V12 Database Engine 3

For Macromedia Authorware

User Manual

integration
new media

Contents

License agreement

PLEASE READ THIS LICENSE AGREEMENT CAREFULLY BEFORE USING V12 DATABASE ENGINE. BY USING V12 DATABASE ENGINE, YOU AGREE TO BECOME BOUND BY THE TERMS OF THIS LICENSE AGREEMENT.

The enclosed computer program(s), license file, manuals, sample files and data (collectively, "Software") are licensed, not sold, to you by Integration New Media, Inc. ("INM") for the purpose of using it for the development of your own product ("Product") only under the terms of this Agreement. INM and its licensors reserve any rights not expressly granted to you. INM and its licensors grant you no right, title or interest in or to the Software. The Software is owned by INM and its licensors and is protected by International copyright laws and treaties.

If you are using V12-Tracker, Section 2 applies to you.

1. **License**. INM grants you a non-exclusive, non-transferable, perpetual (unless terminated in accordance with this Agreement), royalty-free, worldwide license to:

(a) Install one (1) copy of the Software on a single computer. To "install" the Software means that the Software is either loaded or installed on the permanent memory of a computer (i.e., hard disk). This installed copy of the Software may be accessible by multiple computers, however the Software cannot be installed on more than one computer at any time. You may only install the Software on another computer if you first remove the Software from the computer on which it was previously installed.

(b) Make one copy of the Software in machine-readable form solely for backup purposes. As an express condition of this Agreement, you must reproduce on each copy any copyright notice or other proprietary notice that is or included with the original copy supplied by INM.

(c) Reproduce and distribute the files named "V12-DBE for Authorware.XTR", "V12-DBE for Authorware.X32" and "V12DBE-A.X16" (collectively, "Runtime Kit") provided that: (i) you distribute the Runtime Kit only in conjunction with and as part of your own Product, and (ii) own a license for the Software that contains the Runtime Kit.

(d) Any third party who may distribute or otherwise make available a product containing the Runtime Kit must purchase its own license of the Software.

(e) Any third party who will use the Runtime Kit in an authoring environment must purchase its own license of the Software.

(f) If the Software is licensed as an upgrade or update, then you may only use the Software to replace previously validly licensed versions of the same software. You agree that the upgrade or update does not constitute the granting of a second license to the Software (i.e., you may not use the upgrade or update in addition to the software it is replacing, nor may you transfer the software which is being replaced to a third party).

(g) All intellectual property rights in and to the assets which may be accessed through the use of the Software are the property of the respective asset owners and may be protected by applicable copyright or other intellectual property laws and treaties. This Agreement grants you no rights to use such content.

2. **V12-Tracker waiver.** V12-Tracker is a fully functional version of V12 Database Engine and a fully licensed V12 database file named V12-Tracker.V12. You are waived from section 1(c)(ii) provided that:

(a) Your Product uses V12-Tracker.

(b) Your Product does not use a V12 database other than V12-Tracker.

(c) The V12-Tracker.V12 database file has the exact same structure as the one delivered to you by INM.

3. **Restrictions.**

(a) The Software contains a license file (.LIC) that may not be distributed by you in any way.

(b) You may not sublease, rent, loan or lease the Software.

(c) You may not transfer or assign your rights under this License to another party without INM's prior written consent. Assignment application forms can be obtained from INM's sales department.

(d) The Software contains trade secrets and, to protect them, YOU MAY NOT MODIFY, ADAPT, TRANSLATE OR CREATE DERIVATIVE WORKS BASED UPON THE SOFTWARE OR ANY PART THEREOF. YOU MAY NOT REVERSE ENGINEER, DECOMPILE, DISASSEMBLE OR OTHERWISE REDUCE THE SOFTWARE TO ANY HUMAN PERCEIVABLE FORM. YOU MAY NOT ALTER OR CHANGE THE COPYRIGHT NOTICES AS CONTAINED IN THE SOFTWARE.

(e) THE SOFTWARE IS NOT INTENDED FOR USE IN THE OPERATION ENVIRONMENTS IN WHICH THE FAILURE OF THE SOFTWARE COULD LEAD TO DEATH, PERSONAL INJURY, OR PHYSICAL OR ENVIRONMENTAL DAMAGE.

4. **Copyright notices**. You may not alter or change INM's and its licensors' copyright notices as contained in the Software. As well, you must include:

(a) a copyright notice, in direct proximity to your own copyright notice, in substantially the following form: "Portions of code are Copyright ©1995-2003 used under license by Integration New Media, Inc."; and

(b) the "Powered by V12" logo on the packaging of your Product or place the "Powered by V12" logo within your Product in the credits section.

5. **Acceptance**. The Software shall be deemed accepted by you upon delivery unless you provide INM, within two (2) weeks therein, with a written description of any bona fide defects in material or workmanship.

6. **Termination**. This Agreement is effective until terminated. This Agreement will terminate immediately without notice from INM or judicial resolution if you fail to comply with any provision of this Agreement. Upon such termination you must destroy the Software, all accompanying written materials and all copies thereof, and Sections 7, 8, 9 and 10 will survive any termination.

7. **Limited Warranty**. INM warrants for a period of ninety (90) days from your date of purchase (as evidenced by a copy of your receipt) that the media on which the Software is recorded will be free from defects in materials and workmanship under normal use and the Software will perform substantially in accordance with the user manual. INM's entire liability and your sole and exclusive remedy for any breach of the foregoing limited warranty will be, at INM's option, replacement of the disk, refund of the purchase price or repair or replacement of the Software.

8. **Limitation of Remedies and Damages**. In no event will INM or its licensors, directors, officers, employees or affiliates of any of the foregoing be liable to you for any consequential, incidental, indirect or special damages whatsoever (including, without limitation, loss of expected savings, loss of confidential information, presence of viruses, damages for loss of profits, business interruption, loss of business information and the like), whether foreseeable or not, arising out of the use of or inability to use the Software or accompanying materials, regardless of the basis of the claim and even if INM or an INM representative has been advised of the possibility of such damage. INM's liability to you for direct damages for any cause whatsoever, and regardless of the form of the action, will be limited, at INM's option to refund of the purchase price or repair or replacement of the Software.

THIS LIMITATION WILL NOT APPLY IN CASE OF PERSONAL INJURY ONLY WHERE AND TO THE EXTENT THAT APPLICABLE LAW REQUIRES SUCH LIABILITY. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

9. **Indemnity**. By using the Software, you agree to indemnify, hold harmless and defend INM and its licensors from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your Product with the Software

10. **General**. This Agreement will be construed under the laws of the Province of Quebec, except for that body of law dealing with conflicts of law. If any provision of this Agreement shall be held by a court of competent jurisdiction to be contrary to law, that provision will be enforced to the maximum extent permissible, and the remaining provisions of this Agreement will remain in full force and effect.

11. **Language**. The parties acknowledge having requested and being satisfied that this Agreement and its accessories be drawn in English. Les parties reconnaissent avoir demandé que cette entente et ses documents connexes soient rédigés en anglais et s'en déclarent satisfaits.

Introduction

Welcome to V12 Database Engine (V12-DBE), the most versatile and user-friendly cross-platform database management Xtra for Macromedia Authorware® and Director®.

V12-Database Engine for Authorware

V12-Database Engine (DBE) was originally designed in 1996 to be used specifically with Director and Authorware. It extends Director and Authorware's features and helps you speed up the development of your multimedia titles.

You will discover many benefits in using V12-DBE to create multimedia applications, such as electronic storybooks, training material, games and more. Used as a back-end to your multimedia projects, V12-DBE allows you to store and manage content separately from the Authorware development platform.

V12 Database Engine helps you provide advanced functionality to your end-users while bringing down your development and maintenance costs.

V12-DBE is very flexible and scalable and can be used in a wide range of applications; from simple projects where Script Lists and FileIO have become difficult to manage, to true database-driven applications.

Where to start

Before browsing through this User Manual, we recommend that you look at the **First Steps** manual, which includes a tutorial, designed to help you get up and running with V12-DBE in a few short steps. The First Steps tutorial is available for download at: http://www.integrationnewmedia.com/support/v12authorware/manuals/.

You may also benefit from browsing through the Authorware pieces available in the **Demos** section of our website. As with the First Steps tutorial, these sample Authorware pieces are designed to help you understand V12-DBE's various features, but in more depth. Download them at: http://www.IntegrationNewMedia.com/products/v12Authorware/Demos/. Please make sure you understand V12 Database Engine's license agreement before proceeding. The full license agreement is at the beginning of this user manual (see License agreement).

The V12-DBE Knowledge Object library features only the basic essential features of V12-DBE's functionality. Before you commit yourself to using the V12-DBE Knowledge Object in your project, you may first want to ask Support@IntegrationNewMedia.com or other V12-DBE users on V12-L (http://www.IntegrationNewMedia.com/support/list/) for advice.

Do I really need to master scripting to use V12-DBE?

How comfortable do you need to be with scripting to use V12-DBE efficiently? The answer varies according to the complexity of your projects.

Simple projects require no knowledge of scripting at all. If your project uses a single database and shows one record at a time on Authorware's stage, chances are you can implement it using the *V12-DBE Knowledge Objects Library* only. If you don't need any more functionality than the Knowledge Objects library provides, then no script programming will be required. See *Free tools*, below.

For more advanced projects, V12 Database Engine's comprehensive scripted interface requires knowledge of scripts, but it provides as much guidance as possible when programming, such as checking the number of parameters, the types of the parameters, etc.

In a nutshell, before delving into V12-DBE you need to know the following basic scripting concepts:

- Variables.
- Control structures (`if` statements, `for` loops `repeat` loops, etc.).
- Object instances (this is covered in detail later in the Using Xtras section of this manual).

Free tools

If you are looking for a fast and easy way to integrate V12-DBE into your multimedia projects, you will want to check out the V12-DBE Tool and V12-DBE Knowledge Objects library. The V12-DBE Tool helps you perform V12 database management tasks, such as database creation, viewing, editing, importing, exporting, and more. The V12-DBE Knowledge Objects library includes a set of Authorware Knowledge Objects that allow you to quickly implement the most common of V12-DBE's functionalities. Both of these products are available for download from the Free Tools section of Integration New Media's web site at: http://www.IntegrationNewMedia.com/products/v12authorware/tools/.

You're not alone!

Whether you are looking for a quick answer or in-depth information, there are many resources available online and offline to help you. There are many, many registered owners of V12 that are sharing concepts and strategies online daily. And, you can always call or e-mail INM's customer support team to help you with any specific problems you may have. The following resources may be very helpful.

V12-L discussion list

On the V12-L discussion list, you will find developers at every level of expertise, and in a wide variety of expertises in the multimedia arena. This friendly group is the perfect place to bounce ideas around with other V12 developers. Sign up at: http://www.IntegrationNewMedia.com/support/list/

Other online resources

Macromedia's web site at http://www.macromedia.com/support, is also a possible source of information. It contains, among other things, directions on how to subscribe to Macromedia's support Newsgroups (the NNTP server is "forums.macromedia.com").

You may want to check alternate Internet resources such as

- Aware-L join:

http://listserv.cc.kuleuven.ac.be/cgi-bin/wa?SUBED1=aware&A=1

- Dazzle Technologies Corp.(Joseph Ganci) site:

http://www.authorware.com/AWHome.htm

- Stefan van As's website site:

http://www.xs4all.nl/~svanas/

- Macromedia:

http://www.macromedia.com/software/authorware

Customer support

If you need additional assistance, INM's experienced team will be happy to help.

Customer support is available from 9:00 am to 5:00 pm EST, Monday through Friday by email to support@IntegrationNewMedia.com or by phone at:

+1 514 871 1333, Option 6.

Priority will be given to registered V12-DBE users. Customer support covers:

- Helping you to understand V12-DBE, clarifying specifications.
- Supplying you with sample scripts for generic concepts.
- Providing useful tips.

Developer assistance

Where Customer Support stops, Developer Assistance begins. If you are familiar enough with V12-DBE, but want to take your project to a more sophisticated level, Developer Assistance is for you. Our team of programmers can help you discover easier ways to take advantage of databases in your multimedia projects. Here are just some of the services we offer:

- Project design, data structure analysis, planning
- Technical guidance throughout the various steps of your project
- Troubleshooting and debugging your scripts
- Optimization (how to obtain superior performance)
- Assistance with other Xtras, custom development of Xtras

You can think of Developer Assistance as an additional team member, or members that you can add on to your project team. They can fulfill a small or large role on your team, depending on your needs.

About this manual

If you are familiar with other database management systems, you will find V12-DBE very easy to use. If you are only vaguely familiar with database management, the First Steps tutorial, which includes a manual and sample Authorware piece, will guide you, step-by-step, through the basics required to implement simple database management in your multimedia projects.

This manual is organized to help you get the information you need efficiently.

- The *Using Xtras* section deals with basics concerning Xtras.
- The *Database basics* gives an overview of databases.
- In Using V12-DBE, we lead you through the basic steps on how to use V12-DBE in detail. You will learn how to prepare data, create the database and import data.
- Subsequent sections show you how to use the methods available to you in V12-DBE.
- The remaining sections cover the integration of V12-DBE with Macromedia Authorware. Here you can get a sense of how V12-DBE can be helpful to your projects.
- The Appendices deal with very specific issues such as capacities and limitations, errors, end-user delivery, portability, etc.

Typographic conventions in this manual

Important terms, such as the names of methods, are in **bold**.

```
Sample code is indented and printed in this font.
```

Although the sample scripts throughout this manual contain both upper and lower case characters, V12-DBE is *not* case sensitive. This applies to the methods names, the parameters as well as to the actual data. They are described using mixed case in order to improve readability and facilitate debugging, and we recommend you adopt a similar strategy in your Script programming. It really does help to improve the readability of your Scripts!

Note: Special annotations and tips are enclosed in the sidebar, like this one.

Welcome to V12 Database Engine

Welcome to V12-DBE, the most powerful and user-friendly cross-platform database management Xtra for Macromedia Authorware™ (version 4.x and later) on Macintosh and Windows.

If you are familiar with other database management systems, you will find V12-DBE very easy to use. If you are only vaguely familiar with database management systems, the next few sections will give you an overview of what you need to know to help you get started with V12-DBE.

System requirements for running V12-DBE

Macintosh versions

Tip: INM recommends that you always thoroughly test your final project using the minimum standard equipment you have determined for your application, on all supported operating system platforms.

This process will also help you confirm that your product functions correctly within the minimum memory requirement you have recommended to your users.

- PowerMac with System 7.1 +
- Mac OS X

On the Macintosh, V12-DBE (and any other Xtra) will share the same memory partition as Macromedia Authorware.

For simple database applications, you probably do not need to change the memory partition allocated to Authorware or to packaged pieces generated by Authorware. For more advanced development, you may need to increase the memory partition.

In general, we recommend that your Authorware application's memory partition be set at the maximum you can afford to give to it, with enough RAM memory reserved for any other applications you may need to run at the same time as Authorware.

We also recommend that you allocate an absolute minimum of 8 megabytes of RAM to the packaged piece's minimum RAM requirements. You may also want to ensure that Authorware's preference for "Use Temporary Memory" setting is set before creating your packaged piece (this will ensure that if the packaged piece does run out of memory, it will start using free System memory and dynamically increase the memory allocated to the application.

Windows version

- Windows 95, 98, ME, NT4, 2000, XP.

Windows uses a Virtual Memory scheme, which dynamically allocates memory to applications. This means that an application can "borrow" as much memory as needed from the Operating System. It also means that Windows shows unpredictable behaviors when it is short of memory. Try to establish the minimum equipment requirements of your project as conservatively as possible.

Always test your project thoroughly on the minimum required equipment you have determined for your application.

For Windows 3.1 and Mac68K versions, please contact us!

Macromedia Authorware

Macromedia Authorware version 4, 5.2, 6, 6.5 or 7 is required.

Installing V12-DBE

The name of this Xtra is **V12-DBE for Authorware.XTR** on the Macintosh and **V12-DBE for Authorware.X32** on Windows.

To install the V12-DBE Xtra in your authoring environment:

- Make sure that Authorware is closed.
- Move the V12-DBE Xtra to the Xtras folder located in the same folder as Authorware.
- Start Authorware.

To confirm that V12-DBE is properly installed, check the Xtras menu in Authorware. You should see "V12-DBE for Authorware" in the Xtras menu.

What's new in version 3.3?

New Features:

- Now supports two Japanese sort orders: Yomigana and Shift-JIS Japanese.

Bug fixes:

- Searching: in some cases, problems occurred when using multiple operators (ex: <= and >=).
- Record deletion: some data was incorrectly deleted when deleting indexed string fields that contained over 251 characters.

Release history

For details on what changed in previous releases of V12-DBE, please visit the support section of Integration New Media's web site:
http://www.IntegrationNewMedia.com/support/

How to register your V12-DBE license

Evaluation copies of V12 Database Engine are available on Integration New Media's web site (http://www.IntegrationNewMedia.com/) along with full documentation and sample pieces. You can download those files and start developing your project without purchasing a V12-DBE license.

The evaluation copy of V12-DBE is not limited in any way: it only displays a splash screen upon startup. To get rid of the splash screen, you must purchase a V12-DBE license (or as many as required by the V12-DBE license agreement). As a licensed user, you are given a registration number that you must enter in Authorware's Xtras > V12-DBE for Authorware > Register… menu item.

Once your copy of V12-DBE is licensed, all new databases you create are automatically registered and do not show a splash screen. Existing databases are also registered as soon as they are opened by the registered V12-DBE.

Note: Existing V12 databases must be opened once in ReadWrite or Shared ReadWrite mode to be licensed. If you open them in ReadOnly mode or from a CD-ROM, they cannot be licensed and the splash screen will continue to appear on computers that do not have the license file. V12-DBE returns a warning when opening unlicensed databases in such circumstances.

Files needed to use V12-DBE

Only one file is required for the "Runtime" version (also called "end-user" version) of V12-DBE. The name of this file is **V12-DBE for Authorware.XTR** on Mac, **V12-DBE for Authorware.X32** on Windows and **V12DBE-A.X16** on Win16.

The "Development" version requires an additional file – the license file – located in the **System:Preferences** folder of your Macintosh, or the **Windows\System** folder of your PC. This encoded file is generated by V12-DBE upon the registration of your license number.

Although the "Runtime" version of V12-DBE can be distributed freely in as many copies as you wish, you *cannot* distribute your license file.

Using Xtras

This section deals with Xtras and how they are used in Macromedia Authorware. The V12-DBE Xtra is used as an example throughout the manual. You will be introduced to the basic steps involved in using V12-DBE successfully before you actually begin to work with V12-DBE.

This section covers:

- What is an Xtra
- Making an Xtra available to Authorware
- Creating a Script Xtra instance
- Verifying whether the instance was successfully created
- Using the Script Xtra instance
- Freeing the Script Xtra instance
- Where to get additional Documentation

What is an Xtra?

Xtras are components (alternatively know as add-ons, or plug-ins) that add new features to Macromedia Authorware. Many of Authorware's own functions are implemented as Xtras.

Note: In the rest of this document, the term Xtra refers to Scripting Xtras, unless otherwise specified.

A few complex tasks can be performed using Authorware's scripting language. However very complex and time-critical tasks are normally implemented as **Xtra**s. Xtras are a new "add-on" standard introduced with Macromedia Authorware 4.0 that add functionality to your multimedia applications. Four types of Xtras are supported by Authorware: Scripting Xtras, Tool Xtras, Transition Xtras and Sprite Xtras.

Making an Xtra available to Authorware

Xtras are designed to be opened automatically by Authorware (in authoring mode) by its runtime packages (in runtime mode, also called *playback* mode). The Xtras must be placed in the **Xtras** folder, located either in Authorware's folder or Authorware's runtime folder. This feature is supported on both Macintosh and Windows.

Creating an Xtra instance

A mandatory first step before using a Script Xtra is to create an instance of it. This is how a Script Xtra comes to life and, from then on, you can use its methods.

Call `NewObject` to create an instance of an Xtra. Generally, you store a newly created Xtra instance into a global variable for future use. It uses the `NewObject` method of the database Xtra.

Example:

```
gDB := NewObject("V12dbe", "myBase.V12", "Create", "myPassword")
```

Checking if NewObject was successful

Note: This is a generic approach and works with all Xtras. In V12-DBE, the preferred way to check for errors is the V12Status() method. See Errors and defensive programming in this manual.

You should always ensure that the Xtra was created successfully immediately after calling `NewObject`. `NewObject` can fail for many reasons, such as a lack of free memory or as a result of misplaced files.

Example:

```
gDB := NewObject("V12dbe", "myBase.V12", "Create", "myPassword")
if V12Status() <> 0 then GoTo(@"NotifyUser")
```

Using the Xtra instance

Note: In order to learn which methods are supported by an Xtra, use the Xtra's built-in documentation. See Basic documentation below.

Once the preliminary steps have been executed, you can start using the Xtra instance of your database for creating tables, fields and indexes, or for using an existing database. **Methods** of the Xtra need to be called to perform these operations. By convention, V12-DBE method names begin with the letter `m` such as `mGetfield` and `mSelect` (except for `NewObject`). `NewObject` is a compulsory method and scripting Xtra's support it.

This example creates a table called `TableName` in the database referred to by `gDB`:

```
CallObject(gDB, "mCreateTable", "TableName")
```

Closing an Xtra

Note: If a V12-DBE Xtra instance is not properly deleted (DeleteObject), the file it refers to remains open and cannot be re-opened unless the computer is restarted. In some cases, the database could be corrupted.

When the Xtra instance has completed its function and is no longer required, close it by calling `DeleteObject`. Closing an Xtra performs mandatory housekeeping tasks and closes unneeded files. It also frees the memory occupied by the Xtra. All Xtra instances created with `NewObject` must be ultimately closed with `DeleteObject` once they are no longer needed.

Example:

```
DeleteObject(gDB)
```

Although `DeleteObject` does exactly what it is intended for, it is always a good practice to systematically set the deleted objects reference to 0. This makes it simpler to know whether a database is open or closed: you only need to check wheter the variable used to refer to it is zero or not.

Example:

```
DeleteObject(gDB)
gDB := 0
```

Checking for available Xtras

You can learn which Xtras are available to Authorware by opening the Functions window from the Window menu and checking the content of the Category menu. The available Xtras are listed right after the "Xtra's(All)" item.

If V12-DBE is installed, you should see V12dbe and V12table listed in the Category menu as well as all other available Scripting Xtras. Note that this technique applies to Scripting Xtras only.

Dealing with pathnames

The NewObject method in V12dbe requires that you specify the name of the V12-DBE file you want to create or open. If only a file name is specified, the file is

assumed to be located in the same folder as Authorware or the Authorware runtime.

Example:

```
gDB := NewObject("V12dbe", "myBase.V12", "Create", "myPassword")
```

assumes that "myBase.V12" is in the same folder as Authorware or the Authorware runtime.

Most of the time, however, placing the database file in the same folder as the *piece* that uses it is more convenient. Use the `FileLocation` variable to get the current piece's folder.

Example:

```
gDB := NewObject ("V12dbe", FileLocation ^ "myBase.V12", "Create",
   "myPassword")
```

Passing parameters to Xtras

As in any programming language (including Authorware's scripting language), functions, commands and methods require a certain number of parameters. For example, in Script, the `Trace` function expects one parameter: the string to output in the Control Panel window (example: `Trace("Database successfully created")`). Likewise, the `ResizeWindow` function expects two parameters: *width* and *height*.

While the two aforementioned examples require exactly one and two parameters respectively, some commands and functions offer more flexibility by accepting optional parameters. For example, in Authorware, the `Capitalize` function requires at least one parameter: the string to capitalize. However, an additional parameter can be specified to modify its behaviour. In this case, assign the parameter "1" would capitalize only the first word in the string.

Xtras offer the same mechanism: some methods require an exact number of parameters (*fixed number of parameters*), others assume default values if parameters are omitted (*variable number of parameters*). Each of these methods can be easily identified in the Xtras built-in documentation explained below (see Basic documentation).

Basic documentation

In Authorware, Xtras contain a built-in mechanism that provides documentation for developers.

Open the Functions window from the Window menu (or type Ctrl+Shift+F) and select the name of the Xtra you need documentation for in the Category pop-up menu. Then select the method for which you need documentation. The lower part of the Functions window will display a description of that function along with its calling syntax.

For example, selecting `mCreateTable` in the **Xtra V12dbe** category of the Functions window will display the following text in the Description area.

```
CallObject(gDB, "mCreateTable", "tableName")
-- PARAMETERS : tableName = name of table to create. DESCRIPTION :
   Creates a new table named tableName…
```

Methods that expect a fixed number of parameters are those for which each parameter is listed. Methods that accept a variable number of parameters are those followed by a **[*]**.

Following are a few examples:

```
CallObject(object, "mFlushToDisk")
```

means that the `mFlushToDisk` method requires exactly one parameter: the database instance.

```
CallObject(object, "mSetProperty", "property", "value")
```

means that `mSetProperty` requires three parameters: the database instance, the property (a string) and the value of the property (a string).

```
CallObject(object, "mDumpStructure"[, *])
```

means that `mDumpStructure` requires at least one parameter, and possibly more (indicated by the asterisk). You must refer to the documentation of this method to know what additional parameters are accepted.

```
CallParentObject("V12dbe", "mXtraVersion")
```

the `mXtraVersion` is a static method - a method that can be used with a database instance (i.e. `mXtraVersion(gDB)`) and a database library instance (i.e. `mXtraVersion(Xtra "V12dbe")`). Static methods are seldom used in V12-DBE.

```
V12Status()
```

`V12status` is a global method - a method that can be used at any time, regardless of Xtra instances. It is only required that the Xtra be present when that function is called.

Database basics

Overview

If you want a clearer understanding of what a database is and does, we recommend that you read this section. The following topics introduce you to database basics:

- What is a database?
- Records, fields and tables
- Indexes, Compound indexes, and Full-text indexing
- Flat and relational databases
- Field types
- Selection, current record, and search criteria

What is a database?

