

In-Memory Database Driver Reference

COPYRIGHT 1994- 2006 SoftVelocity Incorporated. All rights reserved.

This publication is protected by copyright and all rights are reserved by SoftVelocity Incorporated. It may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from SoftVelocity Incorporated.

This publication supports Clarion. It is possible that it may contain technical or typographical errors. SoftVelocity Incorporated provides this publication “as is,” without warranty of any kind, either expressed or implied.

SoftVelocity Incorporated
2335 East Atlantic Blvd., Suite 410
Pompano Beach, Florida 33062
(954) 785-4555
www.softvelocity.com

Updated for Version 2.0

Trademark Acknowledgements:

SoftVelocity is a trademark of SoftVelocity Incorporated.

Clarion™ is a trademark of SoftVelocity Incorporated.

Microsoft®, Windows®, and Visual Basic® are registered trademarks of Microsoft Corporation.

All other products and company names are trademarks of their respective owners.

Contents:

In-Memory Database Driver	5
Product Overview	5
In-Memory Driver Uses	6
Getting Started - Registering the In-Memory Driver	7
Using the In-Memory Driver	7
Specifications	8
Library Files	8
Data Types Supported.....	8
File Specifications/Maximums	8
Driver String Support	9
Supported Commands and Attributes	10
Notes	12
Application Design Considerations	13
Template Guide.....	15
Registering the IMDD Template.....	15
In-Memory Data Caching Support Extension.....	16
DLL Support.....	17
Generated Table options	18
Dictionary Table options	21
Concurrent Generated and Dictionary Tables	24
InMemoryCachedTableLoad Code Template	25
InMemoryCachedTableSave Code Template	25
IPDriver In-Memory Caching Support.....	26
Server Files.....	27
Client Files	28
Template Notes	29
Assertion Message	29
DLL considerations.....	30
IMDD Class Library Reference	31
Overview	31
FileSynchronization Class Source Files.....	31
Template Support.....	31
Template Embeds	32
FileSynchronization Class Properties	33
BackupOnSave (backup physical file on Save).....	33
Filter (record filter expression).....	33
FreeElement (comparison value for range limits).....	33
HighLimit (range of values upper limit)	34
LimitType (type of range limit process)	34
LoadSuccess (memory table load was successful).....	34
LowLimit (range of values lower limit)	34
Original (data source table file reference)	34
OptimizeSave (custom save option).....	35
PrimaryKey (reference to data source primary key).....	35
RangeKey(reference to data source range limit key)	35
RecordPositionField (key or table position).....	36
RecordsToRead (records to process from data source)	36
SaveOnKill (save changes on exit to data source).....	36
Silent (silent mode flag)	36
StartTransaction (OK to start transaction processing)	37
Target (target Memory table file reference).....	37
UseSQL (SQL data source).....	37
UseLogout (transaction processing enabled).....	37

FileSynchronization Class Methods	38
AsynchronousLoad (load IMDD table on background thread)	38
BindOnLoad (BIND variables on asynchronous load)	38
Close (close data source and target tables)	39
Destruct (dispose the filter)	39
GenerateBackUp (create data source backup table)	40
GetLoadSuccess (successful asynchronous load)	41
GetSyncError (get synchronous error message)	42
GetSyncErrorCode (get synchronous error code)	43
Init (initialize file synchronization)	44
Kill (save changes and dispose the filter)	45
Load (write source data to memory table)	46
Open (open data source and target table)	48
OptimizedSave (customized save)	50
PrimeKey (initialize key element)	50
PrimeRecord (prime fields on load)	51
PrimeSaveRecord (prime fields on save)	51
Reset (clear data source contents)	52
ResetRange (reset key or file range to start)	53
Save (write memory table contents to source)	54
SetFilter (initialize record filter)	56
SetOrderKey (set to key order sequence)	57
SetRange (set to range limit sequence)	58
SetRecordPositionField (set record position)	59
SetRecordsToRead (initialize records to process)	60
SetUseSQL (set the UseSQL property)	60
SyncDelete(delete record in physical table)	61
SyncInsert(add record to physical table)	62
SyncRefresh(refresh IMDD record)	63
SyncUpdate(update record to physical table)	64
TransactionCommit (Commit after save)	65
TransactionRollback(Rollback after save)	65
TransactionStart (begin Logout before Save)	66
ValidateRecord (evaluate filter during load and save)	67

Index:	69
---------------------	-----------

In-Memory Database Driver

Product Overview

The SoftVelocity In-Memory driver is a new add-on file driver technology that does not use physical tables for working with data. This is due to a RAM-based technology known as IMDD (In-Memory Database Driver). All data is stored in Random Access Memory (RAM), which gives the driver a number of unique properties.

- Extremely high performance. Once data is loaded into the In-Memory table, either at program startup or on-demand, all access is virtually instantaneous.
- You are also only limited by your machines memory capacity, and because you can use VIEWS to populate In-Memory tables, and can create VIEWS across multiple In-Memory tables, you have the capacity of storing and working with significant amounts of hierarchical data.
- In-Memory tables can be used as a “Global” lookup table, but unlike a global queue, you do not have to write code to synchronize threaded access, because synchronization is built into the driver.
- The In-Memory driver has a standard file driver interface, so you can use In-Memory tables with any Clarion entity or template that works with files (Browse, FileDrop, Report, etc.), including all 3rd party templates.
- In-Memory tables support an aggregation of all driver data types, so they can be used in conjunction with any supported file driver.
- You use standard Clarion file processing statements to access and update an In-Memory table, the same as you do with other file formats. See *Supported Commands and Attributes* for a list of supported commands.
- The application developer can choose to locate highly dynamic or frequently referenced tables in memory, while leaving the less intensively used parts of the database to be stored on much cheaper disk.
- In-Memory tables provide highly exclusive data access. Storing data in an In-Memory table you can be sure of its security and privacy. Even two programs simultaneously running on the same computer will each use their own data copy.
- It's especially useful for WEB-enabled applications. In addition, all stored data will be automatically destroyed when an application terminates, and you will never have any undeleted temporary files.
- As In-Memory files are recreated each time an application runs, you don't need to convert In-Memory tables after a change to the record structure of your physical file.
- Instantaneous access to data, even when changing the sort order or applying filters.
- Like an ALIAS structure, the In-Memory driver can emulate the structure of any defined file in the dictionary, but unlike the ALIAS, it can be modified with additional columns and indexes without the need for conversion.

In-Memory Driver Uses

Here are just some possible tasks for the In-Memory driver.

- Use it instead of a global queue. You access it using standard Clarion File commands, and it is a thread-safe structure requiring no synchronization.
- Use it for the most intensely used areas of your application, where your users spend most of their time.
- Use it as a temporary table for Parent-Child relations where the end user makes all changes to the child records in the In-Memory table. For example new order items can be added and updated, and only when the invoice is actually saved, write all the child records to the physical file in a single transaction, along with the order record.
- Use it as a buffer for receiving SQL query results. After you have received the result set into the In-Memory table you can do sorting, filtering, perform QBE/QBF requests using standard browse box tools, without any additional access to the SQL server. This approach can dramatically reduce network traffic, and frees server resources as fast as possible.
- Using the In-Memory driver allows you to minimize the number of KEY and INDEX files in your physical tables. For example, if you have a rarely requested report that requires a special index, you can declare this index in the In-Memory table but not in the physical table. This way you can reduce required database resources and speed up data I/O operations.
- Eliminate file conversions of the physical table when you need to add a new key or index for a Report or Browse. Just add the new INDEX to the In-Memory table.
- You can construct your In-Memory table using several data sources, and the sources can combine data tables of various formats. For example, you can load configuration data from a mainframe file and combine it with additional items from a local file.

Getting Started - Registering the In-Memory Driver

You must register the In-Memory Driver with the Clarion development environment before you can use it.

To register the In-Memory Driver:

1. Start the Clarion development environment.
2. Choose Setup ► Database Driver Registry.
3. Press the **Add** button.
4. If you are using Clarion 5.5, highlight **C55MEM.DLL** (by default, in the ..\BIN directory) in the list box, then press the **OK** button. If you are using Clarion 6.0, highlight **C60MEM.DLL** (by default, in the ..\BIN directory) in the list box, then press the **OK** button.
This registers the In-Memory Driver.
5. Press the **OK** button.

Using the In-Memory Driver

After you register the In-Memory Driver, using it is very simple. You can use the In-Memory Driver as you would any other Clarion database driver.

After registering the In-Memory Driver, load the Dictionary Editor. From the Dictionary Editor, add a new table. From the *Table Properties* window, press the down arrow to display a list of available drivers in the **Database Drivers** prompt. From the drop list, select the In-Memory driver.

Tip

To save time and extra work, you can also simply **Copy** and **Paste** any existing table and change the driver to **In-Memory**. From there, you can add new columns and keys as needed.

In addition, when using the template support included in this version, there is no need to define a memory table in your dictionary. There is built-in support for creating memory tables “on the fly”.

Specifications

The In-Memory driver reads and writes data that is stored in the memory of the program. In general, one copy of the data is shared by all In-Memory files that have the same NAME. If no NAME is specified, then the label of the file is used to find the data to be used.

Library Files

C55MEMXL.LIB	Windows Static Link Library (Clarion 5.5)
C60MEMXL.LIB	Windows Static Link Library (Clarion 6.0 and greater)
C55MEMX.LIB	Windows Export Library (Clarion 5.5)
C60MEMX.LIB	Windows Export Library (Clarion 6.0 and greater)
C55MEMX.DLL	Windows Dynamic Link Library (Clarion 5.5)
C60MEMX.DLL	Windows Dynamic Link Library (Clarion 6.0 and greater)

Data Types Supported

BYTE	REAL	CSTRING
SHORT	BFLOAT4	PSTRING
USHORT	BFLOAT8	DATE
LONG	DECIMAL	TIME
ULONG	PDECIMAL	GROUP
SREAL	STRING	

File Specifications/Maximums

File Size:	limited only by memory space
Records per File:	67,108,864
Record Size:	2,147,483,648 bytes
Field Size:	2,147,483,648 bytes
Fields per Record:	Unlimited
Keys/Indexes per File:	255
Key Size:	2,147,483,648 bytes
Memo fields per File:	255
Memo Field Size:	2,147,483,648 bytes
Open Data Files:	unlimited

Driver String Support

Driver Strings are switches that you can set to control the way your application creates, reads, and writes files with a specific driver. Driver strings are simply messages or parameters that are sent to the file driver at run-time to control its behavior. More information regarding these strings can be found in the *Language Reference* PDF

Note:

Some driver strings have no effect after the file is open, so no SEND function syntax is listed for those strings. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

The In-Memory Driver supports the following Driver String:

THREADEDCONTENT

DRIVER('Memory', '/THREADEDCONTENT')

The **THREADEDCONTENT** switch makes an In-Memory table only visible on the thread where it was created. Any In-Memory table defined with the **THREADEDCONTENT** switch active is completely invisible to other threads.

Normally, an In-Memory file with the THREAD attribute shares the same data across all threads. However, if you want to have unique data on each thread, then you can set the driver string to **/THREADEDCONTENT**.

Note:

In Clarion 5.5, you can emulate the /THREADEDCONTENT behavior by simply creating a variable filename, and prior to opening the IMDD on each thread, assign a unique name (i.e., 'MemName' & THREAD()). This will create a unique copy of the IMDD table on each new thread launched.

Tip

To set the THREADEDCONTENT switch at runtime in Clarion 6, use the following syntax:

```
file{PROP:DriverString} = file{PROP:DriverString} & '/THREADEDCONTENT=' & value
```

Supported Commands and Attributes

File Attributes ***Supported***

CREATE	Y
DRIVER(filetype [,driver string])	Y
NAME	Y
ENCRYPT	N
OWNER(password)	N
RECLAIM	N
PRE(prefix)	Y
BINDABLE	Y
THREAD	Y
EXTERNAL(member)	Y
DLL([flag])	Y
OEM	N

File Structures ***Supported***

INDEX	Y
KEY	Y
MEMO	Y
BLOB	Y
RECORD	Y

Index, Key, Memo Attributes ***Supported***

BINARY	Y
DUP	Y
NOCASE	Y
OPT	Y
PRIMARY	Y
NAME	Y
Ascending Components	Y
Descending Components	Y
Mixed Components	Y

Field Attributes ***Supported***

DIM	Y
OVER	Y
NAME	Y

File Procedures ***Supported***

BOF(file)	N
BUFFER(file)	N
BUILD(file)	Y
BUILD(key)	Y
BUILD(index)	Y
BUILD(index, components)	Y
BUILD(index, components, filter)	N
BYTES(file)	N
CLOSE(file)	Y
COPY(file, new file)	Y ¹
CREATE(file)	Y

DUPLICATE(file)	Y
DUPLICATE(key)	Y
EMPTY(file)	Y
EOF(file)	N
FLUSH(file)	N
LOCK(file)	N
NAME(label)	Y
OPEN(file, access mode)	Y ²
PACK(file)	N
POINTER(file)	Y
POINTER(key)	Y
POSITION(file)	Y ³
POSITION(key)	Y ³
RECORDS(file)	Y
RECORDS(key)	Y
REMOVE(file)	Y ¹
RENAME(file, new file)	Y ¹
SEND(file, message)	N
SHARE(file, access mode)	Y ²
STATUS(file)	Y
STREAM(file)	N
UNLOCK(file)	N

Record Access	Supported
ADD(file)	Y
ADD(file, length)	N
APPEND(file)	Y
APPEND(file, length)	N
DELETE(file)	Y
GET(file, key)	Y
GET(file, filepointer)	Y
GET(file, filepointer, length)	N
GET(key, keypointer)	Y
HOLD(file)	N
NEXT(file)	Y
NOMEMO(file)	Y
PREVIOUS(file)	Y
PUT(file)	Y
PUT(file, filepointer)	Y
PUT(file, filepointer, length)	N
RELEASE(file)	N
REGET(file, string)	Y
REGET(key, string)	Y
RESET(file, string)	Y
RESET(key, string)	Y
SET(file)	Y
SET(file, key)	Y
SET(file, filepointer)	Y
SET(key)	Y
SET(key, key)	Y
SET(key, keypointer)	Y
SET(key, key, filepointer)	Y
SKIP(file, count)	Y
WATCH(file)	N

Transaction Processing	Supported
LOGOUT(timeout, file, ..., file)	N
COMMIT	N
ROLLBACK	N

Null Data Processing	Supported
NULL(field)	Y
SETNULL(field)	Y
SETNONNULL(field)	Y

Notes

- 1 COPY and RENAME can be used to make alternative copies of the In-Memory table in memory. If a file has the /THREADEDCONTENTS driver string switch, then COPY and RENAME will only work within the scope of the currently active thread.
- 2 The access mode of OPEN and SHARE is used to enforce access between threads.

