

Dynamic File Driver Reference

COPYRIGHT 1994- 2005 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

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:

Dynamic File Driver (DFD)	5
Overview	5
Setup	5
Working with Dynamic files - conceptual flow	6
Getting started	7
Dynamic File Interface – Overview	9
Creating and Destroying a Dynamic File	9
Defining a file using PROP:SQL	9
Defining a File Manually	11
Interface Specifics	13
Using Dynamic File Support	13
Language Support	13
FIXFORMAT (fix a dynamic file)	13
UNFIXFORMAT (unfix a dynamic file)	13
File Structure Properties	14
Defining the record structure – general rules	16
Field (Column) Properties	16
Defining Keys	17
Defining Memos and Blobs	18
DFD Templates	19
Dynamic File Driver Global Extension	20
“Cache Files on this thread” extension template	23
Requirements	23
Shipping Examples	25
DynFile Class	27
Overview	27
DynFile Class Source Files	27
DynFile Properties	27
DynFile Methods	28
AddField (add field to a Dynamic File Structure)	28
AddFieldToKey (add field to dynamic file Key definition)	29
AddKey (add key to a Dynamic File Structure)	30
AddMemo (add memo to a Dynamic File Structure)	31
CacheFile(load table into in-memory dynamic file)	32
CreateFromFile (build Dynamic File Structure from existing file)	33
CreateFromSQL(create structure from SQL result set)	34
CreateKeyComponents (create key elements of a dynamic file key)	35
CreateStruct (build a dynamic file structure)	36
FillFrom (copy data to a Dynamic File Structure)	37
FillTo (copy dynamic file contents to table)	39
GetCreate (get value of Create property)	42
GetDriver (get value of Driver property)	42
GetEncrypt (get value of Encrypt property)	42
GetField (get field contents)	43
GetFieldName (get label name of field)	44
GetFieldValue (move field value to a reference)	45
GetFileRef (get reference to a dynamic file)	46
GetKeyRef (get reference to a dynamic file’s key)	47
GetName (get value of Name property)	48
GetOEM (get value of OEM property)	48
GetOwner (get value of Owner property)	48
GetPrefix (get value of Prefix property)	49

GetReclaim (get value of Reclaim property)	49
GetCreatedFromSQL (get value of CreatedFromSQL property)	50
LoadFile (load file with data source).....	51
RemoveField (remove a field from a Dynamic File)	52
RemoveKey (remove a key from a Dynamic File).....	53
RemoveKeyField (remove a key field from a Dynamic File)	54
ResetAll (clear all dynamic file values).....	55
SaveFile (write dynamic file contents to data source).....	56
SetCreate (set dynamic file CREATE attribute)	57
SetDriver (set dynamic file DRIVER attribute).....	58
SetEncrypt (set dynamic file ENCRYPT attribute)	59
SetFieldValue (set default value of field).....	60
SetName (set dynamic file NAME attribute).....	61
SetOEM (set OEM attribute to dynamic file)	61
SetOwner (set owner property to Dynamic File)	62
SetPrefix (set prefix value)	63
SetReclaim (set Reclaim property to Dynamic File).....	64
Trace (send debug information)	65
UnfixFormat (close and reset Dynamic File Reference)	66
ViewFormat (read format of active dynamic file).....	67
Annotated Examples	69
Create a simple dynamic ASCII file	69
Load a dynamic SQL or ISAM structure into a Virtual List Box (VLB)	75
Extended ERRORCODES	81
Structural Errors - Snumber codes	82
File Driver Specific Errors - Dnumber Codes.....	84
Index:	85

Dynamic File Driver (DFD)

Overview

The Dynamic File Driver (DFD) provides developers with an interface to define file structures, on-demand, at runtime. File structures can be defined for any of the supported file systems (ISAM or SQL) using standard property syntax. If the dynamic file is defined using an SQL driver then you can use PROP:SQL and issue either a SELECT statement or a call to a server-side Stored Procedure that returns a result set to automatically define a new file structure.

Here are just a few of the things that can be accomplished using the DFD:

- Dynamically create a FILE structure to match the result set from any SQL SELECT command or from a Stored Procedure.
- Dynamically create In-Memory driver FILE structures for use as "cached recordsets" from any data source (ISAM or SQL).
- Create temporary tables that match a user-defined query
- Remove strict binding to the data dictionary.
- Create and process tables that are not defined in the application's dictionary.
- Change Database drivers at runtime using Prop:Driver

Setup

Setup is easy! Just register *DynaDrv.tpl* in the **Template Registry**. You do not need to register the Dynamic File Driver (or *DynaDriver*) in the Database Drivers Registry.

Working with Dynamic files - conceptual flow

When you work with Dynamic files you'll follow these basic steps:

1. Declare reference variables in the Data section:

```
AFile &FILE  
AKey &KEY
```

2. The assignment of necessary properties at runtime in the CODE section:

```
AFile &= NEW(FILE)  
AFile{PROP:Driver} = 'TopSpeed'  
AFile{PROP:Create} = TRUE  
AFile{PROP:Name, 1} = 'Field1'  
AFile{PROP:Type} = 'LONG'  
AFile{PROP:Dim} = 3  
AKey &= AFile {PROP:Key, 1}  
AKey{PROP:Type} = 'KEY'  
AKey{PROP:Primary} = TRUE  
AKey{PROP:Field} = 1
```

3. A call to the FIXFORMAT() to initialize and create the dynamic structure.

```
FIXFORMAT(AFile)
```

And a call to CREATE(), OPEN() and later DISPOSE()

```
CREATE (AFile)  
OPEN(AFile)
```

```
! do cleanup processing  
DISPOSE (AFile)
```

Getting started

The Dynamic File Driver has powerful functionality that helps you to create file structures at runtime. To become familiar with the syntax for defining dynamic file structures read the *Interface Overview* section.

The **DynFile** class was designed to simplify the task of creating dynamic file structures. The class encapsulates all of the functions needed to define dynamic files and its methods can be used to simplify and reduce the code needed to create a dynamic file to a few lines. The class is documented in the *DynFile Class* section.

The **DynaFile** template may provide all the functionality that you need. Please refer to the *DFD Templates* section.

Some of the examples found in `..\Examples\DFD\<example>` are documented in this manual in the *Annotated Examples* section.

We suggest that you skim through this manual and then run some of the DFD examples. Examine the code used in the examples and then revisit this manual when necessary.

Dynamic File Interface – Overview

This section reviews language fundamentals that you will use often when working with dynamic files.

Creating and Destroying a Dynamic File

The NEW(File) statement is used to create a file, which can later be disposed of using DISPOSE (File).

For Example:

```
AFile &FILE
CODE
AFile &= NEW (FILE)
...
DISPOSE (AFile)
```

Defining a file using PROP:SQL

If the destination driver of your dynamic file is an SQL driver, it is possible to use **PROP:SQL** to create the file structure. You will still need to use **FIXFORMAT** to set the format. You can use either a **SELECT** statement, or a call to a stored procedure, that returns a result set to define the file structure.

Example:

```
AFile &FILE
CODE
AFile &= NEW (FILE)
AFile {PROP:Driver} = 'MSSQL'
AFile {PROP:Owner} = 'Connection String'
AFile {PROP:SQL} = 'SELECT * FROM MyTable'
FIXFORMAT (AFile)

UNFIXFORMAT (AFile)
AFile {PROP:SQL} = 'CALL MyStoredProcedure(3)'
FIXFORMAT (AFile) !Now the file will match the columns
                  !returned by the stored procedure

DISPOSE (AFile)
```

Tip

It is valid to change PROP:Driver after using PROP:SQL. You can use this to create, for example, an In-Memory file that can be used to hold the results of an SQL statement.

Example:

```

IMDDFile &FILE
ODBCFile &FILE
bGroup &GROUP
oGroup &GROUP

CODE
  IMDDFile &= NEW(FILE)
  IMDDFile {PROP:Driver} = 'ODBC'
  IMDDFile {PROP:Owner} = 'MyDataSource'
  IMDDFile {PROP:TextAsString} = 400
  IMDDFile {PROP:ImageAsString} = 400
  IMDDFile {PROP:SQL} = 'SELECT * FROM MyTable'
  IMDDFile {PROP:Driver} = 'In-Memory'
  IMDDFile {PROP:Create} = TRUE
  IMDDFile {PROP:Name} = 'Results'
  FIXFORMAT (IMDDFile)
  bGroup &= IMDDFile{PROP:Record}
  CREATE (IMDDFile)
  OPEN (IMDDFile)

  ODBCFile &= NEW(FILE)
  ODBCFile{PROP:Driver} = 'ODBC'
  ODBCFile{PROP:Owner} = 'MyDataSource'
  ODBCFile{PROP:SQL} = 'SELECT * FROM MyTable'
  FIXFORMAT (ODBCFile)
  oGroup &= ODBCFile{PROP:Record}

LOOP
  NEXT (ODBCFile)
  IF ERRORCODE () THEN BREAK.
  bGroup = oGroup
  ADD (IMDDFile)
END
! do processing
CLOSE (IMDDFile)
CLOSE (ODBCFile)
DESTROY (IMDDFile)
DESTROY (ODBCFile)

```

Defining a File Manually

To create a dynamic file definition, you can use the supplied DynFile Class or standard file property syntax. When you call **NEW(FILE)**, the structure is considered dynamic. The file structure is changed using property syntax without any error checking. Error checking is actually done when the **FIXFORMAT(File)** statement is called. Once **FIXFORMAT** has been successfully called, any further attempts to change the structure (beyond that which is supported by all drivers) will result in an "ERRORCODE 80 – Unsupported File Driver Function". To change a file format after **FIXFORMAT** has been called, **UNFIXFORMAT** must be called.

Example:

```
AFile &FILE
AKey &KEY
CODE
AFile &= NEW (FILE)
AFile{PROP:Driver} = 'TopSpeed'
AFile{PROP:Create} = TRUE
AFile{PROP:Name, 1} = 'Field1'
AFile{PROP:Type} = 'LONG'
AFile{PROP:Dim} = 3
AKey &= AFile {PROP:Key, 1}
AKey{PROP:Type} = 'KEY'
AKey{PROP:Primary} = TRUE
AKey{PROP:Field} = 1
FIXFORMAT(AFile)
CREATE(AFile)
DISPOSE(AFile)
```


Interface Specifics

Using Dynamic File Support

Any program that uses dynamic file system must link in **C60DFX%L%.LIB**.

Language Support

The Dynamic File Driver provides the following language statements:

FIXFORMAT (fix a dynamic file)

FIXFORMAT(*file*)

file The label of a FILE structure, which must be a reference variable.

FIXFORMAT fixes a dynamic file so that it can be used like any other file.

If you pass a file that was not created using NEW(file), **FIXFORMAT** posts the following error codes that can be trapped by the ERRORCODE function:

Errorcode	Equate	Reason
80	<i>NoDriverSupport</i>	File passed was not created with NEW(<i>file</i>)
47	<i>InvalidFileErr</i>	Structure is invalid

If Error code 47 is posted, the FILEERRORCODE function can be used to return extended information. See the *ErrorCode 47 Extended information* topic for more information.

UNFIXFORMAT (unfix a dynamic file)

UNFIXFORMAT(*file*)

file The label of a FILE structure, which must be a reference variable.

UNFIXFORMAT changes a dynamic file that has previously been fixed into an unfixed state, thus allowing you to further change the format.

UNFIXFORMAT will set ERRORCODE to NoDriverSupport if you pass a file that was not created with NEW(file)

If you pass a file that was not created using NEW(file), **UNFIXFORMAT** posts the following error codes that can be trapped by the ERRORCODE function:

Errorcode	Equate	Reason
80	<i>NoDriverSupport</i>	File passed was not created with NEW(<i>file</i>)
47	<i>InvalidFileErr</i>	Structure is invalid

If Error code 47 is posted, the FILEERRORCODE statement can be used to return extended information. See the *ErrorCode 47 Extended information* topic for more information.

File Structure Properties

The following properties are used to set the file structure:

File {PROP:Driver} = string

This property will not work in Local linked applications. If you attempt to set this property in a Local linked application, ERRORCODE will be set to NoDrvFunc(80). You need to use PROP:FileDriver in Local linked applications. (See *also* SetDriver)

If both PROP:Driver and PROP:FileDriver are specified, PROP:FileDriver will be used.

File {PROP:DriverString} = string

File {PROP:Owner} = string

File {PROP:Create} = 0 | 1

File {PROP:Owner} = string

File {PROP:Reclaim} = 0 | 1

File {PROP:Encrypt} = 0 | 1

File {PROP:OEM} = 0 | 1

File {PROP:Name} = string

File {PROP:FileDriver} = File

You can use these properties to set the file driver for the file instead of using PROP:Driver. This is the only way to set the driver for *local linked applications*.

If both PROP:Driver and PROP:FileDriver are specified, PROP:FileDriver will be used.

You cannot set PROP:Thread. Dynamically created files are not threaded. If you want to “thread” a dynamically created file, then you need to add the **THREAD** attribute on the file reference and create the file definition on each thread, or cache the data into a THREADED In-Memory table.

The following properties can be read during file creation:

Value = File{PROP:Fields}

Returns the field ID for the maximum field defined so far.

Value = File{PROP:Keys}

Returns the key ID for the maximum key defined so far.

Value = File{PROP:Blobs}

Returns the blob ID for the maximum blob defined so far.

Value = File{PROP:Memos}

Returns the memo ID for the maximum memo defined so far.

Once PROP:Driver or PROP:FileDriver has been set, you can also read the following file driver properties:

Value = File{PROP:SQLDriver}

PROP:SQLDriver returns TRUE if the file drive is an SQL based file driver and therefore supports the PROP:SQL statements and other SQL only features.

Value = File{PROP:SupportsOp}

PROP:SupportsOp is an array property that returns TRUE if the driver supports the file driver function. This list of operation codes is defined in equates.clw

Value = File{PROP:SupportsType}

PROP:SupportsType is an array property that returns TRUE if the driver supports the data type. This list of operation data types is defined in equates.clw

Value = File{PROP:DriverLogoutAlias}

PROP:DriverLogoutAlias returns true if the file driver logs out a file and its aliases when you logout the file. The TopSpeed driver returns true for this op. All other drivers return false.

Defining the record structure – general rules

1. You do not need to specify that you are adding a field to a file structure. You simply have to assign any attribute to a field for it to exist.
2. You do not have to create fields in any specific order. It is valid to define field five, then field one. However, FIXFORMAT will return an error if there are any gaps in the field numbering.
3. Before FIXFORMAT is called File{PROP:Fields} will return the maximum field number currently defined.
4. When defining a file structure using property syntax many properties take values 0 or 1. You can also use an empty string instead of 0.
5. If FIXFORMAT has not been called on a file, all of properties return FALSE and ERRORCODE is set to 80.

Field (Column) Properties

The properties used to specify fields within the record structure are as follows:

File{PROP:Label, n} = string
File{PROP:Name, n} = string

If PROP:Name is specified for a field, and PROP:Label is not specified, then the system will create a label for the field based on the name.

File{PROP:Type, n} = string

The string assigned to this property must be the name of a standard Clarion data type. The case of the string does not matter.

If you specify an invalid string ERRORCODE will be set to TypeDescErr (75).

