MAR 08, 1988@14:06^80^135.125}
87^WT^2^WHEELWRIGHT,MANDY^2^FEB 12, 1988@10:36^83^142}
88^WT^2^WHEELWRIGHT,MANDY^2^JAN 22, 1988@09:08^85^139}
192^WT^2^MILLER,SALLY^3^DEC 05, 1988@10:00^186^328.25}
193^WT^2^MILLER,SALLY^3^NOV 30, 1988@08:51^187^328
379^WT^2^JONES,JODY^4^DEC 08, 1988@10:52^604^181}
380^WT^2^JONES,JODY^4^AUG 19, 1988@12:03^665^176}
381^WT^2^JONES,JODY^4^AUG 04, 1988@08:28^677^176}
382^WT^2^JONES,JODY^4^JAN 21, 1988@15:53^800^167.25}
473^WT^2^ROBERTS,DIANE^5^DEC 16, 1988@07:44^1268^243}
474^WT^2^ROBERTS,DIANE^5^DEC 13, 1988@07:50^1269^239.5}
475^WT^2^ROBERTS,DIANE^5^DEC 09, 1988@07:00^1270^239.25}
476^WT^2^ROBERTS,DIANE^5^DEC 06, 1988@08:00^1271^241}
```

Figure 5-42: Graphic example of result sets from a compound join

5.3.10.4 Nested View Strings

A nested view string is almost identical to an ordinary (top level) view string. However, the following differences apply to the nested VSTG.

1. The MAX parameter is ignored. The resulting set will contain as many data rows as necessary to complete the joins. However, the MAX parameter of the master VSTG *is* utilized by BMX ADO SS.

2. If the START and STOP parameters of the nested VSTG contain lookup IENs, the values will be ignored. The lookup IENs of details files are determined by the IENs of the master file. In fact BMX ADO SS will automatically manipulate the START and STOP values as record generation progresses. If the user supplies START and STOP IENS, they will be ignored and overwritten by BMX ADO SS. Note that in most cases when the AA index is used, START and STOP do not contain IEN values, so they *can* be used to control iteration under these circumstances. For example, if the details file is V MEASUREMENTS, START and STOP will contain dates that limit the values returned by the iterator. The START and STOP values of the master VSTG are *not* ignored by BMX ADO SS.

3. If (1) the master file is A PATIENT file and (2) if the AA index is used by the details file iterator, then the first piece of the PARAM parameter of the nested VSTG should be null. This piece normally contains the patient IEN. However, in a join, the value is automatically supplied by BMX ADO SS based on the contents of the master data rows. If the user supplies a value, it will be overwritten.

5.3.10.5 Sample VSTG Strings

Assume schema 10 is connected to the PATIENT file. Using IENs, iterate through IEN 10000. Start with the first entry in the list:

```
BMX ADO SS^10^0^~0~~100000
```

Figure 5-43: Sample of string

Assume schema 3 is connected to the VISIT file. Using the “AC” index, return all visits for patient IEN = 1. Return 5 records at a time:

```
BMX ADO SS^3^0^AC~1~1~5
```

Figure 5-44: Sample of string

Assume schema 3 is connected to the VISIT file. Using the “AC” index, return all visits for patient IEN = 1. Return 5 records at a time. Seed the iteration with DA = 2944:

```
BMX ADO SS^3^2994^AC~1~1~5
```

Figure 5-45: Sample of string

Assume schema 4 is connected to the VA PATIENT file. Using the “B” index, return all patients whose last name begins with “N”. Return 10 records at a time:

```
BMX ADO SS^4^0^B~N~NZZZ~10
```

Figure 5-46: Sample of string

Assume schema 8 is connected to the V MEASUREMENT file. Return all WEIGHTS for patient 1 in the date range 5/1/82 to 5/7/2000. Iterate in reverse chronological order:

```
BMX ADO SS^8^0^AA~5/1/1982~5/7/2000~10~~~~1|WT|R
```

Figure 5-47: Sample of string

Assume schema 3 is connected to the VISIT FILE. Assume schema 4 is connected to the VA PATIENT file. Join these files using the .05 field (PATIENT) of the visit file and the .001 field of the VA PATIENT file. Return all visits in the date range 1/3/85 to 1/4/85. Return no more than five master records (i.e., visits). Start with the first record in the date range:

```
BMX ADO SS^3^0^B~1/3/1985~1/4/1985~5~~~~~4,.05,.001
```

Figure 5-48: Sample of string

Assume schema 1 is connected to the TEST file. Assume schema 9 is connected to a sub-file in TEST. The IENs of the TEST file will drive the iteration. Start with the first records. Return an ANR for the file entries and related sub-file entries as shown in the following example.

```
BMX ADO SS^1^0^~~~~~~~~9,SUB
```

Figure 5-49: Sample of string

5.3.10.6 Sample Custom Iterator Code

If a custom iterator is used, the START, STOP and MAX parameters may be ignored. All iteration can be controlled by the value of PARAM. All custom iterators are extrinsic functions that return the null value (“”) or the last DA accessed (the “seed” of the next iteration group – if continuation is enabled). Consider the following example:

```
BMX ADO SS^3^0^~~~~~DAIT~BMXADOV~1|2|3~
```

Figure 5-50: Sample of iterator code screen

The PARAM “1|2|3” is passed to the extrinsic function $$DAIT^BMXADOV. This custom iterator loops through the IENs in the PARAM string (delimited by the “|”) to generate the record set.

