

CLARION 5

Learning Your ABCs

**Making A Smooth Transition
from Legacy to ABC**

COPYRIGHT 1999 by TopSpeed Corporation
All rights reserved.

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

This publication supports Clarion 5. It is possible that it may contain technical or typographical errors. TopSpeed Corporation provides this publication “as is,” without warranty of any kind, either expressed or implied.

TopSpeed Corporation
150 East Sample Road
Pompano Beach, Florida 33064
(954) 785-4555

Trademark Acknowledgements:

TopSpeed® is a registered trademark of TopSpeed Corporation.

Clarion 5™ is a trademark of TopSpeed Corporation.

Btrieve® is a registered trademark of Pervasive Software.

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

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

TABLE OF CONTENTS

Preface: A New Car	6
1 - INTRODUCTION	7
Before We Begin - All in Good Fun	8
What You'll Find in this Book	9
How to Use this Handbook	10
Acknowledgments	11
2 - LEARNING THE ABC DEVELOPMENT MODEL	13
Preface	13
The New ABC Templates	13
The Benefits of Clarion Objects (over Legacy)	14
Why Bother with Conversion?	16
Leaving Legacy and Getting Started	19
3 - CONVERSION TOOLS	21
Tools to Port Legacy Applications to ABC	21
Application Converter	21
The ABC Templates are (mostly) Backward Compatible	22
Application Conversion	22
Conversion Hints and Messages	27
Other Issues	28
Making Your Own Rules	29
After Conversion: A little clean up	37
Mapping Legacy Embeds to the ABC equivalent	39
4 - THE ABC DEVELOPMENT ENVIRONMENT	47
The Abstract View	47
Terminology Review	48
Clarion Objects	49
Programming with Objects in Clarion	50
Step 1 - How the ABC Templates generate Clarion Objects	50
Step 2 - How Template Code is Customized when using the ABC Templates	50
What is an Embed Point?	50

The Concrete View	52
Source Embeds in the ABC IDE	52
Embed Tree	53
Using the Embeditor	58
Derivation - more support through the IDE.....	60
How do I find an embed point?	62
Looking at Generated Source	64
5 - GENERAL APPLICATION TECHNIQUES	71
Introduction	71
Error Handling (The ABC Error Class).....	71
Naming Conventions	78
The Local Map	78
INI File Management with ABC	79
ABC Based Toolbars	81
The power of the ABC Translator Class	83
Popup Menus	84
6 - DATA AND FILE ACCESS TECHNIQUES	89
Data and File Access	89
Introduction	89
Overview	90
ABC FileManager, RelationManager, BufferedPairsClass, ErrorClass	91
FieldPairsClass Concepts	93
Files, ABCs, and Legacy Applications	95
New ABC File Handling Capability	96
Advanced References	100
7 - WINDOW AND CONTROL TECHNIQUES	109
Introduction	109
Overview	109
Resizer	110
The Resizer - Overview	110
WindowResize	110
Translator	117
Multi-Language Support - Overview	117

8 - BROWSE PROCEDURE TECHNIQUES	121
Introduction	121
Overview	121
Browse Box Template Features	122
Calling a Lookup from Edit-in-place	124
Combining Edit-in-place and an Update procedure	127
9 - REPORT AND PROCESS TECHNIQUES	133
Introduction	133
Overview	133
Processes & Reports	134
The Progress window	134
Pause and Go	135
Child file processing	136
TakeRecord method	136
Sorting	136
Joins	137
Just Reports	139
The ABC Print Previewer	139
Date and Time	140
Single record printing	141
INDEX	143

Preface: A New Car

A man walks into a new car dealer. You see, he has been driving the “old reliable” Legacy model for many years now. Although he can still push it a few more miles, he knows that its time is running short, and it’s time for a new model.

Much to his dismay, the Legacy model is no longer in production. He even dares to look at other dealerships, but nothing resembles the old Legacy model he is driving now. So, with resignation, he is back in the original dealership, and decides to “kick the tires” on a new model.

He sees a sign that says “ABC Model” and decides to look under the hood. Again, anxiety fills his heart as he sees that the engine of the old Legacy model is a distant memory. There are parts under the hood that he has never seen before, and other recognizable parts are in completely different positions. He closes the hood in horror.

As usual, the salesman is starting to get on his nerves. “C’mon, this baby can do this and that and is the best thing since sliced bread.” To get him off his back, he decides to get in for a test drive.

At first everything seems alien to him...there are a lot of new buttons and dials to learn. Even the seat needs adjusting, and it uses adjustment controls like he’s never seen.

Nevertheless, there is still a key. And a gas pedal. And a brake.

He starts the ABC model, and the engine purrs. This baby can move! He cruises off the lot, and begins to drive like he’s never driven before. A small smile begins to curl on his lips, and he begins to forget why he was so worried in the first place. A quick spin back to the dealership and our hero is now the proud owner of a new car.

Here is your key, dear reader, to the ABC model. Along with this owner’s manual.

By the way, some of you may have a Legacy model that no longer runs, or if it does run, runs in the slow lane. This owner’s manual also has a special section for you “do-it-yourselfers”, to help you rebuild that vehicle with object power!

1 - INTRODUCTION

Welcome to the “Learning Your ABCs - Making a Smooth Transition from Legacy to ABC” handbook.

This is the definitive guide for anyone using (or planning to use) the Clarion ABC (Application Builder Class) templates found in our Enterprise, Professional, or Web Editions. Although this document is written with the legacy user (pre-ABC or Clarion templates) in mind, you should also find it valuable if you are new to Clarion and the ABC templates.

Most of the information contained here is compiled from a wide variety of sources. These include the existing TopSpeed documentation, TopSpeed Education material, news groups and web sites. However, you will find new information and techniques with ABC templates that are specifically created for this handbook.

In addition to the main chapters outlined here, we focus on other key issues, including:

- The benefits of Object Oriented Programming
- The similarities between ABC and Legacy generated source code.
- The key features of the ABC templates and available tools.
- A detailed description of Embed Mapping.
- A list of the most commonly used embed points, and why they are common.

Our goal for this handbook is to increase the number of users that employ the ABC templates in their application projects. We also hope to help the existing users with tips and techniques which will increase your productivity.

Before We Begin - All in Good Fun

Overheard at a recent TopSpeed DevCon:

With Legacy (Clarion) templates we had to code our own bugs. In ABC we can inherit them.

Q. How many ABC programmers does it take to screw in a lightbulb?

A. None. You just send a ChangeBulb message to the socket object!

Legacy gives you enough rope to hang yourself. ABC also gives you the tree object to tie it to. OR....

...Legacy allows you to shoot yourself in the foot. ABC allows you to reuse the bullet.

“Our application is an object-oriented system.
If we change anything, the users object.”

And one more...

An ABC programmer was walking along the beach when he found a lamp. Upon rubbing the lamp a genie appeared who stated "I am the most powerful genie in the world. I can grant you any wish you want, but only one wish."

The ABC programmer pulled out a map of the Mediterranean area and said "I'd like there to be a just and lasting peace among the people in the Middle East."

The genie responded, "Gee, I don't know. Those people have been fighting since the beginning of time. I can do just about anything, but this is beyond my limits."

The ABC programmer then said, "Well, I am an ABC programmer and my programs have a lot of users. Please make all the users satisfied with my programs, and let them ask sensible changes"

Genie: "Umm, let me see that map again."

What You'll Find in this Book

This handbook is divided into the following parts:

- **Introduction**

Chapter 1 describes the purpose, scope and goals of this book.

- **Learning the ABC development model**

Chapter 2 provides a series of topics designed to familiarize you with the ABC style of application development. The benefits of Clarion Objects, features of ABC, and getting started with conversion projects is featured here.

- **Conversion Tools**

Chapter 3 documents in detail the available conversion tools designed to help the user convert legacy (Clarion) applications to applications that are ABC template based.

- **The ABC Development Environment**

Chapter 4 is intended to get you up to speed with the new environment in as little time as possible. Key areas of the Clarion Integrated Development Environment (IDE) is highlighted, including the Classes control and Embed areas. Sample ABC generated code is also examined.

- **General Application Techniques**

Chapter 5 provides application specific (Global) tips and techniques when using the ABC templates.

- **Data and File Access Techniques**

Chapter 6 provides tips and techniques using the ABC libraries to access and control your application's data.

- **Window & Control Techniques**

Chapter 7 provides information and helpful tips to configure your window and specific controls using the ABC templates.

- **Browse, List and Tree Techniques**

Chapter 8 provides specific information concerning Browse procedures, list boxes, and relational tree controls when using the ABC templates.

- **Report and Process Techniques**

Chapter 9 focuses on the processing of your application's data into reports using the ABC templates. Import and Export techniques using the ABC Process template are also discussed.

How to Use this Handbook

A primary goal in creating this book is to provide useful information to *anyone* who is using, or plans to be using, the ABC templates. Based on your current level and needs, there are three areas of useful information:

- If you need to get your legacy (Clarion template based) applications upgraded to the ABC template classes, you should start your reading in the *Conversion Tools* chapter. This chapter provides information to help you “Quick Start” your conversion project, and what to do after conversion is completed.
- If you haven’t decided whether or not you need to convert an existing legacy application, or you are still in the dark concerning the benefits of ABC templates, or you need an update concerning the ABC template support in the Clarion IDE (Integrated Development Environment), you should start reading the *Learning the ABC Development Model* chapter. This chapter presents the benefits of OOP and the ABC templates, identifies and clarifies necessary terminology, and offers a thorough examination of the application development’s built-in support options.
- For the experienced ABC template user, there are a variety of *Techniques* chapters, providing a quick review of the ABC template support, related class libraries and their important properties and methods, and popular embed techniques which were accumulated from many sources. Each chapter is divided into specific program functions, so you can locate and reference the specific information you need.

Acknowledgments

There were many kind and knowledgeable people who helped to contribute valuable information to this book. We would like to acknowledge them here:

- ◆ Mr. Bryce Campbell, who contributed information derived from a helpful utility (TPLINFO) that every legacy to ABC user needs to have in their tool kit. For more information about TPLINFO and other related products, please visit Bryce at:
<http://www.cix.co.uk/~bryce/>
- ◆ Mr. Mike Hanson, who contributed contents of an article that was published a little while ago concerning Application Convertor Rules development. Mike's products and services are available at:
<http://www.BoxsoftDevelopment.com>
- ◆ A heart felt thanks to Mr. Roy Hawkes at the TopSpeed Development Centre, who contributed many ideas and kept our crew on track.
- ◆ To the Rens brothers, Peter and Arie, at Advantage Software. Thank you again for your suggestions and encouragement. Your attention to detail has helped to make this a better book. Visit them at:
<http://www.advantages.nl/gbhoofd.htm>
- ◆ Last, but certainly not least, many thanks to Mr. David Bayliss, who helped to open our eyes to the mountain of ABC benefits. His tips and insights to Class implementation are invaluable. Many thanks again, David!

2 - LEARNING THE ABC DEVELOPMENT MODEL

Preface

Much of the material in this chapter contains invaluable information and tips concerning the benefits of migration from legacy applications to ABC. This information is also found in your Clarion on line help, but also contains additional material.

Legacy applications are defined as those applications using any template set that is not derived from the base ABC template chain. Most legacy applications are those that have been generated with the Clarion template set.

Contrary to popular belief, Clarion programming has not experienced a radical shift in the way that applications are designed, developed, generated, compiled, and linked. The only thing that has really changed is the prewritten code base supplied by the Clarion environment. What follows is an explanation of the subtle differences in the way the two template sets, the Application Builder Classes and the Clarion Templates, function.

This section is designed to get you, the Legacy user, into the right frame of mind for learning Clarion's Objects, as defined in the Application Builder Classes. When discussing template sets, and embedded code it is important to understand that what we are dealing with is canned code: code that is dropped into an application to perform a specific function.

The New ABC Templates

Although the ABC templates first shipped in December 1997, if you have any legacy applications, these templates could still be new to you.

The Clarion Template chain (Legacy) was designed to write pre-tested, procedural, Clarion code. Each template would generate all the code necessary for whatever function it was designed for every time that functionality was needed in an application. This is in stark contrast to the elegant object oriented model of the ABCs, where the actual code is only placed into the application once, and then referenced by the templates as needed. Both template sets are designed to place prewritten code into your application.

The ABC Templates are significantly different than the Clarion for Windows 2.0 (and earlier) templates. Specifically, the new templates generate less code overall. A much higher portion of the generated code is object oriented code, taking advantage of the Clarion Application Builder Class Libraries. Therefore, much of the functionality in the templates is moved out of the generated source and into the static source of the Application Builder Class Libraries.

From the developer's viewpoint, the ABC templates possess many similarities with your legacy applications. The core programming paradigms (Menu, Browse, Form, Report, etc.) have not changed, only many features are added to make your template based development more efficient.

Moving functionality to the Application Builder Classes does not mean the runtime behavior of your application is inaccessible. You can still control your program's behavior through the use of embedded source code (usually with the same embed points available in the CW2.x templates, or comparable embed point in ABC), plus you can derive your own class methods and properties to override the CLARION Application Builder Classes. The new Embeditor feature makes source modification easier than ever before.

The Benefits of Clarion Objects (over Legacy)

You will find when searching our existing documentation, company web site, or on-line newsgroups, that a lot of effort has been expended to explain the benefits and virtues of Object Oriented Programming, or OOP.

In order to better understand the concepts presented in this handbook, let's start thinking less about OOP and more about *Clarion objects*.

Definition: A Clarion *object* is a logical grouping of *properties* (legacy: think variables) and *methods* (legacy: think procedures) that are designed to accomplish a related collection of program tasks.

Properties are values (data) that control an objects' behavior. Methods are specific actions or tasks (prototypes) that are performed by program instructions within the object.

An object binds together data and prototypes. A *class* defines the rules for creating different objects of the same type, that is, they have the same data members and member prototypes. Relationships between classes are expressed using inheritance, containment (encapsulation) and templates. The result is that Clarion objects are defined to the user through their behavior, while hiding the data and specific procedures that cause that behavior. This allows the specific implementation of an object to change without modifying the way it is used by a program at all.

In contrast, legacy programming paradigms lack these features. The modular paradigm allows abstraction of data through the partitioning of the program into modules, but software re-use is restricted because of the lack of a mechanism to express relationships between modules.

The procedural paradigm does not group data and subroutines together in any way. A program is looked upon as a series of unrelated procedures operating on the same set of global data.

Before we introduce additional terminology, let's look at the benefits of using Clarion Objects:

- Lines of code are reduced.
Clarion objects are more flexible than a single procedure or routine written in legacy. They can change their characteristics based on the type of event or message that you can pass to it. So one browse object can be used for every browse procedure in your applications.

Example: A simple browse generated in Clarion produced 1202 lines of code. The same browse with file and features identical generated with ABC based templates produced about 166 lines!

- Development time is decreased (long-term).
Some may argue with this point, especially those who are having trouble with their conversion projects. But look at the growth of the ABC templates over recent months. Each release produces a more solid set of features, with new capabilities and options. For example, the edit-in-place code that you had to hand code in older versions of Clarion are a seamless feature today.
- Code is reusable.
Clarion Objects and the IDE offer the possibility of writing a body of code once, and then reusing that code over and over again. The built-in support for templates, inheritance, and polymorphism ensures that object code can be used in many different contexts.
- Program maintenance is easier
Clarion objects and ABC templates have revolutionized the technology of application development, to allow large projects to be completed and updated using multiple levels of developers. To simplify the process, common programming problems are first split into conceptual areas which can be addressed as separate programming tasks (templates). Each of these tasks are controlled by the ABC library, a collection of classes which perform the tasks needed in a variety of areas in your applications. The ABC libraries are highly optimized and have a particular published set of rules for their use (See the *Application Handbook*). They serve as the primary tool for managing the complexity of large programming projects.