Also regarding OPEN, you cannot OPEN an IMDD table and its Alias in Open Exclusive MODE. The ability to OPEN a FILE and its ALIAS is currently only a feature of the TopSpeed driver.
- 3 POSITION(file) returns a STRING(4). POSITION(key) returns a STRING the size of the key fields + 4 bytes.

Application Design Considerations

- The In-Memory driver does not support transaction framing. The solution is to access the **Individual File Overrides** in the *Global properties* and set the **Use RI transaction frame** to *No* for all of your In-Memory tables.
- If you are using table names in your data dictionary that are greater than 8 characters, it is good practice to specify an external name for your In-Memory table. This will avoid the problem of possible clashes with duplicate labels, and an ERRORCODE 47 (Invalid Record Declaration) when attempting to open the In-Memory table.
- When using SQL tables as the source for memory caching, the following caveats are used:
 1. No backups of the SQL table will be performed. Backups are automatic for all ISAM data sources.
 2. The **Re-write physical file on Application Exit** template option will be disabled. (The **SaveOnExit** property is set to FALSE)
 3. Any call to the **Save** method will call the **OptimizeSave** method, which must be derived by the developer to add additional functionality.

See the *IMDD Class Library Reference* and *Template Guide* for more information.

Template Guide

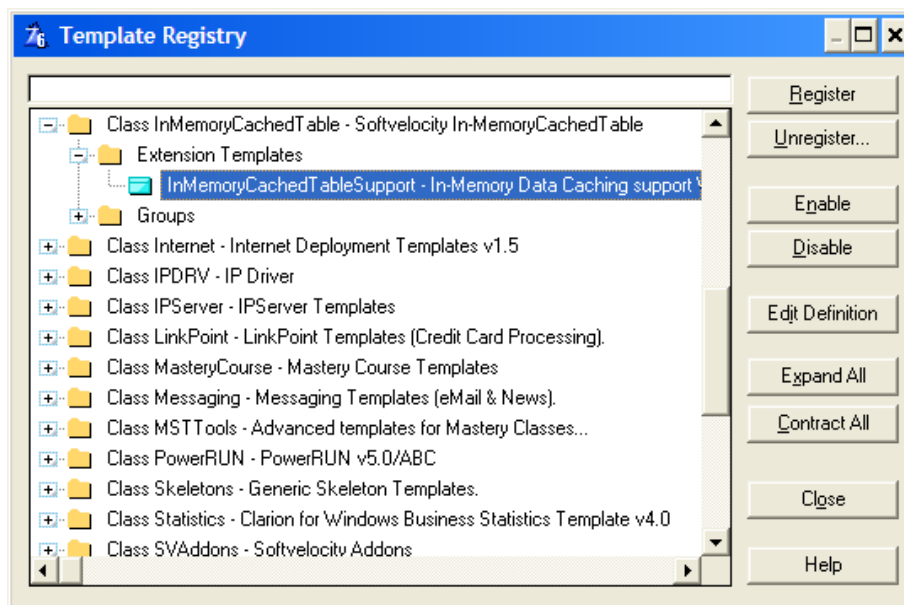
This section documents the built-in template support provided for the In-Memory Database Driver.

Registering the IMDD Template

The first step is to register the template in the IDE Template Registry. This is accomplished by selecting the **Template Registry** menu item from the Clarion IDE **Setup** main menu. You will need to register **MemTable.TPL (or MemTableC55.TPL for Clarion 5.5)**.

Note:

The **InMemoryCachedTable** template set is *only* valid for the ABC template chain in Clarion 5.5, but compatible for *both* Clarion and ABC template chains in Clarion 6 versions and higher.



In-Memory Data Caching Support Extension

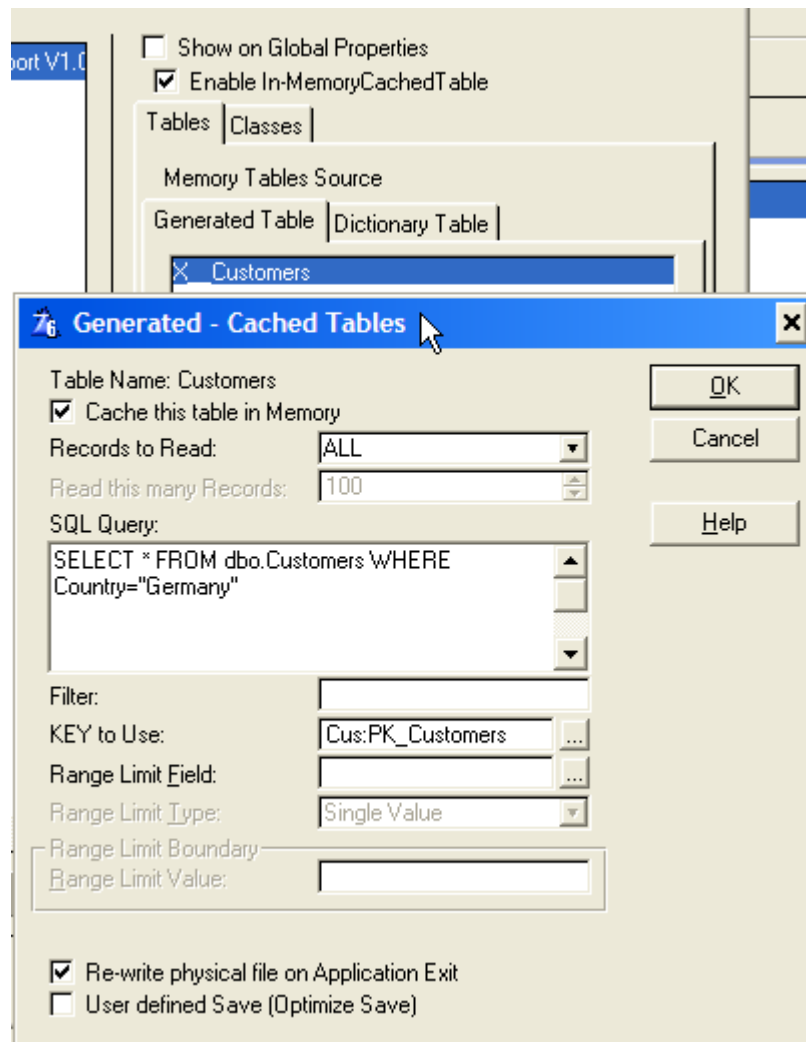
The In-Memory Data Caching Support global extension template is designed to cache data from a physical file into a memory table. The template allows you to cache data from any table (source) defined in your application's data dictionary to a Memory table (target), the Memory table *does not need to be defined in your Data Dictionary*. The template will generate a FILE declaration that matches the source physical file, and generate the code to read from the source physical file and write to the target memory file.

The template supports caching the entire contents of the data source into the target memory table, or you can specify the criteria to cache a subset of the data, filtered according to your requirements.

Alternatively, the template also allows you to choose from a Memory table defined in your data dictionary to be the target.

The main use of this template is to create memory lookup tables to use in your application. However, the template is supported by an underlying **FileSynchronization Class**, and you can use the available properties and methods to aid in the processing of *any* memory table. This class is documented in detail in this manual.

The following template prompts are provided:



Enable In-MemoryCached Table

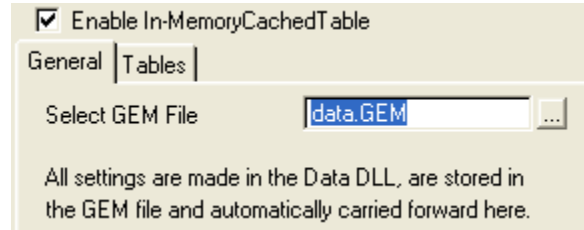
Check this box to enable the In-memory caching support (the default). You can disable the support if you need to compatibility test with other third-party or custom templates.

DLL Support

This version of the IMDD also includes built-in DLL support. If you have selected the “All External” option located in the Global File Control tab, this indicates that your IMDD tables are defined in another external DLL, and the following template prompt is available here:

Select GEM File

When the IMDD In-Memory Data Caching Support global extension template is included in your Data DLL, a GEM (Global Extension Module) is automatically generated for you. This file transfers all information and settings from your Data DLL to any other application that references that DLL. Press the ellipsis button to select the GEM file that was generated automatically by the data DLL.



Note:

If any change is made the core data DLL, and a new GEM file is generated, the changes will not be visible in other application that references that DLL until the application source is regenerated, or you re-import the target GEM file.

An example application is shipped with this version that demonstrates using the IMDD in a DLL configuration. It is located in the <ClarionRoot>\Examples\In-Memory Driver\Multi-DLL folder.

Generated Table options

This template supports two methods for caching data to a memory table: **Generated (Memory) Table** and **Dictionary (Memory) Table**. **These methods identify where the source of the memory table is defined.**

The template can automatically generate a FILE structure for a Memory table that exactly matches the FILE declaration of the data source (**Generated Memory Table**), or you may choose to use a Memory table that you have already defined in your data dictionary (**Dictionary Memory Table**). The **Generated Table** does not exist in your dictionary, but the **Dictionary Table** is defined by you in the data dictionary.

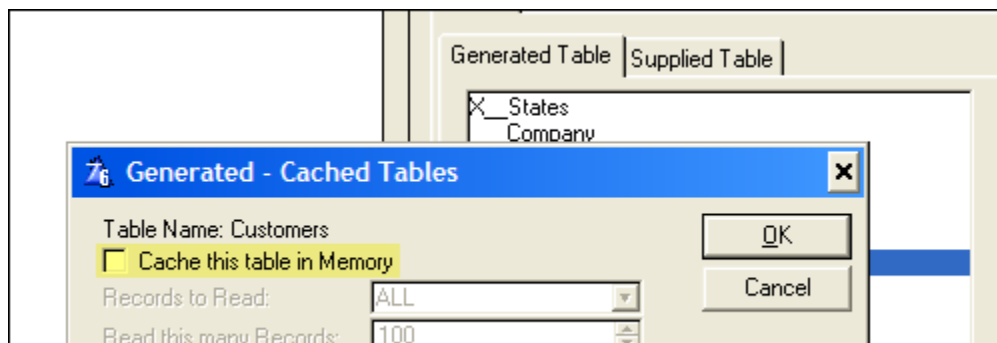
Tip

Use the **Generated Table** option for “lookup tables” that will be loaded at program startup, and will change infrequently.

In the *Generated - Cached Tables* dialog (when not in DLL mode):

Cache this table in memory

Highlight the Physical table you wish to load into a Memory table and click the Properties button. Check the “Cache this table in Memory” checkbox to activate Memory caching for this table. An “X” displays in the list box indicating that this table is active for IMDD caching. Caching occurs at the program start.



Records to Read

You can control how many records to load into the target Memory table. Select *ALL* from the drop list to load all records, or *Limited* in order to set a maximum number of records to cache.

Read this many Records

If *Limited* is selected in **Records to Read**, specify the maximum number of records to load into the Memory table.

SQL Query

If the original file to cache is an SQL based table, this text box is enabled to allow you to enter a valid SQL query that will be used to filter the memory table contents. Using an SQL based query is more efficient than using a standard filter.

Note:

If you are using a single quote in the filter expression, replace it with a double single quote. (i.e., `SELECT * FROM dbo.Customers WHERE Country=' 'Germany' '`).

Also, you can still filter your query results using the next option if needed.

An example application that demonstrates the use of SQL caching can be found in the `<Clarion Root>\Examples\In-Memory Driver\IMDD-SQLcache` folder, and is named `IMDDAndSQLEx.APP`.

Filter

Specify an optional Record Filter here. Type a valid Clarion expression to qualify which records should be cached into the Memory table. Only those records which match the filter expression are loaded into the Memory table. It is generally slower to Filter records rather than use Range Limits based on a KEY. Auto binding of global or local variables is not implemented in this template.

Note:

Since the generated table will always be loaded at program startup, you must ensure that all filter components are primed properly prior to the memory table load.

Key to Use

You can choose a KEY from the physical table so that the records cached into the Memory table are in an optimal order. Press the ellipsis button and select the table's KEY here.

If you plan to use a **Range Limit** to qualify which records are cached into your Memory table, you must select the table's KEY here.

Range Limit Field

Used in conjunction with the **Range Limit Type**, this option specifies a field component from the KEY you selected. The field you choose here is used to match the group of records for inclusion in the memory table.

Choose a field by pressing the ellipsis (...) button. The subsequent dialog displays the field components from your selected KEY. Range Limits are generally much faster than filters.

Range Limit Type

Specifies the type of range limit to apply. Choose one of the following from the drop-down list.

Single Value

This option allows you to specify a single value for which every record from the physical file Range Limit field must match. If you select "Single Value" the **Range Limit Value** entry field is enabled.