A database is a collection of information that can be structured and sorted. A telephone book is an example of a hardcopy database, and government statistical records are examples of electronic databases. Database management programs such as V12-DBE provide many advantages over hardcopy databases. Unlike a telephone directory, where you can look up data that is sorted in alphabetical order only, database management programs allow you to change the way you sort and view information. Moreover, you can find, modify and update information quickly and easily.

Records, fields and tables

Note: Some database management systems refer to fields as columns and to records as lines or rows. In V12-DBE, the terms remain `fields` and `records`.

An entry in a database is called a **record**.

Each record consists of pieces of information called **fields**.

All records are stored in a **table**.

For example, data entry in an address book typically consists of seven pieces of information called `fields`: last name, first name, street address, city, state, zip code and phone number. All the information relevant to one person makes up one `record`. The collected records make up the `table` and are contained in a `database` file. Entries below are typical of those found in an address book:

This is an example of a **table**:

Last Name	First Name	Address	City	State	Zip	Phone	--- These are **fields**
Jordan	Ann	6772 Toyon Court	San Mateo	CA	94403	349-5353	--- This is the 1st **record**
Brown	Charles	30 Saxony Ave.	San Francisco	CA	94115	421-9963	--- ...the 2nd **record**
Pintado	Jack	22 Hoover Ave.	Bowie	MD	20712	731-5134	--- ...the 3rd **record**
Van Damme	Lucie	87 Main St.	Richmond	VA	23233	315-3545	--- ...the 4th **record**
Peppermint	Patty	127 Big St.	Lebanon	MO	92023	462-6267	--- ...etc…

Indexes

In a telephone directory, information is indexed by last name - a typical way to search for a telephone number. There are directories that index information by order of phone number or address, but such static directories sort information in only one specific predetermined order.

V12-DBE, by contrast, allows you to determine how you want to sort information by defining one or more **indexes** in a table. When a field is indexed, V12-DBE creates an internal list that can be used to sort and search quickly the data it contains. Non-indexed fields can also be searched and sorted, but at a slower speed.

In this example, the address book entries are listed according to an index of the first name field and sorted in ascending order (A to Z), thus appearing in alphabetical order by first name.

Last Name	First Name	Address	City	State	Zip	Phone
Jordan	Ann	6772 Toyon Court	San Mateo	CA	94403	349-5353
Brown	Charles	30 Saxony Ave.	San Francisco	CA	94115	421-9963
Pintado	Jack	22 Hoover Ave.	Bowie	MD	20712	731-5134
Van Damme	Lucie	87 Main St.	Richmond	VA	23233	315-3545
Peppermint	Patty	127 Big St.	Lebanon	MO	92023	462-6267

Compound indexes

A *compound index* — or *complex index* —organizes entries composed of two or more fields, as opposed to simple indexes — or indexes, for short — which organize single-field entries. Compound indexes are useful to determine the sorting order of records when some fields contain identical values.

In this example, three records share the same last names (Cartman). Indexing the field `LastName` alone would certainly force Last Names to be properly ordered. But this would not determine the order in which the Cartman's are sorted.

Last Name	First Name	City	State	Zip
Brown	Charles	San Francisco	CA	94115
Cartman	Wendy	San Mateo	CA	94403
Cartman	Lucy	Richmond	VA	23233
Cartman	Eric	Lebanon	MO	92023
Pintado	Jack	Bowie	MD	20712

If you want your records sorted by Last Name, and by First Name in case of identical Last Names, you define a compound index on the fields `LastName` and `FirstName`. The sorted result would then be:

Last Name	First Name	City	State	Zip
Brown	Charles	San Francisco	CA	94115
Cartman	Eric	Lebanon	MO	92023
Cartman	Lucy	Richmond	VA	23233
Cartman	Wendy	San Mateo	CA	94403
Pintado	Jack	Bowie	MD	20712

If you want them sorted by Last Name, and then by State in case of identical Last Names you define a compound index on the fields `LastName` and `State`. The sorted result would then be:

Last Name	First Name	City	State	Zip
Brown	Charles	San Francisco	CA	94115
Cartman	Wendy	San Mateo	CA	94403
Cartman	Eric	Lebanon	MO	92023
Cartman	Lucy	Richmond	VA	23233
Pintado	Jack	Bowie	MD	20712

Up to twelve fields can be declared in a single compound index in V12 Database Engine.

Full-text indexing

Defining an index on a field allows for quick sorting and searching *of the first few characters* of a field. In some applications – typically when fields contain extensive information – you need to search for words that appear *anywhere* in a field efficiently. This is where you need to define a full-text index, or **full-index** for short, on that field. A full-index is an index defined on every single word of a field.

Last Name	First Name	Publication Title
Jordan	Ann	Soups and Salads for Dummies
Brown	Charles	The Hunchback of the Empire State Building
Pintado	Jack	Bounds on Branching Programs
Van Damme	Lucie	Natural and Artificial Intelligence
Peppermint	Patty	Mastering Soups in 32,767 Easy Lessons

Note: Each index takes up disk space, so it is not recommended that all fields be indexed. Full-indexes require much more space than regular indexes. Indexed fields should be limited to those most likely to be searched and/or sorted.

In this example, looking for the word "Soup" in the Publication Title field requires a full-index for optimal search performance. If no index is defined on the Publication Title field, the same result can be achieved, but with a slower performance. If a regular index is defined on the Publication Title field, publications that start with the word "Soup" can be quickly located, but publications that contain that word somewhere in the middle of the field require more time. Full-indexes apply only to fields of type `string`, including those that contain styled text (see Field types and International support).

Defining full-text indexing options

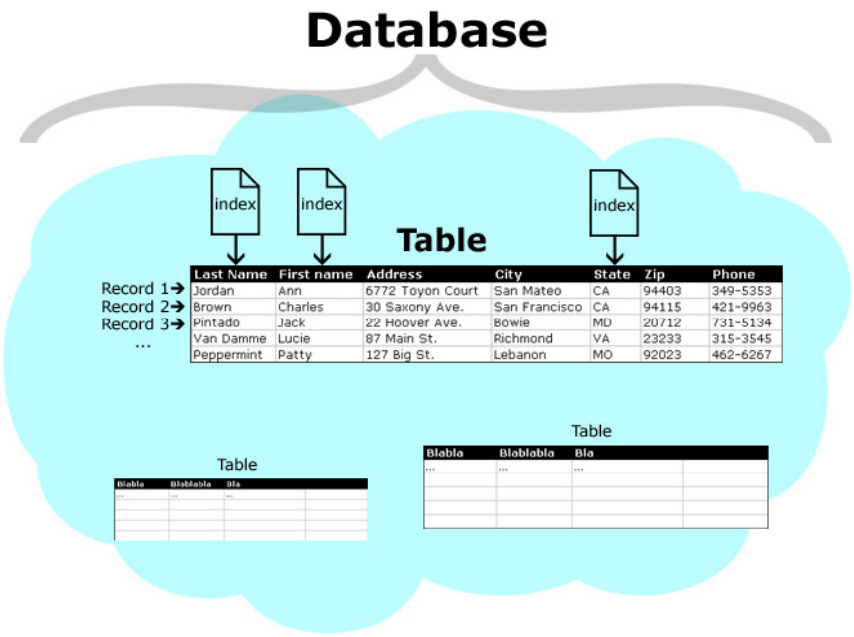
For optimal full-text search efficiency, some level of control is required on the way it is performed. For example, indexing trivial words such as "and", "or", "the", etc. (or equivalent words that appear frequently in your application's language) is useless as most records would contain one or more occurrences of those words.

Likewise, some applications or languages require that digits be full-indexed whereas others would prefer to ignore them. V12-DBE enables you to fine-tune the behavior

of the full-indexes by allowing for the definition of **Stop Words** (words that must be ignored), **Delimiters** (characters that delimit word boundaries) and **MinWordLength** (the size of the shortest word that must be considered for full-indexing).

Database

A table, its fields and the indexes defined are stored in a **database**. A database can contain one or more such tables.



Flat and relational databases

Note: There is a practical limit to the number of tables you can have in a V12 database. See Appendix 1: Capacities and Limits for specific limitations.

A flat databaseusually consists of one table. In flat database management systems such as FileMaker Pro, the terms table and database are interchangeable.

A relational database presents a more sophisticated use of information. In relational database management systems, two or more tables are contained in the database. Therefore, you can store as many tables as you need in a single database file and each table could have one or more indexes. Tables can be linked so that information can be shared, saving you the trouble of copying the same information into several locations and eliminating the maintenance of duplicate information. Linking is important if there are relationships between the various pieces of information.

For example, if you want to add information to the entries contained in the address book in our first example, such as the company address and phone number, one way to do this would be to add them to the table:

Last Name	First Name	Address	City	State	Zip	Phone	Company	Phone
Jordan	Ann	6772 Toyon Court	San Mateo	CA	94403	349-5353	Rocco & Co.	526-2342
Brown	Charles	30 Saxony Ave.	San Francisco	CA	94115	421-9963	National Laundry	982-9400
Pintado	Jack	22 Hoover Ave.	Bowie	MD	20712	731-5134	Rocco & Co.	526-2342
Van Damme	Lucie	87 Main St.	Richmond	VA	23233	315-3545	Presto Cleaning	751-5290
Peppermint	Patty	127 Big St.	Lebanon	MO	92023	462-6267	Presto Cleaning	751-5290

However, adding this information might lead to duplication of information, given that some people might be working for the same company. To prevent duplication and to save on disk space and time required to update, you could create a new table containing only the business information. For example, the new table could be called: Companies. Each record of that new table would have a unique ID number, *Company Ref*, which would also be stored in the first table.

The database now contains two related tables, each having a field containing the common information, named "Company Ref":

Table 1: containing information about each person:

Last Name	First Name	Address	City	State	Zip	Phone	**Company Ref**
Jordan	Ann	6772 Toyon Court	San Mateo	CA	94403	349-5353	**RO**
Brown	Charles	30 Saxony Ave.	San Francisco	CA	94115	421-9963	**NA**
Pintado	Jack	22 Hoover Ave.	Bowie	MD	20712	731-5134	**RO**
Van Damme	Lucie	87 Main St.	Richmond	VA	23233	315-3545	**PR**
Peppermint	Patty	127 Big St.	Lebanon	MO	92023	462-6267	**PR**

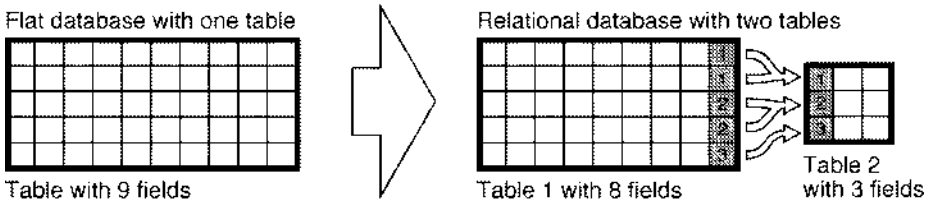
Table 2: containing information about the companies:

Company ref	Company	Phone
NA	National Laundry	982-9400
PR	Presto Cleaning	751-5290
RO	Rocco & Co.	526-2342

Note: Relational Database Management Systems (RDBMS) are usually programmed with SQL (System Query Language) statements, which have the ability to automatically resolve relations between related tables.

Although V12 Database Engine can store multiple tables per database, it relies on Lingo scripts to resolve relations. It cannot automatically resolve such relations.

The two databases could also be compared as follows:



The relational database is smaller because it avoids useless data duplication. In order to retrieve full information about any given individual in your address book, you would perform a search in your first table, retrieve the company reference, and then perform a search in the second table. The flat model may be easier to manage when retrieving data given that only one search is required, however it tends to consume valuable disk space. The flat model also introduces the risk of data discrepancies due to the duplication of data.

Field types

For optimal data sorting and searching, you can specify the kind of information to be stored in each field. In V12-DBE, fields can be designated to contain strings, integers, floating-point numbers, dates, pictures, sounds, palettes, etc. A field would then be of type **string**, **integer**, **float**, or **date**. See Appendix 1: Capacities and Limits at the end of this manual for a formal definition of each field type.

For example, if you wanted to organize a contest where each person listed in your address book were collecting points, you would need to keep track of the number of

points accumulated by each person. Therefore, you would update your address book to include a new field: *number of points*. Since you would want to search and sort this new field quickly, you would need to define an index. This new field could be one of two types: string or integer.

If you define the new field as type string, you might end up with the following listing when the table is sorted by ascending order of points:

Jordan	Ann	1
Brown	Charles	12
Peppermint	Patty	127
Pintado	Jack	6
Van Damme	Lucie	64

This order occurs because the string "12" is alphabetically lower than the string "6" given that the ASCII code for "1" is 49 which is smaller than the ASCII code for "6", 54. To sort the list in the expected ascending order, you must define the field *number of points* to be of type integer to get this result:

Jordan	Ann	1
Pintado	Jack	6
Brown	Charles	12
Van Damme	Lucie	64
Peppermint	Patty	127

Typecasting

Typecasting (or *casting*, for short) is the process of converting a piece of data from one type to another. This is a common mechanism to most programming languages, including Authorware scripting language.

For example, the integer *234* can be casted to the string "234". Conversely, the string "3.1416" can be casted to the float *3.1416*.

Typecasting can be performed explicitly in Authorware's scripting language using the `Integer`, `String` and `Float` functions (i.e., `String(234)` returns the string "234") or automatically (i.e., `12&34` returns the string "1234").

V12-DBE has the same ability as Authorware's scripting language to typecast data when it is required by the context. However, some borderline conditions can lead to ambiguous results such as trying to store the value " 123" in a field of type `Integer` (note the leading space).

You must always make sure that the data supplied to V12-DBE does not contain spurious characters, otherwise typecasting will not be performed properly.

International support

Although the 26 basic letters of the Roman alphabet sort in the same order in all roman languages, the position of accented characters (also called *mutated characters*) varies from one language to another. For example, the letter **ä** sorts as a regular **a** in German whereas it sorts after **z** in Swedish. Likewise, in Spanish, **ch** sorts after **cz** and **ll** sorts after **lz.**

V12-DBE's default `string` was designed to satisfy as many languages as possible. It can sort and search texts in English, French, Italian, Dutch, German, Norwegian,

etc. See Appendix 4: String and custom string types in the appendices of this manual for a detailed description of `string`'s behavior.

V12-DBE also offers the option of defining fields of type `Swedish`, `Spanish`, `Hebrew`, etc. that index and sort data in a way that is compliant with these languages. See Appendix 4: String and custom string types for an exhaustive list and description of those behaviors called *custom string types*.

The Regular Edition of V12-DBE allows you to create custom string types , where you define a sort/search description table for each. Therefore, you can define your own string type for any language supported by single-byte characters, including Klingon.

Note: Everything that applies to the type `string` also applies to custom string types. Throughout this manual, the term `string` is used to designate both the default V12-DBE string and custom string types.

Selection, current record, search criteria

Besides sorting a table through indexes, you can find information based on search criteria. You can define simple search criteria, also called **simple queries**, such as:

- First name is Jack
- State is California
- Number of points is less than 30
- Last name begins with P

Or you can define **complex search criteria**, also called Boolean queries using and/or, such as:

- First name is Jack **or** Last name begins with P
- State is California **and** Number of points is less than 30
- State is California **and** Number of points is less than 30 **and** Last name contains "pe"

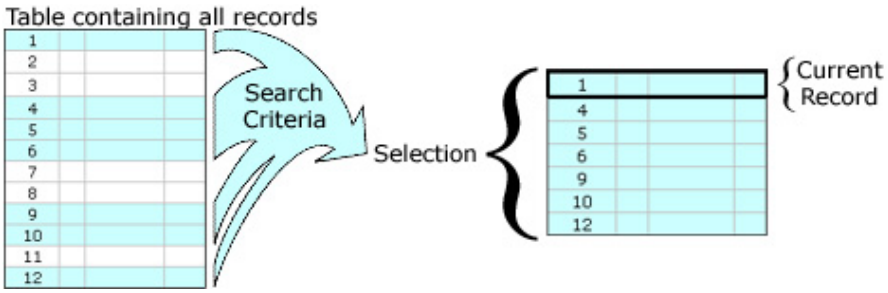
Note: Database Management Systems that use SQL as their programming language can define search criteria such as: *(Dish is soup or appetizer)* and *(Main Ingredient is celery or eggplant or pumpkin)*. V12 Database Engine does not support alternating uses of ANDs and ORs.

See technical notes on http://www.IntegrationNewMedia.com/support/v12director/technotes for possible workarounds.

The **selection** is a set of records currently available in the table. When a table is opened the selection contains all the records of the table. If you search through a table after having defined search criteria, the resulting set of records that satisfy the search is the new selection. When a selection is first defined, the current record is the first record of that selection.

- If exactly one record satisfies the search criteria, the selection contains only that record, which automatically becomes the current record.
- If two or more records satisfy the search criteria, the selection is the set of those records, and the first record of the selection becomes the current record.
- If no record satisfies the search criteria, then the selection is empty and the current record is undefined. Any attempt to read or write in a field will result in an error.

This figure illustrates the idea of searching a table for records satisfying a certain criteria. The result is placed in a selection, the first record of which becomes the current record.



All operations on any fields (such as reading and writing data) are done on the current record. Therefore, before performing these operations, you must designate the record on which you wish to work as the current record by selecting it, and by using methods such as `mGoFirst`, `mGoLast`, `mGoNext`, `mGoPrevious` and `mGo`.

You can read the contents of a field in the current record, modify its contents or delete the entire record. When you move from one record to then next in the selection, the current record pointer changes. Note that if you modify the fields of the current record, you must call `mUpdateRecord` to save your changes to the database before moving to another record, otherwise your changes will be lost.

Using V12-DBE: step-by-step

Overview

This section covers the main steps in using V12-DBE. If you have looked at the *First Steps* manual (recommended), you should already be familiar with some of these steps. Here we will explain in detail how to prepare the data, design a data model and create a V12-DBE database. A script review and a detailed explanation of table and database Xtra methods will follow.

V12-DBE components

V12-DBE is a powerful database management engine, composed of two Xtras libraries: a **database** Xtra named "V12dbe" and a **table** Xtra named "V12table". The database Xtra is used to create a new database or to open an existing database in a given mode (Read Only, ReadWrite or Create). The table Xtra is used to manage the content of the table in your database.

The main steps

If you have read the **First Steps** manual and followed the accompanying ShowMe, you have seen a typical step-by-step use of V12-DBE. The individual steps to using V12-DBE are explored in greater detail in this section. Other useful examples that cover these steps are illustrated in the **ShowMe** pieces named V12Glossary and V12Quiz. These pieces are available for download from our website at: http://www.IntegrationNewMedia.com/products/v12authorware/demos/.

Note: Although steps 1 and 2 do not involve any production work or programming, they are the **most critical** ones to the success of your project.

A well-designed project will yield high-quality results, on time, on budget. Similarly, failing to lay solid foundations at steps 1 and 2 will lead to an unmanageable project with fragile results. If you don't feel comfortable with steps 1 and 2, we recommend that you seek advice or hire professional help.
(See You're not alone!)

Step 1: Decide on a Data Model: Before you create your database, decide which fields are needed, the data type of each field, how the fields should be grouped in the tables and which fields should be indexed. This design effort does not require a special tool (with the possible exception of a word processor to help you edit your ideas). If your original data is managed in FileMaker Pro, MS Access, or a similar database management product, that database's model is probably the best starting point for your V12 database model.

Step 2: Prepare the Data: If your original data is managed in FileMaker Pro, MS Access, or a similar database management product, in step 2, you make sure that your data is properly entered and that it is in a format readable by V12 Database Engine (Text file, DBF file or one of V12-DBE's ODBC-compliant formats).

Step 3: Create a database: Use the V12-DBE Tool to create the V12 database you designed at Step 1. Alternatively, you can use the database Xtra's (i.e. Xtra *V12dbe*'s) methods to write an automated database creation script in Script.

Step 4: Import data into a V12-DBE database**:** Use the V12-DBE Tool to import the Text or DBF file exported at Step 2. Alternatively, you can write Scripts to automate the process of importing data into your V12-DBE databases.

Step 5: Implementing the user interface: In this step you develop the means by which users will search for, retrieve and modify data at runtime. This interaction with the database can be developed either using V12-DBE Knowledge Objects attached to Authorware sprites, or as functions in Authorware Script.

Download the Knowledge Objects from the Free Tools section on Integration New Media's web site: http://www.IntegrationNewMedia.com/products/v12authorware/tools/.

Sample Authorware pieces are provided on the Demos page: http://www.IntegrationNewMedia.com/products/v12authorware/demos/.

Each of the aforementioned steps is discussed more in depth in subsequent sections. Since V12-DBE offers more than one way to attain a goal, the simplest approach is explained first; then alternate and more powerful or versatile approaches are discussed.

Step 1: Decide on a Data Model

Before creating a database file, you need to decide how you want to organize your data. If your original data is managed in FileMaker Pro, MS Access, or a similar database management product, that database's model is probably the best starting point for your V12 database model. The questions you need to address are:

- which fields are required and what are their respective types?
- which fields should be indexed for quick searching and sorting?
- how many tables are required to group the fields?
- are there any relationships between the various tables?

In the stationary catalog example below, only one table is needed. It is called "Articles". The seven fields you need are :

- Field "ItemName" of type String
- Field "Category" of type String
- Field "Description" of type String
- Field "Price" of type Float
- Field "CatalogNumber" of type Integer
- Field "Date" of type Date

Since only the fields "ItemName", "Price" and "CatalogNumber" will be searchable, only they are indexed.

Defining identifiers

Tables, fields and indexes are given names called **identifiers**, and V12-DBE makes reference to them by use of these identifiers. An identifier must start with a low-ASCII alphabetic character (a..z, A..Z) and can be followed by any combination of alphanumeric characters (0..9, a..z, A..Z, à, é, ö, …). The maximum length for an identifier is 32 characters. No two fields or indexes of a table can have the same name.

V12-DBE is not case-sensitive. That is, upper cases and lower cases are identical. These identifiers are considered identical in V12-DBE: "articles", "ARTICLES", "Articles", "aRtICleS".

Step 2: Prepare the Data

Step 2 is relevant only if your original data is managed in FileMaker Pro, 4th Dimension, DBase or any other database management system that has the ability to export TEXT or DBF files.

If you plan to use an ODBC driver to import your data from MS Access, MS FoxPro, MS Excel or MS SQL Server, or if the records must be keyed-in by the user, skip to Step 3.

In brief, Step 2 consists in making sure that your original data is properly structured and in exporting it as Text or DBF files. Those files are then imported to V12 databases at Step 4: Import data into a V12-DBE database.

TEXT file formats

Text files are the most popular data interchange file formats. Usually, *TAB-delimited* Text files are used to exchange data between database management systems.

A typical TAB-delimited file is in this format:

```
Field_A1 TAB Field_A2 TAB Field_A3 TAB ... TAB Field_An CR
Field_B1 TAB Field_B2 TAB Field_B3 TAB ... TAB Field_Bn CR
Field_C1 TAB Field_C2 TAB Field_C3 TAB ... TAB Field_Cn CR
```

where `Field_A1, Field_A2,` etc. represent the actual data in those fields. `TAB` is the ASCII character 9, indicating the end of a field.

On the Mac, `CR` is the ASCII character 13, indicating the end of a record. On Windows, `CR` is the ASCII character 13 followed by the ASCII character 10 (Line Feed). Since V12-DBE always ignores Line Feed characters, you need not worry about exceptional cases between the Mac and Windows with respect to Record Delimiters.

Generally, using the V12-DBE Tool or the `mImport` method to import a text file into a V12-DBE database is a straightforward process, unless your fields contain `TAB` or `CR` characters. In such cases, V12-DBE confuses the real delimiter with the legitimate content of your field. See Dealing with delimiter ambiguity below.

Field descriptors

V12-DBE requires a special type of Delimited Text file format. The file's first line must contain **field descriptors**, or the names of the fields into which the data that follow must be imported. This file format is sometimes referred to as **mail merge format**. This is an example of such a file:

Name	*Price*	*CatNumber*
Ruler	1.99	1431
Labels	1.19	1743
Tags	6.19	…

You can easily have FileMaker Pro and MS Access export those field names before exporting the records data (See Exporting a FileMaker Pro database to text or Exporting a MS Access database to text).

Dealing with delimiter ambiguity

Most of the time, TABs are used to delimit fields in a Text file, and CRs to delimit records. If your fields contain TABs or CRs as part of their actual data, the legitimate content of your fields would be confused with those delimiters once exported in a text file. There is more than one way to deal with this problem. Choose the one — or combination — that best fit your project's needs in the list below.

Virtual carriage returns

Some database management systems (e.g., FileMaker Pro) export a special character other than ASCII #13 instead of the CRs that appear in your fields. For example, FileMaker Pro exports ASCII #11 (Vertical Tab) instead of ASCII #13. Those characters are called *Virtual Carriage Returns* or *VirtualCR* for short.

V12 Database Engine can recognize those characters and convert them to real Carriage Returns (ASCII #13) once they are imported. See Import data with mImport, in Step 4: Import data into a V12-DBE database, and VirtualCR in Properties of databases.

Text qualifiers

A *text qualifier* is a special character used to begin and end each Text field. In most database management systems, the quotation mark (") is the default text qualifier. Its main purpose is to group a field's content between two identical marks so as to enable the occurrence of field and record delimiters without the risk of confusion.

Example:

"Name","Description" *CR*

"Hat","high-quality, excellent fabric, available in:*CR*Red*CR*Green*CR*Blue"

"Shoe","this description, field, contains, commas, and, Carriage*CR*Returns"

Text qualifiers are automatically placed in text files exported from MS Access, FileMaker Pro (Mail Merge format) and MS Excel (only for fields that contain commas).

Text files containing Text Qualifiers are easily imported to V12 databases by setting the mImport method's TextQualifier property to the right character. See Import data with mImport, in Step 4: Import data into a V12-DBE database.

Note: Since V12-DBE always ignores Line Feed characters, (ASCII Character 10), those cannot be used as field or record delimiters.

Custom delimiters

Another way to avoid delimiter ambiguity is to choose delimiters other than TAB and CR. Some database management systems allow you to select appropriate delimiters before exporting to a TEXT file (e.g., 4th Dimension). Some others allow only the selection of a custom field delimiter and always use CRs as record delimiters (e.g., MS Access). FileMaker Pro and MS Excel do not allow for any customization.