- Databases can contain multimedia and complex object types. Yes, we do have data object in Clarion, expressed as BLOBs (Binary Large Objects). Although not specific to the ABC paradigm, it is nevertheless worthy to mention here.

Why Bother with Conversion?

Depending on the size of your existing legacy projects, there will be some time that you will need to invest during the conversion process. What are the real benefits of crossing over to the ABC templates? Do they compensate for the time you will invest at the start of the application life cycle?

In addition to the benefits of Clarion Objects just discussed, there are many other benefits for you to consider that are ABC template specific. Although the following list is not comprehensive, it should give you a good idea of what the ABC templates have to offer:

General

- Seamless integration of 3rd party substitutions on ABC code.
- Ability to name prefixes and other related elements used in generated code (i.e. - *PeopleBrowse*. rather than *BRWI*.)
- Wizatron support (See the *Wizatron Handbook*)
- Legacy template features are “frozen”. On the other hand, ABC templates continue to improve with each release.
- Avoidance of many ISL project errors in 16 bit applications (specifically reduces the limitations on the number of browses per window and files in the data dictionary).
- Automatic implementation of local maps (reducing project compile time).
- Ability to preserve global variables between sessions (improved INI Manager).

Toolbar

- Automatically references an active browse control populated on a tab control. In legacy, this does not work for child browses without hand (embed) coding.
- Configurability of text using the ABC Translator Class.

Popup Menus

- Restricted control in legacy, but fully configurable in ABC
- Icon support
- Configurability of text (via the Translator Class) leaving code intact.
- Floating toolbox capability (dockable control)

QBE

- Query by Example is only available in the ABC template chain.

Reports

- Ability to skip report preview under program control.
- Far more powerful previewer, fully configurable.
- Icons (and other arbitrary text) on progress bar.
- A better (more accurately calibrated) progress bar.
- Ability to easily hide progress bar.
- Child file view capability
- Date and Time control templates
- Pause / Go capability during printing
- Configurability of preview text
- Sort reports on any value
- Join files on the fly
- Ability to use report for single record printing

Browse

- Edit-in-place
- Improved efficiency (especially using SQL-based data)
- Ability to avoid delay loading the browse until visible (vital when using update procedures with many children).
- File loaded browse support
- Filtered locators, which also can be configured to float right.
- Print button template
- Sort order on any value (non-keyed)
- Ability to switch between multiple update forms (including edit-in-place)
- Selection bar stays in same position when returning from update.
- Selection bar stays in same position when switching tabs
- Ability to support ‘partially filled’ browses (vital for SQL-based data)
- Ability to locate to ‘current’ location when first entering the browse
- Smoothing of the browse refresh “flicker”

Files

- New lazy open capability
- Support for on-server Referential Integrity
- Improved record buffer integrity during RI updates or deletes.
- All priming / validation done in one place (localized in an object)
- Improved integrity when using field validation.
- Support for INLIST validation.
- Improved recovery from a sequential read of a locked record
- Handling of the 'between procedure' alias problem

Forms

- Cancelling a form with child records does not leave orphans.
- Additional field assignments capability on field lookup interface.

Errors

- Configurable text / priority levels
- Ability to override the built in error reporting screen

ASCII Files

- ASCII file driver enhancements.

Drop Combos

- Type-ahead lookup capability
- Page-loaded option for large files
- Support for multiple field linking keys
- Ability to add to a child file automatically even if the child has an auto-incrementing key.

Resizer

- New and improved base strategies and configurability

Translator Class

- Translate any screen to the language of your choice. Fully configurable.

Each of these features mentioned here are covered and explained in more detail in the related *Techniques* chapter found later in this handbook.

Leaving Legacy and Getting Started

After reviewing the benefits of programming with Clarion Objects and examining the features of ABC templates, the next step is to begin the conversion cycle from your legacy (Clarion) based application to ABC. Here is the suggested sequence of tasks that are recommended if you are ready to get started.

Run the Converter Tool

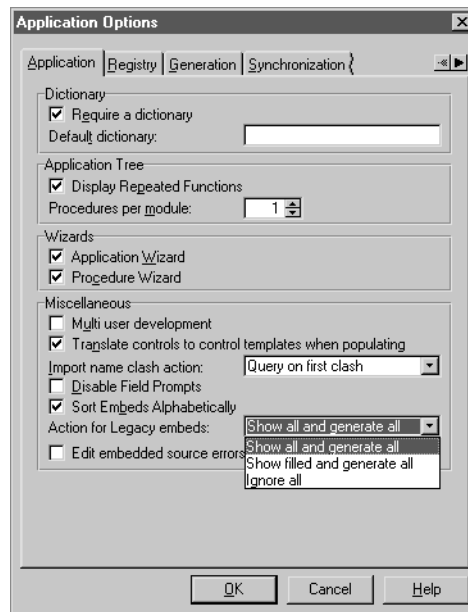
- Run one of your existing legacy applications through the Application Converter in Manual mode. Take notes as the converter explains what it is changing and why (See the *Conversion Tools* chapter for an in-depth description of this process).

Compare the Legacy and ABC Embed Points

- To see the Clarion Template Embed Points side by side with their corresponding ABC Template Embed Points:

From the Clarion IDE Menu,

1. Choose **Setup ► Application Options**
2. From the **Application** tab, set the **Action for Legacy Embeds** option to *Show all and generate all*.



After setting this option, when you have an application file active (loaded) and the Application Tree displayed:

1. RIGHT-CLICK on a selected procedure, and **choose Embeds** to open the Embeditor.

Upgrade your source to Object Syntax

The *Conversion Tools* chapter contains specific guidelines for replacing selected legacy template code with appropriate object syntax.

Object syntax is defined as explicitly referencing any member of any complex structure by prepending the label of the structure containing the field to the field label, separated by a period (StructureName.FieldLabel).

Summary

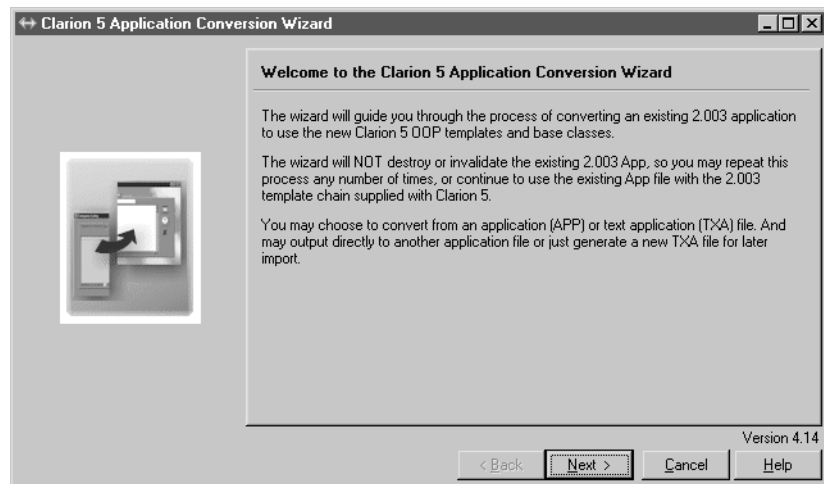
It should be obvious that there are many benefits to upgrading to the ABC templates. Their object-based foundation will certainly create a more efficient application, and the additional features of ABC should eliminate a good majority of the hand-coded embeds that were required with legacy templates.

3 - CONVERSION TOOLS

Tools to Port Legacy Applications to ABC

For applications with embedded source code, TopSpeed provides the following tools to help you get them up and running with the ABC templates:

Application Converter



A porting utility exists to help you move your legacy applications to the ABC Template chain. This tool automates the most common porting tasks. In addition, when you use the converter in Manual mode, you can learn the new coding techniques used in the ABC Templates.

To access this special utility, choose **File Convert ► Application...** from the Application Generator main menu.

2.0 (Legacy) Templates

The 2.0 Templates that ship with Clarion let you continue to use the classic templates for immediate, conservative, and indefinite use (CW.TPL and Wizard.TPL) in your Clarion applications. When you feel ready to enter the Object Oriented power of the ABC templates, you must have these templates registered to use the Application Converter.

The ABC Templates are (mostly) Backward Compatible

TopSpeed has made every reasonable effort to maintain full backward compatibility between the template sets, with a high degree of success. Pure template generated applications (with no embedded code) are fully compatible. That is, you can simply convert the application, generate, compile, and run. Legacy 2.x applications that contain embedded code may require some additional modification.

The ABC Templates optionally support the Clarion Template embed points. See *Action for Legacy Embeds* in the **Application Options** dialog to configure the default handling of these Legacy embed points.

To see the Clarion Template Embed Points side by side with their corresponding ABC Template Embed Points as a default, set Action for Legacy Embeds to Show all and generate all, then open the Embeditor (choose **Edit ► Source**).

You can easily differentiate between legacy and ABC embed points by viewing the embed points general description:

```

! [Priority 3800]
! Parent Call
PARENT.SetQueueRecord()
! [Priority 5500]
! Start of "Legacy: End of Format an element of the browse queue"
! [Priority 4000]
!Legacy Code here
! End of "Legacy: End of Format an element of the browse queue"
SELF.Q.FullName=FullName           !Assign formula result to display queu
! [Priority 8000]
! End of "Browser Method Executable Code Section"

```

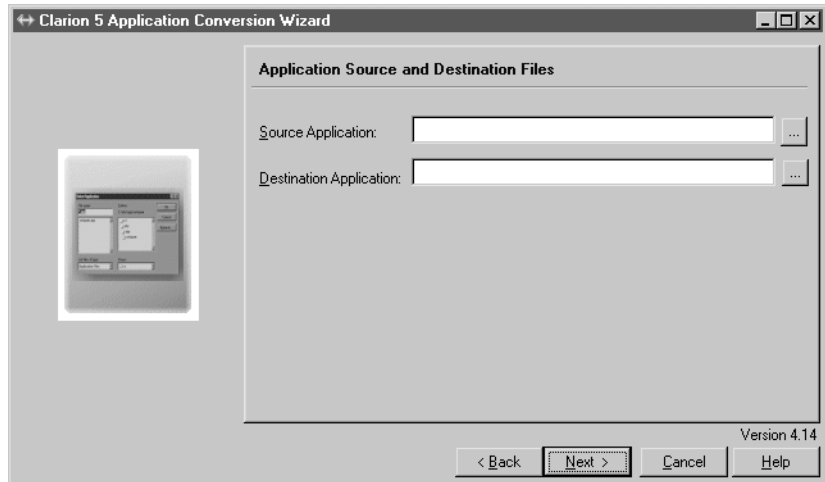
Application Conversion

If you have Legacy applications, and you want to convert them to the ABC templates, one must run them through the conversion process.

The converter is a tool, shipped with Clarion. The entire source code is included as well. This is so that if you have some 3rd party templates, special code that you have used, one can simply add the rules of conversion to the source.

Note: The source code for the Application Converter is located in the `..\ConvSrc` sub folder.

To start the conversion, simply select it off of the File menu. You will see an opening splash window describing the procedure. Just click on *Next*. You now see this window:



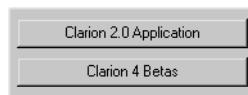
Source and Destination

While you can specify a legacy application file, you may also select a legacy TXA file too. You may also specify either an Application (APP) or Application Text (TXA) for a destination. The converter defaults to the first 7 characters of the old application plus the number 5 to give it a unique name.

Tip: The Application Converter assumes that your source and destination folders will be the same. If you wish to create the destination application in a different folder, make sure to copy the active dictionary also to that folder.

Conversion Options

There are two buttons on the next window. These buttons simply direct you to which set of conversion rules you are applying to your application.



Press the Clarion 4 Betas button to access the Clarion 4 Beta to current ABC conversion rules.

Typically, you will need to use the 2003 application button.

Press the Clarion 2.0 Application button to configure the Clarion 2.00x to Clarion ABC conversion rules. Choosing this option displays the following window:

Configure Clarion 2.0 Application Rule Family

Redundant Locals:	Manual	Report Procedures:	Manual
File Accesses:	Manual	Process Procedures:	Manual
2.0 Std Functions:	Manual	Report Use Details:	Manual
Browse Feq's:	Manual	UnLinked Procs:	Manual
UnTerm OIMITS:	Manual	Change Tpl Chain:	Manual
Toolbar Equates:	Manual	Browse Formula:	Manual
Browse Routines:	Manual	Hints:	Manual
Browse Queue:	Manual	Change ASCII Box:	Manual
Window Routines:	Manual		

Before proceeding, it is important to note several issues:

- Although we have made every effort to insure a smooth conversion, please **BACK UP** all applications and dictionaries before starting the conversion process. Your conversion project should be stored in a separate folder from your actual legacy applications.
- Your legacy apps are not modified by the Application Converter. (i.e., After converting a CW2.003 application, you can still load the CW2.003 application back into the CW2.003 environment.)
- The Application Converter was designed for applications created in Clarion versions 2.x and higher. If you are migrating a CW 1.0 or 1.5 application, you need to first upgrade the application to version CW2.X before proceeding. In most cases, this is simply loading (and saving) the older application into the upgraded environment.

Conversion Rules

All of the defaults shown above are set to “manual”, meaning:

- You can watch the code being converted.
- The converter will stop when it encounters certain pieces of source code.
- You will be presented with several options as to how you would like to convert certain sections of source code.

Tip: If you right-click any of the drop down lists, you will be presented with a menu that will apply to all drop-down lists on this window. The options for each list are *None*, *Manual* and *Automatic*. The pop up menu has the same options, except for they apply to *All* options here.

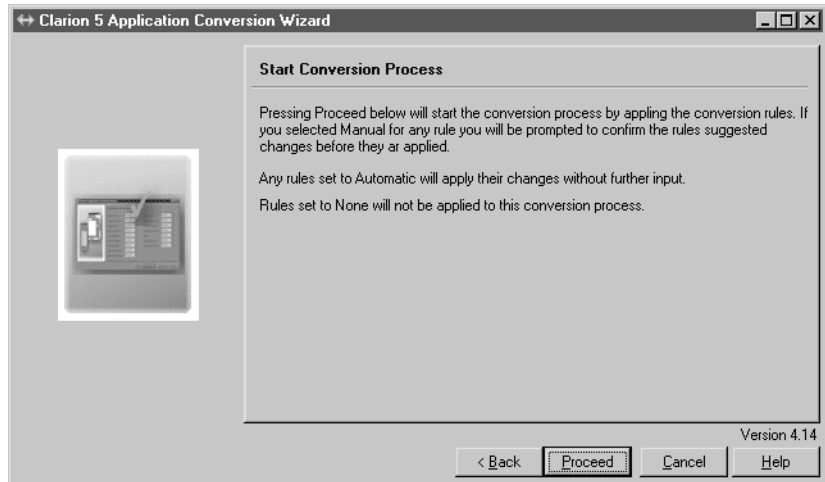
- *None* means that you will ignore any conversion issues for this section, in other words, skip it.
- *Manual* means that you will decide to accept, change or reject the converter’s suggestions.

- *Automatic* means that you will accept the converters recommendation and not to bother you about it.

It is suggested that you use Manual for most issues. The only exception may be the **Change Tpl Chain** option. In this case you may want to change this to Automatic, as the converter knows all the old template symbols and what the new ones are.