In the **Range Limit Value** you must specify either a value, or a variable that will contain the value used to limit the records added to the Memory table.

Range of Values

Choosing "*Range of Values*" for the *Range Limit Type* lets you specify upper and lower limits. Specify the values or the variables that will contain the limits in the **Low Limit** and **High Limit Value** boxes.

Range Limit Boundaries

This option is enabled only when you have selected a Range Limit Type. If you have selected a *Single Value*, enter a variable containing that value in the **Range Limit Value** box. If *Range of Values* is selected, enter the variables containing the limits in the **Low Limit** and **High Limit Value** boxes.

Include Blobs

If the data source used to cache the target memory table contains BLOBs (Binary Large Objects), check this box to specify that the BLOB contents will also get loaded into the target memory table. Leaving this check box off can greatly improve the memory table performance and program memory allocation.

Re-write physical file on Application Exit

Check this box if you would like to write the contents of the memory table back to the original source when the program exits. By default, a backup of the original data source will be created, and the entire contents of the memory table will update the original data source. This option is only applicable to ISAM data sources, and only available if you are globally caching ALL records. Filtered records are loaded and saved based on the filter criteria, but any records not in the filter set are maintained in the original data source. If you wish to write to an SQL based table, use the appropriate *Save* or *OptimizeSave* embed point to write the process.

User Defined Save (Optimize Save)

If you wish to write the contents of the memory table to the data source, but need to add additional validity checking or need to preserve some of the original data source's contents, check this box. This will override the template's default save strategy, and call the *OptimizeSave* virtual method. No code is generated in this method; you will need to write the source needed to save the memory table's contents. This option is only applicable to ISAM data sources.

See the *IMDD Class Library Reference* in this PDF for more information.

Load the file on a background thread

By default, all generated tables are loaded from the main program module. Check this option if you would like to load the target generated table on a background (or separate) thread. This will allow the program to continue its startup code while the target memory table is loading in the background. This will allow a faster program start if a generated memory table is larger than normal design.

Keep physical file synchronized

Check this option to allow automatic update of the physical memory table source any time the generated memory table is updated. This feature uses global triggers that update on a record-by-record basis. Note: concurrency checking is not handled by this option.

Dictionary Table options

This section of the In-Memory Data Caching Support global extension template allows you to use a Memory table that you have already defined in your data dictionary (Dictionary Table). One of the uses for this option is that you may want to add additional fields to the Memory table that does not exist in the Physical table. Examples of this are calculated fields, variable used for tagging, or a date/time stamp.

You might also intend to load the Memory table with data from more than one physical source. For example you may have normalized data stored in two or more tables, and for presentation or reporting might want to de-normalize it into a single table, for convenience or to optimize a report or process.

A third option is that you wish to Load the memory table *conditionally* in the application. The *Generated* table option always loads the memory table at program startup, and you may wish to load the table later on demand later on into the application. Use the *Dictionary* based memory table to do this.

The following prompts are provided:

Dictionary - Cached Tables

Physical Table: Detail

In-Memory Table: MemDetail

☒ Clear Record Before Assign

Field Assignment

Records to Read: ALL

Read this many Records: 100

SQL Query:

Filter:

KEY to Use: DTL:InvoiceNumberKe

Range Limit Field: DTL:InvoiceNumber

Range Limit Type: Range of Values

Range Limit Boundaries

Low Limit Value: GLO:LowLimit

High Limit Value: GLO:HighLimit

☐ Cache this table Globally

☐ Re-write physical file on Application Exit

☐ User defined Save (Optimize Save)

OK Cancel Help

Physical Table

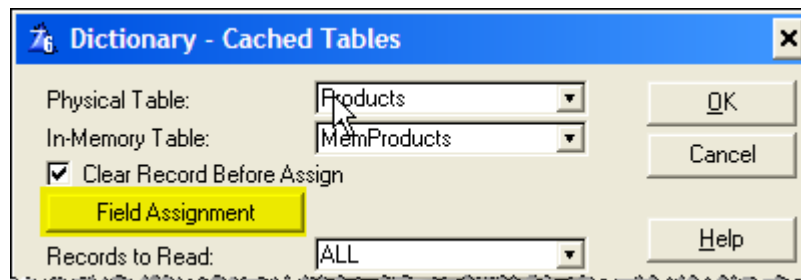
Select the physical table (the data source for the memory table) in the drop list provided.

In-Memory Table

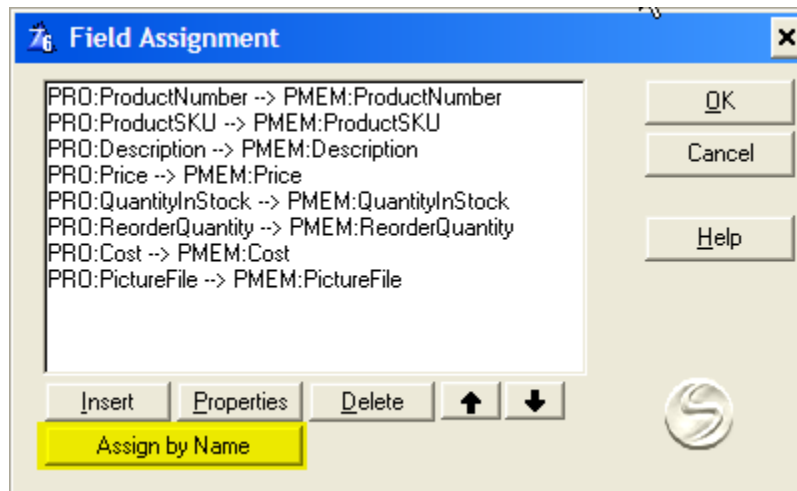
Select the In-Memory table (the target memory table) in the drop list provided. The template only lists the In-Memory tables that you have defined in the application's dictionary.

Clear Record Before Assign

Check this box to clear the In-Memory table record buffer before assignment from the physical table when loading records. Normally you will want this checkbox **ON**.



Press the **Field Assignment** button to access a list box where your custom column assignments are defined. The *Field Assignment* dialog allows you to map any field from the physical data table to any field in your Memory table. Press the **Assign by Name** button to auto map all matching names in the physical data table to your Memory table:



Records to Read

You can control how many records to load into the target Memory table. Select *ALL* from the drop list to load all records, or *Limited* in order to set a maximum number of records to cache.

Read this many Records

If *Limited* is selected in **Records to Read**, specify the maximum number of records to load into the Memory table.

SQL Query

If the original file to cache is an SQL based table, this text box is enabled to allow you to enter a valid SQL query that will be used to filter the memory table contents. Using an SQL based query is more efficient than using a standard filter.

Note:

If you are using a single quote in the filter expression, replace it with a double single quote. (i.e., `SELECT * FROM dbo.Customers WHERE Country=' 'Germany' '`).

Also, you can still filter your query results using the next option if needed.

An example application that demonstrates the use of SQL caching can be found in the <Clarion Root>\Examples\In-Memory Driver\IMDD-SQLcache folder, and is named *IMDDAndSQLEx.APP*.

Filter

Specify an optional Record Filter here. Type a valid Clarion expression to qualify which records should be cached into the Memory table. Only those records which match the filter expression are loaded into the Memory table. It is generally slower to Filter records rather than use Range Limits based on a KEY. Auto binding of global or local variables is not implemented in this template.

Key to Use

You can choose a KEY from the physical table so that the records cached into the Memory table are in an optimal order. Press the ellipsis button and select the table's KEY here.

If you plan to use a **Range Limit** to qualify which records are cached into your Memory table, you must select the table's KEY here.

Range Limit Field

Used in conjunction with the **Range Limit Type**, this option specifies a field component from the KEY you selected. The field you choose here is used to match the group of records for inclusion in the memory table.

Choose a field by pressing the ellipsis (...) button. The subsequent dialog displays the field components from your selected KEY. Range Limits are generally much faster than filters.

Range Limit Type

Specifies the type of range limit to apply. Choose one of the following from the drop-down list.

Single Value

This option allows you to specify a single value for which every record from the physical file Range Limit field must match. If you select "Single Value" the **Range Limit Value** entry field is enabled.

In the **Range Limit Value** you must specify either a value, or a variable that will contain the value used to limit the records added to the Memory table.

Range of Values

Choosing "*Range of Values*" for the *Range Limit Type* lets you specify upper and lower limits. Specify the values or the variables that will contain the limits in the **Low Limit** and **High Limit Value** boxes.

Range Limit Boundaries

This option is enabled only when you have selected a Range Limit Type. If you have selected a *Single Value*, enter a variable containing that value in the **Range Limit Value** box. If *Range of Values* is selected, enter the variables containing the limits in the **Low Limit** and **High Limit Value** boxes.

Cache this table Globally

This option causes this Memory table to be loaded on program start up with the records from the physical table. If this option is not checked, the Memory table is CREATED on start up, but must be loaded manually using an appropriate embed point or code template.

Note:

If the **THREADEDCONTENT** switch is set in the data dictionary for this table, the **Cache this table Globally, Re-write physical file on Application Exit** and **User Defined Save (Optimize Save)** options are disabled (discussed below). If your memory table has different content on different threads, you are responsible for loading the table and updating the original data source (you can use the associated **Load** and **Save** IMDD code templates to do this).

Re-write physical file on Application Exit

Check this box if you would like to write the contents of the memory table back to the original source when the program exits. By default, a backup of the original data source will be created, and the entire contents of the memory table will update the original data source. This option is only applicable to ISAM data sources, and only available if you are globally caching ALL records. Filtered records are loaded and saved based on the filter criteria, but any records not in the filter set are maintained in the original data source. If you wish to write to an SQL based table, use the appropriate **Save** or *OptimizeSave* embed point to write the process.

User Defined Save (Optimize Save)

If you wish to write the contents of the memory table to the data source, but need to add additional validity checking or need to preserve some of the original data source's contents, check this box. This will override the template's default save strategy, and call the *OptimizeSave* virtual method. No code is generated in this method; you will need to write the source needed to save the memory table's contents. This option is only applicable to ISAM data sources.

Concurrent Generated and Dictionary Tables

When using the In-Memory Data Caching Support global extension, it is possible to cache a memory table from both Generated and Dictionary tables, using the same physical data source. However, it is necessary to know the protocol of naming conventions when doing this.

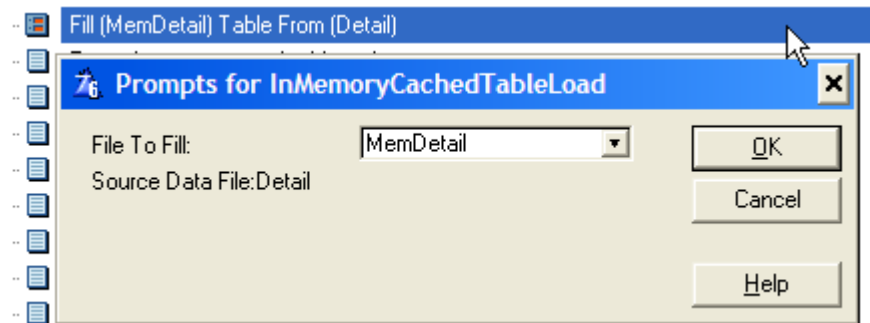
When you have the same physical table active on the Generated and Dictionary tabs, the Generated table is first cached to a memory table, and is always performed prior to the Dictionary table being cached. What is actually cached in the Dictionary table is the memory table which was first cached by the Generated option.

The global auto generated table creates a memory table with the original file name, and renames the Physical table using the `MEMAUX_tableName` convention. All of the FileManager/RelationManager and Clarion RI/RU code is performed using the memory table, not the Physical table. The Dictionary cached tables code is generated after the global auto generated tables, and therefore it works with the memory table.

Since memory tables do not support transaction processing, your Dictionary table, when concurrently used with a Generated table, is actually processing a memory table during Save and Load operations. The templates internally know this and handle the change effortlessly behind the scenes.

InMemoryCachedTableLoad Code Template

The **InMemoryCachedTableLoad** code template is used to load a defined *Dictionary* table with the contents of the physical table. This table must be activated by the **In-Memory Data Caching Support** global extension.



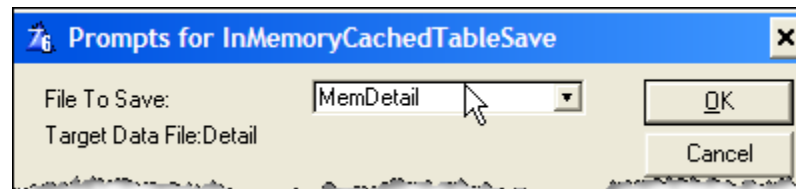
File To Fill

Select the memory table to load in the drop list provided. Only the memory tables that are active in the *Dictionary* tables global list will be available. The associated **Source Data File** will automatically be displayed.

Use this template when you do not wish to auto-load the memory table at program start up.

InMemoryCachedTableSave Code Template

The **InMemoryCachedTableSave** code template is used to save the contents of a *Dictionary* memory table to the physical table. This table must be activated by the **In-Memory Data Caching Support** global extension.



File To Save

Select the memory table to save in the drop list provided. Only the memory tables that are active in the *Dictionary* tables global list will be available. The save destination's **Target Data File** that is linked to the memory table will automatically be displayed.

Note:

Both of the code template described here are only available if the **In-Memory Data Caching Support** global extension is enabled.

Use this template when you do not wish to save the memory table at program shutdown. If an SQL table is used as the original data source, you must write the necessary functionality in the overridden **OptimizeSave** method located in the Global embeds.

ATTENTION! The **InMemoryCachedTableSave** code template will completely overwrite your original data source with the current contents of the memory table that match your global filter settings. If you are limiting your memory table to a fixed amount of records, only that number or records that match your filter criteria will be written to the original data source, and all other records will be lost.