```
DAIT(PARAM,IENS,MAX,OUT,TOT)  ; EP - SET OF IENS ITERATION.
   ; THE PARAM CONTAINS A "|" SET OF DAS STRINGS
   ; ALL VALUES ARE RETURNED.  MAX IS NOT CHECKED.
   ; START AND STOP ARE IRRELEVANT
   ; NOT TO BE USED WITH SUBFILES
   N PCE,DA,XIT,IENS,L,DAS
   S L=$L(PARAM,”|”)
   F PCE=1:1:L S DA=$P(PARAM,B,PCE) Q:’DA  D DATA^BMXADOV1(IENS,DA)
   Q ""
   ;
```

Figure 5-51: Sample of iterator code screen

This extrinsic function demonstrates the characteristics of that should be included in all custom iterators (shown in **bold** text). There must be five parameters in the entry point: PARAM,IENS,MAX,OUT,TOT. PARAM comes from VSTG. Even though they do not appear to be used by the extrinsic function, do not change the value or NEW the variables IENS, MAX, OUT or TOT. The entry point **DATA^BMXADOV1** must be called with the IENS and DA parameters. *DATA^BMXADOV1 generates the record set data rows and places them in the OUT array*. Assume that the schema IEN and other required variables are defined and available to DATA^BMXADOV1.

Here is another example:

```
BMX ADO SS^3^0~~~~~DUPV~BMXADOV~1|4/19/04@1PM|I|4585|A~
```

Figure 5-52: Sample of iterator code screen

Given a date and other visit attributes, this extrinsic function returns possible duplicate visits in the record set. The required components of the function are shown in red. The iterator (sort) code is shown in blue. The filter (search) code is shown in green.

```
DUPV(PARAM,IENS,MAX,OUT,TOT)  ; EP - DUPLICATE VISIT ITERATION
   ; PARAM: 'DFN|VISIT TIMESTAMP|TYPE|LOCATION|CATEGORY
   ; PATIENT DFN AND VISIT TIMESTAMP (EXTERNAL DATE FORMAT) MUST EXIST.
   ; THE OTHER 3 DUP PARAMETERS WILL BE CHECKED ONLY IF THEY ARE DEFINED.
   ; ALL DUPS ARE RETURNED.  MAX,START,STOP ARE IGNORED
   N DFN,TIME,TYPE,LOC,CAT,IDT,VIEN,DAY,X,PATIENT,Y,%DT,FMTIME,DA,IENS
   S DFN=+PARAM,TIME=$P(PARAM,”|”,2),TYPE=$P(PARAM,”|”,3),
S LOC=$P(PARAM,”|”,4),CAT=$P(PARAM,”|”,5)
   I $D(^DPT(+$G(DFN),0)),$L($G(TIME))
   E  Q ""
   S X=TIME,%DT="T" D ^%DT I Y=-1 Q
   S FMTIME=Y
   S (IDT,DAY)=9999999-(FMTIME\1),IDT=IDT-.0000001
   F  S IDT=$O(^AUPNVSIT("AA",DFN,IDT)) Q:$E(IDT,1,7)'=DAY  S
VIEN=999999999999 F  S VIEN=$O(^AUPNVSIT("AA",DFN,IDT,VIEN),-1) Q:'VIEN  D
   . S X=$G(^AUPNVSIT(VIEN,0)) I '$L(X) Q  ; VISIT DATA MUST EXIT
   . I $P(X,U,11) Q  ; MUST BE AN 'ACTIVE' VISIT - NOT 'DELETED'
   . I $L(TYPE),TYPE'=$P(X,U,3) Q
   . I $L(LOC),LOC'=$P(X,U,6) Q
   . I $L(CAT),CAT'=$P(X,U,7) Q
   . S DA=VIEN,IENS=DA_”,”
   . D DATA^BMXADOV1(IENS,DA)
   . Q
   Q ""
```

Figure 5-53: Sample of possible duplicate visits

Note that the purpose of custom iterator code is simply to generate DAs. It is the *schema* that actually determines how the data is to be formatted in the record set.

5.3.10.7 Managing Chunks

The VSTG parameter MAX determines the maximum number of records that will be passed back in a given BMX ADO SS transaction. If more than MAX records exist, BMX ADO SS will return records in MAX sized “*chunks*”. The secret of managing chunks is to understand the “*SEED*” parameter. The SEED is the IEN of the last record in the DataTable. *It will only have a non-zero value if there are* ***more*** *records to display*. Retrieve the seed from DataTable.ExtendedProperties[“fmSeed”]. If there is a positive integer in that piece, that value can be used to start the next round of iteration. The following C# sample code demonstrates the how the SEED is used to return data in chunks.

```
CHUNK ; ITERATE IN CHUNKS
   ; RE-ITERATE USING THE "B" INDEX
   ; START WITH PT IEN 5 AS THE "SEED", STOP AFTER PATIENT NAME = "D", MAX #
RECORDS RETURNED = 5 PER CHUNK
   N %,SIEN,SEED,INTRO
   S SEED=0
   S SIEN=$$SCHEMA("PATIENT")
   F  D  I '$G(SEED) Q
   . D SS^BMXADO(.OUT,SIEN,SEED,"B~C~D~5")
   . D DISP(OUT) R %:$G(DTIME,60) E  S SEED="" Q
   . I %?1"^" S SEED="" Q
   . S INTRO=$P(@OUT@(1),U,1)
   . S SEED=$P(INTRO,"|",3)
   . K ^TMP("BMX ADO",$J)
. Q
Q
```

Figure 5-54: Sample of screen used in managing chunks

5.3.11 Updating FileMan DataTables from .NET Applications

BMXNet supports DataTable updates through its BMXNetDataAdapter and the BMXNetCommand classes. The following C# code, copied from the BMXSchemaBuilder sample application, shows a generic routine that updates changes to an ADO.NET DataTable. Note that no RPCs are called and that the code is standard .NET. See Section 7.0 for References and Sources on using .NET and ADO.NET.