V12-DBE's mImport method assumes, by default, that the field and record delimiters are TAB and CR. However, other delimiters can be specified. See Import data with mImport, in Step 4: Import data into a V12-DBE database.

Calculated fields

If your database management system does not support alternative delimiters you can nonetheless force it to export your own delimiters by creating an additional field and setting it as the result of the concatenation of all the other fields with the desired delimiter in between each two fields. Then, export only the new field in a text file.

Processing the exported text file

If the database management system used to store your data is not flexible enough, or if the data themselves are not properly structured, you can export them in a text file and use Third Party tools to search and replace sequences — or patterns — of characters.

Below is a non-exhaustive list of helpful tools:

- BBEdit from Bare Bones Software (http://www.BareBones.com/) For MacOS.
- TextPad from Helios Software Solutions (http://www.Textpad.com/). For Windows.
- UltraEdit from IDM Computer Solutions (http://www.Ultraedit.com/). For Windows.
- Microsoft Excel from Microsoft Corp. (http://www.Microsoft.com/. For MacOS and Windows.

BBEdit, TextPad and UltraEdit feature GREPs (General Regular Expression Parsers), which are very convenient to reorganize unstructured data.

Character sets

Character sets are not standard across operating systems and file formats. For example, the letter "é" is the 233rd on Windows, whereas it is the 142nd on Macintosh and the 130th on MS-DOS.

Although all three operating systems use the ASCII characters set, only low-ASCII characters (i.e., those below #127) are common to the many variants of the ASCII set. Therefore, the rest of this topic is of interest to you only if you deal with high-ASCII characters (such as **€, å, æ, ß, ê, ï, ø, ž, ‰, §, ¥,** etc.)

Note: Most applications import/export DBF files using the MS-DOS character set.

V12-DBE's `CharacterSet` property can be set to translate Windows, Macintosh or MS-DOS character sets when importing or exporting Text or DBF files. Optionally, `mImport` accepts the `CharacterSet` property to use only once to import a single file (as opposed to the `CharacterSet` property which permanently affects `mImport` and `mExportSelection`, or until it is set to another value). See Step 4: Import data into a V12-DBE database and Import data with mImport.

MS Word documents, V12 databases as well as many other proprietary file formats are cross-platform compatible. You should not worry about this portability issue if your data contains only low-ASCII characters (e.g. English alphabet).

Dealing with dates

Note: If you fail to initialize a field of type Date in a new record, or try to store an invalid date in it, it is automatically set to 1900/01/01 (January 1st, 1900).

Although V12-DBE can output dates in highly customizable formats, it requires that they be input in a single unambiguous format called the *raw* format: YYYY/MM/DD.

- YYYY: year in 4 digits (e.g., 1901, 1997, 2002)
- MM: month in 1 or 2 digits (e.g., 01 or 1 for January)
- DD: day in 1 or 2 digits (e.g., *04* or *4* for the 4th day of the month)

The separator between the three chunks of values can be any non-numeric character, although slash (`/`), hyphen (`-`) and period (`.`) are most commonly used.

Any date that needs to be imported in a V12-DBE field of type `date` needs to be in this raw format. This rule applies to the V12-DBE Tool as well as to V12-DBE's Script methods that accept dates as input parameters (e.g., `mImport`, `mSetField` and `mSetCriteria`).

Exporting a FileMaker Pro database to text

In FileMaker Pro, choose File > Import/Export > Export Records and select "Merge (*.MER)" in the Save as Type menu. As a side effect, FileMaker Pro exports your data with quotation marks surrounding each field and a comma as field separator. Your file can easily be imported to the V12 database with quotation marks as Text Qualifiers (see Text qualifiers) and commas as field delimiters (see Custom delimiters).

Exporting a MS Access database to text

Note: MS Access databases can be imported directly to V12 databases either by using the V12-DBE Tool, or through Lingo. See Loading a descriptor from a source file.

In MS Access, choose File > Export, to an external file or database. Then, select Text Files in the Save as Type menu. Click Export or Save. Make sure that Delimited is selected and click Next. Choose an appropriate field delimiter for your data (see Dealing with delimiter ambiguity), choose a Text Qualifier from the list (see Text qualifiers), and check "Include Field Names in First Row"; then click Next.

DBF file formats

Note: Years ago, DBF files were convenient, given that they contained fewer variants than TEXT files. However, since the introduction of Windows and the popularization of DBF to other Operating Systems, DBF files now contain many categories and have become difficult to manage. V12-DBE's preferred file importing format is Text.

V12 Database Engine can import DBF files two ways:

- on both MacOS and Windows, it can read DBF files of type Dbase III, Dbase IV, Dbase V, FoxPro 2.0, FoxPro 2.5, FoxPro 2.6, FoxPro 3.0 and FoxPro 5.0.
- on Win32 only, DBF files can be imported using the FoxPro ODBC driver.

You may want to export your data as DBF files, if that format is supported by your database management system.

DBF is an old file format. It was enhanced over the years but most common applications still use the popular Dbase III format whose features are common to all other DBF file variants. Limitations include:

- Field names are limited to ten characters, all in upper case,
- The number of fields per DBF file is limited to 128,
- Records are of fixed length, determined upon the creation of the DBF file,

- There is more than one way to deal with high-ASCII characters (accented characters) with DBF files. This depends on the operating system and application used to manage the DBF file,
- Indexes are saved in separate files with extensions such as IDX, MDX, NDX or CDX (depends on the managing application),
- DBF files cannot be password-protected. However, some applications protect DBF files by encrypting/decrypting them,
- Character fields (roughly, the equivalent of V12-DBE's string fields) are limited to 255 characters. Any text longer than 255 characters, must be stored in separate files called DBT files and referred to by Memo fields,
- Media (either Binary or Text) are stored in external DBT files pointed to by Memo fields in the DBF file. DBF fields of type Media are not supported by V12-DBE.

Various flavors of the DBF file format were introduced over the years, such as DBase IV, DBase V, FoxPro 2.0, FoxPro 2.5, FoxPro 2.6, FoxPro 3.0 and FoxPro 5.0. They all include DBase III's features as core specifications and add new data types or extend certain limits. See Appendix 2: Database Creation and Data Importing Rules for more details.

In summary, the exact structure and limitations of your DBF files largely depend on how your database management system deals with them.

Note: Database management systems that use a fixed-length record format, such as the DBF file format, use the maximum record length to allocate data space on disk. Consequently, that amount of space is lost for each record of the database regardless of the actual data stored in it.

V12 Database Engine uses a variable-length record format. This means that it uses the exact amount of space needed for the storage of a record on disk, with no space loss at all. The Field Buffer Size refers to the RAM buffer, used while transferring data between Director and the V12 database files.

Field buffer size

Prior to creating your database structures, you need to determine the size of the largest chunk of data for each field of type `string` in your database. This helps you optimize the size of the buffers needed to manage V12-DBE's internal data structures for each of those fields.

If you are confident that your `string`s will not exceed 256 bytes, you do not need to worry about the buffer size. Default buffers are set to 256 bytes for `string`s.

Step 3: Create a database

At Step 3, you formalize the database you designed at Step 1: Decide on a Data Model into a **database descriptor**. Then, you provide that descriptor to the V12-DBE Tool (if you choose to use the V12-DBE Tool), or to V12-DBE's `mReadDBstructure` method (if you decided to script the database creation process).

The V12-DBE Tool is a convenient point-and-click environment for small projects. Scripting the database creation process requires a little more effort upfront but may end up saving you a lot of time, if you need to experiment with your database structure or data before committing to a final form. It enables you to automate the database creation process.

If you use the V12-DBE Tool, just read the next two sections (Database descriptor and Using the V12-DBE Tool) and skip toStep 4: Import data into a V12-DBE database. If you wish to script the database creation process, read Script the database creation as well.

Database descriptors

Following is the format of text (and literal) database descriptors required by both the V12-DBE Tool and the `mReadDBstructure` method. It is used to build a database structure from scratch.

If you build your V12 databases from other databases (e.g., MS Access, MS Excel, etc.), you can directly skip to Using the V12-DBE Tool or Script the database creation.

The desired V12-DBE database structure is stored in a text file (or Authorware variable) called the database descriptor in this format.

```
[TABLE]
NameOfTable
[FIELDS]
FieldName1 FieldType1 IndexType1
FieldName2 FieldType2 IndexType2
FieldName3 FieldType3 BufferSize3 IndexType3

(* if there are more than one table, their descriptors follow each
   other *)

[Table]
NameOfTable2
[FIELDS]
FieldName1 FieldType1 IndexType1
FieldName2 FieldType2 IndexType2
etc.

[END]
```

The `[TABLE]` tag is followed by one parameter: the name of the table. This is an identifier (see Defining identifiers).

The `[FIELDS]` tag is followed by as many lines as you need to define fields in the above defined table. The syntax of each line is as follows (see Database basics for a thorough explanation of these concepts):

- `FieldName`: the name given to the field to be created. This is an identifier (see Defining identifiers),
- `FieldType`: `string`, `integer`, `float`, `date` or a custom string type (see Field types),
- `BufferSize`: the amount of RAM to allocate for the internal management of the field's content. This parameter is relevant only for fields of type `string.` If omitted, fields of type string are created with a default buffer size of 255. Characters are created with a default buffer size of 64K. See Field buffer size in Step 2: Prepare the Data.
- `IndexType`: the word "indexed" if the field must be indexed, or the word "full-indexed" if the field must be full-indexed, or nothing if `no` indexing is required. If you need to both index and full-index a field, see Defining both an index and a full-index on a field.

Note: If you try to store text longer than the size of the buffer allocated for the field type `string`, V12-DBE signals a warning and stores the truncated text into the field. `Media` that are larger than the maximum buffer size of a field are not stored at all.

The `[END]` tag indicates the end of the descriptor. It is a mandatory tag.

In each line of the descriptor file, tokens (i.e. field name, index name, value, etc.) must be separated by one or multiple Tabs and/or Space characters.

Note: A convenient way to build a Descriptor File for a database containing a large number of tables, fields or indexes is to type it into a spreadsheet thus taking advantage of advanced editing functions. The results can then be saved into a TAB-delimited file or Copied and Pasted into a Director field for processing by `mReadDBStructure`.

Example:

```
[TABLE]
Recipes
[FIELDS]
NameOfRecipe string indexed
Calories integer indexed
CookingTime integer
TextOfRecipe string 5000 full-indexed
[END]
```

Defining both an index and a full-index on a field

In exceptional cases, you would need to define both an index and a full-index on a field. Since the `IndexType` parameter defined above can represent only one of "indexed" or "full-indexed", you would need to set it to "indexed" and define the full-index separately under an additional tag named `[FULL-INDEXES]`.

Note: A valid database needs at least one table, and each table requires at least one field and at least one index.

The `[FULL-INDEXES]` tag must follow the `[FIELDS]` section and must be followed by a list of fields to be full-indexed, one per line.

Example:

```
[TABLE]
Recipes
[FIELDS]
NameOfRecipe string indexed
Calories integer indexed
CookingTime integer
TextOfRecipe string 5000 indexed
[FULL-INDEXES]
TextOfRecipe
[END]
```

Alternate syntax for creating indexes

Database descriptors support an alternate syntax for the creation of indexes. The `[INDEXES]` tag can be used right after the field definitions to explicitly name and define the desired indexes.

This alternate syntax is used by `mDumpStructure` for clarity (see View the structure of a database). It also allows the definition of unique-valued indexes and

descending indexes which are used only in exceptional cases (in summary, if you don't know what they mean, you probably don't need them).

This database descriptor example is equivalent to the one above:

```
[TABLE]
Recipes
[FIELDS]
NameOfRecipe string
Calories integer
CookingTime integer
TextOfRecipe string 5000
[INDEXES]
NameOfRecipeNdx duplicate NameOfRecipe ascending
CaloriesNdx duplicate Calories ascending
TextOfRecipeNdx duplicate TextOfRecipe ascending
[FULL-INDEXES]
TextOfRecipe
[END]
```

Compound indexes

Compound indexes are indexes defined on two or more fields (see Database basics/Compound indexes). Compound indexes can be defined after the `[INDEXES]` tag, as in:

```
[TABLE]
Students
[FIELDS]
LastName string
FirstName string
Age integer
[INDEXES]
CompoundNdx duplicate LastName ascending FirstName ascending
[END]
```

The general syntax of a compound index definition is

```
[INDEXES]
Indx1 UniqueOrDup [FieldName AscOrDesc]1..10
```

where:

- `Indx1` is the name of the compound index
- `UniqueOrDup` is either "unique" or "duplicate", depending upon whether or not you allow duplicate entries for that index
- `FieldName` is the name of a field defined under the `[FIELDS]` tag
- `AscOrDesc` is "ascending" if you want that field sorted low-to-high, or "descending" otherwise.

Up to ten `FieldName AscOrDesc` couples can be defined for a single compound index.

Adding comments to database descriptors

Database descriptors can also contain comments in much the same way Authorware Scripts do. In Script, comments are preceded by double hyphens ("`--`") and must be followed by a `CARRIAGE_RETURN`. In database descriptors, comments must be preceded by `(*` and be followed by `*)`. They can include any sequences of characters, including `CARRIAGE_RETURN`s.

Example:

```
(*
   description of the Mega-Cookbook recipes table version 1.1
   by Bill Gatezky, 14-Feb-97
   This is a valid comment despite the fact that it contains
   Carriage Returns
*)
[TABLE]
Recipes
(* this is also a valid comment *)
[FIELDS]
NameOfRecipe string indexed
...
[END]
```

The comment opening tag for database descriptors must be followed by a blank character such as a space, tab or `CARRIAGE_RETURN`. Likewise, a comment closing tag must be preceded by a blank character. Thus,

```
(*invalid comment: will generate an error*)
```

is an invalid comment, whereas

```
(* valid comment *)
```

is valid.

Multiple tables in a descriptor

If your database has more than one table, each new table follows the description for the previous table, before the `[END]` tag.

Example:

```
[TABLE]
NameOfTable1
[FIELDS]
FieldName1 FieldType1 IndexType1
FieldName2 FieldType2 IndexType2
FieldName3 FieldType3 BufferSize3
[Full-INDEXES]
FieldName3Ndx duplicate ascending

[TABLE]
NameOfTable2
[FIELDS]
FieldName1 FieldType1 IndexType1
FieldName2 FieldType2 IndexType2

etc.

[END]
```

Using the V12-DBE Tool

To create a V12 database using the V12-DBE Tool:

1. Choose File > New…
2. Fill out the Database Descriptor field according to the syntax described in Database descriptors, or load a descriptor from an external file (see Loading a descriptor from a source file),
3. Provide a name, and optionally a password, for your new V12 database,
4. Click the Create button

Loading a descriptor from a source file

Instead of filling out the Database Descriptor field manually in the V12-DBE Tool, you can load one from an external file. Click the **Source** list to select the type of the file that contains the descriptor information. File types that are supported by the V12-DBE Tool include:

- Text
- DBF file
- V12 file
- Template
- FoxPro file
- Access file
- Excel file
- SQL Server Database

If your data is already in one of the database file formats listed here, you can simply use that database to retrieve the descriptor information. Click the **Load…** button and browse to the file that contains the descriptor information.

Once the descriptor information appears in the Descriptor box, you may edit it. For instance, your source database may contain several tables, and you may only want to include a subset of these in your V12 database.

For more information, see the V12-DBE Tool's User Manual. The V12-DBE Tool and User Manual are available for download from the Free Tools section of Integration New Media's web site:

http://www.integrationnewmedia.com/products/v12authorware/tools/.

Script the database creation

Automating the creation a V12 database through Scripts with V12-DBE consists in three steps:

a. Create an Xtra instance of the database with NewObject

b. Define its structure with `mReadDBstructure`

c. Build the database with `mBuild`

The general form of a database creation Script is:

```
gDB := NewObject("V12dbe", FileLocation ^ "filename.v12", "create",
   Password)
if V12Status() <> 0 then GoTo(@"NotifyUser")
mReadDBStructure(gDB, InputType, other params)
if V12Status() <> 0 then GoTo(@"NotifyUser")
CallObject(gDB, "mBuild")
if V12Status() <> 0 then GoTo(@"NotifyUser")
DeleteObject(gDB)
gDB := 0
```

where:

- `FileName` is the full pathname of the V12 database to create
- `Password` is the password to protect `FileName`

- `InputType` is one of "Text", "Literal", "DBF", "V12", "FoxPro", "Access", "Excel" or "SQL Server".
- *`other params`* are one or more parameters depending on the selected `InputType`.
- The resulting V12-DBE database can be immediately verified with `mDumpStructure` (see View the structure of a database).

`mReadDBStructure` reads the *structure* of a DBF file, not its content. To import the content of a DBF file, see Importing from a DBF File.

Step 3a: Create a database Xtra instance

Use the `NewObject` method to create a database Xtra instance.

Syntax:

```
gDB := NewObject("V12dbe", "Name", "create", "Password")
```

The parameters you provide are:

- `Name`: the name of the new database file, including its path if needed (see Dealing with pathnames in Using Xtras).
- `"Create"` or the `Mode`: the mode in which the Xtra instance is defined. In this case, the mode is `Create` (create a new database file). Other possible modes are `ReadOnly, ReadWrite` and `Shared ReadWrite`. See Open an existing database.
- `Password`: the password is required if you wish to protect your database against tampering and/or data theft. You can lock the database with a password, but make sure to record it in a safe place. If you forget it, you will not be able to open your database again.

Example:

```
gDB := NewObject("V12dbe", "Catalog.V12", "Create", "top secret")
```

Note: For a number of reasons, the creation of an Xtra instance can fail (insufficient memory, invalid file path, etc.) Always make sure that your database instance is valid by checking V12Error (see Errors and defensive programming) or ObjectP (see Checking if *NewObject* was successful in Using Xtras) before pursuing the database creation process.

Step 3b: Define the database structure

The next method, after successfully creating a database Xtra instance, is to call `mReadDBstructure` to read in the database structure you designed at Step 1: Decide on a Data Model.

`mReadDBstructure` requires one the following inputs:

- a database descriptor as defined in Database descriptor above. Such as descriptor is supplied either as a text file or as a literal (i.e. a Authorware field or variable),
- a DBF file (DBase) which serves as a table template,
- a V12 database which serves as a database template,
- a directory containing one or more MS FoxPro files which serve collectively as a database template (Windows-32 only, requires the FoxPro ODBC driver),
- a MS Access database which serves as a database template (Windows-32 only, requires the Access ODBC driver),
- a MS Excel workbook which serves as a database template (Windows-32 only, requires the Excel ODBC driver),
- a MS SQL Server data source which serves as a database template (Windows-32 only, requires the MS SQL Server ODBC driver).

Note: `mReadDBStructure` reads the *structure* of a DBF file, not its content. Use `mImport` to import the content of a DBF file.

Note: A valid database needs at least one table, and each table must contain at least one field and at least one index.

(See Appendix 2: Database Creation and Data Importing Rules for complete examples of each of the above variations of `mReadDBStructure`)

It is always a good practice to check the value returned by `V12Error()` or `V12Status()` after calling `mReadDBstructure` (see Errors and defensive programming) to find out if an error occurred. You may also call `mDumpStructure` right after calling `mReadDBstructure` to check the actual database structure V12-DBE will build once `mBuild` is called.

Database structure translation rules from the above ODBC-compliant databases to V12 Databases vary according to the specific ODBC driver installed on your computer.

Step 3c: Build the database

Note: For `mBuild` to create a licensed database (that is, one that does not display an UnRegistered splash screen when opened), a V12-DBE license file must be present on your Mac or PC. Since the V12-DBE license file cannot be delivered to the end-user, `mBuild` cannot be used to create new databases at runtime. If your application needs to create new databases at runtime, use `mCloneDatabase` (see Cloning a database).

Once the database structure is read by `mReadDBstructure`, whether from a text file, a DBF file or otherwise, build the database by calling `mBuild`. `mBuild` checks if the database is well defined and creates the file on your disk.

Syntax:

```
CallObject(database_instance, “mBuild”)
```

Once the database file is built, the database instance remains valid and data can be immediately imported into the file. It is as if the database was opened in ReadWrite mode.

View the structure of a database

You can view the structure of a database with `mDumpStructure`.

Syntax:

```
CallObject(database_instance, "mDumpStructure")
```

Example:

```
DBdump := CallObject (gDB, "mDumpStructure")
```

The above example sets the Authorware variable `Dbdump` to the structure of the database referred to by `gDB`.

```
(*
    Structure of file 'HardDisk:myDatabase.V12'
    created on Thu Apr 25 15:55:07 2002,
    last modified on Tue May 14 15:31:53 2002,
    file format version = V12,3.3,Multi-User
*)

[TABLE]
Articles

[FIELDS]
name string 256
category string 256
price Float
catalognumber Integer
description string 600

[INDEXES]
nameNdx duplicate name ascending     (* Default index *)
categoryNdx duplicate category ascending
priceNdx duplicate price ascending
cat#Ndx unique catalognumber ascending
catNameNdx duplicate category ascending name descending
```

```
[FULL-INDEXES]
description

[END]"
```

Note that the date/hour of the last modification mentioned in the header of the above output is provided by the Operating System. Therefore, it reflects the date/hour at which the V12 database was *closed* regardless of when the *modification* occurred.

This output is fully compatible with the database descriptors discussed in *Database Descriptors* and thus, can be used as is with `mReadDBstructure`.

Step 4: Import data into a V12-DBE database

Note: `mImport` was introduced with V12-DBE version 3.0. It replaces the former `mImportFile` method and is more comprehensive. Although `mImportFile` is still supported in V12-DBE version 3.2, it will be phased out in future versions.

In Step 3: Create a database, you created a properly structured (although empty) V12 database. Step 4 explains how to import the data prepared at Step 2: Prepare the Data into your V12 database.

You can import data into a V12 database through one of the two following methods:

- using the V12-DBE Tool. This is a convenient point-and-click environment for small projects.
- using V12-DBE's mImport method in a Script handler. This approach is efficient when you need to experiment with your database structure or data before committing to a final form. However, it requires a bit more up-front effort to write/adapt Script handlers than simply using the V12-DBE Tool.

For extensive examples on how to import databases from a variety of sources, including Microsoft Access, Microsoft Excel, FoxPro

Import data with the V12-DBE Tool

To import data using the V12-DBE Tool:

1. Choose File > Open… to open the V12 database you want to import data to. A newly created V12 database automatically opens and data can be immediately imported to it.
2. In the File menu, you will see the following options for importing data:
   - Import Text File…
   - Import DBF File…
   - Import from V12…
   - Import from FoxPro…
   - Import from Excel…
   - Import from Access…
   - Import from SQL Server…

   Choose the appropriate option for the format for your data.
3. Browse through your disk to locate the file containing the data to import and fill in any other information necessary to open the file. For some formats you may also need to specify a table name. Click Import.

If the source data is in more than one file, you can successively import them by repeating the above steps.

Script the data importing

`mImport` imports data to a V12-DBE table both at authoring time (i.e., in Authorware's development environment) and at runtime (i.e., from a packaged piece).

`mImport` is very flexible and can be adapted to a large number of situations. It can import data from:

- a Text file
- a literal value, such as a string, a variable, etc.

- a DBF file
- a V12 database
- a Script list
- a MS Access database through an ODBC driver (Win-32 only)
- a FoxPro file through an ODBC driver (Win-32 only)
- a MS Excel file through an ODBC driver (Win-32 only)
- a MS SQL data source through an ODBC driver (Win-32 only)

Data type translation rules from the above ODBC-compliant databases to V12 Databases vary according to the specific ODBC driver installed on your computer.

The general form of a table importing script is:

```
-- create a V12dbe instance
gDB := NewObject("V12dbe", database_filename, mode, password)
if V12Status <> 0 then GoTo(@"NotifyUser")
-- create a V12table instance
gTable := NewObject("V12table", CallObject (gDB, mGetRef),
  TableName)
if V12Status <> 0 then GoTo(@"NotifyUser")
-- import data
CallObject(gTable, "mImport", InputType, InputSource, other params)
if V12Status <> 0 then GoTo(@"NotifyUser")
-- free the V12table and V12dbe instances
DeleteObject(gTable)
DeleteObject(gDB)
gTable := 0
gDB := 0
```

As for any V12table method, valid instances of V12dbe and V12table must exist before the method is invoked. This is explained in details in Creating Instances.

`mImport`'s syntax varies significantly according to the selected input source. This is explained in details in Import data with mImport below.

Deleting Xtra instances when they are no longer needed is mandatory, as explained in Closing an Xtra, to make sure that the imported data is secured on hard disk.

Import data with mImport

The general syntax for `mImport` is:

```
CallObject(table_instance, "mImport", InputType, InputSource, other
   params)
```

where:

- `InputType` is one of "Text", "DBF", "Literal", "List", "PropertyList", "V12", "Access", "FoxPro", "Excel" or "SQLserver".
- `InputSource` is the data to import or a reference to the data to import. It varies according to the selected `InputType`.
- *`other params`* are parameters that depend upon the selected `InputType`. For example, if `InputType` is "text", *`other params`* is an optional property list that specifies the source text file's field delimiter, record delimiter, etc. If `InputType` is "Access", *`other params`* are the user name, password and table to import. The details are explained below.

See Appendix 2: Database Creation and Data Importing Rules for complete examples of each of the above variations of `mImport`.

Step 5: Implementing the user interface

Tip: If you chose to script the database creation and importing processes, once the database file is ready you do not need those scripts any longer. Moreover, they do not necessarily need to be delivered to the end-user.

At this point, you may want to consider removing those scripts, or storing them in an appropriate place. You may also want to keep all the scripts related to project creation in a single Authorware piece.

Steps 1 through 4 (Step 1: Decide on a Data Model through Step 4: Import data into a V12-DBE database) explain how to design, build and import data into a V12-DBE database.

This section discusses the elements needed to manage your V12-DBE database at runtime.