IPDriver In-Memory Caching Support

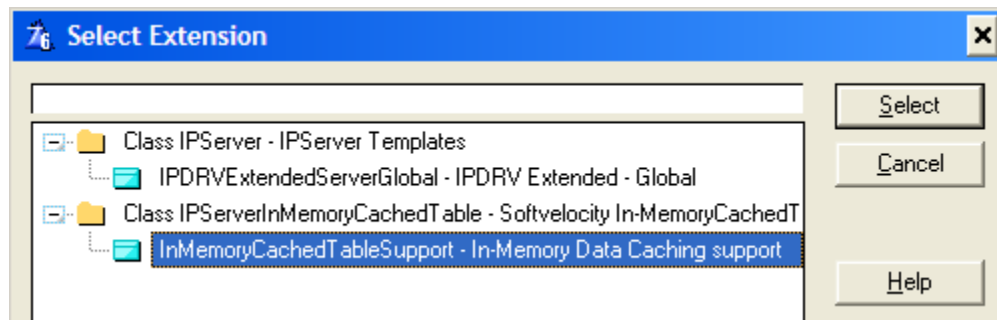
The In-Memory Database Driver also includes template support for another Clarion add-on product, the IP (Internet Protocol) Driver. **This new support is only compatible with the IP Driver Version 2.0 templates and greater.**

The In-Memory Data Caching Support global extension template allows you to easily and automatically cache any Server-side table to a Client-side In-Memory table. The support is added to the Data Manager DLL application, through the global IPServer template support.

The intended use of this support is the caching of IMDD “lookup” tables to the client from a server side table source. Usually these tables are read-only. Although you can certainly write the IMDD tables' contents back to the server table at any time, it is not a recommended practice in a multi-user environment, and careful programming steps must be considered. **In this version, the template based Save option is not supported in an IP-enabled configuration.**

You must first register this support template. This is accomplished by selecting the **Template Registry** menu item from the Clarion IDE **Setup** main menu. You will need to register **IPMemTable.TPL**.

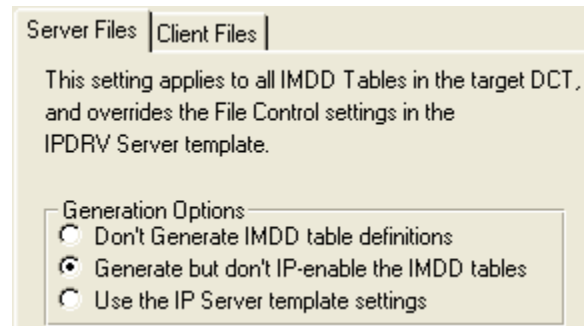
To add the In-Memory Data Caching Support extension, open your application used to create your IPServer Data Manager. In the *Global Properties* dialog, press the **Extensions** button, and in the *Extensions* dialog press the **Insert** button. The following dialog appears:



This extension controls how the IMDD tables defined in your target dictionary will be generated on both server and client locations.

The following template prompts are provided:

Server Files



Server Files | Client Files |

This setting applies to all IMDD Tables in the target DCT, and overrides the File Control settings in the IPDRV Server template.

Generation Options

- ☐ Don't Generate IMDD table definitions
- ☒ Generate but don't IP-enable the IMDD tables
- ☐ Use the IP Server template settings

Generation Options

This option controls how your IMDD tables declared in the IP Data Manager's target dictionary are processed. These server settings **ONLY** affect the IMDD tables defined in your data dictionary.

Select *Don't Generate IMDD table definitions* if you do *not* need the IMDD tables declared and visible in the Data Manager DLL. In this scenario, your IMDD dictionary tables are only used locally by the Client application, and no IMDD tables can be IP enabled.

Select *Generate but don't IP-enable the IMDD tables* if you plan to access IMDD tables via a remote procedure defined in the Data Manager DLL, and you do *not* want to "IP enable" the IMDD tables that are located on the server.

Finally, the *Use the IP Server template settings* will not override the individual **File Control** settings located in the IP Data Manager DLL global extension, which are used to control *all* of the dictionary tables (including IMDD) that can be IP enabled in the IP Client application.

In summary, all IMDD tables defined in your data dictionary can exist on both Server and Client in any IP enabled application. If an IMDD table is IP enabled, you should remember that data does not exist in this table and you must first pre-process the IMDD table from a remote server procedure. Once an IMDD table is IP enabled, it essentially has the same behavior as any other IP enabled table (TopSpeed, Btrieve, etc.) as seen from the Client side.

A nice feature of this template is allowing the existence of IMDD tables to occur on the server, where remote procedures can process their data directly.

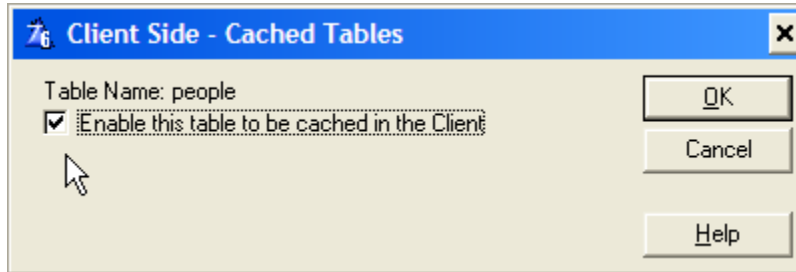
Client Files

This section of the template controls the extent of the IMDD caching support that you need in any target IP Client application.

Don't Allow Cached Tables in the Client

Check this box if you need to disable the IMDD caching support of all dictionary tables (that are not In-Memory) to the Client side location. By checking this box, you are essentially only using this template extension for the server side processing of IMDD tables.

To make a table available for use with the IMDD client template, highlight the table and press the **Properties** button.



Check the **Enable this table to be cached in the Client** check box to make a table available for use with the IMDD client template. You must include the **In-Memory Data Caching Support** global extension template in your IP Client application to activate this caching support.


Template Notes

The following items need to be considered when using this template:

When using the **Generated Table** option, the label used in the application for the original physical data source is renamed, using the form:

MEMAUX_OriginalLabel

For example:



```
!This table was modified by the In-MemoryCachedTableSupport Template
!the driver attribute was changed to the In-Memory driver
MEMAUX_States FILE, DRIVER('TOPSPEED'), PRE(MEMAUX_STA), CREATE, BINDABLE, THREAD, NAME('States')
StateCodeKey KEY(MEMAUX_STA:StateCode), NOCASE, OPT
Record RECORD, PRE()
StateCode STRING(2)
Name STRING(25)
END
END

!This table was generated by the In-MemoryCachedTableSupport Template
States FILE, DRIVER('MEMORY'), PRE(STA), CREATE, BINDABLE, THREAD, CREATE
StateCodeKey KEY(STA:StateCode), NOCASE, OPT
Record RECORD, PRE()
StateCode STRING(2)
Name STRING(25)
END
END
```

The template implementation for the Memory table is designed to seamlessly replace the use of the original data source throughout the application. However, if you need to reference both the physical data source and Memory table, and wish to retain the original Label for the physical table and its KEYs, then instead of the **Generated Table** option choose the **Dictionary Table** option. Doing so allows you to make use of the power features of the template for loading the Memory table, and all of your Procedures will continue to use the physical tables.

- This template uses the **FileSynchronization** Class to perform the necessary operations. The available properties and methods are discussed in the next section. By default, the template will generate a module name using the following naming convention:

<applicationname>Mn.CLW

where *n* is an integer from 0 to 9.

This module contains any overridden methods of the base **FileSynchronization** Class.

- The embed points for each memory table virtual methods can be found in the application's global embeds.

Assertion Message

If you are using the **InMemoryCachedTableLoad** code template on the same IMDD table more than once in a single procedure, and are linking in *Debug* mode, you may see an assertion message:

Assertion is on line:480 in file ABFILE.CLW

This is not a problem and can safely be ignored.

DLL considerations

Using the IMDD In-Memory Data Caching Support global extension template in a DLL configuration is fairly straightforward.

- Make sure to include the global extension in the DLL where all tables are referenced (e.g. – not defined as external).
- The executable file and other DLLs using the IMDD caching must also include the global extension. The template will disable all global options if template data is referenced as External (e.g., the **Generate template globals and ABC's as EXTERNAL** global option is active).
- Any initialization needed for generated tables (i.e., priming a filter prior to Load) must be implemented in the *Program Setup – Load In-Memory Tables* global embed point located in the target DLL (not the executable). See the shipping DLL example application for more information.

IMDD Class Library Reference

Overview

The In-Memory Database Driver includes a special class library that is used to support the *In-MemoryCachedTable* extension template that ships with the IMDD. The **FileSynchronization** Class contains the necessary properties and methods needed to effectively process a memory table from any alternative data source.

More information regarding the *In-MemoryCachedTable* extension template can be found in the *Template Guide* section of this document.

FileSynchronization Class Source Files

The source code for the FileSynchronization class is installed by default to the Clarion \LIBSRC folder. The source code and their respective components are contained in:

ABTbISyn.INC	Class declarations
ABTbISyn.CLW	Class method definitions

Template Support

A simple, yet robust extension template allows easy implementation of this class into your applications. See the *Template Guide* section in this PDF for a detailed description of this template tool.

Essentially, the global extension includes the proper class derivations needed. Other template options implement the appropriate methods needed for file loading, filtering, and “save on exit” features.

Each table activated by the template support generates a distinct derived FileSynchronization object. For example, if memory caching support is enabled for the *Customer* table, the following object will be instantiated:

CachedManager:Customers

In addition, the template generates two procedures that are used for housekeeping of all *CachedManager* objects.

MemDriverInit is used to initialize all of the *CachedManager* objects.

MemDriverKill is used to kill all of the *CachedManager* objects.

Template Embeds

Each memory table activated by the template support will generate a set of embed points that correspond to the following Virtual methods used in the FileSynchronization Class:

Close	Close data source and target table
GenerateBackUp	Make a data source backup on Save
Init	Initialize Class
Kill	Kill Class
Load	Move data from source to memory table
Open	Open data source and target table
OptimizedSave	Perform a custom save
PrimeKey	Initialize key element
PrimeRecord	Initialize record on Load
PrimeSaveRecord	Initialize record on Save
Reset	Clear all data source contents
ResetRange	reset key or file range to start
Save	Move data from memory table to data source
SetFilter	Initialize record filter
SetOrderKey	set to key order sequence
SetRange	set to range limit sequence
SetRecordsToRead	Set memory table maximum records
TransactionCommit	Complete data source transaction frame
TransactionRollback	Abort data source transaction frame
TransactionStart	Begin data source transaction frame
ValidateRecord	Validate active record before write

FileSynchronization Class Properties

The following **FileSynchronization** Class properties are available:

BackupOnSave (backup physical file on Save)

BackupOnSave **BYTE**

The **BackupOnSave** property is a BYTE value that when *TRUE* (1) indicates that a backup copy of the physical data source (specified by the **Original** property) will be created prior to the **Save** method.

Implementation:

The **BackupOnSave** property is set by default to *TRUE* by the **Init** method. This will activate the **GenerateBackUp** method in the **Save** method prior to any writing to the **Original** data source.

See Also: **GenerateBackUp**, **Save**

Filter (record filter expression)

Filter **&STRING**

The **Filter** property contains a reference to a valid filter expression used to filter records loaded into a memory table, and also to filter records saved to the original source.

Implementation:

The **Filter** property is set by the **SetFilter** method, and later EVALUATED by the **ValidateRecord** method, which is called in both **Load** and **Save** methods. The **Filter** prompt in the **In-Memory Data Caching Support** global extension template dialog sets this property.

FreeElement (comparison value for range limits)

FreeElement **ANY**

The **FreeElement** property identifies the Free Key element (column) used for range limiting records loaded into the memory table, and also used to range limit records saved to the original source.

Implementation:

The **FreeElement** property is set by the **SetRange** method. The **Range Limit Field** prompt in the **In-Memory Data Caching Support** global extension template dialog sets this property

HighLimit (range of values upper limit)

HighLimit	ANY
------------------	------------

The **HighLimit** property sets the upper filter range of range-limited records loaded into the memory table, and saved to the original source.

Implementation:

The **HighLimit** property is set by the **SetRange** method, and later compared in the **ValidateRecord** method, which is called in both **Load** and **Save** methods. The **High Limit Value** prompt in the **In-Memory Data Caching Support** global extension template dialog sets this property.

LimitType (type of range limit process)

LimitType	BYTE
------------------	-------------

The **LimitType** property indicates the type of filtering active when a memory table is first loaded from and later saved to a data source. If **LimitType** is 0, no filtering or key is used. If **LimitType** is 1, no filtering is used but the memory table is processed in key order. If **LimitType** is 2, a single range limit value is active. If the **LimitType** is set to 3, a Range of Values filtering type is active.

Implementation:

The **LimitType** property is set by the **SetRange** and **SetOrderKey** methods.

LoadSuccess (memory table load was successful)

LoadSuccess	BYTE, PROTECTED
--------------------	------------------------

The **LoadSuccess** property is a protected BYTE value that when **TRUE** (1) indicates that a successful load of the memory table (identified by the Target property) was executed. If any error is encountered during the **Load** method, this property is used to prevent writing incomplete or corrupted data to the original data source when saving via the **Save** method.

Implementation:

The **LoadSuccess** property is first cleared by the **Init** method. If no errors are encountered during the **Load** method, a *Level:Benign* is set, and **LoadSuccess** is subsequently set to **TRUE**. During the **Save** method, if the **LoadSuccess** property is set to **FALSE**, the save process is aborted, and the data source is unchanged.