File{PROP:Over, n} = number < n
File{PROP:Dim, n} = number

You cannot define multiple dimensions.

If number < 0, ERRORCODE () will be set to InvalidFileErr (47)

File{PROP:Places, n} = number

Only valid if PROP:Type is DECIMAL or PDECIMAL

If number < 0, ERRORCODE() will be set to InvalidFileErr (47)

File{PROP:Size, n} = number

Only valid if PROP:Type is DECIMAL, PDECIMAL, STRING, CSTRING or PSTRING.

If number < 0 or number > 4,190,208, ERRORCODE() will be set to InvalidFileErr(47).

If you specify an even number for the size of a PDECIMAL or DECIMAL field, it will be converted to the next largest number. This is due to the way these fields are stored in memory.

File{PROP:Field, n} = "

Use this form to **delete** a field from the dynamic file definition.

Defining Keys

To define a key, use `File{PROP:Key, n}` to obtain a reference to a key and then set the attributes of the key using that key reference.

You are limited to 255 keys in a file definition. Attempting to get a key reference for a key greater than 255 will return a NULL reference.

You are limited to 255 components in a key. Attempting to set a property of a component outside this range will return an error of **NoDrvFunc(80)**

The properties used to define a key are:

Key{PROP:Label} = string

Key{PROP:Name} = string

If `PROP:Name` is specified for a key, and `PROP:Label` is not specified, then the system will create a label for the key based on the name.

Key{PROP:Dup} = 0 | 1

If not specified, then 0 is assumed

Key{PROP:Primary} = 0 | 1

If not specified, then 0 is assumed

Key{PROP:Type} = 'KEY' or 'INDEX'

If not specified, then KEY is assumed

Key{PROP:NoCase} = 0 | 1

If not specified, then 0 is assumed

Key{PROP:Opt} = 0 | 1

If not specified, then 0 is assumed

Key{PROP:Field, n} = number

If number ≤ 0 , `ERRORCODE()` will be set to `InvalidFileErr(47)`

Key{PROP:Ascending, n} = 0 | 1

If not specified, then 1 is assumed

To delete a key definition, use `File{PROP:Key, n} = "`

To delete a key component, use `Key{PROP:Component, n} = "`

The following properties can be read during file creation:

Value = Key{PROP:Fields}

Returns the component ID for the maximum component defined so far.

Defining Memos and Blobs

To define a memo or a blob you use file properties with negative indexes.

You are limited to 255 memos and blobs in a file structure. Attempting to set a property of a memo or a blob outside this range will return an error of NoDrvFunc(80)

The properties used to define a memo or a blob are:

File{PROP:Type, -n} = 'BLOB' | 'MEMO'

If not specified, then MEMO is assumed

If you specify an invalid string ERRORCODE will be set to TypeDescErr (75).

File{PROP:Label, -n} = string

File{PROP:Name, -n} = string

If PROP:Name is specified for a memo, and PROP:Label is not specified, then the system will create a label for the memo based on the name.

File{PROP:Binary, -n} = 0 | 1

If not specified, then 0 is assumed.

File{PROP:Size, -n} = number

If number < 0 or number > 4,194,304, ERRORCODE() will be set to InvalidFileErr(47)

To delete a blob or a memo from the definition, use:

File{PROP:Memo, n} = ''

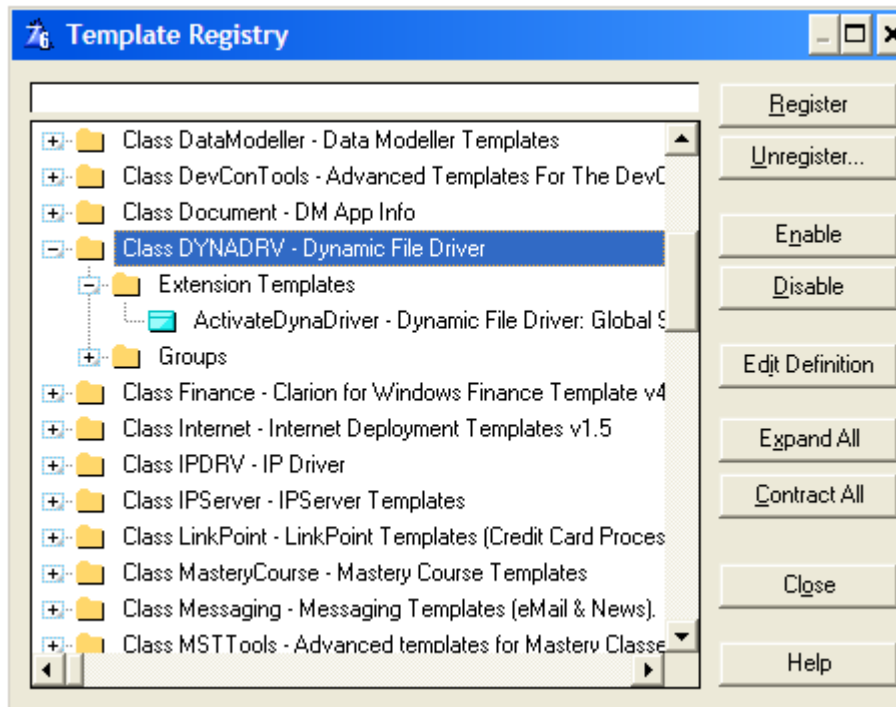
or

File{PROP:Memo, -n} = ''

DFD Templates

The Dynamic File Driver library includes a basic support template set that helps you to easily integrate dynamic file support into your existing applications.

To register the template, select the **Template Registry** menu item from the Clarion IDE **Setup** main menu. You need to register **DYNADRV.TPL**:

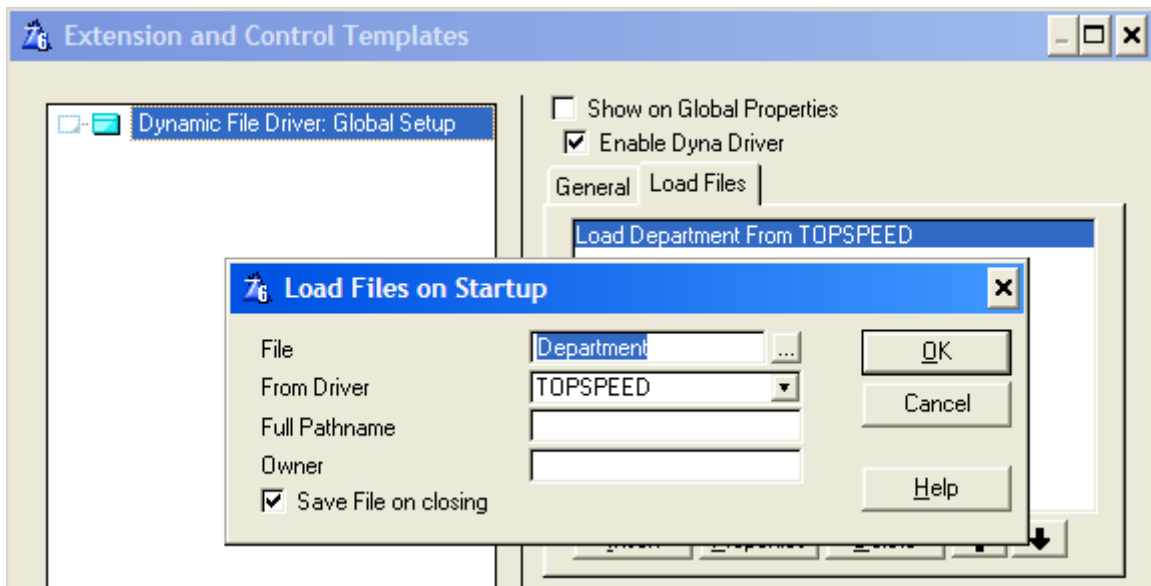
**Note:**

For purposes of this template documentation, the term “DynaDriver” refers to a class library used for Dynamic File Driver support. The class properties and methods are discussed in more detail in this document.

Dynamic File Driver Global Extension

The Dynamic File Driver Global Extension has two fundamental features:

1. By including the global extension into your application, the required libraries and the DynFile classes (containing a robust set of methods written to help you easily manage your dynamic files) are included into your application.
2. The “Load File” option, where a dictionary table used in your application can load values from any alternate data source. For example, an In-Memory table at program start up can read values from a TopSpeed file or SQL data source. At program shutdown, the modified values can optionally be written back to the alternate data source.



The following template prompts are provided:

Enable Dyna Driver

Check this box to automatically make the DynFile class available in your applications. The complete set of available methods is documented in this manual.

The **General** tab simply contains version and copyright information.

Load Files

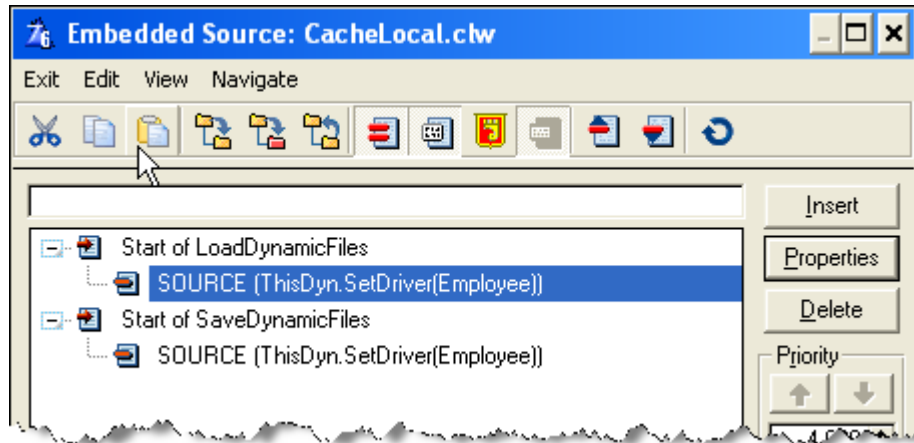
The Load Files option allows the Dynamic Driver to be used to easily load one file from another data source. The file to load should usually be an In-Memory table, but you are not limited to this particular driver. The template creates two global procedures, *LoadDynamicFiles* and *SaveDynamicFiles* that handles all of the logic and housekeeping needed to do this.

In the **Load Files** tab, a list box is displayed that displays the tables that are created by the Dynamic File Driver. To add, or modify their existing behavior, press the appropriate update buttons.

Note:

If you are linking in local mode, you must override this method and include the modified form of the *SetDriver* method. See the documentation for *SetDriver* for more information.

To override the SetDriver method, simply use the following global embeds:



In the example shown above, *Employee* is the label of a file that contains the matching driver type of the dynamic file that will be created. *ThisDyn* is the default template name of the DynaFile class object.

In the *Load Files on StartUp* dialog, the following prompts are displayed:

File

Press the ellipsis button, and select the dictionary table that will be created and populated by the Dynamic File Driver using the DynFile class methods. In most cases, this should be an In-Memory table, but you are not limited to this.

If the *file* selected is an In-Memory table, you need to select the driver type (From Driver), full pathname, and owner information (if applicable) of the alternate data source. If the *file* selected is not an In-Memory table, it is assumed that this file will be the alternate data source. The remaining prompts are disabled, and the dynamic file created will be an In-Memory table that matches the *File* definition.

From Driver

Select the driver type from the alternate data source that will be used to populate the dictionary *file*. If the full pathname and owner is *not* specified, the alternate data source should default to the first eight letters of the dictionary table, with the appropriate extension as identified by the driver.

For example, if your *File* named is "Department", and the driver type is *TOPSPEED*, the file that will be loaded will be "Departme.TPS"

Full Pathname

Enter the actual name and path of the alternate data source, if needed.

For Example:

```
C:\DATA\DEPARTMENT.TPS
```

```
Dbo.department (SQL source)
```

Owner

If the alternate data source is an encrypted ISAM type, enter the valid password information here. If the alternate data source is an SQL source, enter the valid connection string information here.

Example:

Myserver,northwind,sa,mypassword

Save File on closing

Check this box to automatically save the contents of the dynamic file to the original data source.

Tip

If you want to save (all the tables) while your program is still running, then use:

`SaveDynamicFiles ()`

In the appropriate embed

Even if you have *not* ticked the **Save File on closing** option, you can still force a single table to write back to the disk using the **SaveFile** method.

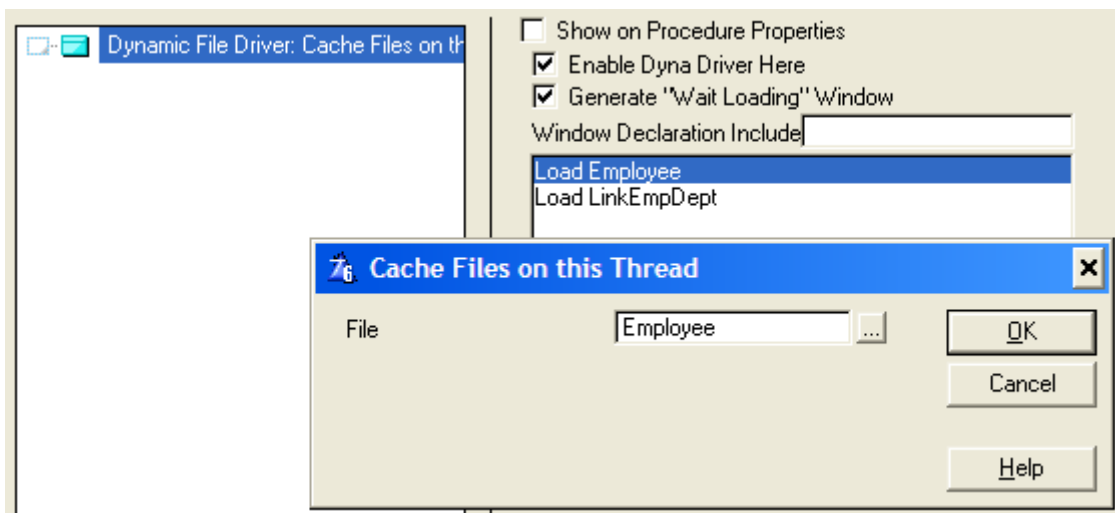
“Cache Files on this thread” extension template

The **CacheFiles on this thread** extension template is designed to improve the performance of a dynamic file load when used in a single process thread (i.e., any report procedure). The template interface handles the proper instantiation of the built-in DynFile class and calls the appropriate method.

Use this template when a dynamic file is used on a single thread, and the file to process is large. Its use will substantially improve the initialization and loading of the dynamic file.

Requirements

This template (as with the global extension Dynamic File Driver Global Extension) requires that you have the In-Memory Database Driver (IMDD) installed.



The following template prompts are provided:

Enable Dyna Driver Here

Check this box to include the DynFile class support locally in this procedure. If this procedure is part of a larger module, and the DynFile class is already declared, you can uncheck this option.

Generate “Wait Loading” Window

If the file to cache to the dynamic in-memory file is a large file, you may wish to present the user with a “Wait Loading” window. Check this option to implement this feature.

Window Declaration Include

This template includes a built-in “Wait Loading” window. If you wish to include your own custom WINDOW structure, enter the include file name in this prompt.