Press the **OK** button when you are ready to proceed or **press** the “X” button in the upper right corner of the window to quit or cancel conversion.

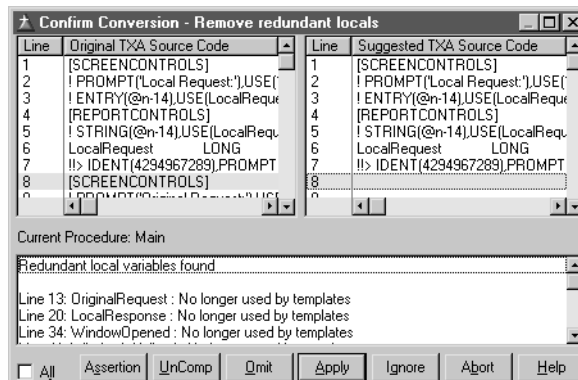
If you pressed OK, you have a final confirmation window that is displayed:



Press the **Proceed** button to begin the conversion process.

During Conversion

Once the conversion process starts, the converter will pass through each section. If the section is set to manual mode, you are asked to accept, change, or ignore certain issues that it finds. Here is a typical response:



This screen has notified us that there are certain locally defined variables used in the legacy templates that are no longer required by the ABC template set.

The buttons at the bottom are options on how to handle this line of code. The default is to apply the suggestions. However, if the new code is close to what you want, but the converter does not quite have it right, you can double click on the highlighted line and type in the correct code. This is handy if you have your own templates and the conversion rule differences are not that great. Also, since the converter does not make any assumptions it cannot 100% rely on, the suggestion may not be applicable to what you need.

Several options are available here:

All

Check this box to apply a buttons action to all the proposed changes for this rule. Clear the box to dispose of each proposed change individually.

Assertion

The proposed (red) code replaces the original code in the new application. The converter inserts a runtime marker (ASSERT(False)) before the new code so the program offers to GPF prior to executing the new code.

UnComp(UnCompile)

The proposed (red) code replaces the original code in the new application. The converter inserts a compile time marker (***) before the new code so the compiler issues a message locating/identifying the new code, and so the program will not successfully compile.

Omit

The proposed (red) code replaces the original code in the new application. The converter inserts an OMIT statement and terminator around the new code so it is neither compiled nor executed.

Apply

The proposed (red) code replaces the original code in the new application.

Ignore

The original code remains in place in the new application.

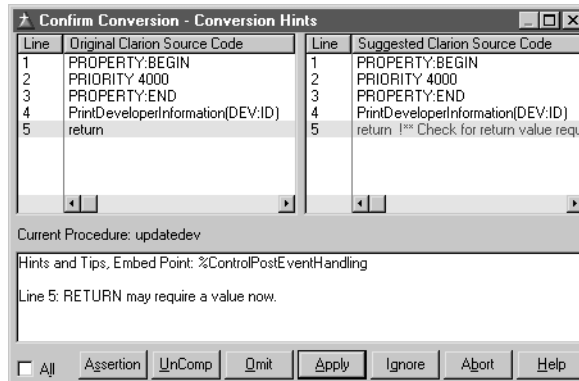
Abort

Press this button to halt and cancel the conversion process.

Tip: As an aid to learning the conversion tool, try converting one of the example applications first. Also, you may also use a TXA instead of the APP file on a live application. Even if you get a few lines wrong, it will not hamper your active application.

Conversion Hints and Messages

During the conversion process, you may be presented with special hints that are incorporated into the Application Converter source:



The hint shown in the window above is alerting you to consider the RETURN statement located in certain embed points. At this time, you should examine the *Language Reference* manual to review the RETURN statements' modifications and samples. Once you have reviewed any proposed changes, you can then decide to remove, resolve or reject (ignore) the suggested hint.

After conversion, when your new application is successfully converted and loaded into the Application Generator environment, begin to compile the application.

At any time during the compile process, you may encounter one or more of the following errors:

Missing %FormulaInstance

This is actually a bug inherited from CW2.003 applications. When you generate an application that contains formulas, the Application Generator issues a message something like “Missing %FormulaInstance.” To correct this problem, simply go to each formula, open it and reselect the formula class, then save the application.

Removed Embed Points

Some embed points have been removed because they are redundant. Mostly these were either hidden or marked for Internal Use Only. Where code exists inside a removed embed point, that code is preserved in an OMITted “Orphaned Code Section” at the end of the procedure. You can cut and paste the code as needed to appropriate embed points.

GETINI and PUTINI Calls

Calls to these runtime library functions should be redirected to the globally instantiated INIMgr object where appropriate.

POPUP Calls

Popups can usually be more efficiently handled using the PopupManager Class and/or associated code template. The PopupManager class will mimic buttons and/or post events to nominated controls without additional coding. A lot of popup related embed code can usually be removed.

Other Issues

PARENT is now a reserved word. Variables and fields cannot use **PARENT** as a label.

Dimensioned groups used to have elements accessed by placing the index on the member field. Although this is still supported for this version, code should be ported to the new syntax that puts the index on the group label, for example:

```
MyGroup[idx].SomeField
```

Orphaned Embed Points

The ABC Templates include a few embed points whose names begin with three asterisks (***) . For example, the “***After opening VIEW” embed point in a Browse procedure. These embed points are provided for backward compatibility only. Any code embedded at these points is generated, but is not compiled because there is no meaningful corresponding point in the ABC Template code. This preserves any code you embedded here with prior versions of Clarion so you can cut and paste the code to an appropriate new embed point.

Making Your Own Rules

The following section is adapted from an article originally written by Mike Hanson, who graciously allowed us to revise and reprint. - Editors

Overview

As prior versions of Clarion were released, we were often left feeling that our old applications were abandoned. There was never a "suitable" upgrade path: If we wanted our applications migrated to the new version, in most cases we had to rewrite them.

TopSpeed has finally broken this trend with the Application Conversion Wizard. It scans through your application file and makes appropriate changes to accommodate the new platform. It handles a wide variety of issues, from changing the template chain to adding a ProgressWindow to Reports.

Some Template Housekeeping

There's one problem, though. If you write your own custom templates, you would probably like to convert them too. If your templates are fully described by the PROMPTS, the biggest need is to change the template chain to match the ABC compatible templates. (e.g., if the old template chain was "MikeHanson", the new chain is "MikeHansonABC".)

Prior to writing conversion rules, the first step is to convert the template itself. This may involve changing file access statements, (Example: #INSERT(%GroupName(Clarion)) to #INSERT(%GroupName(ABC)), and other assorted modifications.

However, template conversion is not the focus of this section. We are more concerned about converting the application *after* the template has been updated.

The Application Converter performs its magic by exporting the old .APP file to a .TXA (the application's text based representation), modifying that .TXA, then importing it into a new .APP file. If we had to change the template's chains ourselves, we might use a similar process. One solution would be to perform a manual search and replace of the chain name in the TXA file. A more preferable option is to create an automatic conversion facility.

Creating a New Rule DLL

The entire source for the Application Converter is located in the \CONVSRC sub folder of Clarion 5. The main engine is contained in CNVENG.PR, CNVENG.CLW, CNVENG.INC and CNVENG.TRN. This source has been used to create C5CNVENG.DLL, which is the foundation for the rest of the conversion modules.

There are two conversion rule sets supplied with C5. The first, CNVRULES, is responsible for converting regular APPs from CW 2.003 to Clarion 5. The second, CNVBETA, is used to convert your applications from interim beta versions of Clarion 4. We'll be copying bits and pieces from both of these rule sets to achieve our goals.

A first look at the conversion source seems difficult to understand. However, its a fairly straightforward process to create your own rules. Just perform the following steps, and you're there. (The example shown is used for both the BoxSoft SuperTemplates and Mike Hanson's public domain offerings)

Step 1 – Make C5CNVENG.LIB

TopSpeed does not include the library (LIB) file necessary for you to link to C5CNVENG.DLL. However, you can use LIBMAKER.EXE (supplied with Clarion 5, and located in the \BIN folder) to create C5CNVENG.LIB. Copy the library file into the \LIB folder of Clarion 5.

Step 2 – Create Your Export (EXP) file

Ultimately, we are trying to create our own dynamic linked library (DLL) that Clarion can use during the conversion process. When creating a DLL, the linker must know what elements to make visible to the outside world. It gets these instructions from an EXP, or export file. When we create applications with the Application Generator, it automatically creates the EXP file for us. In this case, we'll have to do it ourselves.

This is really quite simple. Copy CNVBETA.EXP to STAB_CNV.EXP (the designated name of our DLL), then edit the EXP file as necessary.

In our example, we need to edit the first line only. Your new export file should look something like this:

```
LIBRARY stab_cnv
CODE MOVEABLE DISCARDABLE PRELOAD
DATA MOVEABLE SINGLE PRELOAD
HEAPSIZE 1024
STACKSIZE 1024
EXETYPE WINDOWS
SEGMENTS
ENTERCODE MOVEABLE DISCARDABLE PRELOAD
EXPORTS
InitializeDLL @?
;
;
```

Step 3 – Create the Project File

The TopSpeed development center uses the .PR extension for their project files. Copy either CNVRULES.PR or CNVBETA.PR and rename it to use the .PRJ extension. Make a few modifications with any text editor. The resulting STAB_CNV.PRJ should look like this:

```
-- TopSpeed Converter Rules
#noedit
#system win
#model clarion dll
#pragma define(maincode=>off)
#pragma debug(vid=>full)
#compile "stab_cnv.clw"
#pragma link("C5cnveng.lib")
#link "stab_cnv.dll"
#run copyconv.bat
```

The only other change needed is to add the **#run** command. This executes a batch file to copy STAB_CNV.DLL to the BIN directory.

Step 4 – Create your Source File

A good place to start for this is the CNVBETA.CLW, because it's much smaller than CNVRULES.CLW. There are a number of sections that you'll have to deal with.

```
MEMBER()
INCLUDE('CNVENG.INC')
MAP
    InitializeDLL,NAME('InitializeDLL')
END
OwnerName EQUATE('BoxSoft SuperTemplates')
```

The **MEMBER()** statement (instead of **PROGRAM**) tells the system that we don't have a "Main" program module. This corresponds to the **#pragma define(maincode=>off)** in the project file.

CNVENG.INC contains the class declarations for the conversion engine.

InitializeDLL is the only routine to be exported as a callable procedure. This is a hook for C5CONV.EXE to call into the DLL.

OwnerName is the selected name for the conversion rule set. It will be displayed on a new button in the conversion wizard.

The next step is to continue with the custom class declarations:

```

ChangeSuperTplClass CLASS(RuleClass)
Construct          PROCEDURE
TakeSection       FUNCTION(SectionClass SectionMgr |
                        ,InfoTextClass Info |
                        ,STRING SectionHeader) |
                        ,BYTE,VIRTUAL
                        END

ChangeSuperProcCall CLASS(RuleClass)
Construct          PROCEDURE
TakeSection       FUNCTION(SectionClass SectionMgr |
                        ,InfoTextClass Info |
                        ,STRING SectionHeader) |
                        ,BYTE,VIRTUAL
                        END

```

There are two rules within our rule set. Each rule is responsible for performing one task. In our case, **ChangeSuperTplClass** is responsible for changing all of the template chains, and **ChangeSuperProcCall** is responsible for converting some procedure calls that have changed from the old templates.

All of your Rule Class declarations must contain at least the two methods shown. You can add extra properties and methods of your own, if required.

After our class definitions, we can define the InitializeDLL procedure:

```

InitializeDLL PROCEDURE
CODE

```

This is the empty hook procedure defined in our map.

```

ChangeSuperTplClass.Construct PROCEDURE
CODE
  SELF.Register(100,OwnerName,'Change SuperTemplate Chains'|
    ,'%Change Tpl Chain:', '[COMMON][ADDITION][PROMPTS]')
!----
ChangeSuperProcCall.Construct PROCEDURE
CODE
  SELF.Register(230,OwnerName,'Change Procedure Calls' |
    ,'%Procedure Calls:', '[SOURCE]')

```

These two Construct methods (SELF.Register) define the priority, owner, description, prompt, and applicable template sections for each rule.

The **priority** parameter controls the order in which the rules are applied.

Tip: The best way to determine an appropriate priority is to look at existing conversion rules that do similar things.

The **owner** parameter ensures that the rules are grouped together in the conversion wizard.

The **description** parameter is used for display in various areas of the conversion wizard.

The **prompt** parameter controls the prompt within the group of rules. You should try to keep the highlighted letter unique for each rule in the group.

The **sections** parameter tells the conversion engine which template sections your rule needs to process. There are a number of benefits to this. The conversion process is faster, because each rule applies only to its appropriate sections. In addition, the rules can be simplified to handle the syntax found only within the specified sections. If you're not sure which sections to include, examine other rules that do similar things.

```

ChangeSuperTplClass.TakeSection FUNCTION(SectionClass SectionMgr|
                                     ,InfoTextClass Info,STRING SectionHeader)

cLine CSTRING(MaxLineLen),AUTO
i      LONG(1)

StrQ   QUEUE
Olds   CSTRING(50)
NewS   CSTRING(50)
      END

CODE
DO BuildStrQ
SELF.Buttons=Action:Apply
Info.AddTitle('Template Name Changed: '&SectionHeader&' section')
LOOP
  SectionMgr.GetLine(i,cLine)
  SELF.Lexer.TakeLine(cLine)
  IF SectionHeader='[COMMON]'
    IF SELF.Lexer.GetToken(1)='FROM'
      DO TryReplace
    END
  ELSIF SectionHeader='[ADDITION]'
    IF SELF.Lexer.GetToken(1)='NAME'
      DO TryReplace
    END
  ELSE
    DO TryReplace
  END
  IF SectionMgr.LineChanged(i,cLine)
    SectionMgr.SetLine(i,cLine)
  END
  i+=1
WHILE i<=SectionMgr.GetLineCount()
RETURN Level:Benign
!-----
BuildStrQ ROUTINE
StrQ.Olds = 'SuperSecurity'
StrQ.NewS = 'SuperSecurityABC'
ADD(StrQ); ASSERT(~ERRORCODE())
!
StrQ.Olds = 'MikeHanson'
StrQ.NewS = 'MikeHansonABC'
ADD(StrQ); ASSERT(~ERRORCODE())
!
StrQ.Olds = 'MHResize'
StrQ.NewS = 'MikeHansonABC'
ADD(StrQ); ASSERT(~ERRORCODE())
!
StrQ.Olds = 'SuperOddsAndEnds'
StrQ.NewS = 'MikeHansonABC'
ADD(StrQ); ASSERT(~ERRORCODE())

```

```

!-----
TryReplace ROUTINE
  DATA
  j BYTE,AUTO
  k BYTE,AUTO
  TokenStart USHORT,AUTO
  TokenEnd USHORT,AUTO
  CODE
  LOOP j = 1 TO RECORDS(StrQ)
    GET(StrQ, j); ASSERT(~ERRORCODE())
    k = SELF.Lexer.FindToken(StrQ.01dS)
    IF k
      TokenStart = SELF.Lexer.GetStartChrPos(k)
      TokenEnd = SELF.Lexer.GetEndChrPos(k)
      cLine = SUB(cLine, 1, TokenStart-1) & StrQ.NewS |
              & cLine[TokenEnd+1 : LEN(cLine)]
      SELF.Lexer.TakeLine(cLine)
      Info.AddLine(StrQ.01dS & ' template changed to ' |
                  & StrQ.NewS & ' template', i)
    BREAK
  END
END

```

This method is responsible for changing the template chain. It's similar to Clarion's chain converter, except that it understands multiple template chains. First, it loads a local queue with old chains and their new counterparts. Notice that we are amalgamating several of the old public domain template chains into a single chain.