There are two basic strategies you can follow when creating your Macromedia Authorware front-end to your data:

1. Using the V12 Knowledge Objects Library
2. Using Scripting

Using the V12-DBE Knowledge Objects library

Tip: Before you commit yourself to using the V12-DBE Knowledge Objects Library in your project, you may first want to ask support@IntegrationNewMedia.com or other V12-DBE users on V12-L <http://www.IntegrationNewMedia.com/support/list/ > for advice.

The fastest and easiest way to implement V12-DBE into your project's user interface is to use the V12-DBE Knowledge Objects. Download them from the V12 for Authorware product page on our web site:
http://www.IntegrationNewMedia.com/products/v12authorware/tools/

Using scripts

As with any V12-DBE method, a valid V12dbe or V12table Xtra instance (depending on which Xtra the method belongs to) must exist before the method is invoked.

Generally, you create instances of V12dbe and V12table at the beginning of your movie, store their references in variables and use those instances throughout your project.

Likewise, at the end of the execution of your movie, you delete those variables, thus disposing of the Xtra instances and closing the V12 database file.

The creation of such Xtra instances is often referred to as Opening a Database and Opening a Table. Disposing the Xtra instances is often referred to as Closing the Database and Closing the Table instances.

Open and close a database, a table

Open an existing database

Use the `NewObject("V12dbe"…)` method to open an existing V12 database. If your V12 database is not created yet, see Step 3: Create a database to learn how to create it.

Syntax:

```
gDB := NewObject("V12dbe", database_filename, mode, password)
```

Opening a database means creating a `V12dbe` Xtra instance with the following parameters:

- `database_Filename`: the `name` if the database file. This is usually a filename preceded by the Scripting function `FileLocation` to indicate that the file is located in the same folder as the current Authorware piece (see Dealing with pathnames in Using Xtras).

- `mode`: the mode in which the Xtra instance is opened. To allow for modifications to the database, open it in "Shared ReadWrite" or "ReadWrite" mode. If you open your database in "Shared ReadWrite" mode, up to 128 users can access your database simultaneously (see Multi-user access). If you open it in "ReadWrite" mode, only one user at a time can access your database. If you do not allow modifications to your database, open it in "ReadOnly" mode.
- `password`: the password. If `you` do not use the correct password, the database cannot be opened.

Example:

```
gDB := NewObject("V12dbe", FileLocation ^ "Catalog.V12", "ReadWrite",
   "top secret")
```

Always make sure that the `NewObject` method succeeded by checking the validity of the returned reference with `V12Status`.

Example:

```
gDB := NewObject("V12dbe", FileLocation ^ "Catalog.V12", "ReadWrite",
   "top secret")
if V12Status() <> 0 then GoTo(@"NotifyUser")
```

Open a table

Records belong to tables. Creating new records, reading the contents of records, and searching and sorting records are operations that are performed on tables. Prior to performing any of these operations, you must create a table Xtra instance

Syntax:

```
gTable := NewObject("V12table", CallObject(gDB, "mGetRef"),
    TableName)
```

To create a table Xtra instance, use the `NewObject` method with the following parameters:

- `gDB`: the database Xtra `instance` to which the current table belongs.
- `TableName`: the name of the `table` to open.

Example:

```
gTable := NewObject("V12table", CallObject(gDB, "mGetRef"), "Articles")
```

`mGetRef` is a standard Xtra method that returns the exact reference of an Xtra instance.

Always make sure that the `NewObject` method succeeded by checking the validity of the returned reference with `V12Status`.

Example:

```
gTable := NewObject("V12table", CallObject(gDB, "mGetRef"),
    "Articles")
if V12Status() <> 0 then GoTo(@"NotifyUser")
```

This is a complete example of a script that open a database and one of his table:

```
gDB := NewObject("V12dbe", FileLocation ^ "Catalog.V12",
    "ReadWrite", "pwd")
if V12Status() <> 0 then GoTo(@"NotifyUser")

gTable := NewObject("V12table", CallObject(gDB, "mGetRef"),
    "Articles")
if V12Status() <> 0 then GoTo(@"NotifyUser")
-- other init instructions
```

Close a table

To close a table, call the DeleteObject method and set the variable that refers to it to 0.

Example:

```
DeleteObject(gTable)
gTable := 0
```

Close a database

To close a V12 Database, call the DeleteObject method and set the variable that refers to it to 0.

Example:

```
DeleteObject(gDB)
gDB := 0
```

Always make sure to dispose of all V12table instances before you dispose of the V12dbe instance that contains them.

Selection and current record

To read or write data to a record, set it as the current **record**. The current record concept is strongly related to the concept of **selection**. Both concepts are fundamental to this section. See Database basics earlier in this manual for more details.

At any time, the selection is sorted according to one of its fields. You can enforce that sorting order with `mOrderBy` (see Sort a selection (mOrderBy)). Otherwise, the selection's sorting order would be defined by the index chosen by V12-DBE for its last search. The field that determines the selection's sorting order is called the **master field**.

Selection at startup

When a table is first opened, its selection is the entire content of that table sorted by the field that is indexed by the default index. The first record of that selection – which is also the first record of the table – *is* the current record. The default index is the first index that was defined for the table in the database descriptor. You can use `mDumpStructure` to verify which of the table's indexes is the default index (see View the structure of a database).

You never need to explicitly manage indexes in V12-DBE. The best index is always chosen by V12-DBE to perform a search.

Select all the records of a table

Note: If you want the results of your query to be sorted, always call `mOrderBy` before calling `mSelectAll`. See Sort a selection (mOrderBy)

Call `mSelectAll` at any time to set the selection to the whole table..

Syntax:

```
CallObject(table_instance, "mSelectAll")
```

Example:

```
CallObject(gTable, "mOrderBy", "StudentID", "ascending")
CallObject(gTable, "mSelectAll")
```

This example sets the selection to the whole table as referred by `gTable`, in ascending order of StudentID's (lowest to highest). The field "StudentID" must be indexed for `mSelectAll` to work efficiently. Otherwise, it would be very slow.

Browse a selection

Browsing a selection means changing the position of the current record. The following methods enable you to change the current record in a selection (to set the current record to various values related to a given selection).

mGetPosition

`mGetPosition` checks the position of the current record in a table and returns an integer between one and the total number of records in the selection.

Example:

```
currRec := CallObject (gTable, "mGetPosition")
-- the current record's position is assigned to the variable currRec
```

mGoNext

`mGoNext` sets the current record to the record following the current record.

Example:

```
CallObject (gTable, "mGoNext")
```

Suppose that the current record is the tenth item in the selection. After calling `mGoNext`, the current record becomes the eleventh. If the selection contains only ten records, the current record does not change and a warning is reported by V12-DBE (see Errors and defensive programming)

mGoPrevious

`mGoPrevious` sets the current record to the record preceding the current record.

Example:

```
CallObject(gTable, "mGoPrevious")
```

Suppose that the current record is the tenth item in the selection. Upon calling `mGoPrevious,` the current record becomes the ninth. If the current record is the first record of the selection, upon calling `mGoPrevious` the current record does not change and a warning is reported by V12-DBE (see Errors and defensive programming)

mGoFirst

`mGoFirst` sets the current record to the first record of the selection.

Example:

```
CallObject(gTable, "mGoFirst")
```

mGoLast

`mGoLast`sets the current record to the last record of the selection.

Example:

```
CallObject(gTable, "mGoLast")
```

mGo

`mGo` takes an integer parameter (call it *n*) and sets the current record to the n^{th} item of the selection.

Example:

```
CallObject(gTable, "mGo", 11)
```

This example sets the current record to the eleventh record of the selection. If no such record exists, `mGo` signals a warning.

mFind

> **Note:** Because `mFind` uses the selection's Master Field, it is advised that you call `mOrderBy` with the appropriate field before calling `mSelect` and `mFind`. (see Search data with mSetCriteria).
> If you don't call `mOrderBy`, `mFind` sets the current record based on the Master Field chosen by default by V12-DBE, which is either the one indexed by the default index (if the table was just opened), or the one indexed by the best index chosen by V12-DBE during the last search.

`mFind` sets the current record to one, in the selection, whose Master Field equals or starts with the keyword provided as a parameter (see definition of Master Field in Selection and current record).

`mFind` is a great complement to `mGo,` which can set the current record only based on its position in the selection.

The syntax is:

```
CallObject(gTable, "mFind", "First", Keyword)
CallObject(gTable, "mFind", "Next")
CallObject(gTable, "mFind", "Previous")
```

where `Keyword` is the value to look for in the Master Field. If the Master Field is of type String, the matching record's content must *start with* `Keyword`. If it is of type Integer, Float or Date, it must *equal* Keyword.

Use the first form (with the "First" parameter), if you want the new current record to be the first one of the selection that matches `Keyword`

Use the second form (with the "Next" parameter) if you want it to be the next record in the selection relative to the present current record. Use the third form ("Previous") if you want it to be the previous record in the selection relative to the present current record.

If, for example, you run this script:

```
CallObject(gTable, "mSetCriteria", "Age", ">", 30)
CallObject(gTable, "mOrderBy", "LastName")
CallObject(gTable, "mSelect")
```

and get this selection:

FirstName	LastName	Age
Marie	Curie	39
Albert	Einstein	75
Kurt	Gödel	36
Mona	Karp	53
Joe	Karp	31
Richard	Karp	62
Eric	Kartman	31
Marshall	McLuhan	48
Claude	Shannon	33
Alan	Turing	36
John	Von Neumann	51

The selection's Master Field is "LastName". Thus, a call to `mFind` would automatically look for values in this field. For example:

```
CallObject(gTable, "mFind", "First", "Kar") -- current rec becomes
    Mona Karp's
CallObject(gTable, "mFind", "Next") -- current rec becomes Joe
    Karp's.
CallObject(gTable, "mFind", "Next") -- current rec becomes Richard
    Karp's.
CallObject(gTable, "mFind", "Next") -- current rec becomes Eric
    Kartman's.
CallObject(gTable, "mFind", "Next") -- current rec remains Eric
    Kartman's.
CallObject(gTable, "mFind", "Previous") -- current rec becomes
    Richard Karp's.
```

mFind can be used to quickly locate one occurrence of a keyword in a selection where many duplicate values exist, as opposed to mSetCriteria and mSelect, which find all occurrences but need more time.

Read data from a database

In order to read or write the content of a record, you must first set it as the current record. Setting the appropriate current record is accomplished by use of the mGoNext, mGoPrevious, mGoFirst, mGoLast, mGo and mFind methods (see Browse a selection).

Read fields of type string, integer, float and date

Once the current record is properly set, mGetField retrieves the data from a specific field.

Syntax:

```
var := CallObject(table_instance, "mGetField", fieldName [,
   dataFormat])
```

Example:

```
sID := CallObject(gTable, "mGetField", "StudentID")
```

This example stores the content of the StudentID field from the current record in the variable sID. You do not need to specify the type of field you are reading. The Script variable is automatically set to the appropriate type after a successful call to mGetField (see Typecasting in Database basics).

Example:

```
Avg := CallObject(gTable, "mGetField", "AverageScore", "99.99")
```

This example retrieves the formatted content of the AverageScore field to the Avg variable. The formatting is according to the pattern "99.99". That is, if the field AverageScore contains the value 1245.5, the string "1,245.50" is returned by mGetField. Note that the result of a formatted value is always a string.

Data formatting applies to mGetField the same way it does to mDataFormat. If two distinct formatting patterns are applied to a field with the mGetField option and mDataFormat, the mGetField option overrides mDataFormat. See Data formatting for a complete explanation on formatting patterns.

> **Note**: mGetField retrieves only unformatted text. If you store styled text in a V12 record, you can retrieve the text without the style formatting using mGetField.

Read one or more entire records

mGetSelection allows for the retrieval of one or more fields in one or more records of the selection. The result is one of the followings:

- a string where fields are delimited by TABs and records by CARRIAGE_RETURNs (the default delimiters), or by any other custom delimiters you specify.
- a list of lists, where each sub-list represents a record and each item of each sub-list is the data contained in the corresponding field.
- a list of property lists, where each sub-list represents a record and each item is a property/value pair: the property is the name of the field and the value is the data contained in it.
- mGetSelection is powerful and flexible. Its behavior depends on the syntax used to call it. The syntax for mGetSelection to return a result of type String is:

CallObject(table_instance, "mGetSelection" [, "LITERAL", [From [, #recs [, FieldDelimiter [, RecordDelimiter [, FieldNames]*]]]])

The syntax for `mGetSelection` to return a Script list is:

```
CallObject(table_instance, "mGetSelection", "LIST" [, From [, #recs [,
   FieldNames ]* ]])
```

The syntax for `mGetSelection` to return a Script property list is:

```
CallObject(table_instance, "mGetSelection", "PROPERTYLIST" [, From [,
   #recs [, FieldNames ]* ]])
```

where:

- `gTable` is the instance of the table from which records must be retrieved (mandatory parameter).
- `From` is the number of the first record to retrieve data from. It is optional. The default value is 1.
- `#recs` is the number of records to retrieve starting from record number `From`. It is optional. The default value is the number of records between `From` and the end of the selection plus 1 (convenient to retrieve all the records of a selection starting from record number `From`).
- `FieldDelimiter` is the character to use as the field delimiter. It is optional. The default field delimiter is a `TAB`.
- `RecordDelimiter` is the character to use as the record delimiter. It is optional. The default field delimiter is a `CARRIAGE_RETURN`.
- `FieldNames` are the names of the fields to retrieve, in the specified order. If the field names are omitted, `mGetSelection` returns the contents of all the fields of `gTable`, in their order of creation.
- Besides `gTable`, all other parameters are optional. However, if a parameter is present, all its preceding ones must also be present. For example, if `#recs` is present, `result_format` and `From` must also be present.
- `mGetField` requires that you set the current record to the record you need to retrieve data from. `mGetSelection` does not.

See Appendix 3: mGetSelection examples for complete examples of each of the above variations of `mGetSelection`.

Note Because `mOrderBy` uses the selection's Master Field, it is recommended that you call `mOrderBy` with the appropriate field before calling `mSelect` and `mGetUnique`. If you don't call `mOrderBy`, `mGetUnique` returns unique values from the Master Field chosen by default by V12-DBE, which is either the one indexed by the default index (if the table was just opened), or the one indexed by the best index chosen by V12-DBE for the last selection. See Selection and current record.

Read unique values of a field

`mGetUnique` returns unique values of the Master Field in a string or a list (See Selection and current record for a definition of Master Field).

Syntax:

```
a := CallObject(gTable, "mGetUnique", "LITERAL")
b := CallObject(gTable, "mGetUnique", "LIST")
```

`mGetUnique` is very convenient to populate a user interface element (such as scrolling list or pull-down menu) with search values that are relevant only for a specific database and context.

Example: In a clothing catalog, you want to display only the available colors for a specific category and size of product (e.g., T-shirt and XXL). You run this script:

```
CallObject(gTable, "mSetCriteria", "category", "=", "T-shirt")
CallObject(gTable, "mSetCriteria", "and", "size", "=", "XXL")
CallObject(gTable, "mOrderBy", "color")
CallObject(gTable, "mSelect")
ScrollList := CallObject(gTable, "mGetUnique", "LITERAL")
```

This script retrieves unique values of the "color" field (which is the Master Field) to the field "ScrollList". Assuming that your selection contains 30 records (10 with Color = "Red", 10 with Color = "Green" and 10 with Color = "Blue"), the above script puts the string:

```
Blue
Green
Red
```

in the variable "ScrollList".

Running this script:

```
CallObject(gTable, "mSetCriteria", "category", "=", "T-shirt")
CallObject(gTable, "mSetCriteria", "and", "size", "=", "XXL")
CallObject(gTable, "mOrderBy", "color")
CallObject(gTable, "mSelect")
ScrollList := CallObject(gTable, "mGetUnique", "LIST")
```

returns the list:

```
[ "Blue", "Green", "Red" ]
```

Data formatting

`mDataFormat` assigns a display pattern to a field so that all data read from that field are formatted according to that pattern. All V12-DBE methods that read data from a formatted field are affected. These include `mGetField` and `mGetSelection`

Syntax:

```
CallObject(table_instance, "mDataFormat", FieldName, Pattern)
```

The following example forces all data retrieved from the field `AverageScore` to be formatted with 2 integral digits and 2 decimal places.

Example:

```
CallObject(gTable, "mDataFormat", "AverageScore", "99.99")
```

`mDataFormat` can be applied to fields of type `float`, `integer` and `date`. `String` fields cannot be formatted.

To reset the formatting of a pattern to its original value, call `mDataFormat` with an empty string.

Example:

```
CallObject(gTable, "mDataFormat", "AverageScore", "")
```

Format integers and floats

Valid patterns for fields of type `integer` and `float` contain:

- `9` designates a digit at that position (possibly 0),
- `#` designates a digit or a space at that position,
- `.` (period) designates the decimal point,
- any other character is interpreted literally.

This example forces the output of the field `ratio` to 2 integral digits, 2 decimal places and a trailing "%" sign:

```
CallObject(gTable, "mDataFormat","ratio", "99.99%")
TheRatio := CallObject(gTable, "mGetField","ratio")
```

If the value in field `ratio` is 34.567, the displayed string is "34.57%".

The pattern "`###9999`" forces the output of an integer field to be formatted with no less than four digits and with three leading spaces, if necessary. Thus:

```
4           is formatted as    "   0004"
123         is formatted as    "   0123"
314159      is formatted as    " 314159"
3141592     is formatted as    "3141592"
31415926    is formatted as    "#######"
```

The last formatting in the above example fails because an eight-digit integer does not fit in a seven-digit pattern.

The pattern "(999) 999-9999" is convenient for formatting phone numbers stored as integers. For example:

```
CallObject(gtable, "mDataFormat", "phone", "(999) 999-9999")
thePhone := CallObject(gTable, "mGetField", "phone")
-- returns something formatted as "(514) 871-1333"
```

Format dates

Valid patterns for fields of type `date` are combinations of:

- `D` for days,
- `M` for months,
- `Y` for years,
- any other character is interpreted literally.

This example formats the date in the "Year-Month-Day" numerical format:

```
CallObject(gTable, "mDataFormat", "TheDate", "YY-MM-DD")
MyDate := CallObject(gTable, "mGetField", "TheDate")
```

> **Note:** If a formatting pattern is assigned to a field, all values retrieved from that field become strings (see Typecasting in Database basics).

Assume the content of field `TheDate` for the current record is April 30, 1945 – the returned string is "45-04-30".

`D`s, `M`s and `Y`s can be combined in the following way:

To format	Use this sequence
Days as 1-31	D
Days as 01-31	DD
Weekdays as Sun-Sat	DDD
Weekdays as Sunday-Saturday	DDDD
Months as 1-12	M
Months as 01-12	MM
Months as Jan-Dec	MMM
Months as January-December	MMMM
Years as 00-99	Y or YY
Years as 1900-9999	YYY or YYYY

Examples:

The pattern	Formats 5 January 1995 as
D	5
DDDD	Thursday
MM	01
DD-MM	05-01
MMM DD, YY	Jan 05, 95
On D MMMM, YYYY	On 5 January, 1995
'Weekday='DDDD; 'Month=' MMMM	Weekday=Thursday; Month=January

In this last example, apostrophes around 'Weekday' and 'Month' are mandatory, otherwise the "d" in *Weekday* and the "m" in *Month* would interfere with the pattern itself. To specify real apostrophes within date patterns, use two consecutive apostrophes.

When a table is first opened, the default format of all its Date fields is set to "YYYY/MM/DD".

By default, the names for the months in V12-DBE are (`MMMM`)

```
January, February, March, April, May, June, July, August, September,
   October, November, December
```

The short names for the months are (`MMM`)

```
Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
```

The names for the weekdays are (`DDDD`)

```
Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
```

The short names of the weekdays are (`DDD`)

```
Sun, Mon, Tue, Wed, Thu, Fri, Sat
```

All of these names can be replaced by custom names through the properties of the database (see Properties of databases).

Add records to a database

To add a new record to a V12-DBE table, use `mAddRecord`, then call `mSetField` as many times as needed (typically once for each field of your table) and finally call `mUpdateRecord`.

In this example, a new record is created for the item "goggles" and its price is set to $158.99:

```
CallObject(gTable, "mAddRecord")
CallObject(gTable, "mSetField", "ItemName", "Goggles")
CallObject(gTable, "mSetField", "Price", 158.99)
CallObject(gTable, "mUpdateRecord")
```

If `mUpdateRecord` is not called, the record created with `mAddRecord` is not saved to the database. After calling `mUpdateRecord`, the record is created and kept in a cache: it is not immediately written to disk. Thus, if the computer crashes or a power failure occurs, the database file on disk may become corrupt. To ensure that the newly added records are saved onto the hard disk, Flush the database to the disk (see FlushToDisk).

Note: Only `mSetField` can be called after `mAddRecord` and before `mUpdateRecord`. Calling any other method aborts the new record adding process and sets the current record to the previous current record. Thus, if you started to add a record and wish to abort the operation, simply call `mGetField` instead of calling `mUpdateRecord`.

New records are always added to the end of the selection regardless of the criteria used to form the selection.

Update data in a database

> **Note:** Updating the contents of fields that have a full-index may take more time to write to the database than equivalent fields without full-indexes.

Writing data is very similar to reading data. Writing data is accomplished with `mSetField`. Prior to updating a field, you must set the current record, and your intentions must be indicated in V12-DBE with `mEditRecord`. Once this is completed, V12-DBE will update your database with `mUpdateRecord`.

After calling `mUpdateRecord`, the modified record is kept in a cache: it is not immediately written to disk. Thus, if the computer crashes or a power failure occurs, the database file on disk may become corrupt. To ensure that the newly added records are saved onto the hard disk, Flush the database to the disk (see FlushToDisk).

Write to fields of type integer, float and string

In this example, student #666's first name is modified for "Bill" and his average score for 2.72.

The Authorware scripts is as follows:

```
-- first make sure the current record is properly set
CallObject(gTable, "mEditRecord")
CallObject(gTable, "mSetField", "FirstName", "Bill")
CallObject(gTable, "mSetField", "AverageScore", 2.72)
CallObject(gTable, "mUpdateRecord")
```

Like `mAddRecord`, every call to `mEditRecord` must be balanced with a call to `mUpdateRecord`. Calls to `mSetField` will result in an error if not preceded by `mEditRecord`.

If an error occurs when updating a record (e.g. Duplicate Key error in a given field), none of the preceding calls to `mSetField` are taken into consideration.

When writing to a field whose type is not the same as the supplied parameter, V12-DBE tries to cast the parameter to the appropriate type and to interpret it as accurately as possible (see Typecasting).

Example

```
CallObject(gTable, "mSetField", "score", "2.72")  -- stores 2.72
CallObject(gTable, "mSetField", "score", "xyz")   -- stores 0.00
CallObject(gTable, "mSetField", "score", "12.34") -- stores 12.34
```

Write to fields of type date

Writing to a field of type Date is similar to writing to field of type Integer, Float or String, except that V12-DBE requires the date to be supplied in raw format (YYYY/MM/DD).

Example:

```
CallObject(gTable, "mSetField", "theDate", "1993/02/22") -- is valid
CallObject(gTable, "mSetField", "theDate","02/22/1993") -- is not valid
```

Storing the current date in Raw format may be difficult as Authorware's `Date` function returns the current date as formatted in the Control Panel settings of the computer it is running on. In this case, use `mGetProperty(gDB, "CurrentDate")` to retrieve the current date in raw format (see CurrentDate in Properties of databases).

Delete a record

Note: Use this method with caution. There is no way to *undelete* records in V12-DBE. As a general rule, avoid giving direct access of this method to the end-user through your user interface.

Call `mDeleteRecord` to delete the current record.

Syntax:

```
CallObject(table_instance, "mDeleteRecord")
```

Example:

```
CallObject(gTable, "mDeleteRecord")
```

After calling `mDeleteRecord`, the record that follows the record being deleted becomes the new current record. If no record follows the deleted record, the preceding record becomes the new current record. If no record precedes the deleted record, the selection is then empty and the current record is not defined.

Delete all the records of a selection

Call `mSelDelete` to delete all the records of a selection at once.

Syntax:

```
CallObject(table_instance, "mSelDelete")
```

After `mSelDelete` has been completed, the selection is empty and the current record is undefined.

Search data with mSetCriteria

When searching data, you often need to isolate a specific group of records that satisfy a common condition in a table. These conditions are called **search criteria** and the subset of isolated records is the **selection** (see Database basics for an explanation of selections and current records). `mSetCriteria` is the method used to specify search criteria, followed by `mSelect` to trigger the search process.

Syntax:

```
CallObject(table_instance, "mSetCriteria", FieldName, operator, Value)
CallObject(table_instance, "mSelect")
```

Simple search criteria

A search criterion has at least three characteristics:

- `FieldName`: this is a valid field name in the table instance,
- `operator`: this is a comparison keyword. Valid operators are `=`, `<`, `<=`, `>`, `>=`, `<>`, `starts, contains`, `wordStarts` and `wordEquals`.
- `value`: this is the value to which the field contents must be compared, in order to be selected.

This example selects all records with scores lower than 12.

```
CallObject(gTable, "mSetCriteria", "score", "<", 12)
CallObject(gTable, "mSelect")
```

Upon completion of `mSelect`, the resulting selection contains the set of records that satisfy the defined criteria. In the above example, all records that contain a `score` field with a value that is strictly smaller than *12* are selected. In addition, the selection is sorted with an increasing order of prices given that a search with a defined ascending index was performed on that field.

The current record is the first record of that selection. In our example, it would be the record with the lowest score.

If you want the selection sorted in an order other than the one proposed by `mSelect`, you can do so by calling `mOrderBy` right before calling `mSelect`. However, keep in mind that this may cost some additional processing time.

Values provided to `mSetCriteria` need to be in the same type as `FieldName`. As discussed in Database basics / Typecasting, V12-DBE tries to automatically typecast `value` to the proper type. Borderline conditions such as criteria containing extra spaces, carriage returns or other unwanted characters must be avoided.