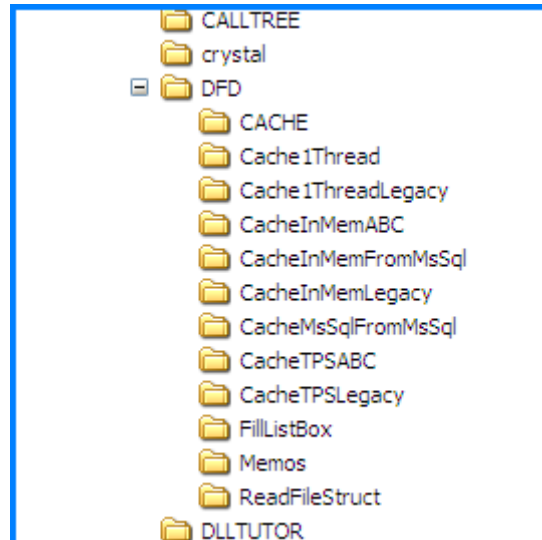
When you press the **Insert** button to add a table to cache, the following option is available:

File

In the *Cache Files on this Thread* dialog, press the ellipsis button to select any valid ISAM or SQL file that will be automatically cached to an In-Memory dynamic file.

Shipping Examples

There are many examples that are shipped with the Dynamic File Driver. Each one is designed to illustrate a different usage or configuration of the dynamic file driver. Clarion and ABC templates are both showcased. Some examples are hand coded. They are all located by default in the Clarion Root Examples folder, in a default *DFD* subfolder:



CACHE

If you were to examine your application's data dictionary, you would most likely be able to identify a number of tables that are infrequently changed. This might be a department list, ZIP codes, country names, etc. If these tables are set as In-Memory files you will gain quite a bit of performance.

This core application demonstrates loading a TopSpeed table, which not defined in the dictionary, into an In-Memory table defined in the dictionary, using the DynFile class methods via the template support. There is also a similar application (*LocalLinkCache.app*) in this folder that demonstrates the necessary modifications needed for using a local link with your DFD applications.

The other "Cache" examples offer several variations to this:

Cache1Thread

This application demonstrates the use of the **Cache Files on this thread** procedure extension, and demonstrates creating a procedural dynamic file used with an ISAM or SQL source. Refer to the *ReportDepartmentsFast* procedure for more information.

Cache1ThreadLegacy (Clarion template chain)

Same as the Cache1Thread example, but using the Clarion template chain.

CacheInMemABC

Same as the core **CACHE** application, but does not use a Full pathname in the Global extension, and saves any changes made to the dynamic file to the TopSpeed data source on closing.

CacheInMemFromMsSql

Demonstrates caching an SQL table to an in-memory dynamic file.

NOTE:

You will need to create a test database on your MSSQL server that matches the In-memory definition defined in the dictionary.

CacheInMemLegacy (Clarion template chain)

Same as the **CacheInMemABC** example, but uses the Clarion template chain.

CacheMsSqlFromMsSql

This application demonstrates using a dynamic file to read data from an SQL data source, and storing it in a dynamic MSSQL table created by the templates. You will need to set the GLO:Owner string to attach to your MSSQL server.

This application was designed to show you that the In-memory driver does *not* have to be used with your dynamic file implementation.

CacheTPSABC

This application demonstrates an alternate template configuration from the core **CACHE** application. By designating a TopSpeed file as the global file to load on startup, the template implicitly creates an In-Memory dynamic table that does not have to be defined in the data dictionary. The core example had an In-Memory table defined in the dictionary, but referred to the TopSpeed table as an external source.

Both applications perform the same function, but we are showing the flexibility of dynamic file configuration.

CacheTPSLegacy (Clarion template chain)

Same as the **CacheTPSABC** example, but using the Clarion template chain.

Memos

This application demonstrates using a dynamic file with tables that include a MEMO data type.

Other hand coded examples:**FillListBox**

This example uses the DynFile class library, and demonstrates creating dynamic files “on the fly” from both SQL and ISAM sources. It is essential that you examine this program if you plan to hand code any dynamic file implementations.

ReadFileStruct

This hand-coded example shows the dynamic file driver in a simple application, to dump the structure of a file to a dynamic ASCII table. This example is a good start to learning how to use the syntax of the dynamic file driver implementation.

DynFile Class

This section documents the DynFile Class that is used to integrate a rich language and template support layer used with the Dynamic File Driver.

Overview

The Dynamic File Driver has powerful logic built into the runtime library that helps you to create any file format on the fly. The DynFile class was designed to simplify the task of creating dynamic file structures.

DynFile Class Source Files

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

DynFile.INC	DynFile declarations
DynFile.CLW	DynFile method definitions

DynFile Properties

All of the DynFile properties are PRIVATE, and cannot be referenced directly by the programmer.

This section only documents these properties as they are referenced in the DynFile methods documentation.

sDriver

A STRING that identifies the contents of the DRIVER attribute used in the dynamic file.

sFileDriver

A reference to a FILE that identifies the DRIVER type used in the dynamic file.

bCreate

A BYTE that identifies if the CREATE attribute is applied to the dynamic file

bReclaim

A BYTE that identifies if the RECLAIM attribute is applied to the dynamic file

bEncrypt

A BYTE that identifies if the ENCRYPT attribute is applied to the dynamic file

bOEM

A BYTE that identifies if the OEM attribute is applied to the dynamic file

sOwner

A STRING that identifies the contents of the OWNER attribute used in the dynamic file.

sName

A STRING that identifies the contents of the NAME attribute used in the dynamic file.

sPrefix

A STRING that identifies the contents of the PRE attribute used in the dynamic file.

DynFile Methods

AddField (add field to a Dynamic File Structure)

AddField(*fieldgroup*)

AddField Add a field to a dynamic file's record structure

fieldgroup A GROUP that uses the *TFieldGrp* DynFile TYPED group structure.

AddField is a virtual method that allows you to add a field and its associated properties to a dynamic file structure. The *fieldgroup* must match the TYPED *TFieldGrp* defined in the *DynFile* class:

```
TFieldGrp            GROUP, TYPE
FieldNbr            LONG
Label                STRING (DynDrvSize:Label)
Name                 STRING (DynDrvSize:Name)
Type                 STRING (DynDrvSize:Type)
Over                 LONG
Dim                  LONG
Size                 ULONG
Places               ULONG
Fields               LONG    !represents the number of fields the group will contain
                      END
```

Implementation: The **AddField** method should be applied any time prior to the **FixFormat** method

Example:

```
MyFieldGrp          group(TFieldGrp) .

CODE

! Add some fields to record structure

MyFieldGrp.FieldNbr = 1
MyFieldGrp.Label = 'SysID'
MyFieldGrp.Type = 'LONG'
MyDynFile.AddField(MyFieldGrp)

MyFieldGrp.FieldNbr = 2
MyFieldGrp.Label = 'FirstName'
MyFieldGrp.Type = 'STRING'
MyFieldGrp.Size = 50
MyDynFile.AddField(MyFieldGrp)

MyFieldGrp.FieldNbr = 3
MyFieldGrp.Label = 'LastName'
MyFieldGrp.Type = 'STRING'
MyFieldGrp.Size = 50
MyDynFile.AddField(MyFieldGrp)
```

AddFieldToKey (add field to dynamic file Key definition)

AddFieldToKey(*keyname, fieldname, sequence, rank*)

AddFieldToKey Add a field to an existing KEY define in a dynamic file.

keyname A STRING constant, variable, or expression that identifies the label of the key

fieldname A STRING constant, variable, or expression that identifies the label of the field

sequence A BYTE value that identifies an ascending (1) or descending (0) sequence.

rank A BYTE value that identifies the rank, or order, that the *fieldname* is positioned in the *keyname*. A value of 1 indicates that it is the highest ranking (first) key component.

AddFieldToKey is used to add a field to a key that is used in a dynamic file.

Implementation: The **AddFieldToKey** method should be applied any time prior to the **FixFormat** method

Example:

```
MyDynFile            class (cDynFile)
                      End

                      CODE
MyDynFile.AddFieldToKey('kSysID', 'SysID', true, 1)
!Parameters are KeyName, FieldName, Sequence, Component rank in the key
```

AddKey (add key to a Dynamic File Structure)

AddKey(*keygroup*)

AddKey Add a key to a dynamic file

keygroup A GROUP that uses the *TKeyGrp* DynFile TYPED group structure.

AddKey is a virtual method that allows you to add a key and any associated properties to a dynamic file structure. The *keygroup* must match the TYPED TKeyGrp defined in the DynFile class:

```
TKeyGrp            GROUP, TYPE
KeyNbr            LONG
Label             STRING (DynDrvSize:Label)
Name              STRING (DynDrvSize:Name)
Type              STRING(1)        ! K = Key, I = Index
Dup               BYTE             ! False = no duplicate, True = duplicates allowed
Primary          BYTE             ! False = not primary, True = key is primary key
NoCase            BYTE             ! False = case sensitive, True = No case
Opt               BYTE             ! False = not optional, True = optional
                  END
```

Implementation: The **AddKey** method should be applied any time prior to the **FixFormat** method

Example:

```
MyKeyGrp            group(TKeyGrp). !Use TKeyGroup GROUP definition
```

CODE

```
! Now, create the keys
! We want 2 keys:
! 1 - kSysID, field: SysID, key is primary
! 2 - kName, field: LastName, FirstName, duplicate allowed

! Create key number 1 as defined above
clear(MyKeyGrp)
MyKeygrp.KeyNbr = 1
MyKeyGrp.Label = 'kSysID'
MyKeyGrp.Primary = true
MyKeyGrp.Type = 'K'            ! K = Key, I = Index
MyDynFile.AddKey(MyKeyGrp)
! Add the fields in the key
MyDynFile.AddFieldToKey('kSysID', 'SysID', true, 1)

! Now create key number 2
clear(MyKeyGrp)
MyKeygrp.KeyNbr = 2
MyKeyGrp.Label = 'kName'
MyKeyGrp.Type = 'K'            ! K = Key, I = Index
MyKeyGrp.Dup = true
MyDynFile.AddKey(MyKeyGrp)
! Add the fields in the key
MyDynFile.AddFieldToKey('kName', 'LastName', true, 1)
MyDynFile.AddFieldToKey('kName', 'FirstName', true, 2)
```

AddMemo (add memo to a Dynamic File Structure)

AddMemo(*memogroup*)

AddMemo Add a MEMO or BLOB to a dynamic file's record structure

memogroup A GROUP that uses the *TMemoGrp* DynFile TYPed group structure.

AddMemo is a virtual method that allows you to add a MEMO or BLOB data type and its associated properties to a dynamic file structure. The *memogroup* must match the TYPed *TMemoGrp* defined in the DynFile class:

```

TMemoGrp    GROUP, TYPE
MemoNbr    LONG            ! Must be negative, first memo = -1, second memo = -2 etc...
Label      STRING (DynDrvSize:Label)
Name       STRING (DynDrvSize:Name)
Type       STRING(1)        ! M = Memo, B = Blob
Binary     BYTE            ! True = Binary contents, False = Text
size       ULONG
              END

```

Implementation: The **AddMemo** method should be applied any time prior to the **FixFormat** method

Example:

```

MyMemoGrp            group(TMemoGrp). !Create group that matches DynFile TmemoGrp

```

CODE

```

! Create a memo field, Notes

MyMemoGrp.MemoNbr = -1
MyMemoGrp.Label = 'NOTES'
MyMemoGrp.Type = 'M'    ! M is for MEMO
MyMemoGrp.Size = 4096 ! memo is 4k in size
MyDynFile.AddMemo (MyMemoGrp)

! Now create a Blob

MyMemoGrp.MemoNbr = -2
MyMemoGrp.Label = 'Photo'
MyMemoGrp.Type = 'B'            ! B is for blob
MyMemoGrp.Binary = TRUE
MyDynFile.AddMemo (MyMemoGrp)

```

CacheFile(load table into in-memory dynamic file)

CacheFile(*filelabel* , <*drivename*> , <*namestring*> , <*ownerstring*>)

CacheFile	Create a dynamic in-memory table from an ISAM or SQL source.
<i>filelabel</i>	The label of a valid FILE that identifies the data source.
<i>drivename</i>	A string constant, variable or expression that identifies the name of the file driver that applies to the data source.
<i>namestring</i>	A string constant, variable or expression that contains the operating system device name for the structure identified by the <i>filelabel</i> to use with the data source.
<i>ownerstring</i>	A string constant, variable or expression that contains file encryption password, or SQL connection string for the data source.

CacheFile is a virtual method that allows a single-line call that changes an ISAM or SQL table to an In-Memory table, and loads it from the source. **CacheFile** is used to cache a table on a single thread.

Implementation: The **CacheFile** method can be applied to any procedure prior to processing the file.

NOTE: YOU MUST HAVE THE IN-MEMORY DRIVER to use this method. There is a template extension available to help you implement this easily in any procedure. See the *DFD Templates* section in this PDF for more information.

Example:

```

ThisDyn  Class(DynFile)
        End
DynWaitWindow WINDOW('Please Wait'),AT(, ,116,18),FONT('MS Sans Serif',8,,FONT:regular),|
        CENTER,GRAY,DOUBLE
        STRING('Please Wait: Caching Tables. '),AT(0,3,116,12),USE(?DynWaitString),TRN,CENTER
        END
CODE
c = clock()
GlobalErrors.SetProcedureName('ReportDepartmentsFast')
SELF.Request = GlobalRequest ! Store the incoming request
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?Progress:Thermometer
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors !Set this windows ErrorManager to global ErrorManager
CLEAR(GlobalRequest)          ! Clear GlobalRequest after storing locally
CLEAR(GlobalResponse)
OPEN(DynWaitWindow)
DISPLAY()
ThisDyn.CacheFile(Employee,'TOPSPEED',,)
ThisDyn.CacheFile(LinkEmpDept,'TOPSPEED',,)
CLOSE(DynWaitWindow)
Relate:Department.SetOpenRelated()

```

CreateFromFile (build Dynamic File Structure from existing file)

CreateFromFile **PROCEDURE**(*filelabel*)

CreateFromFile Create a dynamic file from an existing file structure.

CreateFromFile is a virtual method used to create a dynamic file structure from an existing file. It was developed to allow a programmer to quickly create a dynamic file structure, assign new attributes as needed, and then fix the new and updated format. The **FillFrom** or **FixFormat** methods must be applied after the **CreateFromFile** method is issued.

Example:

```
!create a dynamic MEMORY file based on the PEOPLE file
MEMProducts      &DynFile
TheFile          &File

CODE
MEMProducts &= new(DynFile)

OPEN(window)
ACCEPT
CASE EVENT()
OF EVENT:Accepted
CASE FIELD()
OF ?FromPeople
MEMProducts.UnfixFormat()
MEMProducts.ResetAll()
MEMProducts.CreateFromFile(people)
MEMProducts.SetName('MEMProduct')
MEMProducts.SetDriver('MEMORY')
MEMProducts.FillFrom(people)
TheFile &= MEMProducts.GetFileRef()
SET(TheFile)
DynFileList.Refresh(TheFile)
END
END
```

CreateFromSQL(create structure from SQL result set)

CreateFromSQL(*querystring*), *errorcode*

CreateFromSQL	Create a dynamic file based on an SQL query.
<i>querystring</i>	A STRING constant, variable, or expression that contains a valid SQL statement.
<i>errorcode</i>	A LONG value that contains an errorcode, if any. If the dynamic file is created successfully, a value of zero(0) is returned.