SELF.Buttons is used to specify which buttons are available when a potential change is being previewed in manual mode. Because this rule applies to internal template sections and not source code, some of the buttons (like UnComp) are not applicable.

Info.AddTitle defines the title describing the type of change being made. It is displayed in the information box in the bottom left corner of the modification preview window.

SectionMgr.GetLine requests the next line from the section. Then **SELF.Lexer.TakeLine** parses the line into tokens.

At this point, each of the section types is checked for the expected tokens. If it's the **COMMON** section, then the template chain name will be preceded by the word "FROM". If it's the **ADDITION** section, then the template chain name will be preceded by the word "NAME". In the **PROMPTS** section, it could be almost anywhere. If applicable, we **DO TryReplace** to change the template chain name.

If the routine is successful, **SectionMgr.SetLine** is called to remember the changes.

The **TryReplace** routine simply checks each queue entry, looking for old template chain names. If any are found, the token is replaced with the new chain name (via string splicing), and the resulting line is passed back into the Lexer class (one of the built-in converter classes) for re-parsing. To perform the slicing, our code uses methods from the Lexer class, including FindToken, GetStartChrPos, and GetEndCharPos. In addition, a descriptive line is added to explain what's happening.

The **TakeSection** method for the other rule is quite similar, and you can always look at the source to see how it works.

Step 5 – Make the DLL

Now that you've got your PRJ, EXP and CLW, it's time to make your DLL. Load the PRJ and hit the lightning bolt. With any luck, you should have a new conversion DLL. Don't forget that the DLL needs to be in the \BIN folder of Clarion 5 for it to work.

Step 6 – Update C5CONV.INI

Finally, you must update the C5CONV.INI file located in the Clarion 5 \BIN folder so that the conversion wizard knows about your new DLL. Just add another entry to the RuleDLLs section, as follows:

```
[RuleDLLs]
1=CNVRULES.DLL
2=CNVBETA.DLL
3=STAB_CNV.DLL
```

Conclusion

Well, that's the basics. Of course, these were very simple conversion rules. We didn't have to worry about parsing entire commands and substituting other commands with different parameters. I would think, however, that these rules would accommodate 99% of people with non-Clarion templates. For those of you with more complex needs, just peruse CNVRULES.CLW. It contains a plethora of rules for you to copy and modify.

After Conversion: A little clean up

- Review the following short list of common code constructs (an explanation of the comments follows the list):

<u>Legacy Clarion 2.00x</u>	<u>ABC Clarion</u>	<u>Comment</u>
LocalRequest	SELF.Request or ThisWindow.Request	Use SELF in a ThisWindow object
LocalResponse	SELF.Response or ThisWindow.Response	Use SELF in a ThisWindow object
DO RefreshWindow	SELF.Reset(TRUE) or ThisWindow.Reset(TRUE) ForceRefresh=TRUE	RefreshWindow routine = Reset method Use SELF in a ThisWindow object Now a parameter of ThisWindow.Reset
DO SyncWindow	SELF.Update or ThisWindow.Update	SyncWindow routine = Update method Use SELF in a ThisWindow object
DO ProcedureReturn	RETURN(Level:Fatal) or ThisWindow.Kill ReturnValue = Level:Fatal	In procedure ROUTINES to postpone RETURN until method end
CheckOpen(file)	Relate:file.Open()	CheckOpen = Open method
CLOSE(file)	Relate:file.Close()	Close method “smarter” than CLOSE
ASCIIBox	ASCIIViewControl	new control template name
ASCIISearchButton	ASCIIViewSearchButton	new control template name
DO BRWn::InitializeBrowse	SELF.ResetFromView or BRWn.ResetFromView	InitializeBrowse routine = ResetFromView method
DO BRWn::NewSelection	SELF.TakeNewSelection() or BRWn.TakeNewSelection()	NewSelection routine = TakeNewSelection method
DO BRWn::AssignButtons	Toolbar.SetTarget(SELF.ListControl)	AssignButtonsroutine= Toolbar.SetTargetmethod
DO BRWn::RefreshPage	SELF.ResetSort(1) or BRWn.ResetSort(1)	RefreshPage routine = ResetSort method
DO BRWn::GetRecord	SELF.UpdateBuffer or BRWn.UpdateBuffer	GetRecord routine call = UpdateBuffer method
DO BRWn::PostNewSelection	SELF.PostNewSelection or BRWn.PostNewSelection	PostNewSelection routine = PostNewSelection method

<u>Legacy Clarion 2.00x</u>	<u>ABC Clarion</u>	<u>Comment</u>
BRWn::	SELF.Q. or BRWn.Q.	Queue:Browse:x available in VIRTUALS as SELF.Q.
NEXT(file) IF ERRORCODE()...	IF Access:file.Next()	Next() returns success/failure flag
ADD(file) IF ERRORCODE()...	IF Access:file.Insert()	Insert() returns success/failure flag
PUT(file) IF ERRORCODE()...	IF Access:file.Update()	Update() returns success/failure flag
GET(file,key) IF ERRORCODE()...	IF Access:file.Fetch(KEY)	Fetch() returns success/failure flag

Let's clear up a few important items at this point:

1. Try to get used to the *dot syntax* as soon as possible. Besides being an industry standard, it becomes easier to read and interpret with time.

Any member of any complex structure can be explicitly referenced by prepending the label of the structure containing the field to the field label, separated by a period (StructureName.FieldLabel). For example, for the following CLASS declaration:

```
MyClass CLASS
MyProc  PROCEDURE
        END
```

you would call the MyProc PROCEDURE(method) as:

```
CODE
MyClass.MyProc
```

2. Notice the use of SELF in many areas. SELF allows the methods to generically reference the data members and methods of the currently executing instance of the CLASS, without regard to how it was derived. Start using SELF in your own embed points for this reason.

Mapping Legacy Embeds to the ABC equivalent

NOTE: Legacy embeds marked with an asterisk have special notes at the end of the embed table.

Old Legacy Name	New ABC Embed	Priority
***After Opening View	NO DIRECT REPLACEMENT	
***After Turning QuickScan OFF	NO DIRECT REPLACEMENT	
***After Turning QuickScan ON	NO DIRECT REPLACEMENT	
***Before Opening VIEW	NO DIRECT REPLACEMENT	
***Before turning QuickScan OFF	NO DIRECT REPLACEMENT	
***Before turning QuickScan ON	NO DIRECT REPLACEMENT	
Accept Loop, After CASE FIELD() Handling	Window Manager - Take Event PROCEDURE(),BYTE	2500
Accept Loop, After TakeEvent	Window Manager - Take Event PROCEDURE(),BYTE	2500
Accept Loop, Before CASE FIELD() handling	Window Manager - TakeFieldEvent PROCEDURE(),BYTE	2500
Accept Loop, Before TakeEvent	Window Manager - Take Event PROCEDURE(),BYTE	2500
Activity for each record	Process Manager - TakeRecord PROCEDURE(),BYTE	2500
After Browse Total Loop	Browser - ResetFromView PROCEDURE()	1000
After Closing the Window	WindowManager - Kill PROCEDURE(),BYTE	5100
After Lookups	Process Manager - TakeRecord PROCEDURE(),BYTE	1000
After Opening Progress Window	WindowManager - Init PROCEDURE(),BYTE	4900
After Opening Report	WindowManager - OpenReport PROCEDURE(),BYTE	7500
After Opening the Window	WindowManager - Init PROCEDURE(),BYTE	4900
After Printing Detail Section	Process Manager - TakeRecord PROCEDURE(),BYTE	1000
After Processing the Window	WindowManager - Ask PROCEDURE()	1000

Old Legacy Name	New ABC Embed	Priority
After Refresh Window for Browse Box	Browser - ResetSort PROCEDURE (BYTE,Force),BYTE	2500
After turning QuickScan On	WindowManager - Init PROCEDURE(),BYTE	8500
After Window Runtime Translation	WindowManager Open PROCEDURE()	1000
After calling DOSFileLookup.Ask Method	NO DIRECT REPLACEMENT	
After first record retrieval	WindowManager Next PROCEDURE(),BYTE	2500
After initializing resizer	WindowManager - Init PROCEDURE(),BYTE	8125
After range and filter check	Browser - ValidateRecord PROCEDURE(),BYTE	5100
Before Browse Total Loop	Browser - ResetFromView PROCEDURE()	1000
Before Closing Report	WindowManager - AskPreview PROCEDURE()	2500
Before Closing the Window	WindowManager - Kill PROCEDURE(),BYTE	5100
Before Lookups	Process Manager - TakeRecord PROCEDURE(),BYTE	1000
Before Opening Progress Window	WindowManager - Init PROCEDURE(),BYTE	4900
Before Opening the Window	WindowManager - Init PROCEDURE(),BYTE	4900
Before Print Preview	WindowManager - AskPreview PROCEDURE()	2500
Before Printing Detail Section	Process Manager - TakeRecord PROCEDURE(),BYTE	1000
Before Refresh Window for Browse Box	Browser - ResetSort PROCEDURE (BYTE,Force),BYTE	2500
Before Resizing Window From INI file	WindowManager - Init PROCEDURE(),BYTE	4900
Before SET() issued	WindowManager - Open PROCEDURE()	2500
Before Turning QuickScan On	WindowManager - Init PROCEDURE(),BYTE	8500

Old Legacy Name	New ABC Embed	Priority
Before Window Runtime Translation	WindowManager - Open PROCEDURE()	1000
Before calling DOSFileLookup.Ask Method	NO DIRECT REPLACEMENT	
Before first record retrieval	WindowManager - Next PROCEDURE(),BYTE	2500
Before subsequent record retrieval	ProcessManager - TakeRecord PROCEDURE(),BYTE	2500
Beginning of Procedure, After Opening Files	WindowManager - Init PROCEDURE(),BYTE	4900
Beginning of Procedure, Before Opening Files	WindowManager - Init PROCEDURE(),BYTE	4900
Browse Box, before calling the update procedure	NO DIRECT REPLACEMENT	
Browse Box, process selected record	Browser - TakeEvent PROCEDURE()	4000
Browse Box, returning from the update procedure	Browser - ResetFromAsk PROCEDURE (*BYTE Request , *BYTE Response)	4000
Browse Initialization	WindowManager - Init PROCEDURE(),BYTE	7750
Browse Preparation, Request Normal Operation	WindowManager - Init PROCEDURE(),BYTE	7750
Browse Preparation, Request to Select Record	WindowManager - Init PROCEDURE(),BYTE	7750
Browse Total Loop	Browser - ResetFromView PROCEDURE()	1000
Browser, After Change	Browser - ResetFromAsk PROCEDURE (*BYTE Request , *BYTE Response)	4000
Browser, After Delete	Browser - ResetFromAsk PROCEDURE (*BYTE Request , *BYTE Response)	4000
Browser, After Insert	Browser - ResetFromAsk PROCEDURE (*BYTE Request , *BYTE Response)	4000
Browser, Before Change	NO DIRECT REPLACEMENT	
Browser, Before Delete	NO DIRECT REPLACEMENT	
Browser, Before Insert	NO DIRECT REPLACEMENT	

Old Legacy Name	New ABC Embed	Priority
Browser, Double Click Handler*	Browser - TakeKey PROCEDURE(), BYTE	2500
Browser, End of FillRecord VIRTUAL for FillBackward	Browser - Fetch PROCEDURE (BYTE Direction)	2500
Browser, End of FillRecord VIRTUAL for FillForward	Browser - Fetch PROCEDURE (BYTE Direction)	2500
Browser, End of FillRecord VIRTUAL, reading backward	Browser - Fetch PROCEDURE (BYTE Direction)	2500
Browser, Format an element of the queue	Browser - SetQueueRecord PROCEDURE()	2500
Browser, Start of Fetch VIRTUAL for FillForward	Browser - Fetch PROCEDURE (BYTE Direction)	2500
Browser, Start of Fetch VIRTUAL, reading forward	Browser - Fetch PROCEDURE (BYTE Direction)	2500
Browser, Start of FillRecord VIRTUAL for FillBackward	Browser - Fetch PROCEDURE (BYTE Direction)	2500
Browser, Start of FillRecord VIRTUAL, reading forward	Browser - Fetch PROCEDURE (BYTE Direction)	2500
Browser, TakeKey inside CASE KEYCODE()	Browser - TakeKey PROCEDURE(), BYTE	2500
Browser, Validate Record: Range Checking	Browser - ValidateRecord PROCEDURE(), BYTE	5100
Browser, no records found*	Browser - ResetQueue PROCEDURE(BYTE ResetMode)	6000
Browser, records found	Browser - ResetQueue PROCEDURE(BYTE ResetMode)	6000
Browser, Start of FillRecord VIRTUAL, reading backward	Browser - Fetch PROCEDURE (BYTE Direction)	2500
CASE EVENT() structure, after generated code	WindowManager - TakeWindowEvent PROCEDURE(),BYTE	2500
CASE EVENT() structure, before generated code	WindowManager - TakeWindowEvent PROCEDURE(),BYTE	2500
CASE FIELD() structure, after generated code	WindowManager - TakeFieldEvent PROCEDURE(),BYTE	2500
CASE FIELD() structure, before generated code	WindowManager - TakeFieldEvent PROCEDURE(),BYTE	2500

Old Legacy Name	New ABC Embed	Priority
Control Event Handling, after generated code	NO DIRECT REPLACEMENT	
Control Event Handling, before generated code	NO DIRECT REPLACEMENT	
Control Handling, after event handling	NO DIRECT REPLACEMENT	
Control Handling, before event handling	NO DIRECT REPLACEMENT	
Data Section, After Report Declaration	NO DIRECT REPLACEMENT	
Data Section, After Window Declaration	NO DIRECT REPLACEMENT	
Data Section, Before Report Declaration	NO DIRECT REPLACEMENT	
Data Section, Before Window Declaration	NO DIRECT REPLACEMENT	
ELSE Clause of CASE ACCEPTED()	WindowManager - TakeAccepted PROCEDURE(),BYTE	2500
End of Format an element of the browse queue	Browser - SetQueueRecord PROCEDURE()	2500
End of Procedure	WindowManager - Kill PROCEDURE(),BYTE	5100
End of Procedure, After Closing Files	WindowManager - Kill PROCEDURE(),BYTE	5100
End of Procedure, Before Closing Files	WindowManager - Kill PROCEDURE(),BYTE	5100
Error checking after record Action	ProcessManager -TakeRecord PROCEDURE(),BYTE	2500
FileDrop, End of Format an Element of the Queue	FileDrop - SetQueueRecord PROCEDURE()	2500
FileDrop, Format an Element of the Queue	FileDrop - SetQueueRecord PROCEDURE()	2500
FileDropCombo, After calling update procedure	FileDropCombo - Ask PROCEDURE(), BYTE	2500
FileDropCombo, Before calling update procedure	FileDropCombo - Ask PROCEDURE(), BYTE	2500