Example:

```
CallObject(gTable, "mSetCriteria", "score", "<", "100")
```

is strictly equivalent to

```
CallObject(gTable, "mSetCriteria", "score", "<", 100.00)
```

but beware of the unpredictable results of

```
CallObject(gTable, "mSetCriteria", "score", "<", "..100.00..")
```

Operations on fields of type `Date` require that Value be supplied in raw format (see Step 2: Prepare the Data /Dealing with dates). This example locates all records where field `theDate` contains a date occurring before May 21st, 1997.

```
mSetCriteria(gTable, "theDate", "<", "1997/05/21")
```

Sort a selection (mOrderBy)

You can define a sort order on a selection by calling the `mOrderBy` method prior to calling `mSelect`. Specify the sorting order (whether `ascending` or `descending`) and the field upon which the sort is performed.

Example:

```
CallObject(gTable, "mSetCriteria", "LastName", "starts", "Mac")
CallObject(gTable, "mOrderBy", "score", "descending")
CallObject(gTable, "mSelect")
```

The above example selects all hats in `gTable` and returns a selection sorted by a descending order of prices (most expensive to least expensive).

If `mOrderBy` is not called before calling `mSelect`, the sort order of the selection depends on the index used to perform the search. That index is automatically chosen by V12-DBE to optimize the search time. See **mOrderBy** in the **V12-DBE Methods Reference** manual.

Operators

Following is a list of valid operators and their meanings. Although comparisons of `integer`s, `float`s and `date`s are straightforward, comparing strings and custom string types depends on how those comparison rules are defined (see Appendix 4: String and custom string types).

Equal (=)

The "=" operator is used to locate data that exactly match the specified value.

Example:

```
CallObject(gTable, "mSetCriteria", "score", "=", 3.14)
```

specifies a search for records that contain exactly the score 3.14 .

Example:

```
CallObject(gTable, "mSetCriteria", "FirstName", "=", "John")
```

specifies a search for records that contain exactly the first name John. Other first names such as Johnny or John-Paul will not be selected. Since V12-DBE does not

differentiate upper case and lower case characters, “john” and “John” will also be selected.

Not Equal (<>)

The "<>" operator has the opposite effect of the "=" operator. It is used to locate data that are different than the specified value.

Example:

```
CallObject(gTable, "mSetCriteria", "score", "<>", 9.99)
```

specifies a search for all records except those that contain a score of 9.99.

Less than (<)

The "<" operator is used to locate data that are strictly smaller that the specified value.

Example:

```
CallObject(gTable, "mSetCriteria", "score", "<", 10)
```

specifies a search for scores lower then 10. Scores equal to or higher then 10 are not selected.

Example:

```
CallObject(gTable, "mSetCriteria", "LastName", "<", "M")
```

specifies an alphabetical search for records with last names that precede the letter “M”. This includes all last names from A to L, and excludes last names that begin with an M.

Less or equal (<=)

The "<=" operator is used to locate data that are smaller or equal to the specified value.

Example:

```
CallObject(gTable, "mSetCriteria", "score", "<=", 10)
```

specifies a search for scores no higher than 10 (including scores of 10).

Example:

```
CallObject(gTable, "mSetCriteria", "LastName", "<=", "Mad")
```

specifies a search for records with last names that alphabetically precede the word “Mad”. This includes all last names from A to M, including names such as “Macaroni” and “MacDonald” but excluding “McGill” and “Mobutu”.

Greater than (>)

The ">" operator is used to locate data that are strictly larger than the specified value.

Example:

```
CallObject(gTable, "mSetCriteria", "studentID", ">", 950)
```

specifies a search for students who’s ID is larger than 950. Sudent ID #950 will *not* be selected.

Example:

```
CallObject(gTable, "mSetCriteria", "birth date", ">", "1961/12/31")
```

specifies a search for records with a "birth date" field occurring *after* Dec 31st, 1961, (excluding that date). The earliest birth date in the selection should be Jan 1st, 1962 or later.

Greater or equal (>=)

The ">=" operator is used to locate data that are larger or equal to the specified value.

Example:

```
CallObject(gTable, "mSetCriteria", "studentID", ">=", 950)
```

specifies a search for student's who's ID is larger or equal to 950. Student ID #950 *will* be selected.

Example:

```
CallObject(gTable, "mSetCriteria", "birth date", ">=", "1961/12/31")
```

specifies a search for records with a "birth date" field occurring *on* or *after* Dec 31st, 1961. Therefore, the earliest birth date in the selection may be Dec 31st, 1961.

Starts

The "starts" operator can be used with fields of type `string` only (including custom string types). It locates records that start with a given sub-string in the specified field.

Example:

```
CallObject(gTable, "mSetCriteria", "LastName", "starts", "Mac")
```

sets records with last names beginning with "Mac", such as MacIntosh, MacDonald and Mac for selection.

If an index is defined on the field `description`, the search process is very fast. If not, the search takes more time but can be performed nonetheless.

Contains

The "contains" operator can be used with fields of type `string` only (including custom string types). It locates records that contain a given sub-string in the specified field.

Example:

```
CallObject(gTable, "mSetCriteria", "Resume", "contains", "DOS")
```

sets records with resumes containing the string "DOS" for selection.

Searches using the "contains" operator are inherently sequential. They cannot take advantage of any index definition and can be very slow.

Note: Although words such as "hamburger" and "hammer" can be quickly found by the example query uing "wordStarts", the word "ham" will never be found because it is shorter than the minimum word length set for full-indexing, and therefore is not stored in the index.

WordStarts

The "wordStarts" operator can be used only with fields of type `string` (including custom string types) with defined full-indexes. It locates records that contain words that fully or partially match the value specified to `mSetCriteria`.

Example:

```
CallObject(gTable, "mSetCriteria", "fld", "wordStarts", "ham")
```

sets records for selection containing descriptions such as "Gigantic hamburger with fries" and "The greatest hammer in the world". It does not find records containing descriptions such as "Champion" or "Gotham City" because the words in these records don't *start* with the sub-string "ham".

Since "wordStarts" operates on full-indexes, searching is performed very quickly.

WordEquals

The "wordEquals" operator can be used only with fields of type `string` with defined full-indexes. It locates records that contain words that fully match the value specified to `mSetCriteria`.

Example:

```
CallObject(gTable, "mSetCriteria", "fld", "wordEquals", "ham")
```

sets records for selection containing descriptions such as "green eggs and ham". Records containing words such as "hams" or "hamburger" would not be selected.

Since "wordEquals" operates on full-indexes, searching is performed very quickly.

Note: Words shorter than the minimum word length set for full-indexing cannot be looked for with the "wordEquals" or "wordStarts" operators. In such cases, you must use the "contains" operator instead.

Difference between 'Contains' and 'WordStarts'

Why should you bother using the slow "contains" operator if "wordStarts" does the job faster?

Because "wordStarts" requires that a full-index be defined on a field. Full-indexes allow for quick searches, but require more disk space and more time when updating data.

Another reason is that "wordStarts" can only search for *words* that match or begin with a given string. For example, if the description field of a certain record contains the text "Dark chocolate with hazelnuts":

```
CallObject(gTable, "mSetCriteria", "descr", "contains", "cola")
```

would locate that record ("chocolate" contains the sub-string "cola"), whereas

```
CallObject(gTable, "mSetCriteria", "descr", "wordStarts", "cola")
```

would not. This is because no word in the description field *starts* with the string "cola".

Complex search criteria

`mSetCriteria` can also be called with four parameters. The additional parameter is the Boolean operator "AND" or "OR". It is added to the second call to `mSetCriteria` and inserted before the field to be searched.

Example:

```
CallObject(gTable, "mSetCriteria", "LastName", "starts", "Mac")
CallObject(gTable, "mSetCriteria", "and", "score", "<=", 50)
CallObject(gTable, "mSelect")
```

The above example selects the records of all students whose last name starts with "Mac" and whose score is lower or equal to 50.

The first call to `mSetCriteria` should use three parameters, and it can be chained with as many four-parameter calls as needed to specify your query. Using `mSetCriteria` with three parameters will reset and ignore the preceding search criteria.

Another example, using the Boolean "OR" operator is:

```
CallObject(gTable, "mSetCriteria", "LastName", "starts", "Mac")
CallObject(gTable, "mSetCriteria", "or", "LastName", "starts", "Mc")
CallObject(gTable, "mSetCriteria", "or", "LastName", "=", "Mobutu")
CallObject(gTable, "mSelect")
```

It selects all records whose "LastName" field starts either with "Mac" or starts with "Mc" or matches exactly "Mobutu".

Note: Complex searches that use the OR operator are always slower than those that use the AND operator. This is true with V12 Database Engine as well as with most other database management systems.

Complex criteria are very powerful but can be tricky to use. This example illustrates complex criteria.

```
CallObject(gTable, "mSetCriteria", "LastName", "=", "MacDonald")
CallObject(gTable, "or", "mSetCriteria", "LastName", "=", "McGill")
CallObject(gTable, "and", "mSetCriteria", "score", "<=", 50)
CallObject(gTable, "mSelect")
```

This section of script selects all MacDonalds and all McGills who's scores are lower than 50. This is very different from:

```
CallObject(gTable, "mSetCriteria", "LastName", "=", "MacDonald")
CallObject(gTable, "and", "mSetCriteria", "score", "<=", 50)
CallObject(gTable, "or", "mSetCriteria", "LastName", "=", "McGill")
CallObject(gTable, "mSelect")
```

where the selection consists of all MacDonalds with scores lower that 50 and all McGills regardless of their scores.

To illustrate the semantic difference between the two requests, we could express the first as:

```
(LastName = "MacDonald" or LastName = "McGill") and score <= 50
```

whereas the second could be written as:

```
(LastName = "MacDonald" and score <= 50) or LastName = "McGill"
```

Important: The current version of V12-DBE does not have the ability to perform searches such as

```
name = "hat" or (name="helmet" and price<=50)
```

(note the parentheses).

The first two criteria are always grouped first and the third criteria is added to the result.

Partial selections

The selection process can be time-consuming if a large number of records match the criteria you specify. The worst-case scenario is when all the records of a table match the specified criteria. This can handicap your project if you have no control over the queries the end-user can express.

To speed up the selection process, you can limit the number of records V12-DBE places in the selection with this syntax of `mSelect`.

```
CallObject(table_instance, "mSelect", from, #recs)
```

Example:

```
CallObject(gTable, "mSetCriteria", "LastName", "=", "Smith")
CallObject(gTable, "mSelect", 1, 100)
```

The above example returns up to a maximum of 100 records in the selection, regardless of the total number of Smith in the database. If less than 100 Smith exist, all of them would be selected.

To retrieve the next 100 records that contain "Smith" in the "LastName" field, call:

```
CallObject(gTable, "mSelect", 101, 100)
```

Note Partial selections also work with complex searches, but not *all* of them. They are only accepted for complex searches that *do not* use full-text indexes (i.e., `WordStarts` or `WordEquals`).

Check the size of a selection

It is sometimes useful to know the number of items localized in a selection. This is the purpose of the `mSelectCount` method.

Example:

```
CallObject(gTable, "mSetCriteria", "FirstName", "=", "Elmo")
CallObject(gTable, "mSelect")
selSize := CallObject(gTable, "mSelectCount")
```

In this example, the number of records in the selection (the number of items named "Elmo") is stored in `selSize`.

Exporting data

`mExportSelection` allows exporting of data from a V12-DBE table to TEXT or DBF files (DBase III). Only the selected records are exported (i.e. those in the selection). To export a complete table, make sure it is entirely selected first (see Selection and Select all the records of a table).

Exporting in TEXT format

The syntax for exporting all the fields of a table's selection is:

```
CallObject(table_instance, "mExportSelection","TEXT", FileName)
```

The above instruction exports all the fields of the selection to the file named `FileName`. The field and record delimiters are `TAB` and `CARRIAGE_RETURN` respectively.

To specify custom field and record delimiters, use:

```
CallObject(table_instance, "mExportSelection","TEXT", FileName,
   FldDelimiter, RecDelimiter)
```

Example:

```
CallObject(gTable, "mExportSelection", "TEXT", FileLocation ^
   "Output.txt", "~", "%")
```

This example exports the selection in a text file named "Output.txt" with the field delimiter "~" and the record delimiter "%".

`mExportSelection` can also export only selected fields in the following way:

```
CallObject(table_instance, "mExportSelection", FileLocation ^ "TEXT",
   FileName, FldDelimiter, RecDelimiter, Field1, Field2, ...)
```

Example:

```
CallObject(gTable, "mExportSelection", "TEXT", "Data.TXT", TAB, RETURN,
   "ItemName", "catalog number", "price")
```

This example exports the selection in a text file named "Data.TXT" with TAB and RETURN delimiters. The only exported fields are `ItemName`, `catalog number` and `price`, in that order.

The first line in the exported file contains the names of the exported fields separated by the selected field delimiter. The resulting text file is in the character set of the current Operating System (this is relevant only if accented characters are present in the exported data).

`mExportSelection` takes the format patterns specified in `mDataFormat` into account. The sorting order of the exported records is identical to the one set on the selection.

Exporting in DBF format

The parameters for exporting DBF files are identical to those of exporting text, without the field and record delimiters.

Example:

```
CallObject(gTable, "mExportSelection", "DBF", "Goliath.DBF")
-- exports all fields of gTable
```

Or:

```
CallObject(gTable, "mExportSelection", "DBF", "Goliath.DBF",
   "ItemName", "catalog number", "price")
-- exports only fields ItemName, catalog number and price.
```

These rules apply when exporting to a DBF file format

- `String` fields are exported to fields of type Character, if the buffer size of the string field is declared to be no larger than 255 characters. Otherwise, they are exported to field of type Memo.
- `Integer` fields are exported to fields of type Numeric.
- `Float` fields are exported to fields of type Numeric with 10 digits after the fixed point.
- `Date` fields are exported to fields of type Date.

Cloning a database

Cloning a database makes a copy of an existing database file, with all the table, field and index definitions but with none of the data. This is similar to creating a database file from a template rather than starting a new project. Contrary to creating a database with `mReadDBstructure` (which requires a V12-DBE license to create legal V12-DBE databases) this method can be used at runtime.

Syntax:

```
CallObject(db_instance,"mCloneDatabase", new_pathname)
```

Example:

```
CallObject( gDB, "mCloneDatabase", FileLocation ^ "myClone.V12")
if V12Status() <> 0 then GoTo(@"NotifyUser")
gDB_cloned := NewObject("V12dbe", FileLocation ^ "myClone.V12",
   "ReadWrite","secret password")
```

In this example, a new database file named "myclone.V12" is created using the same tables, fields and index definitions, as well as the same password as the database file designated by the global variable `gDB`. This implies that the original database file, designated by `gDB`, must be opened with the appropriate password before cloning. After the new database has been cloned, you need to create an instance of it, using `NewObject`, before you can use it.

Freeing up disk space (packing)

Most database management systems, including V12-DBE, do not reclaim the space freed by deleted records, for the sake of performance. Consequently, as records are created and deleted, the size of the database grows continuously. `mPackDatabase` can be used periodically to reclaim lost bytes.

Syntax:

```
CallObject(database_instance, "mPackDatabase", NewFilePathName)
```

Example:

```
CallObject(gDB, "mPackDatabase", FileLocation ^ "Packed_DB.V12")
CallObject(gDB, "mPackDatabase",
   "LAN/Shared/Projects/Barney/KidsStuff.V12")
```

This example compresses `gDB` into a new file named `Packed_DB.V12` located in the same folder as the current Authorware piece.

At the end of the operation, `database_instance` stays valid (referring to the non-packed database) and `NewFilePathName` is a new file that can be opened with V12-DBE.

If you just need to compress your current database without creating a new file, you can do so by compressing it into a new temporary database, deleting your initial database and renaming the temporary database to your initial database's name.

```
-- let fName be your V12 database's name
-- first make sure to set all table instances to 0
DeleteObject(gTable)
gTable := 0
CallObject(gDB, "mPackDatabase", FileLocation ^ "temp.V12")
if V12Status() <> 0 then GoTo(@"NotifyUser")
DeleteFile(FileLocation ^ fName)
RenameFile(FileLocation ^ "temp.V12", FileLocation ^ fName)
```

Fixing corrupted database files

Databases may become corrupt if a power failure or system crash occurs while updating records. Therefore, V12-DBE is unable to reopen the database and returns an explicit error code when trying to create a database instance.

Some of these corrupt databases can be fixed with `mFixDatabase`. The syntax for `mFixDatabase` is:

```
CallParentObject("V12dbe", "mFixDatabase", pathname, new_pathname)
```

`pathname` is the name of the database to fix and `new_pathname` is the name of the fixed database, which may reside on a different volume.

`mFixDatabase` is a static method (its first parameter is the Xtra library itself, not on an instance of V12dbe). In this example:

```
CallParentObject("V12dbe", "mFixDatabase2, "Crash.V12",
   "Recovered.V12")
```

mFixDatabase tries to read data from "Crash.V12" and saves the data to "Recovered.V12".

Note: `mFixDatabase` attempts to save a corrupted file as much as possible, but there is no guarantee on the result. `mFixDatabase` essentially attempts to rebuild the indexes of a damaged V12-DBE file, but if the file's headers or data clusters are damaged, chances are that the recovery process will fail.

Checking the Vversion of the Xtra

At authoring time, you check the V12-DBE Xtra's version by opening its Get Info window in the MacOS Finder, or by checking its Properties in Windows' Explorer.

Both at authoring time and runtime, you can call `mXtraVersion` to retrieve the version of the Xtra.

Example:

```
xtraversion := CallParentObject("V12dbe", "mXtraVersion")
```

Changing a password

You can change the password assigned to a database by using the `mSetPassword` method. The new password can be an empty string. The syntax is as follows:

```
CallObject(database_instance, "mSetPassword", oldPassword, newPassword)
```

Example:

```
CallObject(gDB, "mSetPassword", "houdini", "ali baba")
```

Dynamically downloading databases via the Internet

V12 has the ability to dynamically download an updated database via the Internet to a local storage device. `V12Download` is the method that you use to replace a database on a user's local drive.

The syntax is as follows:

The syntax is as follows:

```
V12Download(url, local_file, password, completion_callback_variable,
   status_callback_variable)
```

`V12download` resumes and returns control to Authorware immediately after initiating the download query. It updates the variable `status_callback_variable` as frequently as possible during download (generally used to display the download status in the user interface) and updates the variable `completion_callback_variable` when the download is complete (generally used to determine when to open the database and start working with it).

If a local V12 database of the same name already exists, the downloaded file replaces it. The Xtra automatically ensures that it is a valid V12 database and its password is supplied and correct.

For an example of its use, see the **V12 Methods Reference: V12Download.**

`V12DownloadInfo` is a method that you can use to determine the size of the database to be downloaded. This can be useful in determining whether or not the database has changed since the last time it was downloaded. Due to the fact that not all HTTP servers support time and date stamps on files, this method helps you diagnose the status of the database files.

Tip: You may also want to consider having an associated text file or a separate database with a time stamp or other important information within it, which you can use to determine whether or not the database has changed or needs to be downloaded.

Errors and defensive programming

Error management in applications

Effective error management is key to any reliable script or program. If you choose to implement your project with the V12-DBE Knowledge Objects Library, you automatically take advantage of this Library's efficient built-in error management. You just need to make sure that the "Show Alert on Error" check box is checked when dragging/dropping a Knowledge Object.

V12-DBE's Script interface provides methods that allow you to keep a close check on your programming. Use the global functions `V12Status()` and `V12Error()`, to confirm each step of database creation and handling.

As well, Authorware has an interesting tool to help you detect your script errors: the Control Panel (by putting traces), available in the Windows menu. Also, a good tool to take advantage of in database projects that write data to a storage device is FlushToDisk. It will help you ensure that all of your data is written to the storage device at important intervals, flushing the disk cache.

Checking the status of the last method called

Call `V12Status()` after each call to V12-DBE methods (both V12dbe and V12table methods) to check its outcome. `V12Status()` returns *0* if no error occurred during the execution of that method. Otherwise, it returns a non-zero error code.

Example:

```
theID := CallObject(gTable, "mGetField", "StudentID")
if V12Status <> 0 then GoTo(@"NotifyUser")
```

`V12Status()` returns a non-zero result, you can call `V12Error()` to get the details of the error. When called with no parameter – as in `V12Error()-` this global function returns a plain-English explanation of the outcome of the last called method. If an error occurred in that last call to V12-DBE, `V12Error()` provides a detailed contextual report on it.

Example:

```
aPrice := CallObject(gTable, "mGetField", "price")
if V12Status <> 0 then
  GoTo(@"NotifyUser"))
end if
```

NotifyUser would then call V12Error() and display the message to the user. For an example of how to implement this type of feedback, download the **V12-DBE Quiz** on the Demos section of our webiste at: http://www.integrationnewmedia.com/products/v12authorware/demos/.

Note: Usually, you call `V12Status()` to get an error or warning code, then `V12Error()` to get a full explanation of that error or warning. Alternatively, your application can choose to handle specific error codes uniquely.

Errors and warnings

Typically, two types of faults can occur in using V12-DBE:

- **Errors**, which lead to major problems that require that you, stop the execution of your script.
- **Warnings**, which happen while executing certain instructions partially or in borderline conditions that you need to be aware of.

An example of an error is *File not found*, when trying to import data. When a file is not found, it does not make sense to continue the importing operation until the problem is solved.

An example of a warning is *No previous record*, when trying to go to the previous record from the first record of the selection. In such a case, the current record remains valid (although unchanged).

`V12Status()` returns negative codes for errors and positive codes for warnings. Often, the term *error* is used to designate faults of both types (i.e. errors and warnings).

Based on the value returned from `V12Status()` you can display specific messages to your end-users.

Example:

```
CallObject(gTable, "mSetCriteria", "UserName", "=", EnteredName)
errCode := V12Status()
if errCode = -8690 then
    ErrorMessage:="Please fill in your username."
    GoTo(@"Show Message and Retry")
end if
CallObject(gTable, "mSelect”)
```

In this example the application reminds the user to fill in his name before doing the search operation.

For a complete listing of all V12 error codes, consult the V12-DBE Methods Reference and Error Codes manual or V12 Help. Both are available for download from:
http://www.integrationnewmedia.com/support/v12authorware/manuals/.

Using the verbose property

The `verbose` property offers a convenient way to monitor and debug your application’s interactions with a V12 database during the testing phase. When `verbose` is turned “on”, both successful v12 method calls and errors are reported to Authorware’s Control Panel window. Turn `verbose` “on” at the beginning of a script segment that includes several V12-DBE method calls; then turn it “off” at the end of that section of code. Although verbose allows you to monitor database activity while in authoring and testing mode, it should not substitute for checking V12Status and handling runtime errors.

Make sure all calls to `CallObject(gDB, "mSetProperty", "verbose", "on")` are deleted or commented out before distributing your application. For more information on the `verbose` property see Verbose in Properties of databases.

Delivering to the end user

V12-DBE is designed in a way that minimizes any last minute changes needed before delivery to the end-user. Unlike other database management systems where you need to swap the development version of certain files with the runtime versions, no swapping is required with V12-DBE.

Standalone packaged pieces

You deliver the Xtra file *V12-DBE for Authorware.XTR* (on Mac) and/or *V12-DBE for Authorware.X32* (on Windows). For an Xtra to be available to an Authorware packaged piece, you must to place it in a folder named "Xtras" located in the same folder as the packaged piece itself.

V12-DBE databases are cross-platform compatible.

As stated in the licensing agreement, you DO NOT deliver the license file "V12-30a.LIC", which is in the System:Preferences folder of your Macintosh, or the Windows\System folder of your PC. It is not needed, nor is it recommended that you distribute this file to end users, because it contains your personal licensing information.

Web-packaged pieces

Authorware can automatically download Xtras via the Internet. If you are using V12 in your project, you need to ensure that all end-users have the Xtras properly installed and running before executing any V12 functions.

After the completion of a piece, you can package it for the web by using the Web Packager tool from Macromedia. In Authorware 6 and higher, the one-button publishing function creates a web package as well as a regular packaged piece. You can embed the web-packaged piece in an HTML page and put the web-packaged piece and all its external files on a web server.

See the TechNote named, How to rename V12-DBE Xtra for web-packaged pieces, for special instructions for web packaging pieces that use V12-DBE for Authorware.

Testing for end-users

It is always a good idea to thoroughly test the product before delivering it to the end-user. Tests must be performed on computers with configurations very similar to those of end-users. However, if you need to perform end-user tests on the computer that contains the V12-DBE license, you can reproduce an end-user environment by proceeding as follows:

- Make sure Authorware or an Authorware packaged piece is not running.
- Open the System:Preferences folder of your Macintosh, or the (Windows\System folder) of your PC.
- Move the V12*.LIC file out of that folder to the destination of your choice, except of course, the trashcan or the recycle bin.
- Open your project either with a packaged piece or Macromedia Authorware and perform the tests.
- Once the tests are completed, close the packaged piece or Macromedia Authorware and put the license file back in its original folder.

Note: DO NOT attempt to rename or tamper with the license file. If you do, you may need to re-register V12-DBE.

Multi-user access

V12-DBE allows for multi-user access to its databases. This means that a V12-DBE file can be shared by many users, provided the V12-DBE file is available to them on a mounted volume.

An icon on the desktop represents a mounted volume on the Macintosh computer. You can mount such a volume by selecting it in the Chooser, from the Apple menu.

On Windows, a mounted volume is either a volume that is mapped to a drive letter or a volume or partition accessible in the Network Neighborhood.

Opening a file in *Shared ReadWrite* mode

To open a V12 database in a multi-user environment, create a V12dbe Xtra instance in "Shared ReadWrite" mode. Syntax:

```
gDB := NewObject("V12dbe", File_Pathname, "Shared ReadWrite",
   "MyPassword")
```

Mac OS example:

```
NewObject("V12dbe", "MyNetworkDrive:Data:Catalog.V12", "Shared
   ReadWrite", "password")
```

Windows example (F is a mapped volume):

```
NewObject("V12dbe", "F:\Data\Catalog.V12", "Shared ReadWrite",
   "password")
```

Windows example:

```
NewObject("V12dbe", "//BigServer/Data/Catalog.V12", "Shared ReadWrite",
   "password")
```

At most 128 users can open a V12-DBE file in `Shared ReadWrite` mode.