CreateFromSQL is a virtual method that is used to create a dynamic file structure based on the components of valid SQL *querystring*. The fields created are extracted from the SELECT criteria, and keys created are extracted by the ORDER BY clause, and assigned to the dynamic file via the use of a PROP:SQL property assignment. The syntax of the *querystring* must match the DRIVER type. The dynamic file MUST be SQL based, to allow proper processing of the SQL statement.

Return Value: LONG

Implementation: The **CreateFromSQL** method should be applied after the Driver and Owner attributes of the dynamic file have been set. These attributes can be set using the DynFile **SetDriver** and **SetOwner** methods.

Error codes returned:

- 2 File not found (Owner attribute incorrect or not set)
- 75 Invalid Field Type Descriptor (Driver attribute incorrect or not set)

Example:

```
MSProducts &DynFile      !Dynamic File reference
lQuery      CSTRING(200) !Hold SQL statement

CODE
!First, set the query
lQuery='SELECT ProductID , ProductName FROM Products ORDER BY ProductName'
MSProducts &= new(DynFile)           !Assign the dynamic file reference
MSProducts.SetDriver('MSSQL')       !Set the Driver
MSProducts.SetOwner('(local),Northwind,sa,') !Set the connect string
MSProducts.CreateFromSQL(CLIP(lQuery)) !Create the file
```

CreateKeyComponents (create key elements of a dynamic file key)

CreateKeyComponents(*keyname, keyfieldgroup*)

CreateKeyComponents	Create key components for keys in a dynamic file
<i>keyname</i>	A reference to a label of a key contained in the dynamic file.
<i>keyfieldgroup</i>	A reference to a queue containing entries matching a TYPED TKeyFieldGrp.

CreateKeyComponents is a protected, virtual method used to create the key components used in a dynamic file structure. It uses the internal queues of the DynFile class to extract necessary key information, and assigns the appropriate properties to the dynamic file structure.

Implementation: The **CreateKeyComponents** method is usually called within the **CreateStruct** method, and assigns all necessary key component properties just prior to a call to FIXFORMAT.

Example:

```
!create the keys

SORT (SELF.KeyQ, SELF.KeyQ.KeyNbr)
rec = RECORDS (SELF.KeyQ)
LOOP ndx = 1 TO rec
  GET (SELF.KeyQ, ndx)
  AKey &= SELF.rDynF{prop:Key, ndx}
  AKey{prop:Type} = CHOOSE (UPPER (SELF.KeyQ.Type) = 'K', 'KEY', 'INDEX')
  AKey{prop:Label} = CLIP (SELF.KeyQ.Label)
  IF CLIP (SELF.KeyQ.Name)
    AKey{prop:Name} = CLIP (SELF.KeyQ.Name)
  END
  AKey{prop:NoCase} = SELF.KeyQ.NoCase
  AKey{prop:Opt} = SELF.KeyQ.Opt
  AKey{prop:Primary} = SELF.KeyQ.Primary
  AKey{prop:Dup} = SELF.KeyQ.Dup
  SELF.CreateKeyComponents (AKey, SELF.KeyQ.FieldQ)
END
```

CreateStruct (build a dynamic file structure)

CreateStruct()

CreateStruct Create a complete dynamic file structure

CreateStruct is a virtual and protected method that is used to assign all file, field, key, memo and blob attributes to a dynamic file structure.

Implementation: The **CreateStruct** method should be called after all field and key name have been assigned, and just prior to the FIXFORMAT language statement. In the DynFile class, it is called in the **FixFormat**, **FillFrom** and **CreateFromFile** methods

Conceptual Example:

```
DynFile.CreateFromFile          PROCEDURE (FILE pFile)

ndx      LONG
ndx2     LONG
Rec      LONG
Rec2     LONG
flds     LONG
AKey     &KEY

CODE
!all file attribute assignments here
  flds = pFile{prop:Fields}
  LOOP ndx = 1 TO flds
    CLEAR(SELF.FieldQ)
!all field assignments here
  ADD(SELF.FieldQ)
  END
  flds = pFile{PROP:Memos}
  LOOP ndx = 1 TO flds
    CLEAR(SELF.MemoQ)
!all memo/blob assignments here
  ADD(SELF.MemoQ)
  END
  rec = pFile{PROP:Keys}
  LOOP ndx = 1 TO rec
!all key property and components assignments here
  ADD(SELF.KeyQ)
  END
  SELF.CreateStruct() !Create the dynamic file structure from information above
```

FillFrom (copy data to a Dynamic File Structure)

FillFrom(| *DynFile |), *errorcode*

FillFrom(| filelabel , <sameflag> |), *errorcode*

FillFrom **Create a dynamic file and fill it with the source's contents.**

DynFile A reference to the DynFile class.

filelabel The label of a FILE structure.

sameflag A BYTE value (default is 0 – not the same) that indicates that the dynamic file structure receiving the contents of the table is the same. This is used to improve record assignment speed.

errorcode A valid error code is returned if the method encountered any errors.

FillFrom is used to create a dynamic file structure, and load existing data from the dynamic file's source. If the parameter is a *filelabel*, **FillFrom** will fix the format of the current dynamic file structure, and load the contents of the referenced *filelabel*.

If the **FillFrom** parameter is the reference to the *DynFile* class, **FillFrom** also creates all file, field, key and memo/blob attributes stored in the *DynFile* Class internal queues, and then proceeded to fix the format and prime the file with values also stored in the internal queues.

Return Value: LONG

Example 1 (using the class parameter):

```

TestFill                    PROCEDURE

MSProducts                &DynFile
MEMProducts               &DynFile

TheFile                    &File

MyQueue                    QUEUE
productID                  LONG
ProductName                STRING(25)
                            END

Window WINDOW('Caption'),AT(,,320,185),FONT('MS Sans Serif' ,,,FONT:regular, CHARSET:ANSI),|
SYSTEM,GRAY, DOUBLE
LIST,AT(6,6,308,146),USE(?List1),VSCROLL,FORMAT(' 72R(2)|M~ProductID~@N_5@20L(2)|
                            M~Product Name~@s25@'),FROM(MyQueue)
BUTTON('Close'),AT(259,161,51,14),USE(?Button1),STD(STD:Close)
END
CODE

MSProducts &= NEW(DynFile)
MEMProducts &= NEW(DynFile)

MSProducts.SetDriver('MSSQL')
!MSProducts.SetOwner(' (local),Northwind,sa,sa2000')
MSProducts.SetOwner(' (local),Northwind,sa,')
MSProducts.CreateFromSQL('SELECT ProductID, ProductName FROM Products')

MEMProducts.SetCreate(true)
MEMProducts.SetName('MEMProduct')
MEMProducts.SetDriver('MEMORY')
MEMProducts.FillFrom(MSProducts)

```

```

DISPOSE (MSProducts)

TheFile &= MEMProducts.GetFileRef()

SET (TheFile)
LOOP
  NEXT (TheFile)
  IF ERRORCODE ()
    BREAK
  ELSE
    MEMproducts.GetFieldValue ('ProductID', MyQueue.ProductID)
    MEMProducts.GetFieldValue ('ProductName', MyQueue.ProductName)
    ADD (MyQueue)
  END
END

OPEN (WINDOW)
ACCEPT
END

DISPOSE (MEMProducts)

```

Example 2 (Using a file label):

```

OPEN (window)
DynFileList.Init (?List1)
ACCEPT
CASE EVENT ()
  OF EVENT:Accepted
    CASE FIELD ()
      OF ?Refresh1
        lQuery = lQuery1
        Do RefreshQuery
      OF ?Refresh2
        lQuery = lQuery2
        Do RefreshQuery
      OF ?Refresh3
        lQuery = lQuery3
        Do RefreshQuery
      OF ?FromPeople
        MEMProducts.UnfixFormat ()
        MEMProducts.ResetAll ()
        MEMProducts.CreateFromFile (people)
        MEMProducts.SetName ('MEMProduct')
        MEMProducts.SetDriver ('MEMORY')
        MEMProducts.FillFrom (people)
        TheFile &= MEMProducts.GetFileRef ()
        SET (TheFile)
        DynFileList.Refresh (TheFile)
    END
  END
END
END

```

FillTo (copy dynamic file contents to table)

FillTo(*filelabel* ,<*sameflag*>)
FillTo(*queueabel*)

FillTo	Copy contents of dynamic file to target file
<i>filelabel</i>	The label of a valid FILE.
<i>sameflag</i>	A BYTE value (default is 0 – not the same) that indicates that the table structure receiving the contents of the dynamic file structure is the same. This is used to improve record assignment speed.
<i>queueabel</i>	The label of a valid QUEUE structure.

FillTo is used to copy the contents of the active dynamic file into an existing table or queue. The *filelabel* must be an existing FILE structure declared in the program. The *queueabel* must be an existing QUEUE structure declared in the program. If the method is successful, it returns no error code (0). Otherwise, an appropriate file processing error code is returned.

Implementation: **FillTo** checks to see if the dynamic file exists via the *StructCreated* property, and if true, extracts the file's contents and adds it to the data source identified by the *filelabel* or *queueabel* parameter. If any error is encountered during this process, an appropriate error code is returned.

Return Value: LONG

Example:

```
CODE
  sts = STATUS(myFile)
  IF sts <> 0
    CLOSE(myFile)
  END

  Self.CreateFromFile(myFile)
  Self.SetDriver('In-Memory')
  Self.SetName(pName)
  Self.SetOwner(pOwner)
  Self.FillTo(myFile)
  Self.UnfixFormat()
```

See Also: FillFrom, LoadFile, SaveFile

FixFormat (fix and create a dynamic file format)

FixFormat(), *errorcode*

FixFormat Fix and create a dynamic file format

errorcode a valid error code that can be returned by the FIXFORMAT language statement

The **FixFormat** method fixes the format of a dynamic file just prior to its reference assignment and creation. It is similar in function to the standard FIXFORMAT language statement, and returns the same *errorcode* if applicable. It also checks the *StructCreated* DynFile property, and if it is not set, it will create the file structure prior to fixing the format.

Implementation: The **FixFormat** method is used internally by the DynFile class by the **FillFrom** methods.

Return Value: LONG

Example:

```
MyFieldGrp        group (TFieldGrp) .
MyKeyGrp         group (TKeyGrp) .
MyMemoGrp        group (TMemoGrp) .

MyDynFile        class (cDynFile)
                  end

TheFile          &File

rBlob            &BLOB
AKey             &Key

LastError        long

locSysID         long
locFirstName     string(50)
locLastName      string(50)
locNotes         string(4096)
locDate          date

Window WINDOW('Caption'), AT(, , 322, 185), GRAY, DOUBLE, AUTO
          PANEL, AT(6, 5, 209, 67), USE(?Panel1), BEVEL(-1)
          STRING('SysID: '), AT(17, 18), USE(?String1)
          STRING(@n_6), AT(63, 18), USE(locSysID)
          IMAGE, AT(223, 5, 87, 134), USE(?Image1)
          STRING('First Name: '), AT(17, 32), USE(?String2)
          STRING(@s50), AT(63, 32), USE(locFirstName)
          STRING('Last Name: '), AT(17, 46), USE(?String3)
          STRING(@s50), AT(63, 46), USE(locLastName)
          STRING('Birthday: '), AT(17, 58), USE(?String8)
          STRING(@d17), AT(63, 59), USE(locDate)
          STRING('Notes: '), AT(10, 76), USE(?String4)
          TEXT, AT(10, 88, 201, 52), USE(locNotes), BOXED
          BUTTON('Previous '), AT(11, 144, 45, 14), USE(?btnPrev)
          BUTTON('Next '), AT(63, 144, 45, 14), USE(?btnNext)
          PROMPT('Select Order: '), AT(168, 148), USE(?Prompt1)
          LIST, AT(223, 147, 87, 10), USE(?List1), DROP(10), FROM('Record Order|kSysID|kName')
          BUTTON('Close '), AT(266, 163, 45, 14), USE(?btnClose)

END
```

```
CODE

! First, let define some file attributes

MyDynFile.SetDriver('TopSpeed')
MyDynFile.SetName('TEST.TPS')
MyDynFile.SetCreate(true)

! Add some fields to record structure

MyFieldGrp.FieldNbr = 1
MyFieldGrp.Label = 'SysID'
MyFieldGrp.Type = 'LONG'
MyDynFile.AddField(MyFieldGrp)

!...add more fields here

! Create a memo field, Notes

MyMemoGrp.MemoNbr = -1
MyMemoGrp.Label = 'NOTES'
MyMemoGrp.Type = 'M' ! M is for MEMO
MyMemoGrp.Size = 4096 ! memo is 4k in size
MyDynFile.AddMemo(MyMemoGrp)

! Now create a Blob

MyMemoGrp.MemoNbr = -2
MyMemoGrp.Label = 'Photo'
MyMemoGrp.Type = 'B' ! B is for blob
MyMemoGrp.Binary = true
MyDynFile.AddMemo(MyMemoGrp)

! Now, create the keys
! We want 2 keys:
! 1 - kSysID, field: SysID, key is primary
! 2 - kName, field: LastName, FirstName, duplicate allowed

! Create key nbr 1 as defined above
clear(MyKeyGrp)
MyKeygrp.KeyNbr = 1
MyKeyGrp.Label = 'kSysID'
MyKeyGrp.Primary = true
MyKeyGrp.Type = 'K' ! K = Key, I = Index
MyDynFile.AddKey(MyKeyGrp)
! Add the fields in the key
MyDynFile.AddFieldToKey('kSysID', 'SysID', true, 1)
! Now create key nbr 2
clear(MyKeyGrp)
MyKeygrp.KeyNbr = 2
MyKeyGrp.Label = 'kName'
MyKeyGrp.Type = 'K' ! K = Key, I = Index
MyKeyGrp.Dup = true
MyDynFile.AddKey(MyKeyGrp)
! Add the fields in the key
MyDynFile.AddFieldToKey('kName', 'LastName', true, 1)
MyDynFile.AddFieldToKey('kName', 'FirstName', true, 2)

! Now call the magic method
MyDynFile.FixFormat()

! Get a reference to the created file, create the file on disk and open it
TheFile &= MyDynFile.GetFileRef()
create(TheFile)
open(TheFile)
empty(TheFile)
```

GetCreate (get value of Create property)

GetCreate()

GetCreate Query CREATE status of current dynamic file structure.

The **GetCreate** method is used to return the current value of the DynFile *bCreate* property. If the value returned is (1) TRUE, then the dynamic file will be able to be created. If the value returned is (0) FALSE, then the dynamic file must exist before opening it.

Implementation: Currently only implemented by the user.

Return Value: BYTE

See Also: **SetCreate**

GetDriver (get value of Driver property)

GetDriver()

GetDriver Query DRIVER attribute contents of current dynamic file structure.

The **GetDriver** method is used to return the current value of the DynFile *sDriver* property. This property holds the contents of the dynamic file DRIVER attribute.

Return Value: STRING

See Also: **SetDriver**

GetEncrypt (get value of Encrypt property)

GetEncrypt()

GetEncrypt Query ENCRYPT status of current dynamic file structure.