Old Legacy Name	New ABC Embed	Priority
FileDropCombo, End of Format an Element of the Queue	FileDropCombo - SetQueueRecord PROCEDURE()	2500
FileDropCombo, Format an Element of the Queue	FileDropCombo - SetQueueRecord PROCEDURE()	2500
Initialize the Procedure	WindowManager - Init PROCEDURE(),BYTE	4900
Other Control Event Handling	NO DIRECT REPLACEMENT	
Other Window Event Handling	WindowManager - TakeWindowEvent PROCEDURE(),BYTE	2500
Preparing Window Alerts	WindowManager - Init PROCEDURE(),BYTE	4900
Preparing to Process the Window	WindowManager - Ask PROCEDURE()	1000
Prime record fields on Insert	WindowManager -PrimeFields PROCEDURE()	2500
Procedure Setup	WindowManager - Init PROCEDURE(),BYTE	4900
Record Priming	NO DIRECT REPLACEMENT	
Refresh Window routine, after lookups	WindowManager - Reset PROCEDURE(BYTE Force = 0)	1000
Refresh Window routine, before DISPLAY()	WindowManager - Reset PROCEDURE(BYTE Force = 0)	1000
Refresh Window routine, before lookups	WindowManager - Reset PROCEDURE(BYTE Force = 0)	1000
Set resize strategy	Resizer -'Init PROCEDURE'(BYTE AppStrategy=AppStrategy...	7000
Setup control resize strategies	Resizer -'Init PROCEDURE'(BYTE AppStrategy=AppStrategy...	7000
Sync Record routine, after lookups	WindowManager - Update PROCEDURE()	7500
Sync Record routine, before lookups	WindowManager - Update PROCEDURE()	7500

Old Legacy Name	New ABC Embed	Priority
Upon field validation failure	NO DIRECT REPLACEMENT	
Validate Record: Filter Checking	Browser - ValidateRecord PROCEDURE(), BYTE	5100
When completed, before writing to disk	WindowManager - TakeCompleted PROCEDURE(),BYTE	2500
When the report is cancelled	WindowManager - SetResponse PROCEDURE(BYTE Response)	2500
Window Event Handling - After Rejected	WindowManager - TakeRejected PROCEDURE(), BYTE	2500
Window Event Handling - Before Rejected	WindowManager - TakeRejected PROCEDURE(), BYTE	2500
Window Event Handling - after generated code	WindowManager - TakeWindowEvent PROCEDURE(), BYTE	2500
Window Event: Open Window, after setting up for read	WindowManager - Next PROCEDURE(), BYTE	2500
Window Event: Open Window, before setting up for reading	WindowManager - Open PROCEDURE()	2500
Window Initialization Code	WindowManager - Open PROCEDURE()	1000

- ◆ With the legacy *Browser, Double Click Handler* embed, use the following translation structure:

```
IF RECORDS(SELF.ListQueue) AND KEYCODE() = MouseLeft2
!Place your code here (you must write the surrounding IF structure, too)
END
```

- ◆ A similar translation is necessary with the *Browser, no records found* legacy embed:

```
IF NOT RECORDS(SELF.ListQueue)
!Place your code here (you must write the surrounding IF structure, too)
END
```


4 - THE ABC DEVELOPMENT ENVIRONMENT

This chapter is divided into two primary sections.

The Abstract View discusses the shift in paradigm for the legacy (Clarion) programmer to the ABC techniques.

The Concrete View examines specific parts of the Clarion Integrated Development Environment that are important to anyone who uses the ABC templates. Information about the derivation of objects, and use of the embeds and Embeditor are discussed here.

The Abstract View

Before we dig into the specifics of Clarion's Application Builder Classes (ABCs) and the specific development environment support, let's take a moment to reintroduce the theory and vocabulary of Object Oriented Programming (OOP). The ABCs are object-based classes of code that have been designed primarily, but not exclusively, for use by data-centric applications.

The primary goal of this handbook is to get you comfortable using *Clarion Objects*. The methodology will be to discuss the theory in general terms, and follow that with a discussion of the specifics of Clarion Objects.

Terminology Review

What is an object?

An object is a physical entity that accomplishes a specific task at run-time. An object exists because there is a bit of code that defines it, and another bit of code that calls it into existence. An object does a specific job, like managing a Window, or displaying a list of database records. An object contains data variables, called **PROPERTIES**, which contain specific information about how the object behaves and appears. Objects also contain code-structures, called **METHODS**, which perform functions to and for the object. Objects are also customizable through a methodology called **INHERITANCE**.

Class

A class is a programming language construct that defines the properties and methods that are required to create a particular object.

Object

An object is a set of properties and methods created at run-time from a class definition. This process is called instantiation. In other words, an object is an instance of a class. Every object has its own set of properties. However all objects instantiated from the same class share the same set of methods.

Property

A data element defined in a class is called a property. Properties typically describe the state, appearance, or functionality of their object.

Methods

Procedures or functions defined in a class are called methods. Methods typically supply required behavior for their object such as, Print or Close.

Inheritance

A class can be derived from another class. Such a class is called a derived class and the class it is derived from is called its parent class. A derived class inherits all properties and methods from its parent class. A derived class may also contain properties and methods which are not declared in its parent class. A class with no parent is called a base class.

Clarion Objects

Clarion objects are based on the Application Builder Classes (ABCs). They contain methods and properties, and rely heavily on derivation to achieve their high degree of power and flexibility.

Classes

The ABCs are a set of object oriented classes that are optimized for creating data-centric, Windows-based applications. They have built in functionality to handle windows, files, relationships, errors, file browsing, query-by-example, and many other data-centric functions necessary for robust business applications.

Objects

Clarion objects are, of course, instantiations of ABCs, and they are expertly and extensively used by the ABC Templates. Clarion developers have relied on templates for years to provide pretested code for their applications. The ABC Templates instantiate Clarion objects and the supporting code to make fully functioning object-oriented applications.

Properties

The ABCs utilize properties as both appearance and functionality determiners. A property contains a value that is evaluated to determine either how the object functions, or how the object appears on a screen or report.

Methods

The ABCs utilize methods to accomplish all of the functions of an object. The methods of a Clarion object are actually small procedures and therefore have their own Data and Code sections like all other procedures in the Clarion language. They also have the same autonomy as any other procedure.

Inheritance

The ABCs and the ABC Templates extensively utilize inheritance to enhance and extend functionality. Some ABCs have been designed to exclusively be parent, or abstract, classes. These classes are never instantiated as objects themselves, but the classes that are derived from them are instantiated as objects. The ABC Templates derive new methods so they can provide the specific functionality required by a particular method. For instance, there is no way for the designer of the ABCs to know which data file a developer will use, so the templates contain code that overrides the initialization method for a browse object to provide the specific file the developer wants to use.

Programming with Objects in Clarion

The power of Clarion lies not only in the ABCs and the Clarion language, but also in the templates that Clarion provides for use with its integrated development environment (IDE). Application, procedure, extension, control, and code templates can be used to create powerful applications quickly. Customizing an application beyond the template-generated code is usually necessary, and the IDE allows for this by providing an interface with the template generated code called embed points. These are the two steps that developers take to create applications in Clarion; they use the templates to do the majority of the work automatically, and then customize by embedding specific Clarion code.

Step 1 - How the ABC Templates generate Clarion Objects

An application file (.APP) contains all of the project settings, global variables, procedures and references to the templates used to create those procedures. When a “make” is done by the developer, Clarion grabs the code contained in the templates and writes it into a source code file for compilation into executable code. The ABC Templates contain code that instantiates and customizes the ABCs. This is the exact same template technology used by the Legacy templates. The big difference being that the Legacy templates were procedural, and therefore wrote much more code into source files without the power and flexibility of Objects.

Step 2 - How Template Code is Customized when using the ABC Templates

Since the ABC Templates are instantiating objects, it stands to reason that if you want to modify the template generated code, the best place to embed custom code is in an object. Objects perform their functionality via methods, and that is one of the places to embed source code. The other place to embed source code is on an Event (Window or Control) which is the same as it was in previous versions of Clarion for Windows.

What is an Embed Point?

An embed point is a predefined “customization” point within the code that is generated by the templates. The code placed within the embed point is stored in the APP file for incorporation whenever the source code is generated. Embed points are provided at every window and control event, as well as predefined points inside the template generated code.

Legacy Embed Points

Legacy template authors created embed points inside their code, usually creating an embed point at every logical break in the procedural code. The labels used for these embed points were usually straightforward and easily understandable. This methodology put the template author in control of where the developer was able to embed custom code. Fortunately, template authors provided many embed points for flexibility.

ABC Embed Points

Embed points within the ABC Templates are available at any point in any object. Custom code can be placed at any point in any method. And because methods are actually mini-procedures, there is a data section for each method, so that the need for implicit variables has been greatly reduced.

Legacy Embedding Methodology

When source code needed to be added to an application using the Legacy Templates, the Embed-Tree was opened in the procedure being worked on and the developer would read the embed points to find the one that was close to the point where he needed his code to be placed. Again, this was usually straightforward because the template author named the individual embed points.

ABC Embedding Methodology

To find the proper embed point when using the ABC templates, it is important to know the Clarion Objects that are being used, and what their methods accomplish. The Embed Tree for each procedure provides a list of all the Clarion Objects and their methods. The Application Handbook is the guide for the Clarion Objects. This manual will also instruct you in the most used methods for embedding your custom code.

The Concrete View

Source Embeds in the ABC IDE

The ABC templates are flexible and powerful. There are times, despite this flexibility, when a programmer must change the behavior of the generated code. This section focuses on what an embed point is, when to use one and (most important) why.

This section of the chapter focuses on four primary topics:

- ◆ **Embed Tree**
A discussion of the Embed Tree, contrasting similarities and differences between legacy and ABC.
- ◆ **Embed Editor (The “Embeditor”)**
A closer look at this new tool, specifically focusing on the aid it now gives the legacy-to-ABC user.
- ◆ **Derivation**
You don’t have to be an expert in OOP to really use it. This section describes the IDE support for adding your own classes derived from a base class.
- ◆ **Looking at ABC Generated Source**
Contrary to what some developers think, ABC code is not a complete “black box”. This section shows a brief overview of ABC generated code using a QuickStart exercise.

Embed Tree

Each template procedure has its own embed tree. Access to this tree is available through two methods:

1. From a procedure properties window, **press** the **Embed** button,
- or
2. **RIGHT-CLICK** on any procedure and **select Embed** from the popup menu.

The tree appears as follows:

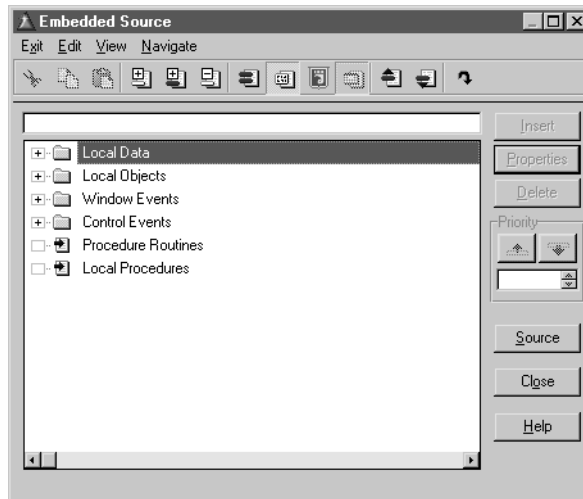


Figure 1 – Collapsed embed tree

Priority

Notice, the Priority group on the right side of the window. Its control is disabled until you add some code.

Priority is defined as the *sequence* in which your code will be generated along with or within the template generated code.

The idea of priority is to keep the amount of embed points manageable without limiting the flexibility that the developer might need. With the Priority scheme, one has access to literally hundreds of *new* embed points! How?

Priority values can range from 1 to 10000, in theory giving the programmer 9999 different entry points. At first, this may seem intimidating. However, most embeds contain a base parent object (or, *parent call*) where template code is generated. This parent object is given a default priority code of 5,000. So the priorities of 1 to 4999 represent code to be executed *before* the parent call, while priorities of 5001 to 10000 represent code to be executed *after* the parent call.

In ABC terms, “Parent call” is defined as calling or running the object that the embed represents. If you assign the same priority as the Parent (5000), then the embed will appear in random order. This may or may not be harmless, but if you want to ensure that it does happen in the order you place it, then use the appropriate priority code.

Selecting a priority now is not an issue. We can use the up and down arrows to move our source code up or down in relation to the template-generated code. Appropriate priorities are then automatically set for us.

What does all this mean?

The Embed window provides a more understandable way to communicate with an application, and we no longer need to worry about priorities directly. We can simply place our embeds in an appropriate position and the correct priority level will be set for us.

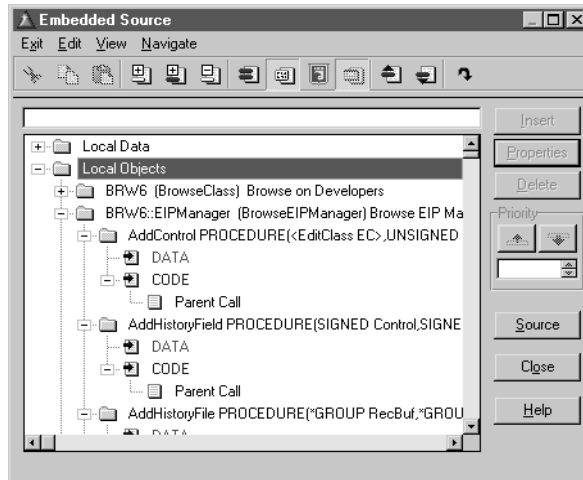
The legacy method of filled embeds included only programmer-generated fragments of code with an associated hard coded priority. Template-generated code did not appear.


The ABC “metaphor” includes programmer-generated fragments plus template-generated fragments. Movement of programmer-generated code within these fragments can be done easily with the up and down arrows rather than setting a priority number. For most of us the ABC metaphor will be much easier to work with.

To summarize, there are hundreds of embed points available in the simplest procedure. The difference between the legacy and ABC templates, is that the developer decides on the specific embed point. In other words, you now have embed points that are *dynamic*.

The ABC Embed Standard

A procedure generated by the ABC templates is actually composed of several “mini” procedures (more accurate, Clarion Object Methods). Each of the embed points available by default is actually showing the available Data and Code sections of these procedures. This is different than the legacy templates, which populate the embed points throughout the code itself.



Click on the Clarion  button to view or use the Legacy embed points when using the ABC template set. All of the legacy embed points become available. The legacy names can help you find the embed that you need, although you are really coding in an ABC embed!

There are five primary embed labels that hide and organize the ABC templates and its associated Clarion objects to any level that you may need to view.

◆ Local Data

Based on the type of ABC template you are using, this section offers embed points into the data types and structures that are necessary for the ABC generated code. Local data is further divided into *Generated Declarations* and *Other Declarations*, so you can modify the properties of the generated data, or declare your own data for anything that you need. Priorities (or positional movement of your embeds) determine if your code statement is applied before or after the generated structure.

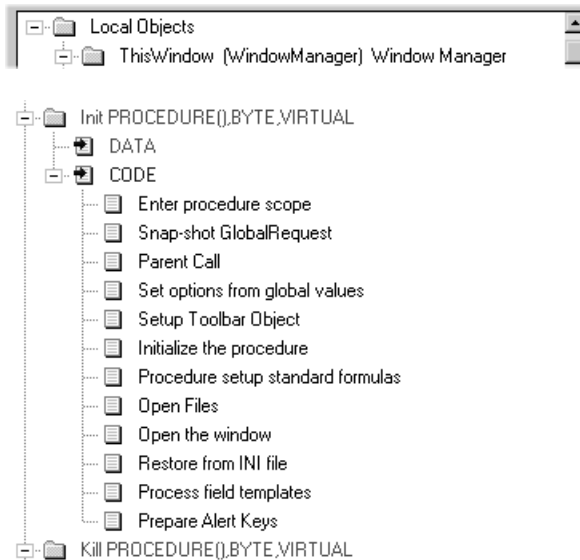
◆ Local Objects

Of all the embed groups discussed here, the *Local Objects* group can be the most challenging to the legacy programmer. A simple ABC Window template contains two local objects (ThisWindow and Toolbar) which contains a combination of 40 methods.