Note: If a user opens a database in "Shared ReadWrite" mode, any attempt to open it in ReadWrite, ReadOnly or Shared ReadOnly mode will fail.
If a database is open in "Shared ReadWrite", other users can also open it in "Shared ReadWrite" mode.
If a database is open in an exclusive mode ("ReadWrite" or "ReadOnly"), it cannot be opened by any other user.

Shared access rules and exceptions

V12 Database Engine relies on Windows and MacOS file and network managers to access shared files on a LAN. Thus, the following exceptions apply:

- If a V12 database is open in Shared ReadWrite mode by one or more V12-DBE clients, new clients can open it only in Shared ReadWrite mode. Shared ReadOnly opening is refused until the last client has dismissed its V12-DBE Xtra instances. If a V12 database is open by one or more V12-DBE clients in Shared ReadWrite mode, any Mac client will also be able to open it in Shared ReadOnly mode.
- If a V12 database is open in Shared ReadOnly mode by one or more V12-DBE clients, both Shared ReadOnly and Shared ReadWrite access are permitted. However, if a client opens the database in Shared ReadWrite mode, all Shared ReadOnly operations are suspended until the Shared ReadWrite instances are dismissed.

As before, only Microsoft and Apple networking protocols are supported. Third party networking software such as Dave, CopsTalks, etc. are not supported.

Note: Although up to 128 users can share the same V12 database over a network, performance degrades rapidly if records must be read or written concurrently. If such situations occur in your project, we recommend using a client-server architecture, such as our GoldenGate Database Connector http://www.GGdbc.com

Shared databases and record locking

If you want to allow users to modify the content of your shared database, note the following rules:

- V12-DBE uses a record locking technique, which means that if a user is editing the current record after a `mEditRecord`, no other user can call `mUpdateRecord` until that user is finished. Any other call to `mEditRecord` fails because the record is locked. Both `V12Error()` and `V12Status()` report such failures, so your application can alert the user with an appropriate message when this occurs.
- Any attempt to retrieve the content of a locked record using `mExportFile`, `mGetSelection,` etc., returns a warning and cancels the action.

If many users proceed to make modifications on the same table simultaneously, synchronization problems may arise between the actual content of the table and the selection as reflected in the users' V12table instances. Such instances must be "refreshed" by invoking `mSelect()`. To detect whether a table has been modified by another user, call `mNeedSelect()` at any time. `mNeedSelect()` returns TRUE if records have been added, deleted or modified since the current instance last called `mSelect()`. In some cases, it is a good idea to check for `mNeedSelect()` on idle and refresh the displayed records when signaled to do so.
Example:

```
If (CallObject(gTable, "mNeedSelect")) then
  CallObject(gTable, "mSelect")
  -- do whatever necessary to refresh the display
end if
```

Counting the number of users

The `sharedRWcount` property is ReadOnly and non persistent. It returns the number of users currently sharing the V12-DBE database file in `Shared ReadWrite` mode.

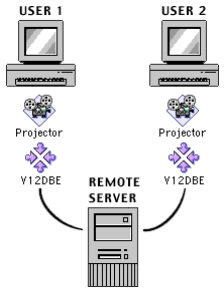
Syntax:

```
CallObject(gDB, "mGetProperty", "sharedRWcount")
```

Possible configurations

Scenario 1

User 1 and User 2 access the same V12 database on a Remote Server. This is the typical multi-user access configuration.

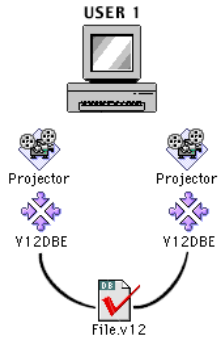


Scenario 2

User 2 accesses the same V12 database file as User 1, on User 1's computer. User 1 and User 2 use separate instances of the projector and V12-DBE Xtra.

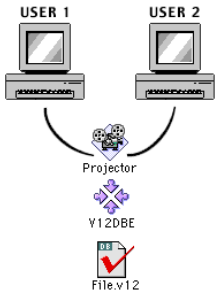
Scenario 3

User 1 with two distinct Projectors, each with its own copy of the V12-DBE Xtra that share a single V12 database. NOTE: for the Mac version to properly run in this scenario, File Sharing must be set and the V12 database must be in a shared folder.



Scenario 4

User 1 and User 2 share the same projector, V12-DBE Xtra, and V12-DBE database file (this scenario requires a locked projector file on a Windows computer).



Progress indicators

> **Note**: `mSelect` preceded by `mSetCriteria` with simple or complex criteria enables the display of a single progress indicator for the selection task, except if the criteria contain `wordStarts` or `wordEquals` operators. In that case, as many progress indicators as criteria are displayed.

V12 Database Engine can display a progress indicator to the user when performing time-consuming tasks such as `mImport`, `mExportFile`, `mGetSelection`, `mSelect`, `mSelDelete`, `mFixDatabase` and `mPackDatabase`. Such a progress indicator can optionally feature a Cancel button to enable users to interrupt the current task. You can also replace the standard V12-DBE progress bar by any custom progress indicator you provide via Authorware and Scripting.

To activate the progress indicator, set the `ProgressIndicator` property to `With_Cancel`, `Without_Cancel` or `UserDefined`. To deactivate it, set it to `None`.

Options of the progressIndicator property

With_Cancel

V12-DBE displays its own progress bar when performing one of the above-mentioned tasks. The user can click on the Cancel button to abort it. You can set the `ProgressIndicator.Message` property to whatever message you wish to display in the upper part of the progress window. If you set the `ProgressIndicator.Message` property to an empty string, V12-DBE displays its own context-dependant message.

Without_Cancel

`Without_Cancel`: V12-DBE displays its own progress bar when performing one of the above-mentioned tasks. No "Cancel" button is shown and the current task cannot be interrupted. You can set the `ProgressIndicator.Message` property to whatever message you wish to display in the upper part of the progress window. If you set the `ProgressIndicator.Message` property to an empty string, V12-DBE displays its own context-dependant message.

None

No Progress Indicator is shown and no callbacks are performed to Script handlers. *None* is the default value of the property.

See also Properties of databases and ProgressIndicator.

Properties of databases

V12-DBE databases contain generic properties that provide for technical information on the current V12-DBE environment (such as the number of available indexes and the state of the active debugger) and allow for the control of the V12-DBE environment (such as custom string types and custom weekday names).

`mSetProperty` and `mGetProperty` are used to assign and read these generic database properties. Certain properties can only be read, not written (i.e. the number of available indexes) while others can be read and written (i.e. custom string types)

Certain properties are *persistent* (i.e. saved to the database and recovered when the database is reopened), others are not.

The syntax for `mGetProperty` is:

```
val := CallObject(gDB, "mGetProperty", PropertyName)
```

The syntax for `mSetProperty` is:

```
CallObject(gDB, "mSetProperty", PropertyName, Value)
```

`PropertyName` is a valid identifier (see Appendix 1: Capacities and Limits for the definition of a valid identifier).

`Value` is always a string, even if `PropertyName` refers to a number.

Note: `Value` is limited to 4096 characters.

`mSetProperty` can be used to define a new property or to change an existing one. Using `mSetProperty` with a value of EMPTY deletes that property. Properties pertaining to Strings (see The String property below) cannot be deleted.

Valid `PropertyName`s and `Value`s are listed below. Both parameters must be of type String. Both are case insensitive (hence "resources", "Resources" and "RESOURCES" are all equivalent).

Note: `mSetProperty` is a *very* powerful tool. If you are unsure about what you're doing, always work on a copy of your original database.

You can retrieve the list of all the properties of a database by calling `mGetPropertyNames`, as in

```
set props = mGetPropertyNames(gDB)
```

Note: V12-DBE properties can only be accessed by the `mSetProperty` and `mGetProperty` methods. They are totally unrelated to Win32 file properties.

Predefined properties

ProgressIndicator

ReadWrite, persistent. Valid values are "None", "With_Cancel", "Without_Cancel", "UserDefined". Default value is "None".

```
x := CallObject(gDB, "mGetProperty", "ProgressIndicator")
CallObject(gDB, "mSetProperty", "ProgressIndicator", "With_Cancel")
```

Enables V12-DBE to show a progress indicator while performing time-consuming tasks, or calls back Scripting handlers to enable custom progress indicator implementations. See Progress indicators.

ProgressIndicator.Message

ReadWrite, persistent.

```
msg := CallObject(gDB, "mGetProperty", "ProgressIndicator.Message")
CallObject(gDB, "mSetProperty", "ProgressIndicator.Message","Exporting
   records. Please be patient…")
```

This property sets the text that is displayed in the upper part of V12-DBE's progress window. If you set it to an empty string, V12-DBE displays a message that depends on the current operation. See Progress indicators.

VirtualCR

ReadWrite, persistent. Valid values: any ASCII character.

When importing or exporting data, convert Carriage Returns (ASCII #13) to this ASCII character. This is convenient to avoid the confusion of real Carriage Returns with Record Delimiters. This property can be overridden by a specific VirtualCR character passed as parameter to `mImport`.

Note: The VirtualCR property requires an ASCII *character* as a parameter, not an ASCII number.

To convert an ASCII number to a character, use the example provided here.

Example:

```
c := CallObject(gDB, "mGetProperty", "VirtualCR")
CallObject(gDB, "mSetProperty", "VirtualCR", Char(10)) -- define ASCII
   character #10 as virtual CR
```

See Step 2: Prepare the Data / Virtual carriage returns.

CharacterSet

ReadWrite, persistent. Valid values: "Windows-ANSI", "Mac-Standard", and "MS-DOS". Default: "Windows-ANSI" on the Windows version of V12-DBE and "Mac-Standard" on the Macintosh version of V12-DBE. This property affects all of V12-DBE's import and export functions. It can be overridden by a specific character set passed as a parameter to `mImport`.

Translates imported and exported files (whether Text or DBF) with the "Windows-ANSI", "Mac-Standard" or "MS-DOS" character set tables.

```
CallObject(gDB, "mSetProperty", "CharacterSet", "Mac-Standard")
```

See Step 2: Prepare the Data / Character sets.

Resources

ReadOnly, non-persistent.

```
CallObject(gDB, "mGetProperty", "resources")
```

Returns information on the number of available indexes and the index used by the last call to `mSelect`.

Example:

```
-- Number of indexes used: 6
-- Current index in table 'students': 'StudentIDNdx', using field
   'StudentID'
```

V12-DBE resources should not be confused with the MacOS resources (those normally edited with ResEdit) - they are completely unrelated.

CurrentDate

ReadOnly, non-persistent.

`mGetProperty` returns the current date in V12-DBE's raw format (YYYY/MM/DD) regardless of the Control Panel settings of the Mac or PC.

Example

```
aDate := CallObject(gDB, "mGetProperty", "CurrentDate")
-- "2002/05/09" is assigned to the variable aDate
```

Verbose

Warning: If you put calls to set the verbose property on in your script, you must remember to remove them before distributing your application.

ReadWrite, non-persistent. Valid values are "on" and "off".

When Verbose is set to "on", V12-DBE constantly displays a detailed feedback on the tasks it is performing in Authorware's Control Panel window,

Example:

```
CallObject(gDB, "mSetProperty", "verbose", "on")
CallObject(gDB, “mGetProperty“, "verbose")
```

This property is extremely useful for debugging database errors, during testing. Turn `verbose` “on” at the beginning of a script segment that includes several V12-

DBE method calls; then turn it "off" at the end of that section of code. Although verbose is convenient for authoring and testing purposes, you should not use it in place of `V12Status and V12Error` to trap potential runtime errors and to display feedback appropriate for your application's end-users. See Errors and defensive programming.

Months

ReadWrite, persistent. Valid values: any 12-word string.

The Month property contains the names of the months used by `mDataFormat` to format dates (the `MMMM` pattern in `mDataFormat`). The `Value` parameter is any 12-word string. Words must be separated by spaces. Names of months that contain spaces themselves must be enclosed between apostrophes.

Example:

```
CallObject(gDB, "mSetProperty ", "Months", "Gennaio Febbraio Marzo
   Aprile Maggio Giugno Luglio Agosto Settembre Ottobre Novembre
   Dicembre")
```

ShortMonths

ReadWrite, persistent. Valid values: any 12-word string.

The ShortMonth property contains the short names of the months used by `mDataFormat` to format dates (the `MMM` pattern in `mDataFormat`). The `Value` parameter is any 12-word string. Words must be separated by spaces. Short names of months that contain spaces themselves must be enclosed between apostrophes.

Example:

```
CallObject(gDB, "mSetProperty", "ShortMonths", "Jan Fév Mar Avr Mai
   Juin Juil Août Sep Oct Nov Déc")
```

Weekdays

ReadWrite, persistent. Valid values: any 12-word string.

The Weekdays property contains the names of the weekdays used by `mDataFormat` to format dates (the `DDDD` pattern in `mDataFormat`). The `Value` parameter is any 12-word string. Words must be separated by spaces. Names of weekdays that contain spaces themselves must be enclosed between apostrophes.

Example:

```
CallObject(gDB, "mSetProperty", "Weekdays", "Montag Dienstag Mittwoch
   Donnerstag Freitag Samstag Sonntag")
```

ShortWeekdays

ReadWrite, persistent. Valid values: any 12-word string.

The ShortWeekdays property contains the short names of the weekdays used by `mDataFormat` to format dates (the `DDD` pattern in `mDataFormat`). The `Value` parameter is any 12-word string. Words must be separated by spaces. Short names of weekdays that contain spaces themselves must be enclosed between apostrophes.

Example:

```
CallObject(gDB, "mSetProperty", "ShortWeekdays", "Lun Mar Mie Jue Vie
   Sab Dom")
```

SharedRWcount

ReadOnly, non-persistent. Returns the number of users currently using the database in Shared ReadWrite mode.

Example:

```
nbUsers := CallObject(gDB, "mGetProperty", "SharedRWcount")
```

DBversion

ReadOnly, non-persistent. Returns the version of the V12-DBE Xtra used to create the database.

Example:

```
dbversion := CallObject(gDB, "mGetProperty", "DBversion")
```

The above example puts "V12,3.3.0,Multi-User" in the variable `dbversion`.

FlushToDisk

If set to "true", this property will set the database to automatically flush the data to disk after every write operation.

V12 uses a disk cache system in order to speed up write operations, and normally flushes to disk on an irregular basis, as needed. This method is useful when you have an application running on a user's machine that may be prone to power failures, or when you want to be absolutely sure that you have stored a write operation to the hard drive.

When a database is opened in Shared ReadWrite mode, the FlushToDisk property is automatically set to "true" to ensure that modified records are immediately available to all users.

If you prefer to manually control when the database is flushed to disk, use the mFlushToDisk method.

Tip: A common use for the FlushToDisk property is in a kiosk situation where you store information into a database, and you want to increase the odds that a power-failure will not affect the database. Power failures may still cause problems with the operating system and/or database, but this command will ensure that the data has been written to the hard drive or storage device.

The String property

The String property is covered in a separate section because other sub-properties (`Delimiters`, `StopWords` and `MinWordLength`) depend on it. Properties below must be modified before fields of the corresponding string types are created in the database.

String.*Language*

ReadWrite, persistent. Valid values: any valid search/sort table (see Appendix 4: String and custom string types)

The String property defines or modifies custom string types (i.e. string fields that obey specific searching and sorting rules). To define a new string type, or modify an existing one, you append its name to "String.". The chosen name must be a valid identifier and cannot contain periods (".").

Example:

```
CallObject(gDB, "mSetProperty", "String.Klingon", CompTable)
```

In this example, the variable "CompTable" contains the search/sort descriptor for Klingon as defined in Appendix 4: String and custom string types. Once this property is defined, you can use the type "Klingon" to define new fields with `mCreateField` or `mReadDBstructure`. You also need to define this property first

before modifying other string properties such as `Delimiters`, `StopWords` and `MinWordLength`.

To modify the sort order of the default string, just omit the *Language* identifier:

```
CallObject(gDB, "mSetProperty", "String", CompTable)
```

String.*Language*.Delimiters

ReadWrite, persistent. Valid values: any valid delimiters descriptor.

Delimiters defines, for an existing string type, the list of characters that are acceptable as word delimiters for full-text indexing. By default, word delimiters for the predefined types are all non-alphanumeric characters (everything except 0-9, A-Z, a-z and accented characters).

Example:

```
CallObject(gDB, "mSetProperty", "String.Spanish.Delimiters",
   "!?@$%?&*()[]^®{}£¢§¨¶==º=+-,./\|")
```

In the above example, the punctuation characters indicated as the `Value` parameter are considered as delimiters.

If you need to specify the double-quote as part of the delimiters, place the delimiters in an Authorware variable and use that variable as a `Value` parameter as follows:

```
CallObject(gDB,"mSetProperty","String.Spanish.Delimiters",
   myDelimiters)
```

All non-printing characters such as TAB, Space, CTRL+J, etc. (i.e. characters lower than ASCII 32) are always considered as delimiters.

To modify the delimiters of the default string, just omit the *Language* identifier:

```
CallObject(gDB, "mSetProperty", "String.Delimiters", newDelimiters)
```

String.*Language*.MinWordLength

ReadWrite, persistent. Valid values: an integer in the range 2..8 passed as a String parameter.

MinWordLength determines the size of the shortest word that must be considered for full-indexing. All words shorter than MinWordLength are ignored and hence refused by the mSetCriteria method when used with the operator "WordEquals".

Example:

```
CallObject(gDB, "mSetProperty", "String.Spanish.MinWordLength", "3")
```

Note that the `Value` parameter is "3" (with quotation marks). This is because mSetProperty expects a `Value` parameter of type String only. The following is also a valid formulation:

```
CallObject(gDB, "mSetProperty", "String.Spanish.MinWordLength",
   String(3)))
```

To modify the `MinWordLength` of the default string, just omit the *Language* identifier:

```
CallObject(gDB, "mSetProperty", "String.MinWordLength", "2")
```

The default value for MinWordLength is 4.

String.*Language*.StopWords

ReadWrite, persistent. Valid values: a string no longer than 32K.

StopWords allows for the definition of a list of words that must be ignored in the full-indexing process. The `Value` parameter is a string containing the stop words in any order separated by spaces, TABs or Carriage Returns).

Example:

```
CallObject(gDB, "mSetProperty", "String.Spanish.StopWords", "el está en la
   de ")
```

To modify the stop words list of the default string, just omit the *Language* identifier:

```
CallObject(gDB, "mSetProperty", "String.StopWords","a the on for in by
   as")
```

By default, the StopWords property is empty. A typical list of stop words in English is:

```
"a by in the an for is this and from it to are had not was as have
of with at he on which be her or you but his that"
```

> **Note**: Remember that at most 4096 characters can be stored in individual properties.

Custom properties (advanced users)

Advanced users may want to define their own properties and make them persistent to a database. This is a convenient way to store preferences in your database and it eliminates the trouble of having to create a table that includes only one record.

For example, if you need to save the name of the user who last accessed your application:

```
CallObject(gDB, "mSetProperty", "LastUser", currentUser)
-- currentUser contains the name of the current user.
```

Next time the application is opened, you can check for the name of the last user and possibly take an appropriate action if it matches/doesn't match the current user.

```
LastUser := CallObject(gDB, "mGetProperty", "LastUser")
if (lastUser = currentUser) then
   -- take appropriate action
end if
```

Custom properties are always read-write and persistent.

Appendix 1: Capacities and Limits

Database

- The size of a V12-DBE database file is limited by disk space.
- The number of database instances and table instances is limited by RAM. Up to 128 instances of a single database can be opened by 128 different executables (Authorware pieces or packaged piece) in `Shared ReadWrite` mode. Each executable is entitled to only one database instance of a given database (although, you may create as many instances of *distinct* databases as you wish in a single executable). Multiple instances of a table can be created on a single computer. Note, however, that you may run into significant performance issues as the number of instances increases.
- A valid V12-DBE database contains at least one table, and each table requires at least one field and one index.
- For multi-user databases, the maximum number of ReadWrite users is 128. The Maximum number of ReadOnly users is unlimited.

Creation

- In order to create licensed V12 database structures at runtime, you must have a registered V12-DBE license file on your workstation. If you do not have a licensed V12-DBE on the workstation, your databases will be unlicensed and will display a splash screen each time they are opened. The best approach is often to clone new databases from existing databases.

Selection

- Up to 100 criteria can be chained with sequences of `mSetCriteria` separated by Boolean operators.

Importing

- DBF files of type DBase III, DBase IV, DBase V, FoxPro 2.0, FoxPro 2.5, FoxPro 2.6, FoxPro 3.0 and FoxPro 5.0 can be imported, exported and used as templates for the definition of V12 databases. Fields of type DateTime are not supported. The following DBF data types are ignored: General, Character-Binary, Memo-Binary.
- On Win32, MS Access databases, MS FoxPro files, MS Excel workbooks and MS SQL Server data sources can be used as templates for creating new V12 databases and as sources of data for importing records through ODBC drivers. The exact database translation/data importing rules varies among ODBC drivers and versions of ODBC drivers.

Table

- Because a maximum of 128 indexes is permitted, and since each table requires at least one index, the maximum number of tables in a V12-DBE database is 128.
- Identifiers (names of fields, tables and indexes) are limited to 32 characters. They must start with a low-ASCII alphabetic character (a..z, A..Z), which can

be followed by any alphanumeric character (0..9, a..z, A..Z, à, é, ö). Keywords such as `NOT`, `AND`, `OR`, `String`, `Integer`, `Float`, or `Date` are not suitable for use as identifiers.

Field

- No two fields or indexes can have the same name in the same table. However, two fields or two indexes might have the same name in different tables.
- All records are of variable length. Fields of type `string` are limited to 64K.
- The range of the type `Integer` is -2^{31} to $2^{31}-1$ (-2147483648 to 2147483647).
- The range of the type `Float` is ±1.79769313486232E+308 to ±2.22507385850720E-308.
- Any date later than January 1st, 1600 can be compared, retrieved and stored to fields of type `Date`. However, date formatting is limited to the range Jan 1st 1904 through Dec 31st 2037.

Index

- A maximum of 128 indexes can be defined on a V12-DBE database. Each index can operate on up to 12 fields.
- Up to 32 custom string types can be defined.
- When indexing fields of type String, up to the 251 first characters of each string are actually entered in the index. The remaining characters are ignored.
- Full-indexes are built with words not exceeding 31 characters. Words longer than 31 characters are truncated to 31 characters for the purpose of indexing (this does not affect the actual data).

Media

Fields of type media are not supported in V12-DBE for Authorware.

Appendix 2: Database Creation and Data Importing Rules

Following are examples of how to work with existing databases from a variety of vendors, in order to use them in V12-DBE.

As well, we have outlined standard Rules that can be referenced when you are trying to import the structures or the data from a specific database format.

The basic steps for converting a database from another database format into V12 are as follows:

1. Determine and/or Import the *structure* of the original database(s) into a readable format (using `mReadDBStructure`).
2. Create the V12 database and indexes, based on the structure of the original database(s).
3. Import the *data* from the original database(s) into the V12 databases you created in Step 2 (using mImport).

Below are examples of how to read the database structure and import files from the following text or database formats, as well as rules for each:

Text Files

mReadDBstructure from a Text File

To read a database descriptor into V12-DBE, use the following Script statement:

```
CallObject(database_instance, "mReadDBStructure", "TEXT",
   File_Pathname)
```

Assuming that the name of the database descriptor's filename is "Def.txt", the following Script creates a new V12-DBE database file named "Catalog.V12" and structures it as described in "Def.txt".

```
gDB := NewObject("V12dbe", "Evaluations.V12", "create", "top secret")
CallObject(gDB, "mReadDBStructure", "TEXT", FileLocation ^
   "Definition.txt")
CallObject(gDB, "mBuild")
```

```
DeleteObject(gDB)
gDB := 0
```

Importing a Text File

The imported text file must begin with a field descriptor line. A field descriptor is a sample record that contains the names of the fields in which subsequent rows of data must be formatted (see Field descriptors in Step 2: Prepare the Data). The field names used in the field descriptor line must match those supplied to the mReadStructure method. However, these fields can be listed in any order. Some of them can even be omitted.

Syntax:

```
CallObject(gTable, "mImport", "TEXT", FileName [, Options])
```

where `FileName` is the pathname of the text file to import, and `Options` is an optional Property list such as:

```
[ #FieldDelimiter:TAB, #RecordDelimiter:RETURN,
#CharacterSet:"Windows-ANSI", #VirtualCR: Char(11),
#TextQualifier:QUOTE ]
```

`Options` may contain some or all of the properties below, or can even be empty:

- `#FieldDelimiter` determines which character is used to delimit fields in the text file. The default character is TAB (ASCII #9).
- #RecordDelimiter determines which character is used to delimit records in the text file. The default character is RETURN (ASCII #13). If the Text file contains Carriage Returns (ASCII #13) followed by Line Feeds (ASCII #10) as records delimiters, Line Feeds are automatically ignored.
- `#CharacterSet` is one of "Mac-Standard", "Windows-ANSI" or "MS-DOS". It determines which character set the Text file is encoded in. Usually, Text files exported on MacOS are encoded in the Mac-Standard character set, and Text files exported on Windows are encoded in the Windows-ANSI character set. See Character sets in Step 2: Prepare the Data. The default character set is the one defined by the `CharacterSet` property (see CharacterSet in Properties of databases).
- `#VirtualCR` determines which character is used as a Virtual Carriage Return, and thus must be converted to ASCII #13 after importing (see Virtual carriage returns in Step 2: Prepare the Data). The default character is the one defined by the `VirtualCR` property, which is usually ASCII #11 (see VirtualCR in Properties of databases).
- `#TextQualifier` determines which character is used to begin and end each Text field. Those qualifiers delimit the field so to allow it to contain special characters, including those used as field and record delimiters. Text qualifiers are removed after importing the file. See Text qualifiers in Step 2: Prepare the Data. The default text qualifier is "|"

For example, this instruction imports the Text file "myTextData.txt" located in the same folder as the current movie into `gTable` with all the default options (field delimiter = TAB, records delimiter = RETURN, Character set = the current operating system's, virtual CR = ASCII #11, Text Qualifier = "|").

```
CallObject(gTable, "mImport", "TEXT", FileLocation ^ "myTextData.txt")
```

This example imports the Text file "myFile.txt" where "%" is used as a field delimiter and "\" as a record delimiter.

```
CallObject(gTable, “mImport”, "TEXT", the MoviePath & "myTextData.txt",
   [#FieldDelimiter: "%", #RecordDelimiter: "\"] )
```

Literals

mReadDBstructure from a Literal

A literal is a Script variable that actually contains the database descriptor (as opposed to containing the pathname of the descriptor Text file). Building a database from a literal description is very similar to the building it from a text file. The literal must contain the database descriptor as defined in Database descriptor. The Script to build the database is:

```
CallObject(gDB, "mReadDBStructure", "LITERAL", variable)
```

For example, assume that the Authorware variable named "Dbdescriptor" contains a database descriptor; the following example creates a V12-DBE database compliant to that description.

```
gDB := NewObject("V12dbe", "Evaluations.V12", "create", "top secret")
CallObject(gDB, "mReadDBStructure", "LITERAL", Dbdescriptor)
CallObject(gDB, "mBuild")
DeleteObject(gDB)
gDB := 0
```

If your tables, fields, index and full-indexes definitions are stored in four Authorware variables named "tbl", "flds", "idx"and "fidx", which don't start with the mandatory tags, the following example creates a valid database structure.

```
gDB := NewObject("V12dbe", "Evaluations.V12", "create", "top secret")
CallObject(gDB, "mReadDBStructure", "LITERAL", "[TABLE] " ^ tbl ^
   "[FIELDS] " ^ fld ^ "[INDEXED] " ^ idx ^ "[FULL-INDEXES] " ^ fidx)
CallObject(gDB, "mBuild")
DeleteObject(gDB)
gDB := 0
```

Import from a Literal

Sometimes, you need to process data with Scripting before importing it into a V12-DBE table. A convenient place to store such data is an Authorware variable.. `mImport` allows you to import the content of such a variable through this syntax:

```
CallObject(gTable, "mImport","LITERAL", variable [, Options])
where variable is an expression of type Text.
```

and `Options` is a property list identical to the one used for importing Text files (see Importing a Text File above).