The **GetEncrypt** method is used to return the current value of the DynFile *bEncrypt* property. If the value returned is (1) TRUE, then the dynamic file's ENCRYPT attribute is active. If the value returned is (0) FALSE, then the dynamic file is not encrypted.

Return Value: BYTE

See Also: **SetEncrypt**

GetField (get field contents)

GetField(*fieldlabel*),*fieldcontents*

GetField Return the contents of a dynamic file field (column)

fieldlabel A string constant, variable or expression that identifies the label of a dynamic file field

fieldcontents The contents of the field named by the *fieldlabel*.

GetField is used to return the contents of a field label used in a dynamic file, and identified by the *fieldlabel* parameter.

Return Value: ANY

Example:

GrabData ROUTINE

DATA

TheFile &FILE

ndx LONG

locANY ANY

CODE

TheFile &= pDynFile.GetFileRef()

!This should use the TheFile instead the queue

IF pDynFile.CreatedFromSQL = false

SET(TheFile)

END

LOOP

NEXT(TheFile)

IF ERRORCODE()

BREAK

ELSE

LOOP ndx = 1 TO RECORDS(pDynFile.FieldQ)

GET(pDynFile.FieldQ, ndx)

IF ~ERRORCODE()

locANY &= pDynFile.GetField(pDynFile.FieldQ.Label)

IF ~(locANY &= null)

SELF.SetFieldValue(pDynFile.FieldQ.Label, locANY)

END

END

END

ADD(SELf.rDynF)

RetVal = ERRORCODE()

IF RetVal

BREAK

END

END

END

See Also: GetFieldValue

GetFieldName (get label name of field)

GetFieldName(*fieldnumber*),*fieldcontents*

GetFieldName Return the label of a dynamic file field (column)

fieldnumber A LONG constant, variable or expression that identifies the ordinal field number in a dynamic file.

fieldcontents A string constant, variable or expression that holds the label of a dynamic file field named by the *fieldnumber*.

GetFieldName is used to return the field label used in a dynamic file, identified by the *fieldnumber* ordinal position on the dynamic file.

Return Value: STRING

Example:

```
DynFileList.Refresh PROCEDURE(*FILE TheDynFile)
lColumns          SHORT
lFormat           CSTRING(2048),AUTO
Idx               SHORT
CODE
  SELF.File &= TheDynFile
  IF NOT SELF.File &= NULL
    lColumns = SELF.File{PROP:Fields}
    lFormat = ''
    SELF.ListControl{PROP:FORMAT}=CLIP(lFormat)
    LOOP Idx=1 TO lColumns
!Column Header is loaded with field label name
lFormat=CLIP(lFormat)&SELF.FormatColumn(CLIP(MemProducts.GetFieldName(Idx)),|
SELF.File{PROP:Type,Idx},SELF.File{PROP:Size,Idx},SELF.File{PROP:Places,Idx})
    END
    SELF.ListControl{PROP:FORMAT}=CLIP(lFormat)
    SELF.Changed = True
  ELSE
    SELF.Changed = False
  END
```

GetFieldValue (move field value to a reference)

GetFieldValue(*fieldlabel*, *varcontents*)

GetFieldValue Set the contents of a dynamic file field (column) to a variable.

fieldlabel A string constant, variable or expression that identifies the label of a dynamic file field

varcontents A reference to an ANY data type used to hold the contents of the field named by the *fieldlabel*.

GetFieldValue is a virtual method used to set the contents of a field label used in a dynamic file, identified by the *fieldlabel* parameter, to an ANY data type named by the *varcontents*.

This method is equivalent to the **GetField** method, but does not return the field contents like **GetField**, but instead passes the contents to a variable by address.

Example:

```
locSysID          LONG
locFirstName      STRING(50)
locLastName       STRING(50)
locNotes          STRING(4096)
locDate           DATE

CODE
DO AssignData

AssignData        ROUTINE

    MyDynFile.GetFieldValue('SysID', locSysID)
    MyDynFile.GetFieldValue('FirstName', locFirstName)
    MyDynFile.GetFieldValue('LastName', locLastName)
    MyDynFile.GetFieldValue('Birthday', locDate)

    locNotes = theFile{prop:Value, -1}

    rBlob &= TheFile{prop:blob, -2}
    if rBlob{prop:size}
        ?Image1{PROP:ImageBlob} = rBlob{prop:Handle}
    end

    ?btnPrev{prop:disable} = bof(TheFile)
    ?btnNext{prop:disable} = eof(TheFile)
```

See Also: GetField

GetFileRef (get reference to a dynamic file)

GetFileRef()

GetFileRef Return a reference to a dynamic file

GetFileRef returns a reference to a file defined in the DynFile class. After any dynamic file has been initialized and formatted, processing the dynamic file is implemented with a file reference assignment.

Return Value: *FILE

Example:

```

MSProducts      &DynFile  !reference to the DynClass
MEMProducts     &DynFile  !reference to the DynClass

TheFile         &File     !reference to a FILE

!window and queue structures defined here
CODE

MSProducts &= NEW(DynFile)           !instantiate new objects
MEMProducts &= NEW(DynFile)

MSProducts.SetDriver('MSSQL')       !set appropriate DF attributes
!MSProducts.SetOwner(' (local) ,Northwind,sa,sa2000')
MSProducts.SetOwner(' (local) ,Northwind,sa, ')
MSProducts.CreateFromSQL('SELECT ProductID, ProductName FROM Products')

MEMProducts.SetCreate(true)
MEMProducts.SetName('MEMProduct')
MEMProducts.SetDriver('MEMORY')
MEMProducts.FillFrom(MSProducts)

DISPOSE (MSProducts)

TheFile &= MEMProducts.GetFileRef() !Get File reference from DynFile class

SET (TheFile)                        !Now start processing
LOOP
NEXT (TheFile)
IF ERRORCODE ()
BREAK
ELSE
MEMproducts.GetFieldValue('ProductID', MyQueue.ProductID)
MEMProducts.GetFieldValue('ProductName', MyQueue.ProductName)
ADD (MyQueue)
END
END

OPEN (WINDOW)
ACCEPT
END

```

See Also: GetKeyRef

GetKeyRef (get reference to a dynamic file's key)

GetKeyRef(*keyname*)

GetKeyRef Return a reference to a dynamic file's key

keyname A string constant, variable, or expression that identifies the key label.

GetFileRef returns a reference to a file defined in the DynFile class. After any dynamic file has been initialized and formatted, processing the dynamic file in key sequence is implemented with a key reference assignment.

Return Value: *KEY

Example:

```
CASE EVENT()
  OF ?FromPeople
    MEMProducts.UnfixFormat()
    MEMProducts.ResetAll()
    MEMProducts.CreateFromFile(people)
    MEMProducts.SetName('MEMProduct')
    MEMProducts.SetDriver('MEMORY')
    MEMProducts.FillFrom(people)
    TheKey &= MEMProducts.GetKeyRef('PEO:KEYNAME')
    IF TheKey &= NULL
      MESSAGE('Key is NULL..processing in FILE order')
      SET(TheFile)
    ELSE
      SET(TheKey)
    END
  DynFileList.Refresh(TheFile)
END
```

GetName (get value of Name property)

GetName()

GetName Query NAME attribute contents of current dynamic file structure.

The **GetName** method is used to return the current value of the DynFile *sName* property. This property holds the contents of the dynamic file NAME attribute.

Return Value: STRING

See Also: **SetName**

GetOEM (get value of OEM property)

GetOEM()

GetOEM Query OEM status of current dynamic file structure.

The **GetOEM** method is used to return the current value of the DynFile *bOEM* property. If the value returned is (1) TRUE, then the dynamic file's OEM attribute is active. If the value returned is (0) FALSE, then the dynamic file's is not using the OEM attribute.

Implementation: **GetOEM** can be called at anytime to query the status of a dynamic file's OEM attribute.

Return Value: BYTE

See Also: **SetOEM**

GetOwner (get value of Owner property)

GetOwner()

GetOwner Query OWNER attribute contents of current dynamic file structure.

The **GetOwner** method is used to return the current value of the DynFile *sOwner* property. This property holds the contents of the dynamic file OWNER attribute.

Return Value: STRING

Implementation: **GetOwner** can be used at any time to read the contents of a dynamic file's OWNER attribute.

See Also: **SetOwner**

GetPrefix (get value of Prefix property)

GetPrefix()

GetPrefix Query PRE attribute contents of current dynamic file structure.

The **GetPrefix** method is used to return the current value of the DynFile *sPrefix* property. This property holds the contents of the dynamic file PRE attribute.

Return Value: STRING

Implementation: **GetPrefix** can be used at any time to read the contents of a dynamic file's PRE attribute.

See Also: **SetPrefix**

GetReclaim (get value of Reclaim property)

GetReclaim()

GetReclaim Query RECLAIM status of current dynamic file structure.

The **GetReclaim** method is used to return the current value of the DynFile *bReclaim* property. If the value returned is (1) TRUE, then the dynamic file's RECLAIM attribute is active. If the value returned is (0) FALSE, then the dynamic file's is not using the RECLAIM attribute.

Implementation: **GetReclaim** can be called at anytime to query the status of a dynamic file's RECLAIM attribute.

Return Value: BYTE

See Also: **SetReclaim**

GetCreatedFromSQL (get value of CreatedFromSQL property)

GetCreatedFromSQL()

GetCreatedFromSQL Query status of DynClass CreatedFromSQL property.

The **GetCreatedFromSQL** method is used to return the current value of the DynFile *CreatedFromSQL* property. If the value returned is (1) TRUE, then the dynamic file was created from SQL syntax. If the value returned is (0) FALSE, then the dynamic file was created using standard DynFile methods.

Implementation: **GetCreatedFromSQL** can be called at anytime to determine if a dynamic file was created using an SQL statement. In the DynFile class, this method is useful in determining how the FillFrom method extracts the requested data to use with the dynamic file.

Return Value: BYTE

LoadFile (load file with data source)

LoadFile(*filelabel*, *drivername*, *<namestring>*, *<ownerstring>*)

LoadFile	Load existing file with contents from another data source
<i>filelabel</i>	The label of a valid FILE that identifies the data target.
<i>drivername</i>	A string constant, variable or expression that identifies the name of the file driver that applies to the data source.
<i>namestring</i>	A string constant, variable or expression that contains the operating system device name for the structure identified by the <i>filelabel</i> to use with the data source.
<i>ownerstring</i>	A string constant, variable or expression that contains file encryption password, or SQL connection string for the data source.

LoadFile is a virtual method that can be used to load an existing table from an alternate data source.

Implementation: **LoadFile** validates that the data source is opened, then calls the CreateFromFile, SetDriver, SetName, and SetOwner methods. It then loads the contents of the data source through the FillTo method.

Example:

```
LoadDynamicFiles    PROCEDURE () !Dyna-Driver Extension
ThisDyn            CLASS (DynFile)
                  END

CODE
  ThisDyn.LoadFile (Department, 'TOPSPEED', 'department.tps', '')

SaveDynamicFiles    PROCEDURE () !Dyna-Driver Extension
ThisDyn            CLASS (DynFile)
                  END

CODE
  ThisDyn.SaveFile (Department, 'TOPSPEED', 'department.tps', '')
```

See Also: FillFrom, FillTo, SaveFile

RemoveField (remove a field from a Dynamic File)

RemoveField(*fieldname*)

RemoveField Remove a field definition from a dynamic file.

fieldname A string constant, variable or expression that identifies the field name to be removed from the dynamic file definition

RemoveField takes care of all housekeeping needed when removing a field from a dynamic file definition. First, the method checks if the *fieldname* is used in any key definitions. If so, it locates and stores all key names that use it. For each key name where that field is used, a call to the **RemoveKeyField** method is made. Also, if the internal File structure is already created, Remove Field kills the field. Additional housekeeping with the DynFile *FieldQ* is performed, and key definitions with the resorted field numbers are also updated.

Example:

```
CODE
TheFile &= null
MyDynFile.Unfixformat()

MyDynFile.SetName('other.tps')
MyDynFile.RemoveKeyField('kName', 'FirstName')
MyDynFile.RemoveField('Birthday')
MyDynFile.RemoveField('SysID')

MyDynFile.Fixformat()
TheFile &= MyDynFile.GetFileRef()
CREATE(TheFile)
```

RemoveKey (remove a key from a Dynamic File)

RemoveKey(*keyname*)

RemoveKey Remove a complete key definition from a dynamic file.

keyname A string constant, variable or expression that identifies the key name to be removed from the dynamic file definition

RemoveKey takes care of all housekeeping needed when removing a key from a dynamic file definition, including proper handling of the key if the dynamic file is already opened, and reindexing of the internal queues used in the DynFile class.

Example:

```
CODE
!here is how the original key was created
clear(MyKeyGrp)
MyKeygrp.KeyNbr = 2
MyKeyGrp.Label = 'kName'
MyKeyGrp.Type = 'K'           ! K = Key, I = Index
MyKeyGrp.Dup = true
MyDynFile.AddKey(MyKeyGrp)
! Add the fields in the key
MyDynFile.AddFieldToKey('kName', 'LastName', true, 1)
MyDynFile.AddFieldToKey('kName', 'FirstName', true, 2)
```

RemoveElements ROUTINE

```
TheFile &= NULL
MyDynFile.Unfixformat()

MyDynFile.SetName('other.tps')
MyDynFile.RemoveKey('kName')
MyDynFile.RemoveField('Birthday')
MyDynFile.RemoveField('SysID')

MyDynFile.Fixformat()
TheFile &= MyDynFile.GetFileRef()
CREATE(TheFile)
```

RemoveKeyField (remove a key field from a Dynamic File)

RemoveKeyField(*keyname*, *fieldname*)

RemoveKeyField Remove a key field from a key in a dynamic file.

keyname A string constant, variable or expression that identifies the key label containing the field to remove.

fieldname A string constant, variable or expression that identifies the field name to be removed from the *keyname* in the dynamic file definition

RemoveKeyField takes care of all housekeeping needed when removing a key field from a dynamic file key definition, including a check to see if removing the field will remove all fields from the target key, and continues to remove the key itself if appropriate.

Example:

```
!here is how the original key was created
clear(MyKeyGrp)
MyKeygrp.KeyNbr = 2
MyKeyGrp.Label = 'kName'
MyKeyGrp.Type = 'K' ! K = Key, I = Index
MyKeyGrp.Dup = true
MyDynFile.AddKey(MyKeyGrp)
! Add the fields in the key
MyDynFile.AddFieldToKey('kName', 'LastName', true, 1)
MyDynFile.AddFieldToKey('kName', 'FirstName', true, 2)

!here is how the element is removed
TheFile &= NULL
MyDynFile.Unfixformat()

MyDynFile.SetName('other.tps')
MyDynFile.RemoveKeyField('kName', 'FirstName')
MyDynFile.RemoveField('Birthday')
MyDynFile.RemoveField('SysID')

MyDynFile.Fixformat()
TheFile &= MyDynFile.GetFileRef()

CREATE(TheFile)
```

ResetAll (clear all dynamic file values)

ResetAll()

ResetAll Clear all dynamic file properties and values.

ResetAll is a virtual method that clears all internal field, key, and memo queues that are used to hold the needed properties and values needed by the DynFile class to create the target structure. This method should be used like an initialization method before creating any new dynamic file.