See Also: Load

LowLimit (range of values lower limit)

LowLimit	ANY
-----------------	------------

The **LowLimit** property sets the lower filter range of range limited records loaded into the memory table, and saved to the original source.

Implementation:

The **LowLimit** property is set by the **SetRange** method, and later compared in the **ValidateRecord** method, which is called in both **Load** and **Save** methods. The **Low Limit Value** prompt in the **In-Memory Data Caching Support** global extension template dialog sets this property.

Original (data source table file reference)

Original &FILE,PROTECTED

The **Original** property is a reference to the data source table used to load the memory table. It is also the target table structure used by the **Save** method.

Implementation:

The **Original** property is initialized in the **Init** method, and used in multiple methods where the data source is processed.

OptimizeSave (custom save option)

OptimizeSave BYTE,PROTECTED

The **OptimizeSave** property indicates (when set to TRUE) that the default save process contained in the **Save** method will be bypassed. Instead, the **OptimizedSave** VIRTUAL method is called containing a developer created custom save process.

The default save process will on save empty the data source, and replace it entirely with the contents of the memory table.

Implementation:

The **OptimizeSave** property is set by the **Init** method to FALSE by default, which activates the **Save** method process to write the memory table contents to the original source.

PrimaryKey (reference to data source primary key)

PrimaryKey &KEY

The **PrimaryKey** property is a reference to a table's primary key. This table is identified as the data source, which is identified by the **Original** property.

Implementation:

The **PrimaryKey** property is initialized in the **Init** method. It is used to set the **RecordPositionField** property used in the **Load** method.

RangeKey(reference to data source range limit key)

RangeKey &KEY

The **RangeKey** property is a reference to the key in the data source table used to range limit the contents loaded into and saved from the memory table. This table is identified as the data source, which is set by the **Original** property.

Implementation:

The **RangeKey** property is initialized in the **SetOrderKey** method. It is used in the **ValidateRecord** and **ResetRange** methods to initialize and reset the range key filter technique.

RecordPositionField (key or table position)**RecordPositionField ANY,PROTECTED**

The **RecordPositionField** property holds the value of a key or table position. Which position value used is based on how the memory table is processed.

Implementation:

The **RecordPositionField** property is set by the **SetRecordPositionField** method. This property is not implemented in any methods, but can be used by the developer if a key or table's position is needed during a custom save process.

See Also: **SetRecordPositionField**

RecordsToRead (records to process from data source)**RecordsToRead LONG**

The **RecordsToRead** property holds the maximum number of records that will be loaded into the target memory table. When set to zero (0), all records from the data source will be processed into the target memory table.

Implementation:

The **RecordsToRead** property is cleared by the **Init** method, and optionally set by the **SetRecordsToRead** method.

See Also: **SetRecordsToRead**

SaveOnKill (save changes on exit to data source)**SaveOnKill BYTE**

The **SaveOnKill** property is a BYTE value that when set to TRUE (1) indicates that a save process to the data source (identified by the **Original** property) will be activated when the program is terminated normally.

Implementation:

The **SaveOnKill** property is cleared (set to FALSE) by the **Init** method. The **Kill** method detects the state of **SaveOnKill**, and will call the **Save** method if the **RecordsToRead** property is also set to zero (0) – no records are filtered by the data source.

Silent (silent mode flag)**Silent BYTE**

The **Silent** property is a BYTE value that indicates when TRUE (1) that a Silent mode is active. This property is currently not implemented in the **FileSynchronization** class.

Implementation:

The **Silent** property is set to TRUE by the **Init** method.

StartTransaction (OK to start transaction processing)**StartTransaction** **BYTE**

The **StartTransaction** property is a BYTE value that indicates when TRUE (1) that transaction processing will be active when writing to the data source (Original or Physical Table).

Implementation:

The **StartTransaction** property is set to TRUE by the **Init** method. This activates transaction-processing support for the single data source during the standard **Save** method. If **UseLogout** is active and **StartTransaction** is FALSE, the transaction logout must be processed manually.

See Also: **Save**

Target (target Memory table file reference)**Target** **&FILE,PROTECTED**

The **Target** property is a reference to the target memory table processed from the data source. It is also the target table structure used by the **Load** method.

Implementation:

The **Original** property is initialized in the **Init** method, and used in multiple methods where the data source is processed.

See Also: **Original**

UseSQL (SQL data source)**UseSQL** **BYTE, PROTECTED**

The **UseSQL** property is a BYTE value that indicates when TRUE (1) that an SQL data source (Original or Physical Table) is used to cache the memory table.

Implementation:

The **UseSQL** property is set to FALSE by the **Init** method. The property is queried in **ValidateRecord** method to disable the range-limiting feature. Use the **SetUseSQL** method to set this property.

UseLogout (transaction processing enabled)**UseLogout** **BYTE**

The **UseLogout** property is a BYTE value that indicates when TRUE (1) that transaction processing will be active when writing to the data source (Original or Physical Table).

Implementation:

The **UseLogout** property is set to TRUE by the **Init** method. This activates transaction-processing support for the data source during the standard **Save** method. The **TransactionStart** method is called when **UseLogout** is active (TRUE).

FileSynchronization Class Methods

The following **FileSynchronization** Class methods are available:

AsynchronousLoad (load IMDD table on background thread)

AsynchronousLoad()

AsynchronousLoad Load an IMDD table on a separate background thread

The **AsynchronousLoad** method is used to load an IMDD table on a background (e.g. separate) thread. By default, all IMDD tables are loaded on the main program thread. For large IMDD tables, this can cause a delay in the program startup. This method allows the load to occur on a separate thread, allowing a faster program startup.

Implementation: **AsynchronousLoad** is implemented by the supporting IMDD templates, and called from within the main program area.

Example:

```
CODE
GlobalErrors.Init(GlobalErrorStatus)
FuzzyMatcher.Init
FuzzyMatcher.SetOption(MatchOption:NoCase, 1)
FuzzyMatcher.SetOption(MatchOption:WordOnly, 0)
INIMgr.Init('IMDDExample.INI', NVD_INI)
DctInit
SelectRegion()
MemAccess:Customers.Load()
MemAccess:Orders.Load()
MemAccess:Products.Load()
MemAccess:States.AsynchronousLoad()
```

See Also: **Load, GetLoadSuccess**

BindOnLoad (BIND variables on asynchronous load)

BindOnLoad(),VIRTUAL

BindOnLoad Allows you to bind variables needed for an asynchronous load.

BindOnLoad is a virtual method that is used to bind any variables needed prior to the loading of an IMDD table via the FileSynchronization Class.

Implementation: **BindOnLoad** is called from the **Load** method. Any variable that needs to be bound (using BIND) should be added in this method.

Example:

CachedManager:States.BindOnLoad PROCEDURE

```
CODE
BIND('GLO:Region',GLO:Region)
PARENT.BindOnLoad
```

See Also: **AsynchronousLoad**

Close (close data source and target tables)

Close(),VIRTUAL

Close is a virtual method that is used to close the data source and target memory table.

Implementation:

Close is called by the **Save** and **Load** methods when processing is completed.

Base Method:

```
FileSynchronizationClass.Close                                PROCEDURE ()
RetVal      BYTE
CODE
    IF NOT SELF.OldStatusOrigin
        CLOSE (SELF.Original)
    END
    IF NOT SELF.OldStatusTarget
        CLOSE (SELF.Target)
    END
    SELF.OldStatusOrigin = True
    SELF.OldStatusTarget = True
```

Example:

```
SELF.CLOSE ()
```

See Also: Save, Load

Destruct (dispose the filter)

Destruct

Destruct is the Destructor method used in the base class to deallocate heap memory used by the **Filter** property.

Base Method:

```
FileSynchronizationClass.Destruct                            PROCEDURE ()
CODE
    IF NOT SELF.Filter &= NULL
        DISPOSE (SELF.Filter)
    END
```

GenerateBackUp (create data source backup table)

GenerateBackUp(),VIRTUAL

The **GenerateBackUp** virtual method creates a backup (i.e., *customer.tpsBAK*) of the **Original** data source prior to saving the target memory table contents to the original copy. This allows restoration of critical data in case of unforeseen errors during the **Save** process.

This method can be overridden with your own custom backup process. Currently this backup process is only designed for ISAM data sources. If your table is not a TopSpeed table, you must always ensure that a valid NAME attribute is active for the data source. Backup of Multi-table TPS files (super files) are currently not supported by this method.

Base Method:

```
FileSynchronizationClass.GenerateBackUp          PROCEDURE ()
FileName CSTRING(FILE:MaxFilePath)
lIndex   SHORT
CODE
    !super table tps file currently not implemented
    !PROP:Name must return a valid file name.
    FileName = SELF.Original{PROP:Name}
    IF NOT INSTRING('.',FileName,1,1)
        FileName=FileName&'.TPS'
    END
    lIndex=0
    LOOP
        lIndex+=1
        IF EXISTS(FileName&'tmp'&LEFT(lIndex)) THEN CYCLE.
        !Copy Original File to a .tmp file
        COPY(FileName,FileName&'tmp'&LEFT(lIndex))
        !IF ERRORCODE()
        !    MESSAGE(ERROR())
        !END
        BREAK
    END
    !Change bak file to baktmp
    RENAME(FileName&'bak',FileName&'baktmp')
    !Rename Original tmp to bak
    RENAME(FileName&'tmp'&LEFT(lIndex),FileName&'bak')
    !Remove baktmp
    REMOVE(FileName&'baktmp')
```

Implementation:

GenerateBackUp will be called by the **Save** method if the **BackupOnSave** property is active (TRUE). Before the memory table is written to the data source, the following sequence is performed:

Original data source *filename* copied to *filename* & 'tmp' (i.e., *customer.tps* to *customer.tpstmp*)

Original data source backup (if it existed in a previous session) renamed to *filename* & 'tmp' (i.e., *customer.tpsbaktmp*)

Original data source renamed to *filename* & 'bak' (i.e., *customer.tpsbak*)

Original data source backup (i.e., *customer.tpsbaktmp*) is removed.

Example:

```
FileSynchronizationClass.Save          PROCEDURE ()
CODE
    IF SELF.LoadSuccess = False
        RETURN Level:Fatal
    END

    IF SELF.BackupOnSave
        SELF.GenerateBackUp()
    END
```

See Also: Original, BackupOnSave

GetLoadSuccess (*successful asynchronous load*)**GetLoadSuccess**()**GetLoadSuccess** Check for successful asynchronous load

The **GetLoadSuccess** method is used to check if an AsynchronousLoad (e.g., loading an IMDD table on a background thread) has completed successfully. You can use this method to delay display of the target IMDD table only after it has loaded completely. If an asynchronous load has completed, **GetLoadSuccess** returns TRUE (1). Otherwise, **GetLoadSuccess** returns FALSE (0).

Return Value: BYTE

Implementation: **GetLoadSuccess** is not implemented by the IMDD templates. You can call it at any time to verify that an asynchronous load has been completed.

Example:

```
SELF.FilesOpened = True
! [Priority 7600]

! Initialize browse
BRW1.Init(?Browse:1,Queue:Browse:1.ViewPosition,|
          BRW1::View:Browse,Queue:Browse:1,|
          Relate:States,SELF) ! Initialize the browse manager
! [Priority 7751]
IF MemAccess:States.GetLoadSuccess()
  MESSAGE('Table Load Completed')
ELSE
  MESSAGE('Browsing partial IMDD table')
END
! Open the window
SELF.Open(QuickWindow)
```

See Also: **AsynchronousLoad**

GetSyncError (*get synchronous error message*)**GetSyncError**()**GetSyncError** Return error string after unsuccessful update

The “Sync” functions handle record synchronization between the IMDD and physical table. **GetSyncError** returns the **ERROR()** generated in the appropriate target “Sync” function.

Return Value: STRING

Implementation: The IMDD templates call the **GetSyncError** method after any unsuccessful Sync method is encountered (SyncInsert, SyncUpdate, SyncDelete, SyncRefresh).

Example:

```
Hide:Access:Customers.PreUpdate PROCEDURE (LONG Pntr, UNSIGNED |
                                     PutLen, *CSTRING ErrCode, *CSTRING ErrMsg)
```

```
ReturnValue          BYTE, AUTO
```

```
Buffer LIKE (CUS:RECORD)
CODE
PUSHERRORS ()
IF SELF.SavePreviousBuffer AND NOT (SELF.PreviousBuffer &= NULL)
    Buffer = SELF.PreviousBuffer
END
IF MemAccess:Customers.SyncUpdate (Customers) <> Level:Benign
    ErrCode=90
    ErrMsg='The file could not be synchronized.|
           (' &MemAccess:Customers.GetSyncErrorCode () &' ) |
           ' &MemAccess:Customers.GetSyncError ()
    POPERRORS ()
    ReturnValue = False
    RETURN ReturnValue
END
ReturnValue = PARENT.PreUpdate (Pntr, PutLen, ErrCode, ErrMsg)
POPERRORS ()
RETURN ReturnValue
```

See Also: **GetSyncErrorCode**

GetSyncErrorCode (*get synchronous error code*)**GetSyncErrorCode**()**GetSyncErrorCode** Return error code after an unsuccessful update

The “Sync” functions handle record synchronization between the IMDD and physical table. **GetSyncErrorCode** returns the **ERRORCODE**() generated in the appropriate target “Sync” function.

Return Value: **BYTE**

Implementation: The IMDD templates call the **GetSyncErrorCode** method after any unsuccessful Sync method is encountered (SyncInsert, SyncUpdate, SyncDelete, SyncRefresh).