```
public int Update(DataTable dt, string sSelectCmd)
{
   DataTable   dtc = dt.GetChanges();
   Int      nRet = 0;
   if (dtc != null)
   {
      //Build UPDATE command based on info stored in table's extended property
set
      BMXNetConnection conn = new BMXNetConnection(m_ci.bmxNetLib);

      BMXNetCommand bmxSelectCmd = (BMXNetCommand) conn.CreateCommand();
      bmxSelectCmd.CommandText = sSelectCmd;
      m_da.SelectCommand = bmxSelectCmd;

      BMXNetCommand bmxUpdateCmd = new BMXNetCommand();
      bmxUpdateCmd.Connection = conn;

      DataTable dtSchema = m_da.FillSchema(dtc, SchemaType.Source);

      //Call BMXBuildUpdateCommand(dtSchema) to set up parameters
      bmxUpdateCmd.BMXBuildUpdateCommand(dtSchema);

      //Link command to data adapeter
      m_da.UpdateCommand = bmxUpdateCmd;

      //Call adapter's Update method
      nRet = m_da.Update(dtc);

      //Accept the changes to the datatable
      dt.AcceptChanges();
   }
   return nRet;
}
```

Figure 5-55: Sample of updating FileMan datatables

5.3.11.1 Testing the BMXNet Updating Process in M

When designing an application, developers may wish to test the BMXNet filing process from within the Cache environment. This is accomplished via the entry point BAFM^BMXADOF1.

```
D BAFM^BMXADOF1(.OUT,CREF)
```

Figure 5-56: Sample of entry point

There is only one input variable: CREF. This is the closed reference that specifies the global array containing the record set. The record set array is stored under ^TMP(“BMX ADO”,$J,… , so CREF = “^TMP(“”BMX ADO””,$J)”. Once the call is made, BAFM gathers all the required information from the ANR, reformats it, and passes the update string to FILE^BMXADOF. BAFM can update multiple records in one “pass”. All the data rows in the record set will be filed – assuming that they can pass the internal verification processes of the both the Filer and FileMan.

Acknowledgement and error messages from the update will be returned in the OUT array. Once BAFM is called, no user intervention is required.

The following code demonstrates the use of BAFM. To run this code on your system, change the values of the patient DFN (“1”) and visit IEN (“71164”) - both are displayed in bold text – to values that are valid in your test system. This code is distributed with BMXNet and can be found in ADD^BMXADOX1.