Example:

```
DynFile.CreateFromFile      PROCEDURE(FILE pFile)
ndx      LONG
ndx2     LONG
Rec      LONG
Rec2     LONG
flds     LONG
AKey     &KEY

CODE
SELF.ResetAll()
SELF.StructCreated = false
SELF.sDriver = pFile{prop:Driver}
SELF.bCreate = pFile{prop:Create}
SELF.bReclaim = pFile{prop:Reclaim}
SELF.bEncrypt = pFile{prop:Encrypt}
```

SaveFile (write dynamic file contents to data source)

SaveFile(*filelabel*, *drivername*, <*namestring*>, <*ownerstring*>)

SaveFile	Save an existing dynamic file contents to another data source
<i>filelabel</i>	The label of the data source's FILE structure.
<i>drivername</i>	A string constant, variable or expression that identifies the name of the file driver to use with the data source named by the <i>filelabel</i> .
<i>namestring</i>	A string constant, variable or expression that contains the operating system device name for the structure identified by the <i>filelabel</i> to use with the data source.
<i>ownerstring</i>	A string constant, variable or expression that contains file encryption password, or SQL connection string for the data source.

SaveFile is a virtual method that can be used to load an existing table from an alternate data source.

Implementation: **SaveFile** validates that the data source is opened, then calls the **CreateFromFile**, **SetDriver**, **SetName**, and **SetOwner** methods. It then writes the contents of the dynamic file to the data source through the use of the **FillFrom** method.

Example:

```

LoadDynamicFiles    PROCEDURE ()  !Dyna-Driver Extension
ThisDyn             CLASS (DynFile)
                    END

CODE
ThisDyn.LoadFile (Department, 'TOPSPEED', 'department.tps', '')

SaveDynamicFiles    PROCEDURE ()  !Dyna-Driver Extension
ThisDyn             CLASS (DynFile)
                    END

CODE
ThisDyn.SaveFile (Department, 'TOPSPEED', 'department.tps', '')

```

See Also: FillFrom, FillTo, LoadFile

SetCreate (set dynamic file CREATE attribute)

SetCreate()

SetCreate Set the CREATE status of current dynamic file structure.

The **SetCreate** method is used to set the current value of the DynFile *bCreate* property. A value of (1) will allow the dynamic file to be created. If set to zero (0), the dynamic file created will have to match an existing file.

Implementation:

SetCreate is used to set the dynamic file CREATE attribute.

Example:

```
MEMProducts.SetCreate(true)
MEMProducts.SetName('MEMProduct')
MEMProducts.SetDriver('MEMORY')
MEMProducts.FillFrom(MSProducts)
```

SetDriver (set dynamic file DRIVER attribute)

SetDriver(*drivername*)
SetDriver(*filelabel*)

SetDriver	Set the driver type for a dynamic file
<i>drivername</i>	A string constant, variable or expression that identifies the name of the file driver to use with the dynamic file. Optionally, the <i>drivername</i> can also be a variable parameter (passed by address) if needed.
<i>filelabel</i>	The label of a FILE structure.

The **SetDriver** method is used to set the database driver used for the dynamic file. The *drivername* must contain a valid string that identifies the driver type (i.e., 'TopSpeed', 'MSSQL', etc.). See the *Database Drivers* PDF for more information.

The **SetDriver** method should also be used with the *filelabel* parameter to set the driver property when the application is linked in local mode. The file passed as a parameter can be any file defined in the application that is also using the driver needed to create the dynamic file.

Implementation: **SetDriver** sets either the PRIVATE *sDriver* or *sFileDriver* property, based on the parameter type used. The appropriate property is used in the **CreateStruct** method to initialize the DRIVER attribute of the target dynamic file. You can override the default **SetDriver** method in appropriate embed points prior to the **LoadDynamicFiles** and **SaveDynamicFiles** template generated procedures.

Example 1 (with the *drivername* parameter):

```
!use with drivername parameter
CODE

! First, let define some file attributes

MyDynFile.SetDriver('TopSpeed')
MyDynFile.SetName('TEST.TPS')
MyDynFile.SetCreate(true)
```

Example 2 (with the *filelabel* parameter):

```
!use with filelabel parameter

dummy FILE,DRIVER('MSSQL')
RECORD
END
END

MyDynFile      &DynFile
lQuery         STRING(128)

CODE

lQuery = 'SELECT * from PRODUCTS'
MyDynFile &= new(DynFile)

MyDynFile.SetDriver(dummy)
MyDynFile.SetOwner('(local),Northwind,sa,')
MyDynFile.CreateFromSQL(CLIP(lQuery))
```

SetEncrypt (set dynamic file ENCRYPT attribute)

SetEncrypt(*flag*)

SetEncrypt Encrypt a dynamic file's contents.

flag A BYTE value or EQUATE that controls the ENCRYPT attribute for a dynamic file. If set to TRUE (1), the dynamic file's contents will be encrypted.

SetEncrypt sets the internal *bEncrypt* property in the DynFile class. When the CreateFromFile or CreateStruc methods are called, this property is used to set the dynamic file's ENCRYPT attribute.

The SetOwner method must also be set properly for the **SetEncrypt** method to be valid. For more details, please refer to the *Language Reference* PDF for information regarding the dependency of OWNER with ENCRYPT.

Implementation: **SetEncrypt** sets the private *bEncrypt* property.

Example:

```
CODE

! First, let define some file attributes

MyDynFile.SetDriver('TopSpeed')
MyDynFile.SetName('TEST.TPS')
MyDynFile.SetOwner('mypassword')
MyDynFile.SetEncrypt(TRUE)
MyDynFile.SetCreate(TRUE)
```

See Also: GetEncrypt, SetOwner

SetFieldValue (set default value of field)

SetFieldValue(*fieldname*, *fieldvalue*)

SetFieldValue Enter a value into a dynamic file field

fieldname A string constant, variable, or expression that identifies the name of the dynamic file field that will be initialized with the contents of the *fieldvalue* parameter.

fieldvalue A string constant, variable, or expression that contains the contents to add to the target *fieldname*.

SetFieldValue is used to enter a default value into a dynamic file's target *fieldname*. If the field name is a numeric type, proper formatting should also be applied.

Example:

```
! Get a reference to the created file, create the file on disk and open it
TheFile &= MyDynFile.GetFileRef()
CREATE(TheFile)
OPEN(TheFile)
EMPTY(TheFile)

! Now the file is created, let add some records to it

CLEAR(TheFile)
MyDynFile.SetFieldValue('SysID', 1)
MyDynFile.SetFieldValue('FirstName', 'Bob')
MyDynFile.SetFieldValue('LastName', 'Jones')
MyDynFile.SetFieldValue('Birthday', date(05,08,1956))

ADD(TheFile)
IF ERRORCODE()
    MESSAGE('Errorcode in add = ' & errorcode())
END
```

See Also: [GetFieldValue](#)

SetName (set dynamic file NAME attribute)

SetName(*nameattribute*)

nameattribute A string constant, variable or expression that contains the operating system device name for the structure identified by the label to use with the dynamic file. Optionally, this parameter can also be a variable parameter (passed by address) if needed.

The **SetName** method is used to set the operating system device name for the structure identified by the label to use with the dynamic file.

Implementation: **SetName** sets the PRIVATE *sName* property, and is used in the **CreateStruct** method to initialize the NAME attribute of the target dynamic file structure.

Example:

```
CODE

! First, let define some file attributes

MyDynFile.SetDriver('TopSpeed')
MyDynFile.SetName('TEST.TPS')

MyDynFile.SetCreate(true)
```

SetOEM (set OEM attribute to dynamic file)

SetOEM(*flag*)

SetOEM Enable OEM translation for a dynamic file.

flag A BYTE value or EQUATE that controls the OEM attribute for a dynamic file. If set to TRUE (1), the dynamic file's will use the OEM ANSI translation.

SetOEM is used to enable OEM translation for the target dynamic file structure.

Implementation: **SetOEM** sets the private *bOEM* property, and is used in the **CreateStruct** method to initialize the OEM attribute of the target dynamic file.

Example:

```
CODE

! First, let define some file attributes

MyDynFile.SetDriver('TopSpeed')
MyDynFile.SetName('TEST.TPS')
MyDynFile.SetOEM(TRUE)
MyDynFile.SetCreate(true)
```

See Also: GetOEM

SetOwner (set owner property to Dynamic File)

SetOwner(*ownerstring*)

SetOwner	Set file encryption password, or SQL connection string for a dynamic file
<i>ownerstring</i>	A string constant, variable or expression that contains file encryption password, or SQL connection string for the dynamic file. Optionally, the <i>ownerstring</i> can also be a variable parameter (passed by address) if needed.

The **SetOwner** method is used to initialize the OWNER attribute for the target dynamic file structure. The OWNER attribute is traditionally used to encrypt the file's header, but in SQL based targets it is used to hold the connection string information.

Implementation: **SetOwner** sets the private *sOwner* property, which is used in the **CreateStruct** method to initialize the OWNER attribute of the target dynamic file.

Example:

```

CODE
ACCEPT
CASE EVENT()
  OF EVENT:Accepted
    CASE FIELD()
      OF ?Refresh1
        lQuery = lQuery1
        Do RefreshQuery
      END
    END
  END
END
END

RefreshQuery    ROUTINE
MSProducts.UnfixFormat()
MEMProducts.UnfixFormat()

MSProducts.ResetAll()
MEMProducts.ResetAll()

MSProducts.SetDriver('MSSQL')
MSProducts.SetOwner('(local),Northwind,sa,')
MSProducts.CreateFromSQL(CLIP(lQuery))

MEMProducts.SetCreate(true)
MEMProducts.SetName('MEMProduct')
MEMProducts.SetDriver('MEMORY')
MEMProducts.FillFrom(MSProducts)

TheFile &= MEMProducts.GetFileRef()
SET(TheFile)
DynFileList.Refresh(TheFile)

```

SetPrefix (set prefix value)

SetPrefix(*name*)

SetPrefix Set the label prefix for a dynamic file

name A string constant, variable or expression that identifies the label prefix to use with the dynamic file. Optionally, the *name* can also be a variable parameter (passed by address) if needed.

The **SetPrefix** method is used to set the label prefix (PRE attribute) used for the dynamic file.

Implementation: **SetPrefix** sets the private *sPrefix* property, which is used in the **CreateStruct** method to initialize the PRE attribute of the target dynamic file.

Example:

CODE

```
! First, let define some file attributes
```

```
MyDynFile.SetDriver('TopSpeed')
```

```
MyDynFile.SetPrefix('TES')
```

```
MyDynFile.SetName('TEST.TPS')
```

```
MyDynFile.SetCreate(TRUE)
```

SetReclaim (set Reclaim property to Dynamic File)

SetReclaim(*flag*)

SetReclaim Enable a dynamic file to reuse deleted record space.

flag A BYTE value or EQUATE that sets the RECLAIM attribute for a dynamic file. If set to TRUE (1), the dynamic file will reuse deleted record space.

SetReclaim is used to allow the reuse of deleted record space for the target dynamic file structure.

Implementation: **SetReclaim** sets the private *bReclaim* property, which is used in the **CreateStruct** method to initialize the RECLAIM attribute of the target dynamic file.

Example:

```
code

! First, let define some file attributes

MyDynFile.SetDriver('TopSpeed')
MyDynFile.SetName('TEST.TPS')
MyDynFile.SetReclaim(TRUE)
MyDynFile.SetCreate(true)
```

See Also: GetReclaim

Trace (send debug information)

Trace(*string*)

Trace Send the string to DEBUGVIEW.

string A string constant or variable that contains information to send to the DEBUGVIEW utility.

Trace allows the class to send debug information to DebugView. DebugView is an external industry standard program used to analyze 32-bit Windows executables.

Implementation: **Trace** can be used anytime after the DynFile class has been initialized.

Example:

```
ThisDyn Class(DynFile)
    End
DynWaitWindow WINDOW('Please Wait'),AT(,,116,18),FONT('MS Sans Serif',8,,FONT:regular),|
    CENTER,GRAY,DOUBLE
    STRING('Please Wait: Caching Tables. '),AT(0,3,116,12),USE(?DynWaitString),TRN,CENTER
    END

CODE
c = clock()
GlobalErrors.SetProcedureName('ReportDepartmentsFast')
SELF.Request = GlobalRequest ! Store the incoming request
ReturnValue = PARENT.Init()
IF ReturnValue THEN RETURN ReturnValue.
SELF.FirstField = ?Progress:Thermometer
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors !Set this windows ErrorManager to global ErrorManager
CLEAR(GlobalRequest) ! Clear GlobalRequest after storing locally
CLEAR(GlobalResponse)
OPEN(DynWaitWindow)
DISPLAY()
ThisDyn.Trace('Caching files prior to generating report')
ThisDyn.CacheFile(Employee,'TOPSPEED',,)
ThisDyn.CacheFile(LinkEmpDept,'TOPSPEED',,)
CLOSE(DynWaitWindow)
Relate:Department.SetOpenRelated()
```

UnfixFormat (close and reset Dynamic File Reference)

UnfixFormat()

UnfixFormat Clears the current structure assigned to the dynamic file.

UnfixFormat is used to close the current dynamic file reference, and clear its current structure contents.

Implementation: UnfixFormat closes and clears the dynamic file structure that was created by the **FixFormat** method.

Example:

```
MSPProducts      &DynFile
CODE
MSPProducts.FixFormat()
!...do some stuff here
MSPProducts.UnfixFormat()
```

ViewFormat (read format of active dynamic file)

ViewFormat()

ViewFormat Read the file format of an active dynamic file.

ViewFormat is virtual method used to extract the FILE structure of an active dynamic file. All elements of a Clarion FILE structure are extracted, with the exception of the file label.

Sample Format:

```

FILE, DRIVER ( ' TOPSPEED ' ) , PRE ( ) , CREATE , NAME ( ' department . tps ' )
DEP : KEY                KEY ( +DEP : DEPARTMENT ) , NOCASE , OPT , PRIMARY
DEP : DESCKEY           KEY ( +DEP : DESCRIPTION ) , NOCASE , OPT
Record                  RECORD , PRE ( )
DEP : DEPARTMENT        LONG
DEP : DESCRIPTION       STRING ( 30 )
END
```

Return Value: STRING(4096)

Implementation: **ViewFormat** should be called after the Dynamic file structure has been initialized. You should also concatenate a label name to complete the full FILE structure.

Example:

```

LoadDynamicFiles    PROCEDURE ( )    !Dyna-Driver Extension
ThisDyn             CLASS ( DynFile )
                    END

CODE
  ThisDyn.LoadFile ( Department , ' TOPSPEED ' , ' department . tps ' , '' )
  GLO:FormatString = ' DEPARTMENT ' & ThisDyn.ViewFormat ( )
!GLO:FormatString is defined as a STRING(4096)
```


Annotated Examples

Note: Each of the examples listed here are included with your Dynamic File Driver install.

Create a simple dynamic ASCII file

The first example is very easy to follow and demonstrates the FILE and KEY properties which are used to define dynamic files.