Some methods are simply created (derived) from a base class unmodified, and marked in black. Other methods are *Virtual*, implying that the objects were derived and modified by the local object (the ABC templates need this modified method to generate the proper code), and are marked in green by default. Finally, *protected* methods are methods that can only be called from their defined local object, and are marked in red.

For the legacy programmer, there are tools that we will discuss and generated source that we will examine in this chapter that will help you choose the proper method embed point, and if and when you actually need to override the behavior of a locally defined object.

In addition, the embed description gives you a lot of clues as to why you would need to use the embed in the first place. Examine the following windows:



ThisWindow is the default name of the Local Object created by the ABC templates. In parentheses, *WindowManager* is the base class that *ThisWindow* was derived from. As a legacy programmer, if I want to know the details of this object, I would reference the *Application Handbook* for all of the properties and methods defined in the *WindowManager* class.

The *INIT* procedure is actually a method of the *ThisWindow* object, and common sense descriptions of the embed points' functions are described here. For example, if I need to perform some kind of user validation prior to entering the window, one possible embed point would be *Initialize the procedure*. If the validation was dependant on a file, I would use the *Open Files* embed. Finally, if the validation required the use of a window variable, I could use any embed point that followed the *Open the window* embed.

We will examine the ABC object code later in this chapter, but let's wrap up the other primary embed classifications.

◆ **Window Events**

Legacy programmers should feel comfortable with this, and the remaining embed classifications. The *Window Events* group controls nine common window events and gives the user the ability to detect and respond to these events in whatever way that is necessary.



◆ **Control Events**

This group is more dynamic than *WindowEvents*, in that the amount and type of events available are dependant on the controls populated in the ABC template procedure. The beauty of the ABC model is that when you give a control a special attribute (like Drag and Drop) an event embed point, if appropriate, is automatically generated for you.

◆ **Procedure Routines**

The legacy programmer will appreciate that the support of the ROUTINE paradigm is alive and well in the ABC templates. Any routine that you have defined is placed into this embed area, and called with the DO statement elsewhere in your embedded code.

◆ **Local Procedures**

In many procedures a simple routine is not appropriate due to the limited scope of use within the active module. This embed classification allows the legacy programmer to embed procedures locally into an ABC template procedure, increasing its accessibility.

Using the Embeditor

Another utility to help you use and understand the ABC templates and their associated embed points is integrated into the Clarion IDE and called the Embeditor. This tool goes a long way to understanding how ABC source is generated.

To access it, simply RIGHT CLICK on any procedure you wish to see. Then **choose Source** from the popup menu. The **Module** menu item also available on the popup menu is the actual generated source, so at times it may not be enabled if you have not yet generated any source. The Embeditor is also accessible via the **Source** button found in the *Embedded Source* window.

Here is a sample of what we see when it is first opened:

```

UpdateDevelopers PROCEDURE
! Start of "Data for the procedure"
! [Priority 1300]

FilesOpened          BYTE
ActionMessage        CSTRING(40)
! [Priority 3000]

! Views & Queues
BRW3::View:Browse   VIEW(Scores)
                    PROJECT(SCO:TestNumber)
                    PROJECT(SCO:Score)
                    PROJECT(SCO:TestType)
                    PROJECT(SCO:DateTaken)
                    PROJECT(SCO:Pass)
                    PROJECT(SCO:CertificateNumber)
                    PROJECT(SCO:ID)
                    END
Queue:3             QUEUE
SCO:TestNumber     LIKE(SCO:TestNumber)
SCO:Score          LIKE(SCO:Score)
SCO:TestType       LIKE(SCO:TestType)
SCO:DateTaken      LIKE(SCO:DateTaken)
SCO:Pass           LIKE(SCO:Pass)
SCO:CertificateNumber LIKE(SCO:CertificateNumber)
SCO:ID             LIKE(SCO:ID)
Mark               BYTE
ViewPosition       STRING(1024)
! [Priority 4000]

History::DEU:Record LIKE(DEU:RECORD),STATIC
  
```

The Layout

The gray areas represent generated code by the templates. The white areas are the actual embeds where you can add your own source. This is very useful as you can see your code *in context* with the generated code.

The Embeditor also generates all code that is *possible* for this procedure. This is not all the code that *is* generated.

Another thing to keep in mind is that if you add pre-written code templates, this will be represented (contained) in the gray areas.

The nice thing about the Embeditor is that the code you add is also seen in the standard embed tree once you close the Embeditor.

Priorities Revisited

Each embed point also has the default priority number listed above the embed point (if you have this option in the Setup menu). While priority numbers were discussed previously, let's examine them once again.

While there is no hard and fast rule about priority numbers, there are some general guidelines for legacy users as follows:

- 1.** Legacy embeds are usually contained between the ranges of 2500 to 7500.
- 2.** CASE structures have gaps of about 500 for each level. For example:

```
#PRIORITY(n)
CASE
OF something
#PRIORITY(n+500)
ELSE
#PRIORITY(n+1000)
END
```

In real world programming, priorities tend to depend on the complexity of the procedure. An INIT procedure could be tighter in the numeric gaps, but most other methods are fairly standard with set priorities.

In summary, do not worry too much about the priority numbering scheme. It is not a precise science, nor is it meant to be. Usually, close is good enough.

Navigation

If you look towards the top of the Embeditor window you will notice four buttons.



These buttons, starting from the left are **Previous embed**, **Next embed**, **Previously Filled embed** and **Next Filled Embed** respectively.

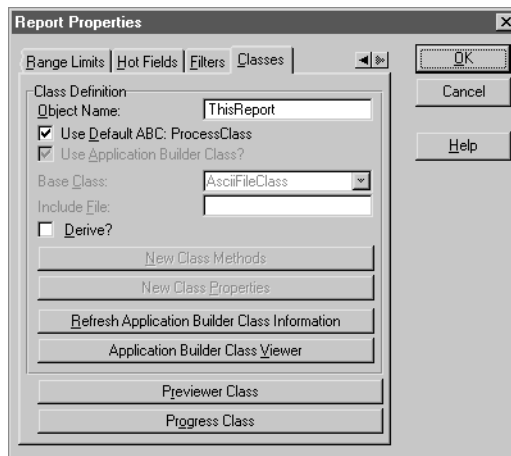
Benefits

With its repeated use, and the legacy users' gradual familiarity with Clarion objects and the ABC templates, the use of the Embeditor becomes invaluable. It can assist you in writing better code by allowing a view of your embed points in context. In simple terms, you write better code by seeing how it is all coming together. Many times, you can determine in the Embeditor that embedded code you used to write in legacy applications is no longer necessary with the ABC templates.

TIP: Examine the example PEOPLE application (located in the \EXAMPLES\PEOPLE folder) using the Embeditor. It illustrates that by using a few key embed points with ABC report templates, you can provide rich features with very little programming effort. An Embeditor study of this application will lead to a better understanding of how Clarion objects and the ABC template embeds can work together.

Derivation - more support through the IDE

Derivation refers to items or things that are taken from other sources and are therefore not original. Clarion objects are derived from base classes, and the ABC templates support this process easily. While the templates generate object names that are very basic and generic, there are times when you may want to use more descriptive names for your derived objects.



The default for the object name is listed in the **Object Name** entry. If you do not like the template default object name, replace it here with your own.

If you do not want to use the default ABC class used by the ABC template (in the window shown, *ProcessClass*), then clear this check box. The Use Application Builder Class becomes enabled, allowing you to pick a new base class. If there is an include file for the class definition, type in the name here. This file must be visible either in the current program folder or the redirection file.

Generally, this technique is used when you (or a third party) have written a base class that more closely emulates the features and tasks that must be performed by your application. The ABC templates are (in most cases) good enough for the great majority, but its nice to know that you have the power to override the default class behavior at any time thorough the IDE.

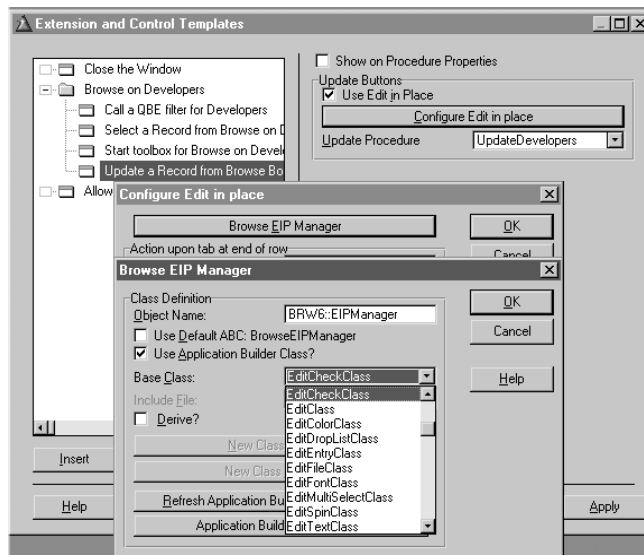
If you want to derive a new class method or property, check the **Derive** check box. This enables the **New Class Methods** and **New Class Properties** buttons. In our *Report Properties* window shown, you may also do the same for the Previewer and Progress class objects.

In the global section of your application, you can also override and/or derive new classes based on the classes that are pertinent to the application.

When to derive

As stated previously, in the vast majority of application development there is little need to override and/or derive new classes due in part to the power of the ABC class library. It is suggested that one use these classes until you find that a class is not going to produce the code or functionality you need.

A popular use of derivation is in the EditInPlace Manager:



See the *Browse Techniques* chapter in this handbook for more detail in using the template-supported edit-in-place configuration.

How do I find an embed point?

First, do I even need one?

While one tries to get a basic understanding of how the ABC embeds work, here are a few pointers:

- ◆ Understand what “derivation” means. Adding an embed is actually causing the ABC templates to derive a new method or property. This is a key to understanding how the ABC embeds really work.
- ◆ Look at a report (any report will do). Where does the print statement, PRINT(RPT:Detail), get generated? Ask yourself why it was placed there. Based on the object name, you can make some logical conclusions:

```

ThisReport.TakeRecord PROCEDURE
ReturnValu          BYTE,AUTO
! Start of "Process Manager Method Data Section"
! [Priority 3500]

SkipDetails BYTE
! [Priority 8500]

! End of "Process Manager Method Data Section"
CODE
! Start of "Process Manager Method Executable Code Section"
! [Priority 500]

! Parent Call
ReturnValu =PARENT.TakeRecord()
! [Priority 5500]
!Do something here before printing
PRINT(RPT:detail)
! [Priority 8000]
!Do something here after printing
! End of "Process Manager Method Executable Code Section"
RETURN ReturnValu

```

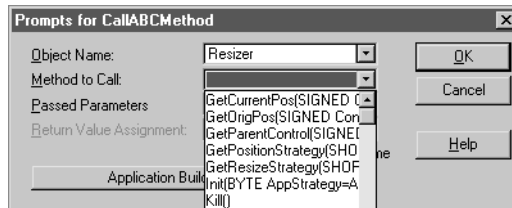
Our embed points refer to the *TakeRecord* method of *ThisReport*, which was derived by the ABC templates from the *ProcessClass*. We’re telling the report to “take a record”, and print it.

- ◆ Before trying to find an embed, try not to use one at all. This may sound strange, but the ABC templates and classes are quite functional. There may be a simple switch to turn on that will do what you are thinking. Test. This has an added benefit of teaching yourself what is going on under the hood.
- ◆ Don’t try to be an expert with all ABC templates overnight. Small “baby steps” in building a procedure will certainly pay off in the near future.
- ◆ Let the IDE find your embed! For example, the Formula Editor will insert an expression into selected embeds based on the type of Formula Class you designate.

If you are performing a complex filter process, enter a dummy expression in the Record Filter entry and using the Embeditor to search for the expression. Once found, the closest embed point will be revealed!

Other places to “ask the IDE” what embed points are needed are browse procedures, additional sort orders, conditional colors, icons, etc. This should take care of about 90% of your embed requirements!

- ◆ Use Code templates to reveal the object’s methods. Go to any embed point and **Insert** the following code template:



Next, choose the object that you want to affect. From the next drop list, you will see method calls that apply to *only* the chosen object. Based on the name only, make a list of possible candidates.

Finally, using the on-line Help, lookup the reference for your list of possible candidates.

- ◆ The same technique above also applies to ABC properties, except you would use the *SetABCProperty* template.

A final step to understanding the ABC templates and the use of embedded code is to study the default ABC generated code. This is presented in the following section.

Looking at Generated Source

The key to understanding any template based code generation is to study the source in its minimal form. From there, any additions can also be gradually studied and learned, along with the accompanying embed points.

On the following pages, we begin with a QuickStart application, using a simple file with two fields and a single key. The source code is then generated.

We begin our study with the main Program module. Legacy template users should pay special note to the many similarities of the ABC templates, as well as the differences.

The PROGRAM Module

The PROGRAM module defines global information, and the launch of the first procedure. The basic structure of a Clarion Object based program is the same as one that is procedure based. The first two statements are EQUATE statements which define constant values that the ABC Library requires. Following those are several INCLUDE statements. The INCLUDE statement tells the compiler to place the text in the named file into the program at the exact spot the INCLUDE statement. The ONCE attribute insures that the contents of this file is included only once, in the event that a duplicate INCLUDE is entered elsewhere in the application.

```
PROGRAM
  _ABCD11Mode_ EQUATE(0)
  _ABCLinkMode_ EQUATE(1)
  INCLUDE('ABERROR.INC'),ONCE
  INCLUDE('ABFILE.INC'),ONCE
  INCLUDE('ABUTIL.INC'),ONCE
  INCLUDE('ABWINDOW.INC'),ONCE
  INCLUDE('EQUATES.CLW'),ONCE
  INCLUDE('ERRORS.CLW'),ONCE
  INCLUDE('KEYCODES.CLW'),ONCE
```

The first four INCLUDE files (all starting with “AB” and ending with “.INC”) contain CLASS definitions for some of the ABC Library classes. The next three INCLUDE files (all ending with “.CLW”) contain a number of standard EQUATE statements used by the ABC Template generated code and ABC Library classes.

```
MAP
  MODULE('PHONEBC.CLW')
  DctInit PROCEDURE
  DctKill PROCEDURE
  END
  !- Application Global and Exported Procedure Definitions -----
  MODULE('PHONE001.CLW')
  Main PROCEDURE !Clarion 5 Quick Application
  END
END
```


The MAP structure contains two MODULE structures. The first declares two procedures *DctInit* and *DctKill* that are defined in the *PHONEBC.CLW* file. These two procedures are generated for you to properly initialize (and uninitialize) your data files for use by the ABC Library. The second MODULE structure simply names the application's first procedure to call (in this case, *Main*).

The PHONES file declaration follows. Nothing new here for the legacy users:

```
Phones FILE, DRIVER('TOPSPEED'), PRE(PHO), CREATE, BINDABLE, THREAD
KeyName KEY(PHO:Name), DUP, NOCASE
Record RECORD, PRE()
Name STRING(20)
Number STRING(20)
      END
END
```

The next two lines of code are your first OOP statements:

```
Access:Phones &FileManager
Relate:Phones &RelationManager
```

The **Access:Phones** statement declares a reference to a *FileManager* object, while the **Relate:Phones** statement declares a reference to a *RelationManager* object. These two references are initialized for you by the *DctInit* procedure, and uninitialized for you by the *DctKill* procedure. These are very important statements, because they define the manner in which you will address the data file in your ABC based code.