```
ADD   ; ADD A NEW ENTRY
   ; THIS IS A 2 STEP PROCESS:
   ;   FIRST GET THE SCHEMA FOR THE FILE YOU WISH TO UPDATE
   ;     THIS SCHEMA TYPICALLY BEGINS WITH THE WORD "UPDATE"
   ;     IT CONTAINS NO ID IR IEN FIELDS
   ;   SECOND ADD THE DATA NODE TO THE ARRAY
   ;     THE DATA NODE HAS EXACTLY THE SAME FORMAT AS A DATA STRING IN THE ANR
   ;     THE FIRST PIECE OF THE DATA STRING CONTAINS THE IEN OF THE RECORD.
;     SINCE THE RECORD HAS NOT BEEN ADDED YET, THE FIRST PIECE IS NULL.
   ;     IN THE DATA STRING, ALL POINTER VALUES ARE PRECEDED BY THE “`”
CHARACTER
   ;     EACH DATA STRING IS ALWAYS TERMINATED WITH $C(30)
   ;     MULTIPLE DATA STRINGS CAN BE APPENDED AS NEW NODES AT THE BOTTOM OF
THE ARRAY
   ;     IN THIS CASE WE ARE ADDING A RECORD TO THE V MEASUREMENT FILE
   ;     DATA STRING="^MEASUREMENT TYPE IEN^PATIENT DFN^VISIT
IEN^RESULT"_$C(30)
   ;   THERE ARE 2 INPUT PARAMS:
   ;     THE CLOSED REF WHERE THE INPUT ARRAY IS STORED
   ;     SINCE IT IS PASSED BY REFERENCE, The "OUT" PARAMETER CAN BE NULL OR
UNDEFIEND.
   ;     OUT WILL BE DEFINED AT THE CONCLUSION OF THE TRANSACTION.
   ;   ACKNOWLEDGEMENT AND ERROR MESSAGES ARE RETURNED IN THE OUT ARRAY
   ;   OUT(1)="OK|ien" WHERE ien IS THE IEN OF THE RECORD THAT HAS BEE ADDED.
   ;   IF THE TRANSACTION FAILED, AN ERROR MSG WILL BE IN THE OUT ARRAY
   ;
   N OUT,%,SIEN,NODE
   S SIEN=$$SCHEMA("UPDATE MEASUREMENTS") ; GET THE SCHEMA IEN FROM “BMX ADO
SCHEMA” FILE
   D SS^BMXADO(.OUT,SIEN,"","") ; PUT ANR IN AN ARRAY
   S NODE=$O(^TMP("BMX ADO",$J,999999),-1)+1 ; GET NEXT AVAILABLE NODE IN THE
ANR ARRAY
   S ^TMP("BMX ADO",$J,NODE)="^`2^`1^`71164^175.75"_$C(30) ; APPEND A DATA
STRING TO ANR ARRAY
   D DISP(OUT) R % ; DISPLAY THE INPUT ARRAY BEFORE UPDATING THE RECORD
   D BAFM^BMXADOF1(.OUT,$NA(^TMP("BMX ADO",$J))) ; EP FOR UPDAING THE RECORD
   K ^TMP("BMX ADO",$J) ; CLEAN UP
   W !!,OUT S %=0 F  S %=$O(OUT(%)) Q:'%  W !,OUT(%) ; RETURN
ACKNOWLEDGEMENT/ERROR MSG
   Q
   ;
DISP(OUT)   ; DISPLAY THE ANR ARRAY
   N I,X
   S I=0 W !
   F  S I=$O(@OUT@(I)) Q:'I  S X=@OUT@(I) S X=$TR(X,$C(30),"}") S
X=$TR(X,$C(31),"{") W !,X
   Q
   ;
SCHEMA(NAME)   ; GIVEN SCHEMA NAME, RETURN THE SCHEMA IEN FROM THE BMX ADO
SCHEMA FILE
   N IEN
   S IEN=$O(^BMXADO("B",NAME,0))
   Q IEN
   ;
```

Figure 5-57: Sample of BAFM

5.4 Raising and Responding to RPMS Events Using BMXNet

5.4.1 BMXNet Events Overview

The event mechanism of BMXNet allows Windows applications to respond to events which occur in RPMS. For example, a developer may want his application to be notified when a patient checks in for an appointment or when a lab result is available. Using BMXNet Events, the .NET application can receive real-time notification of these occurrences.

Support for BMXNet Events is implemented in the BMXNet.ConnectInfo class using the following methods:

5.4.1.1 ConnectInfo.Subscribe Event

ConnectInfo.SubscribeEvent(string EventName)
Use this method to register interest in a particular event. When the event occurs, applications which have registered interest will be notified.

5.4.1.2 ConnectInfo.UnSubscribe Event

ConnectInfo.UnSubscribeEvent(string EventName)
Use this method to indicate that an application is no longer interested in a particular event.

5.4.1.3 ConnectInfo.Raise Event

ConnectInfo.RaiseEvent(string EventName, string Parameter, Boolean RaiseBack)
Use this method to tell the RPMS server to raise ("fire") a particular event passing the given parameter. All windows applications which have registered interest in the event and are currently connected to the same RPMS server will be notified. Events may also be raised directly in RPMS using M code. If *RaiseBack* is true, the event will be reflected back to the application which initially raised the event.

5.4.1.4 ConnectInfo.Event Polling Enabled

ConnectInfo.EventPollingEnabled
Use this Boolean property to start or stop event polling by BMXNet. Default is *false*.

5.4.1.5 ConnectInfo.EventPollingInterval

Use this Integer property to specify the event polling interval, in milliseconds, used by BMXNet. Default is *5000*.

5.4.1.6 ConnectInfo.BMXNetEvent

Use the BMXNetEvent delegate to connect BMXNet Events to client code.

5.4.1.7 ConnectInfo.TransmitLock

ConnectInfo.RPMSDataTable automatically controls access to the connected TCP/IP port to prevent possible simultaneous access of the port by the event polling mechanism and other RPC calls. Using ConnectInfo.RPMSDataTable ensures that no collision will occur. However, if it is necessary to make a BMXNetLib.TransmitRPC call using the same TCP/IP connection, then TransmitLock may be used. Set TransmitLock to true to temporarily block event polling. Set TransmitLock to false to re-enable polling.

5.4.1.8 ConnectInfo.TransmitLockTimeout

TransmitLockTimeout is used internally by BMXNet to set the number of milliseconds that BMXNet will wait to obtain a TransmitLock.

5.4.2 BMXNet Event Code Samples

The following C# code sample, taken from the sample application distributed with BMXNet, illustrates the use of BMXNet events. Note that standard C# technique is used to connect the event to its handler.

```
private void frmBMXNetTest_Load(object sender, System.EventArgs e)
{
   m_ci.EventPollingEnabled = true;
   m_ci.BMXNetEvent += new
BMXNetConnectInfo.BMXNetEventDelegate(BMXNetEventHandler);
}

private void BMXNetEventHandler(Object obj, BMXNet.BMXNetEventArgs e)
{
   MessageBox.Show("BMXNet event " + e.BMXEvent + " fired with parameter " +
e.BMXParam);
}

private void cmdSubscribeEvent_Click(object sender, System.EventArgs e)
{
   m_ci.SubscribeEvent(“MyEvent”);
}

private void cmdRaiseEvent_Click(object sender, System.EventArgs e)
{
   m_ci.RaiseEvent((“MyEvent”, "MyParam", true);
}

private void cmdUnSubscribeEvent_Click(object sender, System.EventArgs e)
{
   m_ci.UnSubscribeEvent(“MyEvent”);
}
```

Figure 5-58: Sample of C# code

From with RPMS, a BMXNet event may be raised by calling the following published entry point in the BMXNet package:

```
D EVENT^BMXMEVN(BMXEVENT,BMXPARAM,BMXORIG,BMXKEY)
```

Figure 5-59: Sample of calling the entry point

BMXORIG represents the event originator's session ($J). The event will not be raised back to the originator if BMXORIG is the session of the originator. BMXKEY is a ~-delimited list of security keys. Only holders of one of these keys will receive event notification. If BMXKEY is "" then all subscribed sessions will be notified.

5.5 Asynchronous Retrieval of RPMS Data

5.5.1 BMXNet Support for Asynchronous data retrieval

The BMXNet ConnectInfo object supplies two methods which support the asynchronous retrieval of data in cases where the retrieval task is expected to be long running. The first method, *ConnectInfo.RPMSDataTableAsyncQue*, is used to submit the task to RPMS for processing.

The second method, *ConnectInfo.RPMSDataTableAsyncGet*, retrieves the queued data.

5.5.1.1 System.Int32 ConnectInfo.RPMSDataTableAsyncQueue(System.String CommandString, System.String EventName)

Call this method with the RPC or SQL SELECT statement you wish to queue and the name of the event which should be used to notify your application when the process is completed. The method returns an integer which corresponds to TaskMan's ZTSK variable and can be used to further route the retrieved data. When the task is complete, BMXNet will raise an event with a parameter. The parameter will consist of a tilde-delimited string containing the original ZTSK number in the first ~-piece, and the queued data location in the second ~-piece. The queued data location should be passed as the first argument to the *RPMSDataTableAsyncGet* method to retrieve the queued data.

5.5.1.2 System.Data.DataTable RPMSDataTableAsyncGet (System.String AsyncInfo , System.String TableName)

Call this method from within an event handler to retrieve RPMS data queued with *RPMSDataTableAsyncQue*. *AsyncInfo* specifies the location of the data in RPMS and is retrieved from the second ~- piece of the *BMXParam* event parameter. *TableName* is the name you want to give to the retrieved data table. An overload of this function allows you to name a DataSet into which you want the DataTable placed.

5.5.2 Asynchronous Data Retrieval Code Sample

The following sample illustrates the three steps to retrieving RPMS data asynchronously: making the asynchronous call, handling the event, and retrieving the queued data.

```
private string  m_sTask;
private void cmdAsyncCall_Click(object sender, System.EventArgs e)
{
   try
   {
      //Use the windows handle to create a unique event name
      m_ci.SubscribeEvent("BMX ASYNC RESULT READY" + this.Handle.ToString());
      //nTask corresponds to TaskMan's ZTSK
      int nTask = m_ci.RPMSDataTableAsyncQue(this.txtAsyncCommand.Text, "BMX
ASYNC RESULT READY" + this.Handle.ToString());
      //Save ZTSK for later comparison
      m_sTask = nTask.ToString();
   }
   catch (Exception ex)
   {
      MessageBox.Show(ex.Message);
   }
}

private void BMXNetEventHandler(Object obj, BMXNet.BMXNetEventArgs e)
{
   try
   {
   //BMXNet events arrive on a separate thread from the UI thread,
   //thus UI methods should be called using BeginInvoke or Invoke
      UpdateEventMessageTextDelegate dlgUEMT = new
UpdateEventMessageTextDelegate(UpdateEventMessageText);
      this.Invoke(dlgUEMT, new object[] {e.BMXEvent, e.BMXParam});

      if (e.BMXEvent == "BMX ASYNC RESULT READY" + this.Handle.ToString())
      {
         string sTask = BMXNetLib.Piece(e.BMXParam,"~",1);
         string sAsyncInfo = BMXNetLib.Piece(e.BMXParam,"~",2);
         DataTable dtAsyncGrid = m_ci.RPMSDataTableAsyncGet(sAsyncInfo,
"AsyncData");
         UpdateAsyncGridDelegate uagd = new
UpdateAsyncGridDelegate(UpdateAsyncGrid);
         this.BeginInvoke(uagd, new object[] {sTask, dtAsyncGrid});
      }
   }
   catch (Exception ex)
   {
      Debug.Write("BMXNetEventHandler exception: " + ex.Message + "\n");
   }
}
```

Figure 5-60: Example of steps to retrieving RPMS data asynchronously

5.5.3 BMX Logging

The following sample illustrates the method of capturing RPMS data into a log file for debug purposes. When started, a file with the following syntax is created in the current working directory. BMXLog concatentated with a space, DayofYear, space, CurrentHour, space, CurrentMinute, space, CurrentSecond, ".log" (for example: BMXLog 160 11 26 44.log). The contents of the text file shows the RPC sent and data received. Because patient sensitive information may be written, developers should be careful to remove both the access to the logging events and remove of the text file.

```
//create a button to start and stop the logging
//add the following methods
//m_ci represents instance of ConnectInfo object

private void cmdStartLogging_Click(object sender, EventArgs e)
        {
            try
            {
                m_ci.bmxNetLib.StartLog();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Unable to start logging:  " + ex.Message);
            }
        }

        private void cmdStopLogging_Click(object sender, EventArgs e)
        {
            try
            {
                m_ci.bmxNetLib.StopLog();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Unable to stop logging:  " + ex.Message);
            }
        }

The following is an example of the data captured:

11:26:54 AM Friday, June 09, 2006
  :
  :Sent: {BMX}0007300058007BMX;;;;000430BMX ADO SS^000260080HW
TEST00100080~1~20~5
-------------------------------
11:26:54 AM Friday, June 09, 2006
  :
  :Received:
@@@meta@@@BMXIEN|1601012|^1601012|.001|N|15|BMXIEN|TRUE|TRUE|FALSE^1601012|.01
|T|30|NAME|FALSE|FALSE|TRUE-1^MOLLY-2^ALICE-4^HORACE-
```

Figure 5-61: Sample of the method of capturing RPMS data into a log file

6.0 Glossary

Term	Definition
ADO.NET	Set of classes that expose data access services to the .NET programmer.
API	Application Program Interface. Callable entry points that enable software to communicate with other software..
Application Context	The Application Context is a read/write part of application programs that contains rules dictating how the application interacts with other applications and users.
BMXNet	A set of software utilities designed to connect to RPMS data by .NET applications.
column_alias	Specifies an alternative name to replace the column name in the query result set. For example, an alias such as “Quantity”, “Quantity to Date”, or “Qty” can be specified for a column named **quantity**
field_name	Specifies from which field (column) the FROM clause should return. The field_name is associated with the preceding table_name by using a dot (.) between the two.
FROM Keyword	Specifies that the system should return the values from the following table_list statement.
INDEX Keyword	Identifies a specific FileMan cross-reference to use when retrieving data. Always use the SHOWPLAN keyword in conjunction with the INDEX keyword to ensure that you achieve the intended result.
Index Row	Part of the SHOWPLAN results that includes the M code created by BMXNet that will execute on the RPMS server and iterate through the FileMan file.
Iterate	To say or perform again; repeat.
join_type	Specifies a join using nonstandard syntax and the WHERE clause. The =* operator is used to specify a One-to-many (OTM) join. Use the OTM join to express relationship between tables A and B such that a record in table A can be referenced by a FileMan pointer field in one or more records in table B.
Log on	Connect to a network
M	A programming language that originated from the medical sector but is currently used in a variety of database applications due to its retrieval capabilities.
M Routines	A collection of command lines, all associated with a single name that can be stored and retrieved as a unit.

Term	Definition
MAXRECORDS Keyword	Specifies the maximum number of records to return
Operator	A symbol that specifies which operation the systems should perform relative to the indicated operator arguments.
Operator Arguments	A value or expression dictating the information upon which the related operator acts.
Overloads	The creation of more than one procedure, instance constructor, or property in a class with the same name but different argument types. Overloading is especially useful when your object model dictates that you employ identical names for procedures that operate on different data types
Parameter	A value given to a variable until the related operation is completed. Parameters are treated by the system as constants. Parameters are often used to customize a program for a particular purpose
Port	Software that links one computer with another using TCP/IP address and port numbers assigned by network administrators.
primary_table_name	The file from which the system should first retrieve data.
related_table_name	The file from which the system should receive data that matches the primary table.
Remote Procedure Call (RPC)	A technique used to constructing distributed, client-server based applications. RPC extends the capabilities of local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure.
Screen Rows	Part of the SHOWPLAN results that includes the M code that BMXNet will execute to filter the results. The “^” character in the query plan is replaced by the “~” character.
search_condition	Restricts the rows returned in the result set by using predicates. There is no limit to the number of predicates that can be included in a search condition.
SELECT Keyword	Specifies that the system should return the values by the following select_list statement.
select_list	Specifies which fields (columns) the system should select for the returned set. The select_list a series of expressions separated by commas.
Server	A computer that hosts RPMS applications.

Term	Definition
SHOWPLAN Keyword	Returns the query plan, including the M code that will be executed to retrieve the records.
table_name	Specifies from which files (tables) the FROM clause should return fields.
table_source	Specifies from which files (tables) the FROM clause should return fields.
WHERE Keyword	Specifies a search condition to restrict the rows returned.

7.0 References and Sources

American Heritage Dictionary of the English Language, 3rd ed. Boston: Houghton Mifflin Company, 1992.

Microsoft Manual of Style for Technical Publications, 3rd ed. Redmond, WA: Microsoft Press, 1997.

C# and the .NET Platform, Andrew Troelsen, Berkeley, CA: Appress, 2001.

Essential ADO.NET, Bob Beauchemin, Boston: Addison-Wesley, 2002.

Professional C#, 2nd ed., Simon Robinson, et. Al, Birmingham, UK: Wrox Press Ltd., 2002.

Windows Forms Programming in C#, Chris Sells, Boston: Addison-Wesley, 2004.

8.0 Index

9.0 Appendix A: Rules of Behavior

9.1 Specific RPMS Rules of Behavior

All users (Contractors and IHS Employees) of RPMS will be provided a copy of the rules of behavior (RoB) and will have to acknowledge them in accordance IHS policy prior to being granted access to a RPMS system. The RPMS system is a United States Department of Health and Human Services, Indian Health Service information system that is FOR OFFICIAL USE ONLY. The system is subject to monitoring; therefore, no expectation of privacy shall be assumed. Individuals found performing unauthorized activities are subject to disciplinary action including criminal prosecution.

RPMS users must follow these RoB in addition to the RoB listed in the IHS General User Security Handbook and, if a privileged user, the RoB listed in the IHS Technical and Managerial Handbook.

Important Note: The RoBs listed in this document are specific to RPMS. For a listing of general RoB for all users, please see the IHS General User Security Handbook and for a listing of system administrators/managers rules, the IHS Technical and Managerial Handbook located at http://home.ihs.gov/ITSC-CIO/security/secpgm/ITproced.cfm.

9.1.1 All RPMS Users

In addition to these rules, each application may include additional RoBs, which may be defined within the individual application's documentation (e.g., PCC, Dental, Pharmacy).

9.1.1.1 Access

RPMS Users Shall:

✓ Only use data for which you have been granted authorization.

✓ Only give information to personnel who have access authority and have a need to know.

✓ Always verify a caller's identification and job purpose with your supervisor or the entity provided as employer *before* providing any type of information system access, sensitive information, or non-public agency information.