This program reads an existing FILE structure, and writes the structure to an ASCII file. The ASCII file is the dynamic file created in this example.

```

PROGRAM

MAP
DumpFileDefinition PROCEDURE(FILE f, STRING dest)
END

Employee FILE,DRIVER('TOPSPEED'),NAME('Employee.tps'),PRE(EMP),BINDABLE,CREATE,THREAD
EmpID_Key KEY(EMP:EmpID),PRIMARY
EmpName_Key KEY(EMP:Lname,EMP:Fname,EMP:Minit),DUP
JobID_Key KEY(EMP:JobID),DUP
PubID_Key KEY(EMP:PubID),DUP
DateKey KEY(-EMP:Hire_date),DUP,NOCASE,OPT
MyMemo MEMO(2000)
MyBlob BLOB,BINARY
Record RECORD,PRE()
EmpID CSTRING(10)
Fname CSTRING(21)
Minit CSTRING(2)
Lname CSTRING(31)
JobID SHORT
Job_lvl BYTE
PubID CSTRING(5)
Hire_date DATE
PictureFile STRING(65)
END

END

LineSize EQUATE(255)
FileIndent EQUATE(20)

```

The CLASS declaration contains methods that use FILE and KEY properties to extract the information from the target FILE structure (the TopSpeed file in this example). These methods are launched from the *DumpFileDefinition* method:

```

FileDumper CLASS
TheFile &FILE,PRIVATE
Dest &FILE,PRIVATE
Line ANY,PRIVATE
DumpFileDetails PROCEDURE,PRIVATE
DumpKeys PROCEDURE,PRIVATE
DumpMemosBlobs PROCEDURE,PRIVATE
DumpGroupDetails PROCEDURE(USHORT start, USHORT total),PRIVATE
DumpFieldDetails PROCEDURE(USHORT indent, USHORT FieldNo),PRIVATE
DumpToFile PROCEDURE,PRIVATE
SetAttribute PROCEDURE(SIGNED Prop,STRING Value),PRIVATE
StartLine PROCEDURE(USHORT indent,STRING label, STRING type),PRIVATE
Concat PROCEDURE(STRING s),PRIVATE
Construct PROCEDURE
Destruct PROCEDURE
DumpFileDefinition PROCEDURE(FILE f, STRING dest)
END

```

The main program contains two lines. The last line simply lets you know that it is completed.

```
CODE
```

DumpFileDefinition needs the label of the source file (*Employee*), and the destination to write the structure to (*'dump.txt'*)

```
DumpFileDefinition(Employee, 'dump.txt')
MESSAGE('Program Completed')
```

```
DumpFileDefinition PROCEDURE(FILE f, STRING dest)
CODE
  FileDumper.DumpFileDefinition(f, dest)
```

The constructor begins to create the dynamic file, setting up the ASCII file characteristics:

```
FileDumper.Construct PROCEDURE()
fGroup &GROUP
CODE
  SELF.Dest &= NEW(FILE)
  SELF.Dest{PROP:Driver} = 'ASCII'
  SELF.Dest{PROP:Create} = TRUE
  SELF.Dest{PROP:Type, 1} = 'STRING'
  SELF.Dest{PROP:Size, 1} = LineSize
```

FIXFORMAT sets up the structure in memory. It does not CREATE or OPEN the file.

```
FIXFORMAT(SELF.Dest)
ASSERT(ERRORCODE()=0, 'FixFormat failed with error ' & FILEERRORCODE())
fGroup &= SELF.Dest{PROP:Record}
SELF.line &= WHAT(fGROUP, 1)
```

At program end, the destructor disposes the file object created by the NEW statement:

```
FileDumper.Destruct PROCEDURE()
CODE
  DISPOSE(SELF.Dest)
  SELF.line &= NULL
```

The dynamic file is opened, created if needed, and then the properties of the TopSpeed file are extracted and saved to the ASCII file in sequence.

```
FileDumper.DumpFileDefinition PROCEDURE(FILE f, STRING dest)
CODE
  SELF.Dest{PROP:Name} = dest
  SELF.TheFile &= f
  OPEN(SELF.Dest)
  IF ERRORCODE()
    CREATE(SELF.Dest)
    OPEN(SELF.Dest)
  END
  ASSERT(ERRORCODE()=0, 'OPEN Dest failed with error ' & ERROR())

  SELF.DumpFileDetails
  SELF.DumpKeys
  SELF.DumpMemosBlobs
  SELF.DumpGroupDetails(0, F{PROP:Fields})
  SELF.StartLine(FileIndent, '', 'END')
  SELF.DumpToFile
  CLOSE(SELF.Dest)
```

Here we see the properties that you can use to create a dynamic file.

```

FileDumper.DumpFileDetails PROCEDURE
CODE
  SELF.StartLine(FileIndent, 'Employee', 'FILE')
  SELF.Concat(',DRIVER('' & CLIP(SELF.TheFile{PROP:Driver}))

  IF SELF.TheFile{PROP:DriverString}
    SELF.Concat(',') & CLIP(SELF.TheFile{PROP:DriverString}))
  END
  SELF.Concat('')')
  SELF.SetAttribute(SELF.TheFile{PROP:Create}, 'CREATE')
  SELF.SetAttribute(SELF.TheFile{PROP:Reclaim}, 'RECLAIM')
  IF SELF.TheFile{PROP:Owner}
    SELF.Concat(',OWNER('' & CLIP(SELF.TheFile{PROP:Owner}) & '''))
  END
  SELF.SetAttribute(SELF.TheFile{PROP:Encrypt}, 'ENCRYPT')
  SELF.Concat(',NAME('' & CLIP(SELF.TheFile{PROP:Name}) & '''))
  SELF.SetAttribute(SELF.TheFile{PROP:Thread}, 'THREAD')
  SELF.SetAttribute(SELF.TheFile{PROP:OEM}, 'OEM')

  SELF.DumpToFile

FileDumper.DumpMemosBlobs PROCEDURE
x UNSIGNED,AUTO
CODE
  LOOP X = 1 TO (SELF.TheFile{PROP:Memos} + SELF.TheFile{PROP:Blobs})
    IF UPPER (SELF.TheFile{PROP:type, -X}) = 'MEMO'
      SELF.StartLine(FileIndent+2, SELF.TheFile{PROP:label, -X}, 'MEMO(')
      SELF.Concat(CLIP(SELF.TheFile{PROP:Size, -X})&')')
    END
    IF UPPER (SELF.TheFile{PROP:type, -X}) = 'BLOB'
      MESSAGE('BLOB FOUND')
      SELF.StartLine(FileIndent+2, SELF.TheFile{PROP:label, -X}, 'BLOB')
    END
    SELF.SetAttribute(SELF.TheFile{PROP:Binary,-X}, 'BINARY')
    IF SELF.TheFile{PROP:Name, -X}
      SELF.Concat(',NAME('' & CLIP(SELF.TheFile{PROP:Name, -X}) & '''))
    END
    SELF.DumpToFile
    MESSAGE('MEMO WRITE')
  END

FileDumper.DumpKeys PROCEDURE
x UNSIGNED,AUTO
y UNSIGNED,AUTO
aKey &KEY
CODE
  LOOP x = 1 TO SELF.TheFile{PROP:Keys}
    AKey &= SELF.TheFile{PROP:Key, x}
    SELF.StartLine(FileIndent+2, AKey{PROP:label}, AKey{PROP:Type})
    SELF.Concat('')
    LOOP y = 1 TO AKey{PROP:Components}
      IF y > 1 THEN SELF.Concat(',').
      IF AKey{PROP:Ascending, y}
        SELF.Concat('+')
      ELSE
        SELF.Concat('-')
      END
    END
    SELF.Concat(SELF.TheFile{PROP:Label, akey{PROP:Field, y}})
  END
  SELF.Concat('')')
  SELF.SetAttribute(AKey{PROP:Dup}, 'DUP')
  SELF.SetAttribute(AKey{PROP:NoCase}, 'NOCASE')
  SELF.SetAttribute(AKey{PROP:Opt}, 'OPT')
  SELF.SetAttribute(AKey{PROP:Primary}, 'PRIMARY')
  IF AKey{PROP:Name}

```

```

        SELF.Concat(',NAME('' & CLIP(AKey{PROP:Name}) & '''))
    END
    SELF.DumpToFile
END

```

```
FileDumper.DumpGroupDetails PROCEDURE(USHORT start, USHORT total)
```

```

fld          USHORT,AUTO
fieldsInGroup USHORT,AUTO
GroupIndent  USHORT,STATIC,AUTO
CODE
    IF start = 0 THEN
        GroupIndent = FileIndent+2
        SELF.StartLine(GroupIndent, 'RECORD', 'RECORD')
        SELF.DumpToFile
    END
    GroupIndent += 2
    LOOP fld = start+1 TO start+total
        SELF.DumpFieldDetails(GroupIndent, fld)
        IF SELF.TheFile{PROP:Type,fld} = 'GROUP'
            fieldsInGroup = SELF.TheFile{PROP:Fields,fld}
            SELF.DumpGroupDetails (fld, fieldsInGroup)
            fld += fieldsInGroup
        END
    END
    GroupIndent -= 2
    SELF.StartLine(GroupIndent, '', 'END')
    SELF.DumpToFile

```

```
FileDumper.DumpFieldDetails PROCEDURE(USHORT indent, USHORT FieldNo)
```

```

FldType  STRING(20),AUTO
CODE
    FldType = SELF.TheFile{PROP:Type,FieldNo}
    SELF.StartLine(indent, SELF.TheFile{PROP:Label, FieldNo}, FldType)
    IF INSTRING('STRING', FldType, 1, 1)
        SELF.Concat('')
        IF SELF.TheFile{PROP:Picture, FieldNo}
            SELF.Concat(SELF.TheFile{PROP:Picture, FieldNo})
        ELSE
            SELF.Concat(SELF.TheFile{PROP:Size, FieldNo})
        END
        SELF.Concat('')
    ELSIF INSTRING('DECIMAL', FldType, 1, 1)
        SELF.Concat('(' & SELF.TheFile{PROP:Size, FieldNo} & ',' & |
            SELF.TheFile{PROP:Places, FieldNo} & ')')
    END
    IF SELF.TheFile{PROP:Dim, FieldNo} <> 0
        SELF.Concat(',DIM(' & CLIP(SELF.TheFile{PROP:Dim, FieldNo}) & ')')
    END
    IF SELF.TheFile{PROP:Over, FieldNo} <> 0
        SELF.Concat(',OVER(')
        IF SELF.TheFile{PROP:Label, SELF.TheFile{PROP:Over, FieldNo}}
            SELF.Concat(CLIP(SELF.TheFile{PROP:Label, SELF.TheFile{PROP:Over, FieldNo}}))
        ELSE
            SELF.Concat('field ' & SELF.TheFile{PROP:Over, FieldNo})
        END
        SELF.Concat(')')
    END
    IF SELF.TheFile{PROP:Name, FieldNo}
        SELF.Concat(',NAME('' & CLIP(SELF.TheFile{PROP:Name, FieldNo}) & '''))
    END
    SELF.DumpToFile

```

```
FileDumper.SetAttribute PROCEDURE (Prop,Value)
CODE
  IF Prop THEN SELF.line = CLIP(SELF.line) & ',' & CLIP(Value).

FileDumper.StartLine PROCEDURE (USHORT indent,STRING label, STRING type)
spaces USHORT,AUTO
clen LONG,AUTO
CODE
  SELF.line = label
  clen = LEN(CLIP(SELF.line))
  IF clen < Indent
    spaces = Indent - clen
  ELSE
    spaces = 4
  END
  SELF.line = CLIP(SELF.line) & ALL(' ', spaces) & type

FileDumper.Concat PROCEDURE (STRING s)
CODE
  SELF.line = CLIP(SELF.line) & s

FileDumper.DumpToFile PROCEDURE
CODE
  ADD(SELF.Dest)
  ASSERT(ERRORCODE()=0, 'Could not add to dump file: ' & ERROR())
```


Load a dynamic SQL or ISAM structure into a Virtual List Box (VLB)

This next example is much more useful and powerful than the previous one.

This program has SQL queries stored in a queue, and one static ISAM file definition. All of these parameters are used to create a dynamic file, which is then loaded into a virtual list box, or VLB (a list box that is created and formatted at run time).

Although this program is an excellent tutorial to learn how to create VLBs, our annotated example comments will only focus on the dynamic file driver elements.

```

PROGRAM
MAP
FillVirtualListBox      PROCEDURE ()
END

```

To have access to the DynFile Class method, this include is needed:

```
INCLUDE('DynFile.inc'), ONCE
```

```

CODE
FillVirtualListBox()

```

The FillVirtualListBox procedure begins with the ISAM FILE and CLASS declarations:

```

FillVirtualListBox      procedure

People                  FILE, DRIVER('TOPSPEED'), PRE(PEO), CREATE, BINDABLE, THREAD
KeyId                  KEY(PEO:Id), NOCASE, OPT, PRIMARY
KeyLastName            KEY(PEO:LastName), DUP, NOCASE
Record                 RECORD, PRE()
Id                     LONG
FirstName              STRING(30)
LastName               STRING(30)
Gender                 STRING(1)
                        END
                        END

MSProducts             &DynFile
MEMProducts            &DynFile

TheFile                &File
TheKey                 &Key

MyQueue                QUEUE
productID              LONG
ProductName             STRING(25)
                        END

```

A class is declared here to simplify processing the list box format strings and refreshing when needed.

```

!=== VLB =====
DynFileList CLASS
Changed          BYTE
ListControl      USHORT
File             &File
Key              &Key
Init             PROCEDURE(UNSIGNED TheList)
Refresh          PROCEDURE(*FILE TheDynFile), VIRTUAL
FormatColumn     PROCEDURE(STRING Label, STRING DataType, |
                        USHORT DataSize, BYTE Places), STRING, VIRTUAL
VLBProc          PROCEDURE(LONG ROW, SHORT COL), STRING, VIRTUAL, PROC
END

```

This is the queue to hold the default SQL queries:

```

QueryQueue  QUEUE
Query       CSTRING(200)
           END

lQuery      CSTRING(200)
INIEntries  LONG
INIEntriesIndex LONG
lFound      BYTE

Window WINDOW('Dynamic File Driver FillVirtualListBox'),AT(, ,320,207),|
           FONT('MS Sans Serif', , ,FONT:regular,CHARSET:ANSI), |
           SYSTEM,GRAY,DOUBLE
           PROMPT('Query: '),AT(8,5),USE(?Query:Prompt1)
           COMBO(@s200),AT(33,5,215,10),USE(?Query:Combo),FORMAT('200L(2)|M@S200@'),|
           DROP(10,400),FROM(QueryQueue)
           BUTTON('From People.TPS'),AT(224,24,78,14),USE(?FromPeople)
           BUTTON('Refresh'),AT(251,2,51,14),USE(?Refresh)
           LIST,AT(6,41,308,132),USE(?List1),VSCROLL
           BUTTON('Close'),AT(259,181,51,14),USE(?Button1),STD(STD:Close)
           END