Following is an example of an Authorware variable containing data to split into V12-DBE fields and records (assume the name of the variable is "Discounts"):

Level-1,Level-2,Level-3

12,14,16

45,58,72

33,56,68

224,301,451

This instruction imports the above Authorware variable (discounts) to `gTable`:

```
CallObject(gTable, "mImport", "LITERAL", discounts, ",", RETURN)
```

Lists or Property Lists

Importing from a List or Property List

Lists, or a Property Lists can easily be imported to V12 tables through `mImport`. This is very convenient for the conversion of projects that use Scripting lists to manage data and that have become difficult to debug and maintain.

It is also convenient to import XML documents into V12 tables.

Syntax:

```
CallObject(gTable, "mImport", "LIST", theList)
CallObject(gTable, "mImport", "PROPERTYLIST", thePropertyList)
```

where:

- `theList` is a list of lists. The first element is a list containing the names of the V12 fields to which subsequent items must be imported, in the right order. If the first item of the list contains field names that are not present in the current V12 table, the corresponding data is ignored.
- `thePropertyList` is a list of property lists, where properties have the same names as the V12 fields into which the corresponding data must be imported.

Examples of valid lists:

```
[ ["LastName", "FirstName", "Age"],  ["Cartman", "Eric", 8],
  ["Testaburger", "Wendy", 9], ["Einstein", "Albert", 75]  ]

[ ["CatalogNumber"],  [8724], [9825], [1745] ]
```

Examples of valid Property lists:

```
[ [#LastName:"Cartman", #FirstName:"Eric", #Age:8 ],
  [#FirstName:"Wendy", #LastName:"Testaburger", #Age:9 ]
  [#LastName:"Einstein", #FirstName:"Albert"] ]

 [ [#CatalogNumber:8724], [#CatalogNumber:9825],
  [#CatalogNumber:1745] ]
```

V12 DBE files

mReadDBstructure from V12-DBE

Any V12-DBE database can be used as a template for the creation of a new V12-DBE database, provided you know the password to unlock it. The syntax is as follows:

```
CallObject (database_instance, mReadDBStructure, "V12", FileName,
   Password)
```

This example uses the database "Catalog.V12" as a template for a new database named "Specials.V12".

```
gDB := NewObject("V12dbe", FileLocation ^ "Specials.V12",
  "create", MyNewPassword")
CallObject (gDB, "mReadDBStructure", "V12", FileLocation ^
  "Catalog.V12", "top secret")
CallObject(gDB, "mBuild")
DeleteObject(gDB)
gDB := 0
```

`mReadDBStructure` reads the *structure* of a V12-DBE file, not its content. To import the content of a V12-DBE file, see Importing from another V12-DBE and Add records to a database.

Importing from another V12-DBE file

Data can be imported from one V12 table into another. The name of the source table need not necessarily match the name of the destination table. However, field names must match. Non-matching field names are ignored. If the source and destination tables have different indexes, the destination table's indexes are used.

Syntax:

```
CallObject(gTable, "mImport", "V12", FileName, Password, TableName)
```

where `FileName` is the pathname of the V12 database to import from, `password` is the password to unlock it and `TableName` is the name of the table to import.

Example:

```
CallObject (gTable, "mImport", "V12", FileLocation ^ "Catalog.V12",
   "top secret", "articles")
```

If two fields have the same name but are of different types when importing data from a V12-DBE database, `mImport` tries to typecast the data fields.

DBF (Database Format)

mReadDBstructure from a DBF File

A DBF file alone represents a flat file, thus a single V12-DBE table. A DBF file can be used as a template for a V12-DBE table in much the same way as a text file or literal can. The name of the created V12-DBE table is identical to the DBF filename without the ".DBF" extension. The syntax is:

```
CallObject (gDB, "mReadDBStructure", "DBF", File_Pathname)
```

Tip: V12-DBE does not check the validity of the content of the index file; therefore you can fool it into creating an index for a field named "MyField" by creating an empty file named "MyField.IDX" in the same folder as your DBF file.

For a DBF file to be used as a complete and valid V12-DBE table descriptor, at least one index must be defined. If that index is defined by an IDX or NDX file located in the same folder as the DBF file, `mReadDBstructure` detects its presence and automatically defines an index for that field in the current table.

This example uses the file VIDEO.DBF as a template to build a table named "video" in the V12-DBE database named "VideoStore.V12". The structure of the file VIDEO.DBF is as follows:

Field	Type	Width
TITLE	Character	30
DESCRIPT	Memo	10
RATING	Character	4
TYPE	Character	10

DATE_ARRIV	Date	8
AVAILABLE	Logical	1
TIMES_RENT	Numeric	5
NUM_SOLD	Numeric	5

Two index files named TITLE.IDX and TYPE.IDX are available in the same folder as VIDEO.DBF.

The Script is as follows:

```
gDB := NewObject("V12dbe", FileLocation ^ "VideoStore.V12",
  "create", "")
CallObject (gDB, "mReadDBStructure", "DBF", FileLocation ^
  "Video.DBF")
CallObject (gDB, "mBuild")
DeleteObject(gDB)
gDB := 0
```

The resulting V12-DBE database can be verified immediately with `mDumpStructure` (see View the structure of a database). The following is a sample output from `mDumpStructure`:

```
[TABLE]
Video
[FIELDS]
TITLE String 30
DESCRIPT String 30000
RATING String 4
TYPE String 10
DATE_ARRIV Date
AVAILABLE Integer
TIMES_RENT Integer
NUM_SOLD Integer
[INDEXES]
TitleNdx duplicate TITLE ascending (* Default Index *)
TypeNdx duplicate TYPE ascending
[END]
```

Importing from a DBF File

Note: DBF is an antiquated file format. It is always assumed to be encoded in the MS-DOS character set. When importing DBF files, make sure to assign the right Character Set. See CharacterSet in Properties of databases.

Importing a DBF file is similar to importing text files, except that you cannot specify a subset of fields to import: all the fields in the DBF file must be imported. The field names of the DBF file must match those in the destination V12-DBE table. Non-matching field names are ignored during the importing process and a warning is reported by `V12Error` (see Errors and defensive programming).

Syntax:

```
CallObject(gTable, "mImport", "DBF", FileName [, Options])
```

where `FileName` is the pathname of the DBF file to import, and `Options` is an optional Property list containing the `#CharacterSet` property:

`#CharacterSet` is one of "Mac-Standard", "Windows-ANSI" or "MS-DOS". It determines which character set the DBF file is encoded in. Most systems automatically encode DBF file in the MS-DOS character set. See Character sets in Step 2: Prepare the Data. The default character set the one defined by the `CharacterSet` property (see CharacterSet in Properties of databases). It is normally "Windows-ANSI" on the Windows version of V12-DBE and "Mac-Standard" on the Macintosh version of V12-DBE.

Example:

```
CallObject (gTable, "mImport", "DBF", FileLocation ^ "Pier1-
   Import.DBF", [#CharacterSet: "MS-DOS"])
```

If a field in the destination table has the same name as a field in the source DBF file, but is of a different type, `mImport` tries to typecast the data to match the destination field type. When importing data from a DBF file that contains Memo fields, the corresponding DBT files are automatically processed and imported by V12-DBE. See *Dealing with dates* and *DBF (Database Format) Rules* below for more details on DBF files and data importing rules.

DBF (Database Format) Rules

The following rules apply to the translation of DBF file structures:

DBF field type	Translated to V12 field type	Notes
Character	String	Buffer size = size of field in DBF file
Integer	Integer	
Numeric with no digit after fixed point	Integer	
Numeric with one or more digits after fixed point	Float	
Float	Float	
Double	Float	
Currency	Float	On Windows 3.1 and Mac68K, acceptable values are in the range -2^k to 2^k-1, where k = 31 minus the number of decimal places.
Date	Date	
DateTime	Date	Data cannot be converted from fields of type DateTime. Only the default date (1900/01/01) is imported.
Logical	Integer	FALSE values are translated to *0*s, TRUE values to *1*s and undefined values (represented by "?" in the DBF file) to *-1*s
Media	String	Buffer size = 32K
General	Ignored	
Character-Binary	Ignored	
Memo-Binary	Ignored	

Memo fields are those typically used to store text longer than 255 characters. Memo fields can also store binary data of arbitrary formats: Binary formatted memo fields *cannot* be imported directly into V12-DBE databases. When importing standard ASCII data from a DBF file that contains Memo fields, the corresponding DBT files are automatically processed by V12-DBE.

Microsoft FoxPro

mReadDBstructure from FoxPro (Win-32 Only)

A FoxPro database is a directory containing a collection of DBF files along with their index files. A directory containing one or more MS FoxPro files can be collectively used as a template for a V12 database. The FoxPro ODBC driver is required to perform this operation. The names of your FoxPro files and their field names must be valid V2-DBE identifiers (see Defining identifiers in Step 1: Decide on a Data Model).

Syntax:

```
CallObject(gDB, "mReadDBStructure", "FoxPro", DirectoryPath)
```

where `DirectoryPath` is the path to a directory — not a file. Thus, it must necessarily end with a "\".

Example:

```
gDB := NewObject("V12dbe", FileLocation^"myDB.V12", "create", "secret")
CallObject(gDB, "mReadDBStructure","FoxPro", FileLocation ^ "FoxDB\")
CalloBject(gDB, "mBuild")
DeleteObject(gDB)
gDB := 0
```

Once the corresponding V12 database structure is created, with mReadDBStructure, the data from your FoxPro tables can be imported.

Importing from Microsoft FoxPro (Win-32 only)

Fox Pro (*.DBF) files can be imported to V12 tables provided a MS FoxPro ODBC driver is present on your PC. No DSN (Data Source Name) is required.

Syntax:

```
CallObject(gTable, "mImport", "FoxPro", FileName)
```

where `FileName` is the path to the source *.DBF file. Always make sure to set V12-DBE's CharacterSet property to the encoding that matches your DBF file's (see CharacterSet in Properties of databases).

Example:

```
CallObject(gTable, "mImport", "FoxPro", FileLocation ^ "Results.XLS",
   TableName)
```

Converting a FoxPro database into a V12 database is a two-step process: First, create the V12 database, and then import data to each of its tables with `mImport`, as explained above.

FoxPro (Microsoft Fox Professional Format) Rules

The following rules apply to the translation of FoxPro databases to V12 databases:

FoxPro field type	Translated to V12 field type	Notes
Character	String	Buffer size is the size of the field in the DBF file
Integer	Float	
Numeric	Float	
Float	Float	
Double	Float	
Currency	Float	
Date	Date	
DateTime	Date	Data cannot be converted from fields of type DateTime. Only the default date (1900/01/01) is imported.
Logical	Integer	
Memo	String	Buffer size = 32K
General	String	Buffer size is the size of the field in the DBF file
Character-Binary	String	Buffer size is 32K
Memo-Binary	String	Buffer size is 32K

Microsoft Access

mReadDBstructure from MS Access (Windows only)

MS Access databases can be used as templates to V12 databases. Like V12-DBE, MS Access can store multiple tables per database. `mReadDBstructure` imports the structure of all such tables to V12-DBE. The MS Access ODBC driver is required to perform this operation.

The names of the tables and fields of your MS Access file must be valid V12-DBE identifiers (see Defining identifiers in Step 1: Decide on a Data Model).

Syntax:

```
CallObject(gDB, "mReadDBStructure", "Access", FileName, Username,
   Password)
```

where:

- `FileName` is the path to the *.MDB file,
- `Username` is a valid user name to access the MDB file, or EMPTY if the MDB file is not protected,
- `Password` is Username's matching password, or EMPTY if the MDB file is not protected.

Once the corresponding V12 database structure is created, the data from your MS Access tables can be imported.

Import from Microsoft Access (Windows only)

MS Access (*.MDB) files can be imported to V12 databases, one table at a time. A MS Access ODBC driver must be present but no DSN (Data Source Name) is required.

Syntax:

```
CallObject(gTable, "mImport", "Access", FileName, UserName, Password,
   TableName)
```

where:

- `FileName` is the path to the source *.MDB file,
- `Username` is a valid user name to access the MDB file, or EMPTY if the MDB file is not protected,
- `Password` is Username's matching password, or EMPTY if the MDB file is not protected.
- `TableName` is the name of the table to import.

Converting an MS Access database into a V12 database is a two-step process: First, create the V12 database (see mReadDBstructure from MS Access (Windows only)). Then, import data to each of its tables with `mImport`, as explained above.

Generally, MS Access databases are encoded in the Windows ANSI character set. Thus, you must make sure that the `CharacterSet` Property is properly set to "Windows-ANSI" before importing the data. ("Windows-ANSI" is the default setting for the `CharacterSet` property. See CharacterSet in Properties of databases).

Microsoft Access Rules

The following rules apply to the translation of MS Access file structures to V12 databases:

MS Access field type	Translated to V12 field type	Notes
Text	String	Buffer size is same as Access field size
Number (byte)	Integer	
Number (integer)	Integer	
Number (long integer)	Integer	
Number (single)	Float	
Number (double)	Float	
Number (replication ID)	Ignored	
Currency	Integer	
Date / Time	Ignored	
Autonumber	Integer	
Yes/No	Integer	
OLE Object	Ignored	
HyperLink	String	URL imported as text
Memo	String	Buffer size is 32K

MS Access unique and duplicate indexes are properly converted to unique and duplicate V12-DBE indexes with ascending field values.

Microsoft Excel

mReadDBstructure from MS Excel (Windows only)

MS Excel workbooks can be used as templates to V12 databases. MS Excel workbooks can contain one or more worksheets, with each worksheet corresponding to a V12 table and each column to a V12 field. The resulting V12 database contains as many tables as there are worksheets in the Excel file. The MS Excel ODBC driver is required to perform this operation.

The names of the worksheets and columns of your MS Excel file must be valid V2-DBE identifiers (see Defining identifiers in Step 1: Decide on a Data Model).

The types of the field defined in the new V12 database depend on the format of the corresponding MS Excel columns. To change the format of a entire column in MS Excel, select it by clicking in its heading, choose Format > Cells… and select the Number tab. It may be necessary to Save As… your workbook with a new name to

force MS Excel to commit to the new column's format (depends on version of Excel).

Syntax:

```
CallObject(gDB, "mReadDBStructure", "Excel", FileName)
```

where `FileName` is the path to the *.XLS file.

Importing from Microsoft Excel (Windows only)

MS Excel workbooks (*.XLS) can be imported to V12 databases, one table at a time, through a PC's ODBC driver. No DSN (Data Source Name) is required.

Note: It is important that the first row contains the field names. This way, V12-DBE can associate an Excel column to a V12-DBE field.

Syntax:

```
CallObject(gTable, "mImport", "Excel", FileName, TableName)
```

where:

- `FileName` is the path to the source *.XLS file. It is assumed to be encoded in the Windows ANSI character set (default encoding on Windows).
- `TableName` is the name of the table to import.

Example:

```
CallObject(gTable, "mImport", "Excel", FileLocation ^ "Results.XLS")
```

Protected MS Excel workbooks cannot be imported

Converting a MS Excel workbook into a V12 database is a two-step process: First, create the V12 database (see Importing from Microsoft Excel (Windows only)). Then, import data to each of its tables with `mImport`, as explained above.

Microsoft Excel Rules

The following rules apply to the translation of MS Excel file structures to V12 databases:

MS Excel field type	Translated to V12 field type	Notes
General	Float	
Number	Float	
Currency	Integer	
Accounting	Integer	
Date	Ignored	Convert to text first if importing to V12-DBE is needed
Time	Ignored	Convert to text first if importing to V12-DBE is needed
Percentage	Float	
Fraction	Float	
Scientific	Float	
Text	String	Buffer size = 255 bytes
Special	Float	
Custom	Float	

MS Excel cannot define indexes on its fields, when reading an Excel workbook; V12-DBE automatically indexes the leftmost field of each worksheet.

Microsoft SQL Server

mReadDBstructure from MS SQL Server (Windows only)

A MS SQL Server version 6 or 7 data source can be used as a template to a V12 database. In contrast to MS Access, MS FoxPro and MS Excel files, `mReadDBstructure` requires a DSN (Data Source Name) to be supplied instead of a pathname. The MS SQL Server ODBC driver is required to perform this operation.

Syntax:

```
CallObject(gDB, "mReadDBStructure", "SQLserver", DSN, Username,
   Password)
```

where

- `DSN` is the name of a valid User DSN, System DSN or File DSN (see Window's Control Panel),
- `Username` is a valid user name to access the DSN,

- `Password` is Username's matching password.

Importing from MS SQL (Windows only)

MS SQL Server data sources can be imported to V12 databases, one table at a time, through a PC's ODBC driver and a valid DSN (Data Source Name). Data sources can be created through Window's ODBC Data Sources Control Panel which is accessible from Start > Settings > Control Panel menu.

Syntax:

```
CallObject(gTable, "mImport","SQLserver", DSN, Username, Password,
   TableName)
```

where:

- `DSN` is a valid Data Source Name.
- `Username` is a valid user name to access the SQL Server.
- `Password` is Username's matching password.
- `TableName` is the name of the table to import.

Example:

```
CallObject(gTable, "mImport", "SQLserver", "InventoryDSN", "Admin",
   "XBF48", "Products")
```

Converting an MS SQL Server data source into a V12 database is a two-step process: First, create the V12 database (see mReadDBstructure from MS SQL Server (Windows only)). Then, import data to each of its tables with `mImport`, as explained above.

Microsoft SQL format Rules

The following rules apply to the translation of MS SQL Server data sources to V12 databases:

MS SQL Server field type	Translated to V12 field type	Notes
Binary	Ignored	
Bit	Integer	
Char	String	Buffer size is same as MS SQL Server field size
DateTime	Ignored	
Decimal	Float	
Float	Float	
Image	String	Buffer size = 32K. Data cannot be imported from Image fields.
Int	Integer	
Money	Float	
Numeric	Integer	
Real	Float	
SmallDateSize	Ignored	
SmallInt	Integer	
SmallMoney	Float	
SysName	String	Buffer size is same as MS SQL Server field size
Text	String	Buffer size = 32K
TimeStamp	Ignored	
TinyInt	Integer	
VarBinary	String	Buffer size is same as MS SQL Server field size
VarChar	String	Buffer size is same as MS SQL Server field size

Note that V12 does not support unicode SQL data type such as nchar, nvarchar and ntext.

Appendix 3: mGetSelection examples

The examples below show various ways of using `mGetSelection`. All examples assume that the table `gTable` contains 3 fields ("name", "price" and "number", declared in that order when creating the table), and that the selection contains 6 records.

Read an entire selection

This example retrieves the entire content of each record of the selection with `TAB`s as field delimiters and `CARRIAGE_RETURN`s (`CR`s) as record delimiters. Fields are sorted in their order of creation. The records' sort order is the one defined by the selection.

```
x := CallObject(gTable, "mGetSelection")
```

sets the variable `x` to the following string:

Batteries	*TAB*	9.20	*TAB*	6780	*CR*
Floppies	*TAB*	1.89	*TAB*	9401	*CR*
Labels	*TAB*	1.19	*TAB*	1743	*CR*
Pencils	*TAB*	5.55	*TAB*	6251	*CR*
Ruler	*TAB*	1.99	*TAB*	1431	*CR*
Tags	*TAB*	6.19	*TAB*	7519	*CR*

Read a range of records in a string variable

This example retrieves the content of 3 successive records in the selection, starting with record #2, with `TAB`s as field delimiters and `CARRIAGE_RETURN`s (`CR`s) as record delimiters.

```
x := CallObject(gTable, "mGetSelection", "LITERAL", 2, 3)
```

sets the variable `x` to the following string:

Floppies	TAB	1.89	TAB	9401	CR
Labels	*TAB*	1.19	*TAB*	1743	*CR*
Pencils	*TAB*	5.55	*TAB*	6251	*CR*

Read a range of records in a list

This is identical to the previous example, except that the result is returned in an Authorware Script list:

```
x := CallObject(gTable, "mGetSelection", "LIST", 2, 3)
```

sets the variable `x` to the following list:

```
[ ["Floppies", 1.89, 9401], ["Labels", 1.19, 1743], ["Pencils",  5.55,
   6251] ]
```

Read a range of records in a property list

Same as the two previous examples, except that the result is returned in an Authorware Script property list:

```
x := CallObject(gTable, "mGetSelection", "PROPERTYLIST", 2, 3)
```

sets the variable `x` to the following list:

```
[ [#name:"Floppies", #price:1.89, #number:9401], [#name:"Labels",
   #price:1.19, #number:1743], [#name:"Pencils",  #price:5.55,
   #number:6251] ]
```

Read the entire contents of the current record

This example retrieves the entire contents of the current record in a single call to V12-DBE.

```
x := CallObject(gTable, "mGetSelection", "LITERAL", CallObject(gTable,
   "mGetPosition"), 1)
```

sets the variable x to the following string:

Batteries *TAB* 9.20 *TAB* 6780 *CR*

The "List" and "ProperyList" would respectively return:

```
[ ["Batteries", 9.20, 6780] ]
```

and

```
[ [#name:"Batteries", #price:9.20, #number:6780] ]
```

Read a record without making it the current record

This example retrieves the content of record #4 without making it the current record.

```
x := CallObject(gTable, "mGetSelection", "LITERAL", 4, 1)
```

sets the variable x to the following string:

Pencils *TAB* 5.55 *TAB* 6251 *CR*

The "List" and "ProperyList" would respectively return:

```
[ ["Pencils", 5.55, 6251] ]
```

and

```
[ [#name:"Pencils",  #price:5.55, #number:6251] ]
```

Read the entire selection with special delimiters

This example retrieves the entire content of each record of the selection with commas (",") as field delimiters and slashes ("/") as record delimiters.

```
x := CallObject(gTable, "mGetSelection", "LITERAL", 1,
   CallObject(gTable, "mSelectCount"), ",", "/")
```

sets the variable x to the following string:

Batteries , 9.20 , 6780 / Floppies , 1.89 , 9401 / Labels , 1.19 , 1743 / Pencils , 5.55 , 6251 / Ruler , 1.99 , 1431 / Tags , 6.19 , 7519 /

Read selected fields in a selection

This example retrieves the content of a single field ("name") for all the records of the selection. Note that the TAB parameter is unused in the result, but it should nonetheless be present.

```
x := CallObject(gTable, "mGetSelection", "LITERAL", 1,
   CallObject(gTable, "mSelectCount"), TAB, RETURN, "name")
```

sets the variable x to the following string:

Batteries *CR*

Floppies *CR*

Labels *CR*

Pencils *CR*

Ruler *CR*

Tags *CR*

The syntax for the Script List result would be:

```
x := CallObject(gTable, "mGetSelection", "LIST", 1, CallObject(gTable,
   "mSelectCount"), "name")
```

and the result would be

```
[["Batteries"],["Floppies"],["Labels"],["Pencils"],["Ruler"], ["Tags"]]
```

(Note: This is a list where each element is itself a single element list).

The syntax for the Property List result would be:

```
x := CallObject(gTable, "mGetSelection", "PROPERTYLIST", 1,
   CallObject(gTable, "mSelectCount"), "name")
```

and the result would be

```
[[#name:"Batteries"],[#name:"Floppies"],[#name:"Labels"],
   [#name:"Pencils"],[#name:"Ruler"],[#name:"Tags"]]
```

Read records with a determined order of fields

This example retrieves the content of all the records of the selection with TABs as field delimiters and CARRIAGE_RETURNs (CRs) as record delimiters, with fields ordered in the sequence "number", "name", and "price".

```
x := CallObject(gTable, "mGetSelection", "LITERAL", 1, CallObject
   (gTable, "mSelectCount"), TAB, RETURN, "number", "name", "price")
```

sets the variable x to the following string:

6780	*TAB*	Batteries	*TAB*	9.20	*CR*
9401	*TAB*	Floppies	*TAB*	1.89	*CR*
1743	*TAB*	Labels	*TAB*	1.19	*CR*
6251	*TAB*	Pencils	*TAB*	5.55	*CR*
1431	*TAB*	Ruler	*TAB*	1.99	*CR*
7519	*TAB*	Tags	*TAB*	6.19	*CR*

The syntax for the Script List result would be:

```
x := CallObject(gTable, "mGetSelection", "LIST", 1, CallObject(gTable,
   "mSelectCount"), "number", "name", "price")
```

and the result would be

```
[ [6780, "Batteries", 9.20], [9401, "Floppies", 1.89], [1743, "Labels",
   1.19], [6251, "Pencils", 5.55], [1431, "Ruler", 1.99], [7519,
   "Tags", 6.19] ]
```

The syntax for the Property List result would be:

```
x := CallObject(gTable, "mGetSelection", "PROPERTYLIST", 1,
   CallObject(gTable, "mSelectCount"), "number", "name", "price")
```

and the result would be

```
[ [#number:6780, #name:"Batteries", #price:9.20], [#number:9401,
   #name:"Floppies", #price:1.89], [#number:1743, #name:"Labels",
   #price:1.19], [#number:6251, #name:"Pencils", #price:5.55],
   [#number:1431, #name:"Ruler", #price:1.99], [#number:7519,
   #name:"Tags", #price:6.19] ]
```

Although, this latter request would not be of much interest because property lists are parsed by property names, not item positions.