Example:

```
Hide:Access:Customers.PreUpdate PROCEDURE (LONG Pntr, UNSIGNED |
                                     PutLen, *CSTRING ErrCode, *CSTRING ErrMsg)
```

```
ReturnVal                    BYTE, AUTO
```

```
Buffer LIKE (CUS:RECORD)
CODE
PUSHERRORS ()
IF SELF.SavePreviousBuffer AND NOT (SELF.PreviousBuffer &= NULL)
    Buffer = SELF.PreviousBuffer
END
IF MemAccess:Customers.SyncUpdate (Customers) <> Level:Benign
    ErrCode=90
    ErrMsg ='The file could not be synchronized. |
            (' &MemAccess:Customers.GetSyncErrorCode () &' ) |
            ' &MemAccess:Customers.GetSyncError ()
    POPERRORS ()
    ReturnVal = False
    RETURN ReturnVal
END
ReturnVal = PARENT.PreUpdate (Pntr, PutLen, ErrCode, ErrMsg)
POPERRORS ()
RETURN ReturnVal
```

See Also: GetSyncError

Init (initialize file synchronization)**Init(TargetFile,OriginalFile),VIRTUAL**

Init	Initializes the FileSynchronization object
<i>TargetFile</i>	The label of the target memory table structure to process.
<i>OriginalFile</i>	The label of the data source table structure to process.

The **Init** method is a virtual method used to initialize the **FileSynchronization** object. A variety of property assignments are performed, the Primary Key of the *OriginalFile* source is detected, and the *TargetFile* (Memory Table) is created.

Base Method:

```
FileSynchronizationClass.Init  PROCEDURE(FILE pTargetFile,FILE pOriginalFile)
lIndex      SHORT
lAuxKey      &KEY
CODE
  SELF.UseLogout      = True
  SELF.StartTransaction = True
  SELF.RecordsToRead    = 0 !Process ALL records of data source
  SELF.SaveOnKill      = False
  SELF.Target          &=pTargetFile
  SELF.Original        &=pOriginalFile
  SELF.LimitType       = 0 !Default to no Range Limit
  SELF.Silent          = True
  SELF.OptimizeSave    = False
  IF NOT SELF.Filter &= NULL
    DISPOSE(SELF.Filter)
  END
  LOOP lIndex = 1 TO SELF.Original{PROP:Keys}
    lAuxKey &= SELF.Original{PROP:Key,lIndex}
    IF lAuxKey{PROP:PRIMARY}
      SELF.PrimaryKey &= lAuxKey
      BREAK
    ELSE
      IF NOT lAuxKey{PROP:DUP}
        SELF.PrimaryKey &= lAuxKey
      END
    END
  END
  END
  CREATE(SELF.Target)
```

Implementation:

The **Init** method should be called just after the object is created.

Example:

```
MemDriverInit  PROCEDURE          ! Initializes the MemDriver definition module
CODE
  MemAccess:Orders &= CachedManager:Orders
  CachedManager:Orders.Init(Orders,MEMAUX_Orders)
  CachedManager:Orders.SetRecordsToRead(100)
  CachedManager:Orders.SetOrderKey(MEMAUX_ORD:InvoiceNumberKey)
  CachedManager:Orders.SetRange(MEMAUX_ORD:InvoiceNumber,GLO:LowLimit,GLO:HighLimit)
```

See Also: Kill

Kill (save changes and dispose the filter)**Kill(),VIRTUAL**

The **Kill** method shuts down the **FileSynchronization** object by freeing any memory allocated during the life of the object and executing any other required termination code.

Base Method:

```
FileSynchronizationClass.Kill                                PROCEDURE ()
RetVal      BYTE
CODE
  IF SELF.SaveOnKill AND SELF.RecordsToRead=0
    RetVal = SELF.Save()
  END
  SELF.LimitType      = 0
  IF NOT SELF.Filter &= NULL
    DISPOSE(SELF.Filter)
  END
```

Implementation:

The **Kill** method should be called at program shutdown. The template support generates a *MemDriverKill* procedure that in turn calls the **Kill** method for all **FileSynchronization** objects that were created.

Example:

```
MemDriverKill      PROCEDURE      ! Kills the MemDriver definition module
CODE
  CachedManager:Orders.Kill()
  CachedManager:States.Kill()
```

See Also: Init

Load (write source data to memory table)**Load(),VIRTUAL**

The **Load** method is used to read the contents of the data source (identified in the **Original** property) and write the validated contents to the target memory table (identified in the **Target** property). If Load is successful, a value of zero is returned. Any other non-zero value can be translated to an error level (Level:Fatal, Level:Notify, etc.) that can be trapped and processed as needed.

Return Value: BYTE

Base Method:

```
FileSynchronizationClass.Load                                PROCEDURE ()

OldStatusTarget      BYTE
OldStatusOrigin      BYTE
RetVal               BYTE
lIndex               LONG
lRecordsToRead       LONG
TargetRecord         &GROUP, AUTO
OriginalRecord       &GROUP, AUTO

CODE
    RetVal = SELF.Reset()
    IF RetVal <> Level:Benign THEN RETURN RetVal.
    !
    RetVal = SELF.Open()
    IF RetVal <> Level:Benign THEN RETURN RetVal.
    !
    IF SELF.RecordsToRead=0
        lRecordsToRead = RECORDS(SELF.Original)
    ELSE
        lRecordsToRead = SELF.RecordsToRead
    END
    TargetRecord  &= SELF.Target{PROP:Record}
    OriginalRecord &= SELF.Original{PROP:Record}
    CLEAR(OriginalRecord)
    SELF.ResetRange()
    lIndex = lRecordsToRead
    LOOP
        IF lIndex<l THEN BREAK.
        NEXT(SELF.Original)
        IF ERRORCODE() THEN
            MESSAGE('Error Reading the Physical file. Error('&ERRORCODE()&')|
                '&ERROR()&'| '&SELF.Original{PROP:Name}')
            BREAK
        END
        IF SELF.ValidateRecord()<>Level:Benign THEN CYCLE.
        !ASSIGN VALUES
        CLEAR(TargetRecord)
        TargetRecord = OriginalRecord
        SELF.PrimeRecord()
        IF NOT SELF.RecordPositionField &= NULL
            IF NOT SELF.PrimaryKey &= NULL
                SELF.RecordPositionField = POSITION(SELF.PrimaryKey)
            ELSE
                SELF.RecordPositionField = POSITION(SELF.Original)
            END
        END
        ADD(SELF.Target)
```

```

        IF ERRORCODE()
            MESSAGE('Error Writing the Memory file. Error('&ERRORCODE()&') |
                '&ERROR()&' | '&SELF.Original{PROP:Name}')
            BREAK
        END
        lIndex -= 1
    END
    !
    SELF.Close()

```

Implementation:

The **Load** method is called just after the INIClass and Dictionary Class have been initialized in the Program start. The template support creates an object for each data source and memory table using the following naming convention:

MEMAccess:*tablename.method*

Example:

```

CODE
GlobalErrors.Init(GlobalErrorStatus)
FuzzyMatcher.Init                                ! Initilaize the browse 'fuzzy matcher'
FuzzyMatcher.SetOption(MatchOption:NoCase, 1)      ! Configure case matching
FuzzyMatcher.SetOption(MatchOption:WordOnly, 0)    ! Configure 'word only' matching
INIMgr.Init('IMDDEExample.INI', NVD_INI)          ! Configure INIManager to use INI file
DctInit
MemAccess:Orders.Load()
MemAccess:States.Load()

```

See Also: Save

Open (open data source and target table)**Open(),VIRTUAL**

Open is a virtual method that is used to open the data source and target memory table, prior to the start of any processing needed. If the method is successful, **Open** returns *Level:Benign* (0). Otherwise, an appropriate error is posted, and *Level:Fatal* (3) is returned.

Implementation: **Open** is called by the **Save** and **Load** methods when processing begins.

Return Value: BYTE

Base Method:

```
FileSynchronizationClass.Open                                PROCEDURE ()
RetVal                                                       BYTE
CODE
    SELF.OldStatusTarget = STATUS(SELF.Target)
    IF NOT SELF.OldStatusTarget
        LOOP
            OPEN(SELF.Target)
            CASE ERRORCODE()
            OF 0! No Error
                SELF.OldStatusTarget = False
                RetVal = Level:Benign
                BREAK
            OF 2! FileNotFound
                CREATE(SELF.Target)
                IF ERRORCODE() THEN RETURN Level:Fatal.
                CYCLE
            OF 52! File Already Open
                SELF.OldStatusTarget = STATUS(SELF.Target)
                BREAK
            ELSE
                MESSAGE('Error Opening the Memory File. Error('&ERRORCODE()&') '&ERROR())
                RETURN Level:Fatal
            END
        END
    END
    IF RetVal = Level:Benign
        SELF.OldStatusOrigin = STATUS(SELF.Original)
        IF NOT SELF.OldStatusOrigin
            LOOP
                OPEN(SELF.Original)
                CASE ERRORCODE()
                OF 0! No Error
                    SELF.OldStatusOrigin = False
                    BREAK
                OF 2! FileNotFound
                    CREATE(SELF.Original)
                    IF ERRORCODE() THEN RETURN Level:Fatal.
                    CYCLE
                OF 52! File Already Open
                    SELF.OldStatusOrigin = STATUS(SELF.Original)
                    BREAK
                ELSE
                    MESSAGE('Error Opening the Physical file. Error('&ERRORCODE()&') |
                        '&ERROR()&' | '&SELF.Original{PROP:Name}')
                    RetVal = Level:Fatal
                    BREAK
                END
            END
        END
    ELSE
        END
    RETURN RetVal
```


Example:

```
FileSynchronizationClass.Load    PROCEDURE()
OldStatusTarget    BYTE
OldStatusOrigin    BYTE
RetVal            BYTE
lIndex            LONG
lRecordsToRead    LONG
TargetRecord        &GROUP, AUTO
OriginalRecord      &GROUP, AUTO
CODE
    RetVal = SELF.Reset()
    IF RetVal <> Level:Benign THEN RETURN RetVal.
    !
    RetVal = SELF.Open()
    IF RetVal<>Level:Benign THEN RETURN RetVal.
    !
```

See Also: Save, Load

OptimizedSave (customized save)

OptimizedSave(),VIRTUAL

OptimizedSave is a virtual method used as an alternative to the **Save** method. It is the developer's responsibility to override the base method and write the custom source code. By default, this method returns FALSE, and can be used to verify that the save to the data source was valid.

Return Value: BYTE

Implementation:

OptimizedSave is called by the **Save** method when the **OptimizeSave** property is set to TRUE. The **OptimizeSave** property is set to FALSE in the **Init** method, and is set to TRUE in the **SetRecordPositionField** method.

See Also: Save, OptimizeSave

PrimeKey (initialize key element)

PrimeKey(),VIRTUAL

The **PrimeKey** method is used to prime the free key element used in an optional filter process when the memory table is created from and saved to the data source. The free element key is initialized if a single limit or range limit type filter is active.

Base Method:

<pre>FileSynchronizationClass.PrimeKey CODE CASE SELF.LimitType OF 1 !No range but using Key order OF 2 !Single Limit Type SELF.FreeElement = SELF.LowLimit OF 3 !Range Limit Type SELF.FreeElement = SELF.LowLimit ELSE END</pre>	<pre>PROCEDURE ()</pre>
--	-------------------------

Implementation:

The **PrimeKey** method is called from the **ResetRange** method, and sets the **FreeElement** property.

See Also: ResetRange, FreeElement

PrimeRecord (prime fields on load)**PrimeRecord(),VIRTUAL**

The **PrimeRecord** is a virtual method used to process the memory table contents prior to the actual write to the table. Contents can be validated and accepted or rejected based on any condition.

Base Method:

There is no code in the base method.

Implementation:

The **PrimeRecord** method is called by the **Load** method after the assignment of the **TargetRecord** property from the **OriginalRecord** property.

See Also: Load

PrimeSaveRecord (prime fields on save)**PrimeSaveRecord(),VIRTUAL**

PrimeSaveRecord is a virtual method used to process the physical table contents prior to the actual write to the data source. Contents can be validated and accepted or rejected based on any condition.

Base Method:

There is no code in the base method.

Implementation:

The **PrimeSaveRecord** method is called by the **Save** method after the target record is read, just prior to writing to the original data source.

See Also: Save

Reset (clear data source contents)**Reset(*ForceReset*),VIRTUAL****Reset** Clear the memory table*ForceReset* A BYTE value that controls the properties that needs to be reset.

The **Reset** virtual method clears (empties) the target memory table, If *ForceReset* is TRUE it also will clear the **RecordsToRead** and **Filter** properties (not implemented yet).

Return Value: BYTE**Base Method:**

```

FileSynchronizationClass.Reset      PROCEDURE (BYTE Force=0)
OldStatus    BYTE
CODE
  OldStatus = STATUS (SELF.Target)
  IF OldStatus
    IF RECORDS (SELF.Target)=0
      RETURN Level:Benign
    END
    CLOSE (SELF.Target)
  END
  OPEN (SELF.Target,18)
  IF ERRORCODE () THEN RETURN Level:Fatal.
  EMPTY (SELF.Target)
  CLOSE (SELF.Target)
  IF OldStatus
    OPEN (SELF.Target,OldStatus)
    IF ERRORCODE () THEN RETURN Level:Fatal.
  END
  RETURN Level:Benign

```

Implementation:

Reset is called by the **Load** method to empty the target memory table prior to loading.

See Also: Load

ResetRange (reset key or file range to start)**ResetRange(),VIRTUAL**

The **ResetRange** virtual method is used to initialize the physical data source that is used to create and seed the target memory table. If the data source is filtered, **ResetRange** will set the data source to process in key sequence. Otherwise it will process the data source in file sequence.