```

CODE

First Load any existing queries from an INI file:

```
DO LoadQueryList
```

Next, initialize a default list of Queries if the INI file is empty:

```

IF RECORDS(QueryQueue)=0
  QueryQueue.Query='SELECT ProductID , ProductName FROM Products ORDER BY ProductName'
  ADD(QueryQueue)
  QueryQueue.Query='SELECT ProductName,ProductID FROM Products ORDER BY ProductName'
  ADD(QueryQueue)
  QueryQueue.Query='SELECT ProductName FROM Products ORDER BY ProductName'
  ADD(QueryQueue)
  QueryQueue.Query='SELECT COUNT(*) Records FROM Products'
  ADD(QueryQueue)
END

```

Create the DynFile objects, to allow access to the methods:

```

MSProducts &= NEW(DynFile)
MEMProducts &= NEW(DynFile)

OPEN(window)
GET(QueryQueue,1)
?Query:Combo{PROP:ScreenText}=QueryQueue.Query
(?Query:Combo{PROP>ListFeq}){PROP:SELECTED}=1
DynFileList.Init(?List1)
ACCEPT
CASE EVENT()
OF EVENT:Accepted
CASE FIELD()

```

If the Refresh button is pressed, we need to make sure that there is a valid Query string before we attempt to create the Dynamic File

```

OF ?Refresh
lFound = FALSE
!Search for Combo text in the queue
LOOP INIEntriesIndex=1 TO RECORDS(QueryQueue)
GET(QueryQueue,INIEntriesIndex)
IF NOT ERRORCODE()
  IF ?Query:Combo{PROP:VALUE}=QueryQueue.Query
    lFound = TRUE
    !Move query to first place
    DELETE(QueryQueue)

```

```

    QueryQueue.Query=?Query:Combo{PROP:VALUE}
    ADD (QueryQueue,1)
    (?Query:Combo{PROP:ListFeq}) {PROP:SELECTED}=1
    BREAK
  END
END
END
!If the combo text was not found in the queue add it
IF lFound=False
  QueryQueue.Query = ?Query:Combo{PROP:VALUE}
  ADD (QueryQueue,1)
END
lQuery = QueryQueue.Query

```

Call the ROUTINE to create the dynamic file after getting a valid query:

```
DO RefreshQuery
```

The “People ISAM” button uses a slightly different approach:

```
OF ?FromPeople
```

Unfix any prior structure attached to the dynamic file

```
MEMProducts.UnfixFormat()
```

Reset the method’s internal queues

```
MEMProducts.ResetAll()
```

This method reads the “people” structure, and calls FIXFORMAT to establish the new structure

```
MEMProducts.CreateFromFile(people)
```

The next two methods change the name attribute and driver type before opening and processing the dynamic file

```
MEMProducts.SetName('MEMProduct')
MEMProducts.SetDriver('MEMORY')
```

The FillFrom method opens the new “In-Memory” dynamic file, and loads the contents of the people file into it.

```
MEMProducts.FillFrom(people)
```

Finally we use assign the file refernce to the file used in the VLB:

```
TheFile &= MEMProducts.GetFileRef()
DynFileList.Refresh(TheFile)
END
END
END
```

Always DISPOSE objects created with NEW!!

```
DISPOSE (MSPProducts)
DISPOSE (MEMProducts)
```

```
Do SaveQueryList
```

Standard INI processing routines are here:

```
SaveQueryList ROUTINE
INIEntries = GETINI('QUERYLIST','QUERYRECORDS',0,'.\TestVLBFill.INI')
LOOP INIEntriesIndex=1 TO INIEntries
  PUTINI('QUERYLIST','QUERY'&INIEntriesIndex,'','.\TestVLBFill.INI')
END
PUTINI('QUERYLIST','QUERYRECORDS',RECORDS(QueryQueue),'.\TestVLBFill.INI')
LOOP INIEntriesIndex=1 TO RECORDS(QueryQueue)
  GET(QueryQueue,INIEntriesIndex)
  IF NOT ERRORCODE()
    PUTINI('QUERYLIST','QUERY'&INIEntriesIndex,QueryQueue.Query,'.\TestVLBFill.INI')
  END
END
```

```

LoadQueryList    ROUTINE
INIEntries = GETINI( 'QUERYLIST', 'QUERYRECORDS', 0, '.\TestVLBFill.INI')
FREE(QueryQueue)
LOOP INIEntriesIndex=1 TO INIEntries
  QueryQueue.Query=GETINI( 'QUERYLIST', 'QUERY' & INIEntriesIndex, ' ', '.\TestVLBFill.INI')
  ADD(QueryQueue)
END

```

The RefreshQuery ROUTINE creates two dynamic files. The MSProducts dynamic file is MSSQL, and needs to be created so that the CreateFromSQL method can correctly construct the table using PROP:SQL.

The second dynamic file loads the contents of the first into a memory file that is used with the VLB.

```

RefreshQuery    ROUTINE
MSProducts.UnfixFormat()
MEMProducts.UnfixFormat()

MSProducts.ResetAll()
MEMProducts.ResetAll()

MSProducts.SetDriver('MSSQL')
MSProducts.SetOwner('(local),Northwind,sa,')
MSProducts.CreateFromSQL(CLIP(lQuery))

MEMProducts.SetCreate(true)
MEMProducts.SetName('MEMProduct')
MEMProducts.SetDriver('MEMORY')
MEMProducts.FillFrom(MSProducts)
TheFile &= MEMProducts.GetFileRef()
DynFileList.Refresh(TheFile)

DynFileList.Init                                PROCEDURE (UNSIGNED TheList)
CODE
SELF.File &= NULL
SELF.ListControl = TheList
SELF.ListControl{PROP:VLBVal} = ADDRESS(SELF)
SELF.ListControl{PROP:VLBProc} = ADDRESS(VLBProc)
SELF.Changed = False

DynFileList.Refresh                            PROCEDURE (*FILE TheDynFile)
lColumns    SHORT
lFormat     CSTRING(2048),AUTO
Idx         SHORT
CODE
SELF.File &= TheDynFile
IF NOT SELF.File &= NULL
  lColumns = SELF.File{PROP:Fields}
  lFormat = ''
  SELF.ListControl{PROP:FORMAT}=CLIP(lFormat)
  LOOP Idx=1 TO lColumns
    lFormat=CLIP(lFormat) & SELF.FormatColumn(SELF.File{PROP:Label, Idx}, |
      SELF.File{PROP:Type, Idx}, SELF.File{PROP:Size, Idx}, SELF.File{PROP:Places, Idx})
  END
  SELF.ListControl{PROP:FORMAT}=CLIP(lFormat)
  SELF.Changed = True
ELSE
  SELF.Changed = False
END

```

```

DynFileList.VLBProc                                PROCEDURE (LONG ROW, SHORT COL)
AttrIndex    UNSIGNED, AUTO
AttrString   ANY
locGroup     &GROUP
lColumns     SHORT
lFormat      STRING(2048),AUTO
lRecords     LONG

CODE
CASE ROW
OF -1
  IF NOT SELF.File &= NULL
    SET(TheFile)
    lRecords=RECORDS(SELF.File)
    RETURN lRecords
  ELSE
    RETURN 0
  END
OF -2
  IF NOT SELF.File &= NULL
    RETURN SELF.File{PROP:Fields}
  ELSE
    RETURN 1
  END
OF -3
  IF NOT SELF.File &= NULL
    IF SELF.Changed
      SELF.Changed = False
      RETURN TRUE
    END
  END
  RETURN FALSE
ELSE
  IF NOT SELF.File &= NULL
    locGroup &= SELF.File{prop:Record}
    GET(SELF.File, ROW)
    IF NOT ERRORCODE()
      AttrString &= WHAT(locGroup, COL)
      RETURN AttrString
    ELSE
      MESSAGE(ERRORCODE())
    END
  ELSE
    RETURN ''
  END
END
RETURN ''

DynFileList.FormatColumn                          PROCEDURE (STRING Label,STRING DataType,USHORT
DataSize,BYTE Places)
lJustification  STRING('L')
lWidth          USHORT(0)
lIndent         BYTE(2)
lPicture        CSTRING(20)
CODE
CASE DataType
OF 'BYTE'
  lPicture = 'n3'
  lWidth   = LEN(Label)*5
OF 'SHORT'
  lPicture = 'n-7'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'USHORT'
  lPicture = 'n6'
  lWidth   = LEN(Label)*5
  lJustification='R'

```

```

OF 'DATE'
  lPicture = 'd17'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'TIME'
  lPicture = 't7'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'LONG'
  lPicture = 'n-14'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'ULONG'
  lPicture = 'n13'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'SREAL'
  lPicture = 'n10.2'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'REAL'
  lPicture = 'n10.2'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'DECIMAL'
  lPicture = 'n10.2'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'PDECIMAL'
  lPicture = 'n10.2'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'BFLOAT4'
  lPicture = 'n10.2'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'BFLOAT8'
  lPicture = 'n10.2'
  lWidth   = LEN(Label)*5
  lJustification='R'
OF 'STRING'
  lPicture = 's'&DataSize
  lWidth   = DataSize*5
OF 'CSTRING'
  lPicture = 's'&(DataSize+1)
  lWidth   = DataSize*5
OF 'PSTRING'
  lPicture = 's'&(DataSize+1)
  lWidth   = DataSize*5
OF 'MEMO'
  lPicture = 's250'
  lWidth   = 80
! OF 'GROUP'
! OF 'BLOB'
ELSE
END
RETURN CLIP(lWidth&lJustification&' ('&lIndent&')|M~'&Label&'~@'&lPicture&'@')

```

Extended ERRORCODES

If an ERRORCODE value of 47 is returned ('Invalid File Declaration') during any attempted access of a dynamic file structure, you can use the FILEERRORCODE statement to check for more information.

FILEERRORCODE returns a string in the following format

"obj : index : property : errorcode"

obj is one of SYSTEM, FILE, FIELD, MEMO/BLOB, KEY, or COMPONENT

index the ordinal of the object in the parent (1 based).

property the name of the property that is invalid

errorcode the format *Snumber* or *Dnumber*

The next few pages display a table of extended error codes, with their corresponding CLAMSG number, default error message, and notes

Dnumber indicates a **driver specific** error code

Snumber indicates a **general** or **structural** error code

Structural Errors - Snumber codes

Below is a table of *Snumber* error codes, with their corresponding CLAMSG number, default error message and notes

<u>Object</u>	<u>ErrorCode</u>	<u>CLAMSG number</u>	<u>Error Message</u>	<u>Notes</u>
SYSTEM	S00001	1000001	Internal Error: Property cannot be set	This indicates a fatal error please report it to SoftVelocity.
	S00002	1000002	Dynamic File Support Not Found	You are trying to change the driver of a static file without having the Dynamic File Driver support library present. This error code is returned by file{PROP:Driver} = 'value'.
FILE	S00001	1100001	No File Driver Specified	
	S00002	1100002	File Driver could not be loaded	The system could not load the file driver DLL. Probably because it is not on the path
	S00003	1100003	The DLL is not a valid file driver	This normally indicates a corrupt file driver
	S00004	1100004	File Driver not defined	The value specified in file{PROP:Driver} does not match any know driver. If you have a third party file driver you may need to add an entry to the list of drivers stored in the windows registry at "HKEY_LOCAL_MACHINE\Software\SoftVelocity\AnyDriver\C60"
	S00005	1100005	The record structure is greater than 4,194,304 bytes in size	
FIELD	S00001	1200001	Not a Valid Clarion Label	
	S00002	1200002	Duplicate Label	
	S00003	1200003	No Type Specified	
	S00004	1200004	Type requires a size and no size specified	
	S00005	1200005	The Group has more fields than has been defined	You have not defined enough fields in the file to fill the group based on the file{PROP:Fields, n} value set for this group
	S00006	1200006	More Decimal Places than specified size	
	S00007	1200007	Size is too big	
	S00008	1200008	Field Count not specified for a GROUP field	

	S00009	1200009	Field must be over a field defined before the field	
	S00010	1200010	The Group is greater than 4,194,304 bytes in size	
	S00011	1200011	The Field is greater than 4,190,208 bytes in size	
	S00012	1200012	Field is larger than the field that it is over	
	S00013	1200013	Invalid Picture	
	S00014	1200014	Field cannot be over a field inside another GROUP	
KEY	S00001	1300001	Not a Valid Clarion Label	
	S00002	1300002	Duplicate Label	
	S00003	1300003	Type set to 'KEY', but no fields specified	
	S00004	1300004	Key is defined as PRIMARY and DUP	
COMPONENT	S00001	1400001	Component number is greater than the number of fields in the file structure	
	S00002	1400002	Component is a dimensioned field	
MEMO/BLOB	S00001	1300001	Not a Valid Clarion Label	
	S00002	1300002	Duplicate Label	
	S00003	1500003	No Size Specified for a MEMO	

File Driver Specific Errors - Dnumber Codes

Below is a table of *Dnumber* error codes, with their corresponding CLAMSG number, default error message

Object	ErrorCode	CLAMSG number	Error Message
FILE	D00001	2100001	Component does not match physical file structure
FIELD	D00001	2200001	Component does not match physical file structure
	D00002	2200002	No Name or Label Specified
	D00003	2200003	Data type not supported
	D0004	2200004	Number of dimensions is greater than that supported by the file driver
	D0005	2200005	Field is larger than the maximum supported by the file driver
KEY	D00001	2300001	Component does not match physical file structure
	D00002	2300002	No Name or Label Specified
	D00003	2300003	Too Many Keys defined
COMPONENT	D00001	2400001	Component does not match physical file structure
	D00002	2400002	Driver does not support Descending Key Components
	D00003	2400003	Driver does not support mixed Ascending and Descending Key Components
MEMO/BLOB	D00001	2500001	Component does not match physical file structure
	D00002	2500002	No Name or Label Specified
	D00003	2500003	Too Many Memos defined
	D00004	2500004	Memo is larger than the maximum size supported by the file driver
	D00005	2500005	Driver does not support BLOBs
	D00006	2500006	Driver does not support MEMOs
	D00007	2500007	Driver does not support BINARY memos or blobs

Index:

AddField	28	GetPrefix	49
AddFieldToKey	29	GetReclaim	49
AddKey	30	Load File	20
AddMemo	31	LoadDynamicFiles	20
CacheFile	32	LoadFile	51
CreateFromFile	33	local link	25
CreateFromSQL	34	local mode	20, 58
CreateKeyComponents	35	Register	
CreateStruct	36	Templates	19
Driver		RemoveField	52
Dynamic File	5	RemoveKey	53
DynaDriver		RemoveKeyField	54
defined	19	ResetAll	55
Dynamic File Driver		SaveDynamicFiles	20
Uses	5	SaveFile	56
Dynamic File Driver Global Extension	20	SetCreate	57
DynFile Class	27	SetDriver	58
Methods	28	SetEncrypt	59
Properties	27	SetFieldValue	60
Examples	69	SetName	61
FillFrom	37	SetOEM	61
FillTo	39	SetOwner	62
FixFormat	40	SetPrefix	63
FIXFORMAT	6, 13	SetReclaim	64
GetCreate	42	Setup	5, 6
GetCreatedFromSQL	50	support templates	19
GetDriver	42	TFieldGrp	28
GetEncrypt	42	Trace	65
GetField	43	UnfixFormat	66
GetFieldName	44	UNFIXFORMAT	13
GetFieldValue	45	Uses	
GetFileRef	46, 47	of DFD	5
GetName	48	of templates	25
GetOEM	48	View Format	67
GetOwner	48		