The next two lines of code declare a *GlobalErrors* object and an *INIMgr* object.

```
GlobalErrors ErrorClass
INIMgr INIClass
```

These objects handle all errors and your program's .INI file (if any), respectively. These objects are used extensively by the other ABC Library classes, so must be present (as you will shortly see).

```
GlobalRequest BYTE(0), THREAD
GlobalResponse BYTE(0), THREAD
VCRRequest LONG(0), THREAD
```

Following that are three Global variable declarations which the ABC Templates use to communicate between procedures. Notice that the global variables all have the *THREAD* attribute. *THREAD* is required since the ABC Templates generate an MDI application by default, which makes it necessary to have separate copies of global variables for each active thread (which is what the *THREAD* attribute does).

The global CODE section only has eight lines of code:

```
CODE
GlobalErrors.Init
INIMgr.Init('Phones.INI')
DctInit
Main
INIMgr.Update
INIMgr.Kill
DctKill
GlobalErrors.Kill
```

The first two statements call Init methods (remember, a procedure in a class is called a method). These are the constructor methods for the GlobalErrors and INIMgr objects.

You'll notice that the INIMgr.Init method takes a parameter. In the ABC Library, all object constructor methods are explicitly called and are named *Init*. There are several reasons for this. The Clarion language does support automatic object constructors (and destructors) and you are perfectly welcome to use them in any classes you write.

However, automatic constructors cannot receive parameters, and many of the ABC Library Init methods must receive parameters. Therefore, for consistency, all ABC object constructor methods are explicitly called and named *Init*. This has the added benefit of enhanced code readability, since you can explicitly see that a constructor is executing, whereas with automatic constructors you'd have to look at the CLASS declaration to see if there is one to execute or not.

The DctInit procedure call initializes the Access:Phones and Relate:Phones reference variables so the template generated code (and any embed code that you write) can refer to the data file methods using Access:Phones.Methodname or Relate:Phones.Methodname syntax. This gives you a consistent way to reference any file in an ABC Template generated program—each FILE will have corresponding Access: and Relate: objects.

The call to the Main procedure begins execution of the rest of your program for your user. Once the user returns from the Main procedure, the INIMgr, DctKill and GlobalErrors.Kill perform some necessary cleanup operations before the return to the operating system.

The Update Module

Let's examine typical source code that was generated for a simple update (Form) procedure.

```
MEMBER('Phones.clw') ! This is a MEMBER module
INCLUDE('ABRESIZE.INC')
INCLUDE('ABTOOLBA.INC')
INCLUDE('ABWINDOW.INC')
MAP
  INCLUDE('PHONE004.INC') !Local module procedure declarations
END
```

The first thing to notice is the **MEMBER** statement on the first line. This is a required statement telling the compiler which **PROGRAM** module this source file “belongs” to. It also marks the beginning of a **Module Data Section**—an area of source code where you can make data declarations which are visible to any procedure in the same source module, but not outside that module. For legacy users, this is nothing new.

The three **INCLUDE** files contain **CLASS** definitions for some of the ABC Library classes. Notice that the list of **INCLUDE** files here is different than the list at the global level. You only need to **INCLUDE** the class definitions that the compiler needs to know about to compile this single source code module. That's why the list of **INCLUDE** files will likely be a bit different from module to module.

Notice the **MAP** structure. By default, the ABC Templates generate local maps for you, that contains **INCLUDE** statements to bring in the prototypes of the procedures defined in the module and any procedures called from the module. This allows for more efficient compilation, because you'll only get a global re-compile of your code when you actually change some global data item, and not just by adding a new procedure to your application. In this case, there are no other procedures called from this module.

The **PROCEDURE** statement begins the **UpdatePhones** procedure (which also terminates the **Module Data Section**).

```
UpdatePhones PROCEDURE !Generated from procedure template - Window
CurrentTab STRING(80)
FilesOpened BYTE
ActionMessage CSTRING(40)
History::PHO:Record LIKE(PHO:RECORD),STATIC
```

Following the **PROCEDURE** statement are four declaration statements. The first two are common to most ABC Template generated procedures. They provide local flags used internally by the template-generated code. The **ActionMessage** and **History::PHO:Record** declarations are specific to a Form procedure. They declares a user message and a “save area” for use by the **Field History Key** (“ditto” key) functionality provided on the toolbar.

After a standard WINDOW structure comes the following object declarations:

```

ThisWindow CLASS(WindowManager)
Ask        PROCEDURE(),VIRTUAL
Init       PROCEDURE(),BYTE,PROC,VIRTUAL
Kill       PROCEDURE(),BYTE,PROC,VIRTUAL
          END
Toolbar    ToolbarClass
ToolBarForm ToolbarUpdateClass
Resizer    CLASS(WindowResizeClass)
Init       PROCEDURE(BYTE AppStrategy=AppStrategy:Resize,BYTE|
          SetWindowMinSize=False,BYTE SetWindowMaxSize=False)
          END

```

The last four are simple object declarations, which create the local objects, enable the user to use the toolbar, and resize the window at run-time. The interesting code here is the ThisWindow CLASS declaration. This CLASS structure declares an object derived from the WindowManager class in which the Ask, Init, and Kill methods of the parent class (WindowManager) are overridden locally. These are all VIRTUAL methods, which means that all the methods inherited from the WindowManager class will be able to call the overridden methods.

Following that comes all of the executable code in your procedure:

```

CODE
GlobalResponse = ThisWindow.Run()

```

That's right—one single, solitary statement! The call to ThisWindow.Run is the only executable code in your entire procedure! So, you ask, “Where's all the code that provides all the functionality I can obviously see happening when I run the program?” The answer is, “In the ABC Library!” or, at least most of it is! The good news is that all the standard code to operate any procedure is built in to the ABC Library, which makes your application's “footprint” very small, since all your procedures share the same set of common code which has been extensively debugged (and so, is not likely to introduce any bugs into your programs).

All the functionality that must be explicit to this one single procedure is generated for you in the overridden methods. In this procedure's case, there are only four methods that needed to be overridden. Depending on the functionality you request in the procedure, the ABC Templates will override different methods, as needed. You also have embed points available in every method it is possible to override, so you can easily “force” the templates to override any method for which you need slightly different functionality by simply adding your own code into those embed points (using the Embeditor in the Application Generator).

OK, so let's look at the overridden methods for this procedure.

```

ThisWindow.Ask PROCEDURE
CODE
CASE SELF.Request
OF InsertRecord
    ActionMessage = 'Adding a Phones Record'
OF ChangeRecord
    ActionMessage = 'Changing a Phones Record'
END
QuickWindow{Prop:Text} = ActionMessage
PARENT.Ask()

```

The really interesting line of code in the `ThisWindow.Ask PROCEDURE` is last. The last statement, `PARENT.Ask`, calls the parent method that this method has overridden to execute its standard functionality. The `PARENT` keyword is very powerful, because it allows an overridden method in a derived class to call upon the method it replaces to “do its thing” allowing the overridden method to incrementally extend the parent method's functionality.

```

ThisWindow.Init PROCEDURE()
ReturnValue    BYTE,AUTO
CODE
SELF.Request = GlobalRequest
IF PARENT.Init() THEN RETURN Level:Notify.
SELF.FirstField = ?PHO:Name:Prompt
SELF.VCRRequest &= VCRRequest
SELF.Errors &= GlobalErrors
CLEAR(GlobalRequest)
CLEAR(GlobalResponse)
SELF.AddItem(ToolBar)
SELF.AddUpdateFile(Access:Phones)
SELF.HistoryKey = 734
SELF.AddHistoryFile(PHO:Record,History::PHO:Record)
SELF.AddHistoryField(?PHO:Name,1)
SELF.AddHistoryField(?PHO:Number,2)
SELF.AddItem(?Cancel,RequestCancelled)
Relate:Phones.Open
FilesOpened = True
SELF.Primary &= Relate:Phones
SELF.OkControl = ?OK
IF SELF.PrimeUpdate() THEN RETURN Level:Notify.
OPEN(QuickWindow)
SELF.Opened=True
Resizer.Init(AppStrategy:Surface,Resize:SetMinSize)
SELF.AddItem(Resizer)
Resizer.AutoTransparent=True
ToolBarForm.HelpButton=?Help
SELF.AddItem(ToolBarForm)
SELF.SetAlerts()
RETURN ReturnValue

```

There are several interesting lines of code in the `ThisWindow.Init` PROCEDURE. This is the `ThisWindow` object's constructor method, so all the code in it performs the initialization tasks specifically required by the `UpdatePhones` procedure.

The first statement, **`SELF.Request = GlobalRequest`**, retrieves the global variable's value and places it in the `SELF.Request` property. `SELF` is another powerful Clarion Object keyword, which always means "the current object" or "me." `SELF` is the object prefix, which allows class methods to be written generically to refer to whichever object instance of a class is currently executing.

The second statement calls the `PARENT.Init()` method (the parent method's code to perform all its standard functions) before the rest of the procedure-specific initialization code executes.

Following that are a number of statements which initialize various necessary properties. The `Relate:Phones.Open` statement opens the `Phones` data file for processing, and if there were any related child files needed for Referential Integrity processing in this procedure, it would also open them (there aren't, in this case).

```
ThisWindow.Kill PROCEDURE()  
CODE  
IF PARENT.Kill() THEN RETURN Level:Notify.  
IF FilesOpened  
    Relate:Phones.Close  
END
```

In addition to calling the `PARENT.Kill()` method to perform all the standard closedown functionality (like closing the window), `ThisWindow.Kill` closes all the files opened in the procedure, then sets the `GlobalResponse` variable.

5 - GENERAL APPLICATION TECHNIQUES

Introduction

This chapter focuses on the features of the ABC templates as they apply to an entire application. Some are provided as an explanation to the Legacy template user to allow them a smooth transition into using the ABC template set. Even non Legacy users will find many useful areas of information to help clarify much of the ABC activity that occurs within an application.

Other sections in this chapter focus on techniques that can be applied to any or all ABC template based procedures.

Although there are *many* techniques to explore in the ABC templates, we will only focus on those that differ significantly from legacy implementations.

NOTE: The code demonstrated and shown in the following examples are available for download at the TopSpeed web site at the same location (URL) where this document was downloaded.

Error Handling (The ABC Error Class)

Overview

The ABC templates have incorporated powerful Error Class objects throughout their design. This gives the developer added confidence and flexibility when using these templates.

To the legacy user, it is important to note a few key implementation features:

- ◆ The ABC Error Class uses a more robust way of trapping errors in an environment where multiple files are accessed. As future file driver enhancements are implemented, the Error Class allows for a more flexible and configurable interface.
- ◆ In legacy programs, a lot of your code was bloated with frequent error checking conditions (IF TheWorldEnds THEN GetYourWings, etc.). The ABC Error Class handles exceptions in a more elegant manner.

In addition, the ABC Error Class is designed with the following features in mind:

- ◆ Clean interaction with the end-user
- ◆ Perform quick and clean corrective action that is required.
- ◆ Allow customizing the error screens, messages, and actions taken.
- ◆ Uses minimum resources to allow clean operation in a potentially hostile environment.

Let's explore a few common techniques using the ABC Error Class.

Altering the Text of an ABC Error Message

1. Locate the file **aberror.trn** found in the \LIBSRC folder of Clarion. This file contains is the data structure that drives the error process.
2. Search for the error text that is closest to the message you want to replace.
Any words starting with a % will match 1 or more words in the string you have seen. For example if you want to replace "Fred key file is invalid. Key must be rebuilt" then the matching one is "%File key file is invalid. Key must be rebuilt". All the words other than the ones preceded by a % may be edited with impunity. The words proceeded by % are macros, they will be expanded out when the user sees them. The full list of potentially available macros is in the aberror.clw file before the SubString procedure definition. You don't have to use a macro value just because the existing text does. Next time you compile you application all the changes will be in place.

Features and benefits:

- One edit applies to all usages in all applications
- No runtime overhead
- No coding
- Particularly suitable for language translations or 'buzzword' translations (i.e., replace "file" with "table"). Can also be used for sheer user friendliness.

Caveats:

- Changes need to be merged with each new TopSpeed release.
- PSTRINGS are limited to 255 characters.
- The title displayed on the window is the line preceding the full error description. This too can be edited although macros are presently *not* available
- Macros are presently case sensitive. This will probably be changed in a future release.

Altering the Severity of an Error Message

The concept and implementation of this looks very simple, as the line above the title is the severity level. You can simply edit that to a new value.

```
DefaultErrors GROUP
Number USHORT(41)
      USHORT(Msg:RebuildKey)           !Errorcode
      BYTE(Level:Notify)              !Severity
      PSTRING('Invalid Key')         !Title
      PSTRING('%File key file is invalid. Key must be rebuilt.') !Message
```

For example, suppose you wanted ‘key building’ to be done only by the DBA. You could change the error text (see above) to “Fred key file is invalid, please call the DBA”, then change the severity level to Level:Fatal (from Level:Notify). The user will now be dropped out of the program rather than the keys rebuilding.

However, things aren’t always quite that easy. Going from Notify to Fatal always works, but the harder ones are the errors that lessen in severity.

A Level:User (which asks a yes/no question) can be softened to a Level:Benign provided you want the default response to be ‘yes’. For example, to make cancels happen without a confirm (globally) change the level of Msg:ConfirmCancel from Level:User to Level:Benign.

In other cases, the Level:User severity is really asking the user if he wants to iterate (e.g. - try again), so hard-wiring the answer to yes can easily leave him locked in an endless loop.

A Level:Notify can safely be softened to a Level:Benign although you risk leaving the user clueless, but can also avoid irritation. For example, to turn off the ‘no records to process’ simply find Msg:NoRecords and turn the level from Notify to Benign.

The riskiest case of all is when you try to soften a Level:Fatal to a Notify (or even Benign!), but there are some applicable cases. (i.e., keep going even with some files missing). A severity decrease of this magnitude usually involves some serious extra work in the ABC embeds trying to keep the ailing system alive.

Adding and using your own error messages

There are two techniques recommended when adding your own custom error messages to the ABC templates. One is easier to implement and is the safest, while the other is a little more efficient, but requires a little more effort.

Both techniques begin in the same manner. You need to create an equate for the new message number, avoiding the TopSpeed defaults. Since the Id is a defined USHORT, you can probably start at 16K and be pretty safe. If you are creating just one new message a simple EQUATE statement will suffice, If there are many, an ITEMIZE is easier.

There are two ways of creating your custom error equates.

Technique #1

1. Edit the **aberror.inc** file (located in the \LIBSRC folder of Clarion), putting the new equate after the ones predefined.

```
UpdateIllegal          EQUATE
UseClosedFile         EQUATE
ViewOpenFailed       EQUATE
QBEColumnNotSupported EQUATE
YourCustomErrorHere  EQUATE
END
```

2. Go to the first field of the group in the **aberror.trn** file and increment the field by the number of new messages you are adding. Then go to the end of the group and append the messages using the format as described before.

Features and benefits of the first technique include its simplicity to implement and how easy the changes proliferate to all of your applications. The only caveat is the need to merge the two files when upgrading to a new TopSpeed release.