Base Method:

```
FileSynchronizationClass.ResetRange      PROCEDURE ()
CODE
  IF NOT SELF.RangeKey &= NULL
    CASE SELF.LimitType
      OF 1 !No range but using Key order
        SET(SELF.RangeKey)
      OF 2 !Single Limit Type
        SELF.PrimeKey()
        SET(SELF.RangeKey,SELF.RangeKey)
      OF 3 !Range Limit Type
        SELF.PrimeKey()
        SET(SELF.RangeKey,SELF.RangeKey)
      ELSE
        !When 0 or other value
        !No Key, using Record Order
        SET(SELF.Original)
    END
  ELSE
    !When 0 or other value
    !No Key, using Record Order
    SET(SELF.Original)
  END
```

Implementation:

ResetRange is called by the **Load** method just prior to processing the physical table (data source). It is also called by the **Save** method just prior to processing (clearing) the physical table (data source). If the physical table is SQL based, the **ResetRange** derived method uses the PROP:SQL statement to query the **Original** table. In addition, the filter is still available to filter each record after the SQL statement is executed.

Example:

```
CachedManager:Customers.ResetRange PROCEDURE
CODE
PARENT.ResetRange
SELF.Original{PROP:SQL}='SELECT * FROM dbo.Customers WHERE Country=''Germany'''
```

See Also: Load, Save

Save (write memory table contents to source)**Save(),VIRTUAL**

The **Save** method is used to write the contents of the target memory table to the original data source, normally at program termination. By default, if the data source was not filtered, the original data source is cleared, and the entire content of the memory table is written to the “new” data source. As an alternative, you can use the optimized save to customize the save process. The method returns the appropriate error level, or no error (Level:Benign), if the save is successful.

Return Value: BYTE

Base Method:

```
FileSynchronizationClass.Save                                PROCEDURE ()
OldStatusTarget      BYTE
OldStatusOrigin      BYTE
RetVal              BYTE
lIndex              LONG
lRecordsToRead       LONG
TargetRecord         &GROUP, AUTO
OriginalRecord       &GROUP, AUTO
TransactionOk        BYTE
CODE
    !WIP
    RetVal = SELF.Open()
    IF RetVal<>Level:Benign THEN RETURN RetVal.
    TransactionOk = True
    IF SELF.UseLogout
        IF SELF.TransactionStart()<>Level:Benign
            TransactionOk = False
        END
    END
    IF TransactionOk = True
        IF SELF.OptimizeSave
            !It will delete first the DeletedRecord from the Original
            !It will later delete the Records that has a RecordPosition in the Original
            TransactionOk = SELF.OptimizedSave()
        ELSE
            !It will delete all the record on the Original file that match the filter and range limit
            TargetRecord  &= SELF.Target{PROP:Record}
            OriginalRecord &= SELF.Original{PROP:Record}
            CLEAR(OriginalRecord)
            SELF.ResetRange()
            LOOP
                NEXT(SELF.Original)
                CASE ERRORCODE()
                OF 0
                OF 33
                    BREAK
                ELSE
                    MESSAGE('Error Reading the Physical file. Error('&ERRORCODE()&') |
                        '&ERROR()&'|'&SELF.Original{PROP:Name})
                    BREAK
                END
                IF SELF.ValidateRecord()<>Level:Benign THEN CYCLE.
                !ASSIGN VALUES
                DELETE(SELF.Original)
                IF ERRORCODE()
                    TransactionOk = False
                    BREAK
                END
            END
            lRecordsToRead = RECORDS(SELF.Target)
            lIndex=1
            CLEAR(SELF.Target)
            SET(SELF.Target)
```

```

        LOOP
            IF lIndex>lRecordsToRead THEN BREAK.
            NEXT (SELF.Target)
            IF ERRORCODE()
                BREAK
            END
            SELF.PrimeSaveRecord()
            CLEAR(OriginalRecord)
            OriginalRecord = TargetRecord
            ADD(SELF.Original)
            IF ERRORCODE()
                TransactionOk = False
                BREAK
            END
            lIndex+=1
        END
    END
    !After that it will loop thru the Target file in record order
    !and add the record to the Original file
    !if any error happens then the TransactionOk will be set to False
END
IF SELF.UseLogout
    IF SELF.StartTransaction
        IF TransactionOk
            RetVal=SELF.TransactionCommit()
            IF RetVal=Level:Fatal
                TransactionOk = False
            END
        ELSE
            SELF.TransactionRollback()
        END
    END
END
SELF.Close()
IF TransactionOk = True
    RETURN Level:Benign
ELSE
    RETURN Level:Fatal
END

```

Implementation:

The **Save** method is called by the **Kill** method if the **SaveOnKill** property is TRUE and the **RecordsToRead** proper is zero (0), indicating that the target table was not filtered.

Example:

```

MemDriverInit      PROCEDURE          ! Initializes the MemDriver definition module
CODE
    MemAccess:Orders &= CachedManager:Orders
    CachedManager:Orders.Init(Orders, MEMAUX_Orders)
    CachedManager:Orders.SetRecordsToRead(100)
    CachedManager:Orders.SetOrderKey(MEMAUX_ORD:InvoiceNumberKey)
    CachedManager:Orders.SetRange(MEMAUX_ORD:InvoiceNumber, GLO:LowLimit, GLO:HighLimit)
    MemAccess:States &= CachedManager:States
    CachedManager:States.Init(States, MEMAUX_States)
    CachedManager:States.SaveOnKill = True
    CachedManager:States.SetFilter("MEMAUX_STA:StateCode = 'FL' ")

MemDriverKill      PROCEDURE          ! Kills the MemDriver definition module
CODE
    CachedManager:Orders.Kill()
    CachedManager:States.Kill() !Calls the Save method if SaveOnKill = TRUE

```

See Also: SaveOnKill, Kill, RecordsToRead

SetFilter (initialize record filter)**SetFilter**(*FilterString*),VIRTUAL**SetFilter** Specifies a filter used to process the target memory table*FilterString* A string constant, variable, or EQUATE containing a valid filter expression.

The **SetFilter** virtual method is used to specify a record filter to use when processing the data source that is used to seed the target memory table.

Base Method:

```
FileSynchronizationClass.SetFilter      PROCEDURE (STRING pRecordFilter)
CODE
    IF CLIP(pRecordFilter)
        IF NOT SELF.Filter &= NULL
            DISPOSE(SELF.Filter)
        END
        SELF.Filter &= NEW(STRING(LEN(CLIP(pRecordFilter))))
    ELSE
        IF NOT SELF.Filter &= NULL
            DISPOSE(SELF.Filter)
        END
    END
END
```

Implementation:

The **SetFilter** method should be called after the derived **Init** and **SetRecordsToRead** methods. The templates use a generated *MemDriverInit* procedure to generate the proper method call. **SetFilter** is used to set the **Filter** property of the **FileSynchronization** class.

Example:

```
MemDriverInit      PROCEDURE      ! Initializes the MemDriver definition module
CODE
    MemAccess:Orders &= CachedManager:Orders
    CachedManager:Orders.Init(Orders, MEMAUX_Orders)
    CachedManager:Orders.SetRecordsToRead(100)
    CachedManager:Orders.SetFilter('MEMAUX_ORD:OrderDate = TODAY()')
    CachedManager:Orders.SetOrderKey(MEMAUX_ORD:InvoiceNumberKey)
    CachedManager:Orders.SetRange(MEMAUX_ORD:InvoiceNumber, GLO:LowLimit, GLO:HighLimit)
```

See Also: Filter

SetOrderKey (set to key order sequence)**SetOrderKey(*KeyLabel*),VIRTUAL****SetOrderKey** Set the active sort order used to process the data source and target memory table.*KeyLabel* A label of a KEY used to specify the processing sequence.**SetOrderKey** is a virtual method used to set the active sort order used to process the original data source contents loaded into the target memory table. It is only needed when a key sequence is active when processing the data source.**Base Method:**

```
FileSynchronizationClass.SetOrderKey  PROCEDURE (KEY parKey)
CODE
    SELF.RangeKey &= parKey
    SELF.LimitType = 1
```

Implementation:

The **SetOrderKey** method should be called after the derived **Init** and **SetRecordsToRead** methods. The templates use a generated *MemDriverInit* procedure to generate the proper method call. **SetOrderKey** is used to set the **RangeKey** property of the **FileSynchronization** class, prior to calling the **Load** method.

Example:

```
MemDriverInit      PROCEDURE      ! Initializes the MemDriver definition module
CODE
    MemAccess:Orders &= CachedManager:Orders
    CachedManager:Orders.Init(Orders, MEMAUX_Orders)
    CachedManager:Orders.SetRecordsToRead(100)
    CachedManager:Orders.SetFilter('MEMAUX_ORD:OrderDate = TODAY()')
    CachedManager:Orders.SetOrderKey(MEMAUX_ORD:InvoiceNumberKey)
    CachedManager:Orders.SetRange(MEMAUX_ORD:InvoiceNumber, GLO:LowLimit, GLO:HighLimit)
```

! Load method is called after this**See Also:** **SetRange**, **SetResetRange**

SetRange (set to range limit sequence)**SetRange(<Field>, <LowValue>, <HighValue>),VIRTUAL**

SetRange	Set the active key or file order to process the original data source
<i>Field</i>	The label of a column to be used to range limit the contents of the target memory table
<i>LowValue</i>	A string constant, variable, or EQUATE that defines the lower range of valid records to include in the memory table.
<i>HighValue</i>	A string constant, variable, or EQUATE that defines the upper range of valid records to include in the memory table.

SetRange is a virtual method used to set the active range limit used to process the original data source contents loaded into the target memory table. **SetRange** can also designate that *no* range limits are used to process the data source. This method is overloaded.

Base Overloaded Methods:

```
FileSynchronizationClass.SetRange      PROCEDURE ()
CODE
    SELF.LimitType = 0

FileSynchronizationClass.SetRange      PROCEDURE (*? Field,*? Limit)
CODE
    SELF.FreeElement &= Field
    SELF.LimitType = 2
    SELF.LowLimit &= Limit

FileSynchronizationClass.SetRange      PROCEDURE (*? Field,*? Low,*? High)
CODE
    SELF.FreeElement &= Field
    SELF.LimitType = 3
    SELF.LowLimit &= Low
    SELF.HighLimit &= High
```

Implementation:

The **SetRange** method should be called just prior to the **Load** method. The templates use a generated *MemDriverInit* procedure to generate the proper method call. **SetRange** is used to set the **LimitType**, and optional **FreeElement**, **LowLimit** and **HighLimit** properties.

Example:

```
MemDriverInit      PROCEDURE      ! Initializes the MemDriver definition module
CODE
    MemAccess:Orders &= CachedManager:Orders
    CachedManager:Orders.Init(Orders, MEMAUX_Orders)
    CachedManager:Orders.SetRecordsToRead(100)
    CachedManager:Orders.SetFilter('MEMAUX_ORD:OrderDate = TODAY()')
    CachedManager:Orders.SetOrderKey(MEMAUX_ORD:InvoiceNumberKey)
    CachedManager:Orders.SetRange(MEMAUX_ORD:InvoiceNumber,GLO:LowLimit,GLO:HighLimit)
    ! Load method is called after this
```

See Also: LimitType, FreeElement, LowLimit, HighLimit

SetRecordPositionField (set record position)**SetRecordPositionField**(*TargetTableField*)**SetRecordPositionField** Set data source table KEY or FILE POSITION.*TargetTableField* A STRING variable used to hold the KEY or FILE POSITION value.

The **SetRecordPositionField** virtual method is used to identify the target variable that is used to store the KEY or FILE position information of the original data source used to process the memory table.

POSITION returns a STRING that identifies a record's unique position within the key or file sequence. POSITION returns the position of the last record accessed in the file. The POSITION procedure is used with RESET to temporarily suspend and resume sequential processing.

Base Method:

```
FileSynchronizationClass.SetRecordPositionField      PROCEDURE(*? FieldFromTargetFile)
CODE
  SELF.RecordPositionField &= FieldFromTargetFile
  SELF.OptimizeSave              = True
```

Implementation:

This method is not implemented, but can be used by the developer to store position information to use in the **OptimizedSave** method. **SetRecordPositionField** also sets the **OptimizeSave** property to TRUE.

See Also: RecordPositionField, OptimizeSave

SetRecordsToRead (initialize records to process)

SetRecordsToRead(*RecordsToRead*),VIRTUAL

SetRecordsToRead Set the maximum number of records to load into the target memory table.

RecordsToRead A LONG constant or variable that specifies the maximum number of records to process into the target memory table.

The **SetRecordsToRead** virtual method is used to set the maximum number of records to load into the target memory table. If *RecordsToRead* is zero (0), all records read from the original data source will be processed into the target memory table.

Base Method:

```
FileSynchronizationClass.SetRecordsToRead     PROCEDURE (LONG pRecordsToRead)
CODE
SELF.RecordsToRead     = pRecordsToRead
```

Implementation:

SetRecordsToRead is used to set the *RecordsToRead* property, which is in turn used by the Load and Save methods.

See Also: *RecordsToRead*

SetUseSQL (set the UseSQL property)

SetUseSQL(*flag*)

The **SetUseSQL** method is used to toggle the **UseSQL** property if the **Original** data source is SQL based.

Return Value: BYTE

Base Method:

```
FileSynchronizationClass.SetUseSQL     PROCEDURE (BYTE pValue)
CODE
IF NOT SELF.Original&=NULL
  IF SELF.Original{PROP:SQLDriver} THEN
    SELF.UseSQL = pValue
  ELSE
    SELF.UseSQL = False
  END
ELSE
  SELF.UseSQL = False
END
```

Implementation:

SetUseSQL is not implemented by the supporting templates, but is a tool for the developer to set the *UseSQL* property as needed.

See Also: *Original*, *UseSQL*

SyncDelete(delete record in physical table)**SyncDelete**(< *filelabel* >)**SyncDelete** Delete a record in the related physical table

filelabel The label of the file opened on the current active thread. If omitted, the file to be updated is assumed to be not threaded.