Appendix 4: String and custom string types

V12-DBE enables you to develop applications containing different types of strings such as English, German, Swedish and Spanish. Basically, each V12-DBE table can contain any combination of those string types.

String comparisons depend on how special characters are defined in their corresponding languages. For example, the letters **a** and **ä** may be considered identical in some languages but different in others. This behavior is determined by the *sorting and searching rules* attached to each type of string.

V12-DBE's default and custom String types' sorting and searching rules are defined by the following tables where equivalent characters are listed on the same line separated by one or more spaces and strict precedence is indicated by characters on successive lines. For example:

```
j J
k K
l L
```

means that:

- `K` sorts after `J` and before `L`,
- `j` and `J` are equivalent (likewise, `k` and `K` are equivalent, and `l` and `L` are equivalent too)

Characters omitted from a sorting and searching rules table are considered to sort *after* those listed in the table, except for Control characters (such as Carriage Return, Horizontal Tab, Vertical Tab, etc.), which are considered to sort *before* those listed in the table.

The default string

The default `string` type has predefined rules that accommodate a large number of languages (English, French, German, Italian, Dutch, Portuguese, Norwegian, etc.).

(If the tables below are not properly formatted in the HTML version of this manual, please refer to the PDF version)

Sorting and searching rules table for the default string type:

1.	' ‘ ’	39.	1
2.	" « » “ ”	40.	2
3.	! ¡	41.	3
4.	? ¿	42.	4
5.	.	43.	5
6.	,	44.	6
7.	:	45.	7
8.	;	46.	8
9.	…	47.	9
10.	#	48.	a à á â ã ä A À Á Â Ã Ä
11.	$	49.	b B
12.	¢	50.	c ç C Ç
13.	£	51.	d D
14.	¥	52.	e è é ê ë E È É Ê Ë
15.	% ‰	53.	f F
16.	°	54.	g G
17.	|	55.	h H
18.	† ‡	56.	i ì í î ï I Ì Í Î Ï
19.	[ ]	57.	j J
20.	{ }	58.	k K
21.	( )	59.	l L
22.	< >	60.	m M
23.	*	61.	n ñ N Ñ
24.	+	62.	o ò ó ô õ ö œ O Ò Ó Ô Õ Ö Œ
25.	-	63.	p P
26.	/	64.	q Q
27.	\	65.	r R
28.	=	66.	s ß S
29.	~	67.	t T
30.	¬ - – —	68.	u ù ú û ü U Ù Ú Û Ü
31.	§	69.	v V
32.	µ	70.	w W
33.	&	71.	x X
34.	@	72.	y ÿ Y Ÿ
35.	©	73.	z Z
36.	ƒ	74.	æ Æ
37.	®	75.	ø Ø
38.	0	76.	å Å

Predefined custom string types

Along with the standard `string` type, V12-DBE contains a number of predefined custom string types. They include `Swedish, Spanish` and `Hebrew`.

Searching and sorting rules for strings of type *Swedish*

(If the tables below are not properly formatted in this version of this manual, please refer to the PDF version)

1.	' ‘ ’
2.	" « » “ ”
3.	! ¡
4.	? ¿
5.	.
6.	,
7.	:
8.	;
9.	…
10.	#
11.	$
12.	¢
13.	£
14.	¥
15.	% ‰
16.	°
17.	|
18.	† ‡
19.	[ ]
20.	{ }
21.	( )
22.	< >
23.	*
24.	+
25.	-
26.	/
27.	\
28.	=
29.	~
30.	¬ - – —
31.	§
32.	µ
33.	&
34.	@
35.	©
36.	*f*
37.	®
38.	0
39.	1
40.	2
41.	3
42.	4
43.	5
44.	6
45.	7
46.	8
47.	9
48.	a à á â ã A À Á Â Ã
49.	b B
50.	c ç C Ç
51.	d D
52.	e è é ê ë E È É Ê Ë
53.	f F
54.	g G
55.	h H
56.	i ì í î ï I Ì Í Î Ï
57.	j J
58.	k K
59.	l L
60.	m M
61.	n ñ N Ñ
62.	o ò ó ô õ œ O Ò Ó Ô Õ Œ
63.	p P
64.	q Q
65.	r R
66.	s ß S
67.	t T
68.	u ù ú û ü U Ù Ú Û Ü
69.	v V
70.	w W
71.	x X
72.	y ÿ Y Ÿ
73.	z Z
74.	æ Æ
75.	ø Ø
76.	å Å
77.	ä Ä
78.	ö Ö

Searching and sorting rules for strings of type *Spanish*

(If the tables below are not properly formatted in this version of this manual, please refer to the PDF version)

1. ' ‘ ’
2. " « » “ ”
3. ! ¡
4. ? ¿
5. .
6. ,
7. :
8. ;
9. …
10. #
11. $
12. ¢
13. £
14. ¥
15. % ‰
16. °
17. |
18. † ‡
19. [ ]
20. { }
21. ( )
22. < >
23. *
24. +
25. -
26. /
27. \
28. =
29. ~
30. ¬ - – —
31. §
32. µ
33. &
34. @
35. ©
36. *f*
37. ®
38. 0
39. 1
40. 2
41. 3
42. 4
43. 5
44. 6
45. 7
46. 8
47. 9
48. a à á â ã ä A À Á Â Ã Ä
49. b B
50. c ç C Ç
51. d D
52. e è é ê ë E È É Ê Ë
53. f F
54. g G
55. h H
56. i ì í î ï I Ì Í Î Ï
57. j J
58. k K
59. l L
60. m M
61. n N
62. ñ Ñ
63. o ò ó ô õ ö œ O Ò Ó Ô Õ Ö Œ
64. p P
65. q Q
66. r R
67. s ß S
68. t T
69. u ù ú û ü U Ù Ú Û Ü
70. v V
71. w W
72. x X
73. y ÿ Y Ÿ
74. z Z
75. æ Æ
76. ø Ø
77. å Å

Searching and sorting rules for strings of type *Hebrew*

(requires a Hebrew font such as "Web Hebrew")

1.	' ' ‘		50.	í î	(mem)
2.	" « » ” “		51.	ï ð	(nun)
3.	! ¡		52.	ñ	(samech)
4.	? ¿		53.	ò	(ain)
5.	.		54.	ó ô	(phe)
6.	,		55.	õ ö	(sadi)
7.	:		56.	÷	(koph)
8.	;		57.	ø	(resch)
9.	…		58.	ù	(sin)
10.	#		59.	ú	(tau)
11.	$		60.	0	
12.	¢		61.	1	
13.	£		62.	2	
14.	¥		63.	3	
15.	% ‰		64.	4	
16.	°		65.	5	
17.	|		66.	6	
18.	† ‡		67.	7	
19.	[ ]		68.	8	
20.	{ }		69.	9	
21.	( )		70.	a A À Á Â Ã Ä	
22.	< >		71.	b B	
23.	*		72.	c C Ç	
24.	+		73.	d D	
25.	-		74.	e E È É Ê Ë	
26.	/		75.	f F	
27.	\		76.	g G	
28.	=		77.	h H	
29.	~		78.	i ı Ì Í Î Ï	
30.	- – ¬		79.	j J	
31.	§		80.	k K	
32.	µ		81.	l L	
33.	&		82.	m M	
34.	@		83.	n N Ñ	
35.	©		84.	o O Ò Ó Ô Õ Ö	
36.	ƒ		85.	p P	
37.	®		86.	q Q	
38.	à	(aleph)	87.	r R	
39.	á	(beth)	88.	s ß S	
40.	â	(ghimel)	89.	t T	
41.	ã	(daleth)	90.	u û ü U Ù Ú Û Ü	
42.	ä	(he)	91.	v V	
43.	å	(vau)	92.	w W	
44.	æ	(zain)	93.	x X	
45.	ç	(heth)	94.	y ÿ Y Ÿ	
46.	è	(teth)	95.	z Z	
47.	é	(iod)	96.	Æ	
48.	ê ë	(caph)	97.	Ø	
49.	ì	(lamed)	98.	Å	

Appendix 5: Character sets

Windows-ANSI character set

32		70	F	108	l	146	’	184	¸	222	Þ
33	!	71	G	109	m	147	“	185	¹	223	ß
34	"	72	H	110	n	148	”	186	º	224	à
35	#	73	I	111	o	149	•	187	»	225	á
36	$	74	J	112	p	150	–	188	¼	226	â
37	%	75	K	113	q	151	—	189	½	227	ã
38	&	76	L	114	r	152	˜	190	¾	228	ä
39	'	77	M	115	s	153	™	191	¿	229	å
40	(	78	N	116	t	154	š	192	À	230	æ
41	)	79	O	117	u	155	›	193	Á	231	ç
42	*	80	P	118	v	156	œ	194	Â	232	è
43	+	81	Q	119	w	157		195	Ã	233	é
44	,	82	R	120	x	158		196	Ä	234	ê
45	-	83	S	121	y	159	Ÿ	197	Å	235	ë
46	.	84	T	122	z	160		198	Æ	236	ì
47	/	85	U	123	{	161	¡	199	Ç	237	í
48	0	86	V	124	|	162	¢	200	È	238	î
49	1	87	W	125	}	163	£	201	É	239	ï
50	2	88	X	126	~	164	¤	202	Ê	240	ð
51	3	89	Y	127		165	¥	203	Ë	241	ñ
52	4	90	Z	128		166	¦	204	Ì	242	ò
53	5	91	[	129		167	§	205	Í	243	ó
54	6	92	\	130	‚	168	¨	206	Î	244	ô
55	7	93	]	131	ƒ	169	©	207	Ï	245	õ
56	8	94	^	132	„	170	ª	208	Ð	246	ö
57	9	95	_	133	…	171	«	209	Ñ	247	÷
58	:	96	`	134	†	172	¬	210	Ò	248	ø
59	;	97	a	135	‡	173	-	211	Ó	249	ù
60	<	98	b	136	ˆ	174	®	212	Ô	250	ú
61	=	99	c	137	‰	175	¯	213	Õ	251	û
62	>	100	d	138	Š	176	°	214	Ö	252	ü
63	?	101	e	139	‹	177	±	215	×	253	ý
64	@	102	f	140	Œ	178	²	216	Ø	254	þ
65	A	103	g	141		179	³	217	Ù	255	ÿ
66	B	104	h	142		180	´	218	Ú		
67	C	105	i	143		181	µ	219	Û		
68	D	106	j	144		182	¶	220	Ü		
69	E	107	k	145	‘	183	·	221	Ý		

Mac-Standard character set

32		70	F	108	l	146	í	184	Þ	222	
33	!	71	G	109	m	147	ì	185	þ	223	
34	"	72	H	110	n	148	î	186	Š	224	‡
35	#	73	I	111	o	149	ï	187	ª	225	·
36	$	74	J	112	p	150	ñ	188	º	226	‚
37	%	75	K	113	q	151	ó	189	ý	227	„
38	&	76	L	114	r	152	ò	190	æ	228	‰
39	'	77	M	115	s	153	ô	191	ø	229	Â
40	(	78	N	116	t	154	ö	192	¿	230	Ê
41	)	79	O	117	u	155	õ	193	¡	231	Á
42	*	80	P	118	v	156	ú	194	¬	232	Ë
43	+	81	Q	119	w	157	ù	195	¯	233	È
44	,	82	R	120	x	158	û	196	ƒ	234	Í
45	-	83	S	121	y	159	ü	197	¼	235	Î
46	.	84	T	122	z	160	†	198	Đ	236	Ï
47	/	85	U	123	{	161	°	199	«	237	Ì
48	0	86	V	124	|	162	¢	200	»	238	Ó
49	1	87	W	125	}	163	£	201	…	239	Ô
50	2	88	X	126	~	164	§	202		240	
51	3	89	Y	127		165	•	203	À	241	Ò
52	4	90	Z	128	Ä	166	¶	204	Ã	242	Ú
53	5	91	[	129	Å	167	ß	205	Õ	243	Û
54	6	92	\	130	Ç	168	®	206	Œ	244	Ù
55	7	93	]	131	É	169	©	207	œ	245	¦
56	8	94	^	132	Ñ	170	™	208	-	246	ˆ
57	9	95	_	133	Ö	171	´	209	—	247	˜
58	:	96	`	134	Ü	172	¨	210	“	248	–
59	;	97	a	135	á	173		211	”	249	š
60	<	98	b	136	à	174	Æ	212	‘	250	²
61	=	99	c	137	â	175	Ø	213	’	251	¾
62	>	100	d	138	ä	176		214	÷	252	¸
63	?	101	e	139	ã	177	±	215	×	253	½
64	@	102	f	140	å	178		216	ÿ	254	³
65	A	103	g	141	ç	179		217	Ÿ	255	¹
66	B	104	h	142	é	180	¥	218			
67	C	105	i	143	è	181	µ	219	¤		
68	D	106	j	144	ê	182	ð	220	‹		
69	E	107	k	145	ë	183	Ý	221	›		

MS-DOS character set

32		70	F	108	l	146	Æ	184	©	222	Ì
33	!	71	G	109	m	147	ô	185	¦	223	_
34	"	72	H	110	n	148	ö	186	¦	224	Ó
35	#	73	I	111	o	149	ò	187	+	225	ß
36	$	74	J	112	p	150	û	188	+	226	Ô
37	%	75	K	113	q	151	ù	189	¢	227	Ò
38	&	76	L	114	r	152	ÿ	190	¥	228	õ
39	'	77	M	115	s	153	Ö	191	+	229	Õ
40	(	78	N	116	t	154	Ü	192	+	230	µ
41	)	79	O	117	u	155	ø	193	-	231	þ
42	*	80	P	118	v	156	£	194	-	232	Þ
43	+	81	Q	119	w	157	Ø	195	+	233	Ú
44	,	82	R	120	x	158	×	196	-	234	Û
45	-	83	S	121	y	159	ƒ	197	+	235	Ù
46	.	84	T	122	z	160	á	198	ã	236	ý
47	/	85	U	123	{	161	í	199	Ã	237	Ý
48	0	86	V	124	|	162	ó	200	+	238	¯
49	1	87	W	125	}	163	ú	201	+	239	´
50	2	88	X	126	~	164	ñ	202	-	240	-
51	3	89	Y	127		165	Ñ	203	-	241	±
52	4	90	Z	128	Ç	166	ª	204	¦	242	_
53	5	91	[	129	ü	167	º	205	-	243	¾
54	6	92	\	130	é	168	¿	206	+	244	¶
55	7	93	]	131	â	169	®	207	¤	245	§
56	8	94	^	132	ä	170	¬	208	ð	246	÷
57	9	95	_	133	à	171	½	209	Ð	247	¸
58	:	96	`	134	å	172	¼	210	Ê	248	°
59	;	97	a	135	ç	173	¡	211	Ë	249	¨
60	<	98	b	136	ê	174	«	212	È	250	·
61	=	99	c	137	ë	175	»	213	i	251	¹
62	>	100	d	138	è	176	_	214	Í	252	³
63	?	101	e	139	ï	177	_	215	Î	253	²
64	@	102	f	140	î	178	_	216	Ï	254	_
65	A	103	g	141	ì	179	¦	217	+	255	
66	B	104	h	142	Ä	180	¦	218	+		
67	C	105	i	143	Å	181	Á	219	_		
68	D	106	j	144	É	182	Â	220	_		
69	E	107	k	145	æ	183	À	221	¦		

Appendix 6: Japanese support

New field types

V12 Database Engine supports Japanese text by adding two new field types: String.SJIS and String.YOMI. The mReadDBStructure method must be used to create those two new fields.

Example:

```
[TABLE]
NameOfTable
[FIELDS]
sjisfield string.SJIS
yomifield string.YOMI
[INDEXES]
sjisfieldndx duplicate sjisfield ascending
yomifieldndx duplicate yomifield ascending
[END]
```

Field of type SJIS

SJIS fields can hold Japanese text in the Shift-JIS format. Katakana, Hiragana, English alphabets, punctuation, numerals and Kanji are available in this representation.

Sorting

A SJIS sort will order the fields (thus the records) according to the Shift-JIS numerical code of the characters (1 or 2 bytes) in it. There is no strict equivalence of characters in this sort order. This means that each character has a distinct location in the table and will always sort the same way in a similar sort operation.

Note: The SJIS field does not support full text indexing. The following operators are not supported: wordStarts and wordEquals.

Searching

All operators are supported and Boolean set operators `AND` and `OR` are also supported.

Field of type Yomi (Yomigana)

This field can hold a subset of Shift-JIS character. This subset is restricted to the full range of Katakana, hiragana including those with voiced diacritic marks (nigori, maru) and small characters (contracted sounds). It can also contain punctuation characters. This field doesn't contain kanji.

Sorting

The exact sorting order of the syllabaries is given by the following table:

Note: requires a Japanese font such as "MS Mincho".

Phonetic characters (hiragana and katakana)	Pronunciation (each sounds corresponds to 2 characters in the left column)
ぁァあアぃィいイぅゥうウぇェえエぉォおオ	(a) a (i) i (u) u (e) e (o) o
かカがガきキぎギくクぐグけケげゲこコごゴ	ka ga ki gi ku gu ke ge ko go
さサざザしシじジすスずズせセぜゼそソぞゾ	sa za shi ji su zu se ze so zo
たタだダちチぢヂっッつツづヅてテでデとトどド	ta da chi dzi (tsu) tsu dzu te de to do
なナにニぬヌねネのノ	na ni nu ne no
はハばバぱパひヒべビぴピふフぶブぷプへヘべベぺペほホぼボぽポ	ha ba pa hi bi pi fu bu pu he be pe ho bo po
まマみミむムめメもモ	ma mi mu me mo
ゃャやヤゅュゆユょョよヨ	(ya) ya (yu) yu (yo) yo
らラりリるルれレろロ	ra ri ru re ro
わワをヲんン	wa wo n

Dash symbol

The dash symbol has a special meaning in Yomigana. This special treatment breaks the uniformity of the comparison method of the sorting procedure. To remove this problem, the dash symbol should be replaced by its respective vowel in the input data. In other words, the compare method does not take into account the semantic of the dash symbol. The mSetCriteria method translates the dash symbol to the correct (preceding) vowel. The translation is defined in the following table.

Reading	Kana	Dashed
1/2 a (hiragana)	ぁ	ぁあ
1/2 a (katakana)	ァ	ァア
a (hiragana)	あ	ああ
a (katakana)	ア	アア
1/2 i (hiragana)	ぃ	ぃい
1/2 i (katakana)	ィ	ィイ

i (hiragana)	い	いい
i (katakana)	イ	イイ
1/2 u (hiragana)	ぅ	ぅう
1/2 u (katakana)	ゥ	ゥウ
u (hiragana)	う	ううう
u (katakana)	ウ	ウウ
1/2 e (hiragana)	ぇ	ぇえ
1/2 e (katakana)	ェ	ェエ
e (hiragana)	え	ええ
e (katakana)	エ	エエ
1/2 o (hiragana)	ぉ	ぉお
1/2 o (katakana)	ォ	ォオ
o (hiragana)	お	おお
o (katakana)	オ	オオ
ka (hiragana)	か	かあ
ka (katakana)	カ	カア
ga (hiragana)	が	があ
ga (katakana)	ガ	ガア
ki (hiragana)	き	きい
ki (katakana)	キ	キイ
gi (hiragana)	ぎ	ぎい
gi (katakana)	ギ	ギイ
ku (hiragana)	く	くう
ku (katakana)	ク	クウ
gu (hiragana)	ぐ	ぐう
gu (katakana)	グ	グウ
ke (hiragana)	け	けえ

ke (katakana)	ケ	ケエ
ge (hiragana)	げ	げえ
ge (katakana)	ゲ	ゲエ
ko (hiragana)	こ	こお
ko (katakana)	コ	コオ
go (hiragana)	ご	ごお
go (katakana)	ゴ	ゴオ
sa (hiragana)	さ	さあ
sa (katakana)	サ	サア
za (hiragana)	ざ	ざあ
za (katakana)	ザ	ザア
shi (hiragana)	し	しい
shi (katakana)	シ	シイ
ji (hiragana)	じ	じい
ji (katakana)	ジ	ジイ
su (hiragana)	す	すう
su (katakana)	ス	スウ
zu (hiragana)	ず	ずう
zu (katakana)	ズ	ズウ
se (hiragana)	せ	せえ
se (katakana)	セ	セエ
ze (hiragana)	ぜ	ぜえ
ze (katakana)	ゼ	ゼエ
so (hiragana)	そ	そお
so (katakana)	ソ	ソオ
zo (hiragana)	ぞ	ぞお
zo (katakana)	ゾ	ゾオ

ta (hiragana)	た	たあ
ta (katakana)	タ	タア
da (hiragana)	だ	だあ
da (katakana)	ダ	ダア
chi (hiragana)	ち	ちい
chi (katakana)	チ	チイ
dzi (hiragana)	ぢ	ぢい
dzi (katakana)	ヂ	ヂイ
1/2 tsu (hiragana)	っ	っう
1/2 tsu (katakana)	ッ	ッウ
tsu (hiragana)	つ	つう
tsu (katakana)	ツ	ツウ
dzu (hiragana)	づ	づう
dzu (katakana)	ヅ	ヅウ
te (hiragana)	て	てえ
te (katakana)	テ	テエ
de (hiragana)	で	でえ
de (katakana)	デ	デエ
to (hiragana)	と	とお
to (katakana)	ト	トオ
do (hiragana)	ど	どお
do (katakana)	ド	ドオ
na (hiragana)	な	なあ
na (katakana)	ナ	ナア
ni (hiragana)	に	にい
ni (katakana)	ニ	ニイ
nu (hiragana)	ぬ	ぬう

nu (katakana)	ヌ	ヌウ
ne (hiragana)	ね	ねえ
ne (katakana)	ネ	ネエ
no (hiragana)	の	のお
no (katakana)	ノ	ノオ
ha (hiragana)	は	はあ
ha (katakana)	ハ	ハア
ba (hiragana)	ば	ばあ
ba (katakana)	バ	バア
pa (hiragana)	ぱ	ぱあ
pa (katakana)	パ	パア
hi (hiragana)	ひ	ひい
hi (katakana)	ヒ	ヒイ
bi (hiragana)	び	びい
bi (katakana)	ビ	ビイ
pi (hiragana)	ぴ	ぴい
pi (katakana)	ピ	ピイ
fu (hiragana)	ふ	ふう
fu (katakana)	フ	フウ
bu (hiragana)	ぶ	ぶう
bu (katakana)	ブ	ブウ
pu (hiragana)	ぷ	ぷう
pu (katakana)	プ	プウ
he (hiragana)	へ	へえ
he (katakana)	へ	へエ
be (hiragana)	べ	べえ
be (katakana)	べ	べエ

pe (hiragana)	ぺ	ぺえ
pe (katakana)	ペ	ペエ
ho (hiragana)	ほ	ほお
ho (katakana)	ホ	ホオ
bo (hiragana)	ぼ	ぼお
bo (katakana)	ボ	ボオ
po (hiragana)	ぽ	ぽお
po (katakana)	ポ	ポオ
ma (hiragana)	ま	まあ
ma (katakana)	マ	マア
mi (hiragana)	み	みい
mi (katakana)	ミ	ミイ
mu (hiragana)	む	むう
mu (katakana)	ム	ムウ
me (hiragana)	め	めえ
me (katakana)	メ	メエ
mo (hiragana)	も	もお
mo (katakana)	モ	モオ
1/2 ya (hiragana)	ゃ	ゃあ
1/2 ya (katakana)	ャ	ャア
ya (hiragana)	や	やあ
ya (katakana)	ヤ	ヤア
1/2 yu (hiragana)	ゅ	ゅう
1/2 yu (katakana)	ュ	ュウ
yu (hiragana)	ゆ	ゆう
yu (katakana)	ユ	ユウ
1/2 yo (hiragana)	ょ	ょお

1/2 yo (katakana)	ョ	ョオ
yo (hiragana)	よ	よお
yo (katakana)	ヨ	ヨオ
ra (hiragana)	ら	らあ
ra (katakana)	ラ	ラア
ri (hiragana)	り	りい
ri (katakana)	リ	リイ
ru (hiragana)	る	るう
ru (katakana)	ル	ルウ
re (hiragana)	れ	れえ
re (katakana)	レ	レエ
ro (hiragana)	ろ	ろお
ro (katakana)	ロ	ロオ
wa (hiragana)	わ	わあ
wa (katakana)	ワ	ワア
wo (hiragana)	を	をお
wo (katakana)	ヲ	ヲオ
n (hiragana)	ん	n/a
n (katakana)	ン	n/a
dash	ー	n/a

Note: The last 3 entries (n, n, dash) wouldn't be followed by a dash in Japanese text, so those can be ignored for the purposes of dash-ing

Searching

The Yomi field has exactly the same search characteristics as the SJIS field. See the details under Fields of Type SJIS, Searching.

Data importation

V12-J import Japanese text through the mImport method. No validation is done on the incoming data. It is up to the user to ensure that the text is valid. Fields and record delimiters remain 1-byte characters.

Index

G

I

L

M

O

P

Q

R

S

T

U

V

W

X