✓ Be aware that personal use of information resources is authorized on a limited basis within the provisions Indian Health Manual Chapter 6 OMS Limited Personal Use of Information Technology Resources TN 03-05," August 6, 2003.

Users Shall Not:

X Retrieve information for someone who does not have authority to access the information.

X Access, research, or change any user account, file, directory, table, or record not required to perform your OFFICIAL duties.

X Store sensitive files on a PC hard drive, or portable devices or media, if access to the PC or files cannot be physically or technically limited.

X Exceed their authorized access limits in RPMS by changing information or searching databases beyond the responsibilities of their job or by divulging information to anyone not authorized to know that information.

9.1.1.2 Logging On To the System

RPMS Users Shall:

✓ Have a unique User Identification/Account name and password.

✓ Be granted access based on authenticating the account name and password entered.

✓ Be locked out of an account after 5 successive failed login attempts within a specified time period (e.g., one hour).

9.1.1.3 Information Accessibility

RPMS shall restrict access to information based on the type and identity of the user. However, regardless of the type of user, access shall be restricted to the minimum level necessary to perform the job.

Users Shall:

- ✓ Access only those documents they created and those other documents to which they have a valid need-to-know and to which they have specifically granted access through an RPMS application based on their menus (job roles), keys, and FileMan access codes. Some users may be afforded additional privileges based on the function they perform such as system administrator or application administrator.
- ✓ Acquire a written preauthorization in accordance with IHS polices and procedures prior to interconnection to or transferring data from RPMS.

9.1.1.4 Accountability

Users Shall:

- ✓ Behave in an ethical, technically proficient, informed, and trustworthy manner.
- ✓ Logout of the system whenever they leave the vicinity of their PC.
- ✓ Be alert to threats and vulnerabilities in the security of the system.
- ✓ Report all security incidents to their local Information System Security Officer (ISSO)
- ✓ Differentiate tasks and functions to ensure that no one person has sole access to or control over important resources.
- ✓ Protect all sensitive data entrusted to them as part of their government employment.
- ✓ Shall abide by all Department and Agency policies and procedures and guidelines related to ethics, conduct, behavior and IT information processes.

9.1.1.5 Confidentiality

Users Shall:

- ✓ Be aware of the sensitivity of electronic and hardcopy information, and protect it accordingly.
- ✓ Store hardcopy reports/storage media containing confidential information in a locked room or cabinet.
- ✓ Erase sensitive data on storage media, prior to reusing or disposing of the media.
- ✓ Protect all RPMS terminals from public viewing at all times.
- ✓ Abide by all HIPAA regulations to ensure patient confidentiality.

Users Shall Not:

- X Allow confidential information to remain on the PC screen when someone who is not authorized to that data is in the vicinity.
- X Store sensitive files on a portable device or media without encrypting.

9.1.1.6 Integrity

Users Shall:

- ✓ Protect your system against viruses and similar malicious programs.
- ✓ Observe all software license agreements.
- ✓ Follow industry standard procedures for maintaining and managing RPMS hardware, operating system software, application software, and/or database software and database tables.
- ✓ Comply with all copyright regulations and license agreements associated with RPMS software.

Users Shall Not:

- X Violate Federal copyright laws.
- X Install or use unauthorized software within the system libraries or folders.
- X Use freeware, shareware or public domain software on/with the system without your manager’s written permission and without scanning it for viruses first.

9.1.1.7 Passwords

Users Shall:

✓ Change passwords a minimum of every 90 days.

✓ Create passwords with a minimum of eight characters.

✓ If the system allows, use a combination of alpha, numeric characters for passwords, with at least one uppercase letter, one lower case letter, and one number. It is recommended, if possible, that a special character also be used in the password.

✓ Change vendor-supplied passwords immediately.

✓ Protect passwords by committing them to memory or store them in a safe place (do not store passwords in login scripts, or batch files.

✓ Change password immediately if password has been seen, guessed or otherwise compromised; and report the compromise or suspected compromise to your ISSO.

✓ Keep user identifications (ID) and passwords confidential.

Users Shall Not:

X Use common words found in any dictionary as a password.

X Use obvious readable passwords or passwords that incorporate personal data elements (e.g., user's name, date of birth, address, telephone number, or social security number; names of children or spouses; favorite band, sports team, or automobile; or other personal attributes).

X Share passwords/IDs with anyone or accept the use of another's password/ID, even if offered.

X Reuse passwords. A new password must contain no more than five characters per 8 characters from the previous password.

X Post passwords.

X Keep a password list in an obvious place, such as under keyboards, in desk drawers, or in any other location where it might be disclosed.

X Give a password out over the phone.

9.1.1.8 Backups

Users Shall:

✓ Plan for contingencies such as physical disasters, loss of processing, and disclosure of information by preparing alternate work strategies and system recovery mechanisms.

✓ Make backups of systems and files on a regular, defined basis.

✓ If possible, store backups away from the system in a secure environment.

9.1.1.9 Reporting

Users Shall:

✓ Contact and inform your ISSO that you have identified an IT security incident and you will begin the reporting process by providing an IT Incident Reporting Form regarding this incident.

✓ Report security incidents as detailed in IHS SOP 05-03, *Incident Handling Guide*.

Users Shall Not:

X Assume that someone else has already reported an incident. The risk of an incident going unreported far outweighs the possibility that an incident gets reported more than once.

9.1.1.10 Session Time Outs

RPMS system implements system-based timeouts that back users out of a prompt after no more than five minutes of inactivity.

Users Shall:

✓ Utilize a screen saver with password protection set to suspend operations at no greater than 10-minutes of inactivity. This will prevent inappropriate access and viewing of any material displayed on your screen after some period of inactivity.

9.1.1.11 Hardware

Users Shall:

✓ Avoid placing system equipment near obvious environmental hazards (e.g., water pipes).

✓ Keep an inventory of all system equipment.