The **SyncDelete** method is used to delete a record from the physical file as the current IMDD record is deleted. As the target record of the IMDD table is deleted, the original record is also deleted in the physical table.

filelabel is used when the IMDD file is threaded and the current buffer is different from the global one.

SyncInsert returns *Level:Benign* (0) if the action executes with out errors and *Level:Fatal* (3) if any error occurs, use the **GetSyncError** and **GetSyncErrorcode** to retrieve the specific error information.

Return Data Type: BYTE

Implementation: **SyncDelete** is called from the templates in the target FileManager **PreDelete** method.

Example:

```
Hide:Access:Customers.PreDelete PROCEDURE(*CSTRING ErrCode,*CSTRING ErrMsg)
```

```
ReturnValue          BYTE,AUTO
```

```
Buffer LIKE (CUS:RECORD)
CODE
PUSHERRORS()
IF SELF.SavePreviousBuffer AND NOT(SELF.PreviousBuffer &= NULL)
    Buffer = SELF.PreviousBuffer
END
IF MemAccess:Customers.SyncDelete(Customers)<>Level:Benign
    ErrCode=90
    ErrMsg='The file could not be synchronized.('|
            &MemAccess:Customers.GetSyncErrorcode()&') '|
            &MemAccess:Customers.GetSyncError()
    POPERRORS()
    ReturnValue = False
    RETURN ReturnValue
END
ReturnValue = PARENT.PreDelete(ErrCode,ErrMsg)
POPERRORS()
RETURN ReturnValue
```

See Also: **GetSyncError**, **GetSyncErrorcode**

SyncInsert(add record to physical table)**SyncInsert**(< filelabel >)**SyncInsert** Add a record to the related physical table.

filelabel The label of the file opened on the current active thread. If omitted, the file to be updated is assumed to be not threaded.

The **SyncInsert** method is used to insert into the physical file the current IMDD record. As a record is added to the IMDD table, the record is also added to the physical table.

filelabel is used when the IMDD file is threaded and the current buffer is different from the global buffer.

SyncInsert returns *Level:Benign* (0) if the action executes with out errors and *Level:Fatal* (3) if any error occurs, use the **GetSyncError** and **GetSyncErrorcode** to retrieve the specific error information.

Return Data Type: BYTE

Implementation: **SyncInsert** is called from the templates in the target FileManager **PreInsert** method.

Example:

```
Hide:Access:Customers.PreInsert PROCEDURE(SIGNED OpCode,UNSIGNED |
                                     AddLen,*CSTRING ErrCode,*CSTRING ErrMsg)
```

```
ReturnValue             BYTE,AUTO
```

```
Buffer LIKE (CUS:RECORD)
CODE
PUSHERRORS()
IF SELF.SavePreviousBuffer AND NOT(SELF.PreviousBuffer &= NULL)
    Buffer = SELF.PreviousBuffer
END
IF MemAccess:Customers.SyncInsert(Customers)<>Level:Benign
    ErrCode=90
    ErrMsg ='The file could not be synchronized.'&|
            (' &MemAccess:Customers.GetSyncErrorCode() &' )'|
            &MemAccess:Customers.GetSyncError()
    POPERRORS()
    ReturnValue = False
    RETURN ReturnValue
END
ReturnValue = PARENT.PreInsert(OpCode,AddLen,ErrCode,ErrMsg)
POPERRORS()
RETURN ReturnValue
```

See Also: **GetSyncError**, **GetSyncErrorcode**

SyncRefresh(refresh IMDD record)**SyncRefresh**(< filelabel >)**SyncRefresh** Update the active memory table record from the contents of the physical (original) table.

filelabel The label of the file opened on the current active thread. If omitted, the file to be updated is assumed to be not threaded.

The **SyncRefresh** method is used to synchronize the contents of the current memory table record with the associated physical table record.

Return Data Type: BYTE

Implementation: This method is currently not implemented in the templates. You will need to call this method through an appropriate embed point.

Example:

```

IF SELF.Request = ChangeRecord
  IF MemAccess:Customers.SyncRefresh(Customers)<>Level:Benign
    MESSAGE('The file could not be synchronized.('|
      &MemAccess:Customers.GetSyncErrorCode()&') '|
      &MemAccess:Customers.GetSyncError())
  ELSE
    MESSAGE('Record was synchronized with server')
  END
END
SELF.Open(QuickWindow)           ! Open window
Do DefineListboxStyle

```

See Also: GetSyncError, GetSyncErrorCode

SyncUpdate(update record to physical table)**SyncUpdate**(< *filelabel* >)**SyncUpdate** Update a record to the related physical table

filelabel The label of the file opened on the current active thread. If omitted, the file to be updated is assumed to be not threaded.

The **SyncUpdate** method is used to write back into the physical file the current IMDD record. As a record of the IMDD table is changed, the related record is also changed in the physical table.

filelabel is used when the IMDD file is threaded and the current buffer is different from the global one.

SyncInsert returns *Level:Benign* (0) if the action executes with out errors and *Level:Fatal* (3) if any error occurs, use the **GetSyncError** and **GetSyncErrorcode** to retrieve the specific error information.

Return Data Type: BYTE

Implementation: **SyncUpdate** is called from the templates in the target FileManager **PreUpdate** method.

Example:

```
Hide:Access:Customers.PreUpdate PROCEDURE (LONG Pntr, UNSIGNED |
                                     PutLen, *CSTRING ErrCode, *CSTRING ErrMsg)
```

```
ReturnValue                BYTE, AUTO
```

```
Buffer LIKE (CUS:RECORD)
CODE
PUSHERRORS ()
IF SELF.SavePreviousBuffer AND NOT (SELF.PreviousBuffer &= NULL)
    Buffer = SELF.PreviousBuffer
END
IF MemAccess:Customers.SyncUpdate (Customers) <> Level:Benign
    ErrCode=90
    ErrMsg ='The file could not be synchronized. (' |
            & MemAccess:Customers.GetSyncErrorcode () & ') ' |
            & MemAccess:Customers.GetSyncError ()
    POPERRORS ()
    ReturnValue = False
    RETURN ReturnValue
END
ReturnValue = PARENT.PreUpdate (Pntr, PutLen, ErrCode, ErrMsg)
POPERRORS ()
RETURN ReturnValue
```

See Also: **GetSyncError**, **GetSyncErrorcode**

TransactionCommit (Commit after save)**TransactionCommit(),VIRTUAL**

The **TransactionCommit** virtual method is used to complete a transaction-framing event when writing to the original data source used to process the target memory table. It simply issues a COMMIT, and returns *Level:Benign* (0) if successful, or *Level:Fatal* (3) if not.

Return Value: BYTE

Base Method:

```
FileSynchronizationClass.TransactionCommit      PROCEDURE ()
CODE
    COMMIT ()
    IF ERRORCODE ()
        RETURN Level:Fatal
    ELSE
        RETURN Level:Benign
    END
```

Implementation:

TransactionCommit is used by the **Save** method, and returns an *Level:Fatal* error level if unsuccessful.

See Also: TransactionRollback, TransactionStart

TransactionRollback(Rollback after save)**TransactionRollback(),VIRTUAL**

The **TransactionRollback** virtual method is used to rollback a transaction-framing event if unsuccessful for any reason. The method simply executes a ROLLBACK statement.

Base Method:

```
FileSynchronizationClass.TransactionRollback    PROCEDURE ()
CODE
    ROLLBACK ()
```

Implementation:

TransactionRollback is called from the Save method if a **TransactionCommit** method was not successful.

Example:

```
IF SELF.UseLogout
IF SELF.StartTransaction
    IF TransactionOk
        RetVal=SELF.TransactionCommit()
        IF RetVal=Level:Fatal
            TransactionOk = False
        END
    ELSE
        SELF.TransactionRollback()
    END
END
END
END
```

See Also: TransactionStart, TransactionCommit, UseLogout, StartTransaction

TransactionStart (begin Logout before Save)**TransactionStart(),VIRTUAL**

The **TransactionStart** virtual method is used to begin transaction framing for the original data source when the contents of the target memory table are written back. **TransactionStart** simply attempts a LOGOUT to the data source specified in the **Original** property, and returns the appropriate error level if unsuccessful.

Return Value: BYTE

Base Method:

```
FileSynchronizationClass.TransactionStart      PROCEDURE ()
CODE
    IF SELF.StartTransaction
        LOGOUT(12,SELF.Original)
        IF NOT ERRORCODE()
            RETURN Level:Benign
        ELSE
            MESSAGE('Error Reading the Physical file. Error('&ERRORCODE()&') |
                '&ERROR() &'| '&SELF.Original{PROP:Name})
            RETURN Level:Fatal
        END
    ELSE
        SELF.Original{PROP:Logout}=True
    END
    RETURN Level:Benign
```

Implementation:

TransactionStart is called by the **Save** method to begin the transaction-framing event for the original data source. If any other error level is returned except for *Level:Benign* (indicating that the LOGOUT was unsuccessful), the save is aborted, and *Level:Fatal* is returned by the **Save** method.

See Also: **Save, UseLogout**

ValidateRecord (evaluate filter during load and save)**ValidateRecord(),VIRTUAL**

The **ValidateRecord** virtual method is used to determine whether or not to include the current record in the active output. **ValidateRecord** provides a filtering mechanism during writes to the target memory table (during **Load**) or the original data source (during **Save**). Any record filter and range limit applied is tested by this method. If the record is successfully validated, an error level of *Level:Benign* (0) is returned. If the target memory table is using range limiting, and the range limit fails, an error level of *Level:Fatal* (3) is posted, and **ValidateRecord** is terminated. If the target memory table is using record filtering, and the record fails, an error level of *Level:Notify* (5) is posted, and **ValidateRecord** continues to process the remaining records.

Return Value: BYTE

Base Method:

```
FileSynchronizationClass.ValidateRecord          PROCEDURE ()
FilterOk      BYTE
RetVal        BYTE
CODE
    RetVal = Level:Benign
    IF NOT SELF.Filter &= NULL
        FilterOk = EVALUATE(CLIP(SELF.Filter))
        IF ERRORCODE()
            MESSAGE(ERRORCODE() & '-' & ERROR(), 'On ValidateRecord')
            RetVal = Level:Benign
        ELSE
            IF FilterOk
                RetVal = Level:Benign
            ELSE
                RetVal = Level:Notify
            END
        END
    END
    IF RetVal = Level:Benign
        IF NOT SELF.RangeKey &= NULL
            CASE SELF.LimitType
                OF 2 !Single Limit Type
                    IF SELF.FreeElement <> SELF.LowLimit THEN RetVal = Level:Fatal.
                OF 3 !Range Limit Type
                    IF NOT (SELF.FreeElement >= SELF.LowLimit AND SELF.FreeElement <= SELF.HighLimit)
                        RetVal = Level:Fatal
                    END
                END
            END
        END
    END
    RETURN RetVal
```

Implementation:

ValidateRecord is called in both **Save** and **Load** methods to validate the contents of each record read. In the **Load** method, it is evaluating the contents of the original data source. In the **Save** method, it is evaluating the contents of the target memory table record in memory.

See Also: Save, Load

Index:

Assertion Message.....	29	PrimeRecord (prime fields on load)	51
BackupOnSave (backup physical file on Save)	33	PrimeSaveRecord (prime fields on save)	51
Caching Support	16, 17, 24, 28	RangeKey(reference to data source range limit key) ...	35
Close (close data source and target tables)	39	RecordPositionField (key or table position)	36
COPY	12	RecordsToRead (records to process from data source)	36
Data Types		registering the driver	7
Supported	8	RENAME.....	12
Design Considerations	13	Reset (clear data source contents).....	52
Destruct (dispose the filter)	39	ResetRange (reset key or file range to start).....	53
Dictionary Table	21	Save (write memory table contents to source)	54, 56
DLL considerations	30	SaveOnKill (save changes on exit to data source).....	36
driver string.....	9	SetFilter (initialize record filter)	56
ERRORCODE 47	13	SetOrderKey (set to key order sequence)	57
File Specifications/Maximums	8	SetRange (set to range limit sequence)	58
FileSynchronization Class		SetRecordPositionField (set record position)	59
Properties.....	33	SetRecordsToRead (initialize records to process)	60
Filter (record filter expression)	33	SetUseSQL (set the UseSQL property).....	60
FreeElement (comparison value for range limits)	33	SHARE	12
GenerateBackUp (create data source backup table)....	40	Silent (silent mode flag)	36
Generated Table	18	SQL design considerations	13
HighLimit (range of values upper limit)	34	StartTransaction (OK to start transaction processing)..	37
Init (initialize file synchronization)	44	Supported Commands and Attributes.....	10
IPDriver In-Memory Caching Support	26	Target (target Memory table file reference)	37
Kill (save changes and dispose the filter)	45	Template Notes.....	29
LimitType (type of range limit process)	34	template support	16
Load (write source data to memory table)	46	THREADEDCONTENT	9
LoadSuccess (memory table load was successful)	34	transaction framing	
LowLimit (range of values lower limit)	34	support for	13
OPEN	12	TransactionCommit (Commit after save).....	65
Open (open data source and target table)	48	TransactionRollback(Rollback after save)	65
OptimizedSave (customized save).....	50	TransactionStart (begin Logout before Save).....	66
OptimizeSave (custom save option)	35	UseLogout (transaction processing enabled)	37
Original (data source table file reference).....	35	UseSQL (SQL data source).....	37
POSITION(file)	12	using the In-Memory Driver	7
PrimaryKey (reference to data source primary key)	35	ValidateRecord (evaluate filter during load and save)..	67
PrimeKey (initialize key element)	50		