✓ Keep records of maintenance/repairs performed on system equipment.

Users Shall Not:

X Do not eat or drink near system equipment.

9.1.1.12 Awareness

Users Shall:

✓ Participate in organization-wide security training as required.

✓ Read and adhere to security information pertaining to system hardware and software.

✓ Take the annual information security awareness.

✓ Read all applicable RPMS Manuals for the applications used in their jobs.

9.1.1.13 Remote Access

Each subscriber organization establishes its own policies for determining which employees may work at home or in other remote workplace locations. Any remote work arrangement should include policies that:

- Are in writing
- Provide authentication of the remote user through the use of ID and password or other acceptable technical means
- Outline the work requirements and the security safeguards and procedures the employee is expected to follow
- Ensure adequate storage of files, removal and non-recovery of temporary files created in processing sensitive data, virus protection, intrusion detection, and provides physical security for government equipment and sensitive data
- Establish mechanisms to back up data created and/or stored at alternate work locations.

Remote Users Shall:

✓ Remotely access RPMS through a virtual private network (VPN) when ever possible. Use of direct dial in access must be justified and approved in writing and its use secured in accordance with industry best practices or government procedures.

Remote Users Shall Not:

X Disable any encryption established for network, internet and web browser communications.

9.1.2 RPMS Developers

Developers Shall:

- ✓ Always be mindful of protecting the confidentiality, availability, and integrity of RPMS when writing or revising code.
- ✓ Always follow the IHS RPMS Programming Standards and Conventions (SAC) when developing for RPMS.
- ✓ Only access information or code within the namespaces for which they have been assigned as part of their duties.
- ✓ Remember that all RPMS code is the property of the U.S. Government, not the developer.
- ✓ Shall not access live production systems without obtaining appropriate written access, shall only retain that access for the shortest period possible to accomplish the task that requires the access.
- ✓ Shall observe separation of duties policies and procedures to the fullest extent possible.
- ✓ Shall document or comment all changes to any RPMS software at the time the change or update is made. Documentation shall include the programmer's initials, date of change and reason for the change.
- ✓ Shall use checksums or other integrity mechanism when releasing their certified applications to assure the integrity of the routines within their RPMS applications.
- ✓ Shall follow industry best standards for systems they are assigned to develop or maintain; abide by all Department and Agency policies and procedures.
- ✓ Shall document and implement security processes whenever available.

Developers Shall Not:

- X Write any code that adversely impacts RPMS, such as backdoor access, "Easter eggs," time bombs, or any other malicious code or make inappropriate comments within the code, manuals, or help frames.
- X Grant any user or system administrator access to RPMS unless proper documentation is provided.
- X Not release any sensitive agency or patient information.

9.1.3 Privileged Users

Personnel who have significant access to processes and data in RPMS, such as, system security administrators, systems administrators, and database administrators have added responsibilities to ensure the secure operation of RPMS.

Privileged Users Shall:

- ✓ Verify that any user requesting access to any RPMS system has completed the appropriate access request forms.
- ✓ Ensure that government personnel and contractor personnel understand and comply with license requirements. End users, supervisors, and functional managers are ultimately responsible for this compliance.
- ✓ Advise the system owner on matters concerning information technology security.
- ✓ Assist the system owner in developing security plans, risk assessments, and supporting documentation for the certification and accreditation process.
- ✓ Ensure that any changes to RPMS that affect contingency and disaster recovery plans are conveyed to the person responsible for maintaining continuity of operations plans.
- ✓ Ensure that adequate physical and administrative safeguards are operational within their areas of responsibility and that access to information and data is restricted to authorized personnel on a need to know basis.
- ✓ Verify that users have received appropriate security training before allowing access to RPMS.
- ✓ Implement applicable security access procedures and mechanisms, incorporate appropriate levels of system auditing, and review audit logs.
- ✓ Document and investigate known or suspected security incidents or violations and report them to the ISSO, CISO, and systems owner.
- ✓ Protect the supervisor, superuser or system administrator passwords.
- ✓ Avoid instances where the same individual has responsibility for several functions (i.e., transaction entry and transaction approval).
- ✓ Watch for unscheduled, unusual, and unauthorized programs.
- ✓ Help train system users on the appropriate use and security of the system.
- ✓ Establish protective controls to ensure the accountability, integrity, confidentiality, and availability of the system.
- ✓ Replace passwords when a compromise is suspected. Delete user accounts as quickly as possible from the time that the user is no longer authorized system. Passwords forgotten by their owner should be replaced, not reissued.
- ✓ Terminate user accounts when a user transfers or has been terminated. If the user has authority to grant authorizations to others, review these other authorizations. Retrieve any devices used to gain access to the system or equipment. Cancel logon IDs and passwords, and delete or reassign related active and back up files.
- ✓ Use a suspend program to prevent an unauthorized user from logging on with the

Privileged Users Shall:

current user's ID if the system is left on and unattended.

- ✓ Verify the identity of the user when resetting passwords. This can be done either in person or having the user answer a question that can be compared to one in the administrator’s database.
- ✓ Shall follow industry best standards for systems they are assigned to; abide by all Department and Agency policies and procedures.

Privileged Shall Not:

- X Access any files, records, systems, etc., that are not explicitly needed to perform their duties
- X Grant any user or system administrator access to RPMS unless proper documentation is provided.
- X Not release any sensitive agency or patient information.

10.0 Contact Information

If you have any questions or comments regarding this distribution, please contact the OIT Service Center by:

Phone:	(505) 248-4371 or
	(888) 830-7280
Fax:	(505) 248-4363
Web:	http://www.ihs.gov/GeneralWeb/HelpCenter/Helpdesk/index.cfm
Email:	ITSCHelp@mail.ihs.gov