Technique #2

1. Create a **myerror.inc** file and add your equates there. Arrange for it to be included in any application that requires the new error messages.
2. Create a **myerror.trn** file containing a group with a leading USHORT (the number of elements) and then a group structure as before. The group should have the static attribute as shown:

```
Myerrors  GROUP,STATIC
USHORT(1)
USHORT(Msg:Boo)
BYTE(Level:Notify)
PSTRING('Surprise')
PSTRING('Boo')
END
```

3. In the Application Generator, **press** the **Global** button, press the embeds button, and locate the *Global objects, Global errors, Init method* embed. In the *Data section* source embed **type**:

```
INCLUDE('myerror.trn')
```

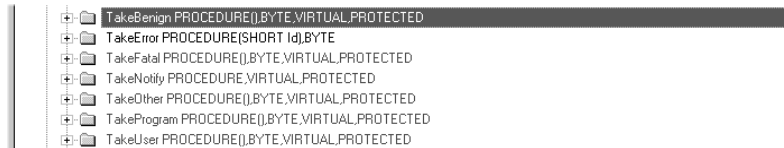
4. In the *Code section*, *After the parent call*, **type**

```
SELF.AddErrors(MyErrors) ! MyErrors is the name of your group
SELF.Throw(Msg:Boo) ! Usually this line is somewhere else in your program
```

Although the second technique is a little harder work, it does not require editing of TopSpeed files. Furthermore, errors can be applied on an application by application basis.

Changing the Presentation of Error Messages

It is easy to tailor the presentation of messages to suit your own style provided you are using Clarion 5. In the global errors embed section, you will see a number of methods all starting with *Take*:



You can override the procedure that has the severity level you wish to alter the appearance of. Caution is advised in some cases (i.e., If the system has just discovered the disk is full, now is not a good time to display a 1024x768 24bit BMP file ...).

Let's modify the TakeFatal Procedure, adding a warning dialog that appears before any fatal error:

1. Insert a source point *before* the Parent call in TakeFatal. Type the following:

```
MESSAGE('You are in deep trouble. Please make a detailed note of the message that follows this and then call 1800 999 9999 with your credit card handy', 'Clang' , ICON:Exclamation, Button:OK, BUTTON:OK, 0)
```

The actual work is still done by the parent call. The very astute may even notice that this MESSAGE could be implemented by giving yourself a Notify level error message and then *Throwing* it as part of the fatal level code.

For the less critical (and presumably more common) error messages it is possible to go to town and use your own window structure. The window structure is first added to the appropriate data section. Next, the detection code is added *before* the parent call and should be terminated by a return of *appropriate level*. The easiest way to see the correct level to return is simply to look at the source code located in **aberror.clw** (don't panic, the longest take method is 6 lines long!)

Altering an Error Message for a Single Procedure (only)

The last example tackled changing an error message globally. The technique described here changes the meaning of an error message for a limited period of time.

1. Construct an ErrorBlock as if you were adding a fresh message. The difference in this case is that the *Msg:* number you use should be the same as the one you want to override.

2. Place this error block in the data section of the procedure you want the error to ‘live’ for (you don’t actually need the static attribute although the compiler generates better code if you use it).
3. In the ThisWindow, Run (no parameters) method, all the following code just before the parent call:

```
GlobalErrors.AddErrors(MyErrors)
```

Looks odd, but the Error Manager operates a LastInFirstOut (LIFO) stack of errors, so if you add an error with a duplicate error number it ‘takes over’ until such time as it is removed.

4. Before leaving the procedure, add the following source:

```
GlobalErrors.RemoveErrors(MyErrors)
```

This removes the new copy of your error and restores the original data. Note also we have used *.Run* rather than *.Init/.Kill*, this is so that the error manager is active throughout the whole life of the procedure. This also illustrates that there *is* an embed point ‘before/after’ the run statement in a procedure, you just have to think a little bit to find it!

One caveat to remember here is that the *scoping* of errors is dynamic (e.g., the new errors applied from the time the *AddErrors* is executed to the time the *RemoveErrors* is executed may or may not be the same as the lexical scope of the procedure). Put another way, if you put these embeds into a browse procedure, and that browse calls an update the errors will *still* be substituted during the update process.

Note: The *.Run* method is very powerful. It really does let you “bite the cherry” before the procedure springs into life. However, you need to remember that you are executing code *before* the init method or after the kill. You cannot assume the WindowManager data elements are set up properly.

Error Checking While in Stand-Alone Mode

Our last example is an interesting, but surprisingly easy task. We produce a program that can run *on its own* without an operator pressing buttons. (Obviously this assumes you created an application that does something without a user being present, like scheduled archiving, data mining etc.). The main thing to decide is precisely which errors you want to fly blind with and which ones are so horrible you *want* the system to grind to a halt and wait for help. For our example, we will assume that all notify messages are trivial and that all user confirmations can be answered in the affirmative.

Of course, we could tackle that problem simply by modifying all of the error message severities. There are two problems with this. First, it is tedious. Second, it is a static implementation (happens all the time for all applications). Let me underline the second defect by further specifying that our program has to be able to fly blind *only when asked* and that normally it is supposed to interact with the user in the normal manner.

Let's start by providing a byte variable in the global data section called **Blind**. This is our flag used to detect stand-alone operation.

Next, in the global embeds, global objects, global errors, TakeNotify and before the parent call we insert an embed :

```
IF Blind THEN RETURN Level:Benign.
```

Next, we go into TakeUser and add (again before the parent call):

```
IF Blind THEN RETURN Level:Benign.
```

The RETURN is actually performing two different tasks. First, it is short-stopping the parent call (the parent call performs the user interaction). Second, it is returning a default value for the user response (Benign means OK, keep going).

Earlier in this section, we said that always returning Benign from a TakeUser could cause the machine to go into an infinite loop (where it keeps retrying something it is simply unable to complete). Our stand-alone mode solution can suffer from precisely this problem.

One possible solution is to put some loop tracking code into the TakeUser function to attempt to spot loops. Such code is always going to be heuristic, and you'll have to decide a heuristic that suits you.

Here is a simple one. If more than 100 TakeUsers are executed inside one minute then a no is returned:

```
!TakeUser : DataSection
LastTime LONG(07FFFFFFFH),STATIC
Count LONG,STATIC
C LONG,AUTO

!Takeuser : before the parent call (instead of previous code):
C = CLOCK()
IF C < LastTime THEN LastTime = C . ! Watch for midnight

IF C-LastTime > 6000 THEN
  LastTime = C
  Count = 0
END

IF Blind
  Count += 1
  RETURN CHOOSE(Count>100?Level:Cancel:Level:Benign)
END
```

Note the use of the **STATIC** attribute. This is because we need these values to persist between invocations of the TakeUser procedure. Note, too, the high initial value for last time, this is so that the code to trap crossings over midnight will also trap the first *TakeUser* usage.

Error Class Summary

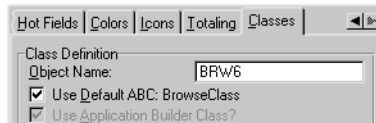
We have moved from simple textual modifications of the TRN file, through more advanced modifications, and on to some significant alterations to base class functionality, through use of the global embeds button. The latter is a new feature in Clarion 5, supported only in the ABC templates.

Naming Conventions

Many developers (on occasion) are guilty of spending insufficient time with their database design, particularly with the labels (names) chosen for their data elements. Often, these errors may lead to unwanted compile errors and increased development time.

The legacy template user needs to be more aware of certain reserved names that are used by the legacy templates. To avoid name conflicts between generated labels and hand coded labels, a double colon convention is used throughout the generated source (i.e., Menu::FileMenu).

Although the same is generally true with ABC templates, more attention has been given to additional flexibility and control over the generated naming conventions.



The window above references *BRW6*, the prefix used in the selected Browse Class. If this prefix happens to clash with a user-defined prefix, the ABC template user has the flexibility to override the name to a value that is more meaningful (Example: BrowseClass).

Another area of importance is how class methods and properties are labeled. For example, a “Try” method in the File Manager Class is consistent in purpose and scope with a “Try” method of the INI Manager Class.

The Local Map

A standard feature that is built in to the ABC template design is the generation of local MAP structures in each module created by the Application Generator. As a result, the global MAP structure is reduced in size and easier to read:

```
EXAMPLE:
!Global Map
MAP
  MODULE('POPUPBC.CLW')
DctInit  PROCEDURE
DctKill  PROCEDURE
  END
!--- Application Global and Exported Procedure Definitions
  MODULE('POPUP001.CLW')
Main    PROCEDURE  !
  END
END
```

The main benefit to this built-in ABC template feature is a significant decrease in compile time, more noticeable with larger projects.

There is no trick, no technique to mention here. Local MAP implementation is automatic.

INI File Management with ABC

Experienced legacy programmers have long understood the power of using an initialization (or INI) file to control many aspects of an application's behavior. Primarily, the use of INI files is centered on two core language statements. The GETINI statement is used to read information from a selected INI file, and the PUTINI statement performs the write operation

INI Manager Overview

The ABC INI Manager is an improved set of methods which allow more power to a developer's INI file processing. There are currently seven methods encapsulated in the INI Base Class. Besides the standard **Init** and **Kill**, there are six others:

Fetch and TryFetch

Gets or returns values from the INI file. At first, you may think of Fetch as a direct replacement of GETINI, but this powerful method also allows you to reference a window structure, returning Maximize, XPos, YPos, Height, and Width.

If the specified section and entry do not exist, the TryFetch method returns an empty string. This allows you to check the return value and take appropriate action when the INI file entry is missing.

FetchField and TryFetchField

Here is where the old language statement GETINI parts ways with the INI Manager. This method supports returning *specific* comma delimited values assigned to a particular entry.

FetchQueue

Although a little more housekeeping is needed in the INI file, this method adds a series of values from the INI file into the specified fields in the specified queue.

Update

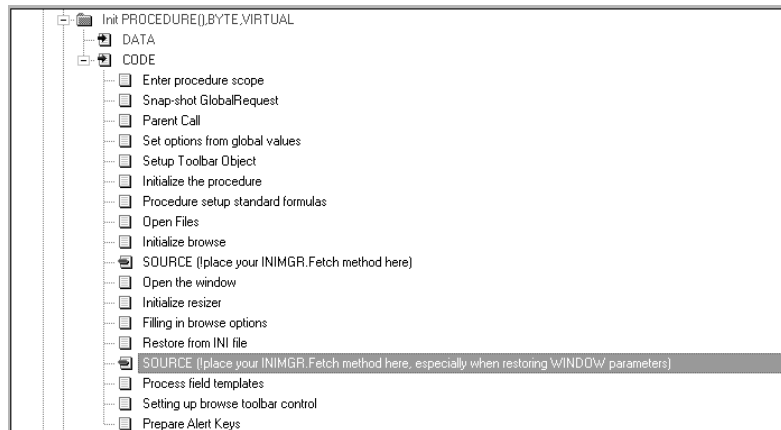
Finally the Update method writes entries to the INI file. If the specified value is null (""), the existing entry is deleted. Based on the parameter(s) referenced, Update writes a single value specified by section and entry, or WINDOW position and size attributes.

The *Application Handbook* and on-line help contains more information about the INI Manager methods, and examples for implementation.

Standard Implementation

In general, you can easily replace your GETINI statements in legacy embeds to the Fetch or TryFetch method of your choice. The ABC templates generate an object named INIMgr automatically, so prepend your methods with that name (i.e., INIMgr.Update).

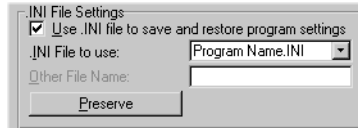
Normally, you will use any ABC embed prior to opening the window to set up INI values. For the WINDOW attributes, use an ABC embed point located just after opening a window:



Likewise, you can replace your legacy PUTINI statements with the Update method and appropriate parameters. You would normally use an ABC embed located just outside of the main procedure's ACCEPT loop, and prior to leaving the procedure's scope.

INI Environment Support - The Global Preservation Society

A powerful new feature of the INI Manager is its ability to save the condition of a program's global data upon exiting the program, and restoring the global values on the next program load. Its implementation is simple, and involves a single button (**P**reserve) in the Application Generator's Global IDE:



Add any number of global variables that your program needs to preserve.

Stay Tuned

Another benefit to using the INI Manager may be seen in the near future. There are concrete plans for the implementation of Class libraries that write to the system registry, or to web based cookies. Because these new classes are designed with the INI compatibility in mind, the ABC template user can virtually leave his code intact, and replace the INI Manager base class with one of these new classes which will be available soon. Stay tuned!

ABC Based Toolbars

Here are a couple of key features to consider when using toolbar controls in the ABC template environment:

Unlimited Browsers

A popular feature with all TopSpeed templates has been the automatic implementation of toolbar buttons which directly affect an active browse control. Standard toolbar buttons are available for data manipulation and record selection.

For the legacy template user, this feature was limited to the primary browse control of a window; if you had multiple browse controls on different tabs, you would have to hand code the focus shift from one browse control to another.

The ABC templates automatically detect an active browse control, and pass the appropriate control to the toolbar buttons.

Sitting on the DOCKABLE way

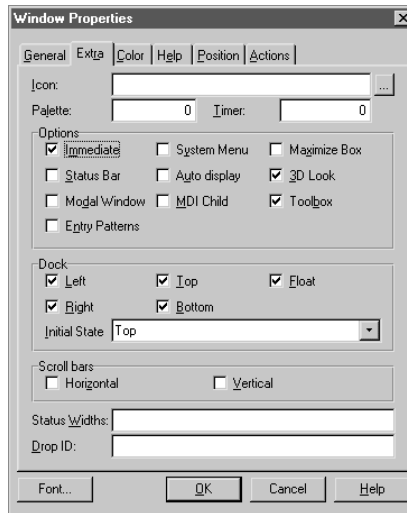
“Watching the haand-code roll away....”

You can all cease your groaning at this time.

A powerful new feature only available through the ABC templates involve the use of dockable toolbar windows.

To implement the dockable effect, you need to consider several rules:

- ◆ As you define your window that you will wish to dock, set the following attributes (found in the Extra tab control):



Note the *Dock* and *Toolbox* sections. Through the IDE, you can control the allowable areas to dock to, and also designate an *Initial State*.

- ◆ Add the *WindowResize* Extension template to the dockable window. This will allow you to select a correct resize strategy for the controls contained on the window.
- ◆ Finally, it is a good idea to add the following Snap properties to your window through the following embeds:

```
!LocalObjects.ThisWindow WindowManager.Init after Open the Window
window{PROP:snapwidth, 1} = 20 ! Vertical size i.e., when made tall
window{PROP:snapheight, 1} = 100
```

```
window{PROP:snapwidth, 2} = 100 ! Horizontal size i.e., when made wide
window{PROP:snapheight, 2} = 20
```

```
window{PROP:snapwidth, 3} = 50 ! normal
window{PROP:snapheight, 3} = 50
```

The Snap properties allow specific width and height parameters to help control and manage the dockable window sizes.

The power of the ABC Translator Class

The ABC Translator Class provides very fast runtime translation of user interface text. You can deploy a single application that serves all your customers, regardless of their language preference. Use the TranslatorClass to display several different user interface languages based on end user input or some other runtime criteria, such as INI file or control file contents. There are three techniques of translator implementation:

1. Locally defined data structures.
2. Use an INI to store and configure your text.
3. Alternatively, you can use the Clarion translation files (*.TRN) to implement a single non-English user interface at compile time.

Implementation

Earlier in this chapter, we discussed at length the implementation and use of the ABC Error Class. You may be pleasantly surprised that there are striking similarities in the implementation of the Translator Class.

- ◆ The Translator Class uses the AddTranslation method, referencing a predefined GROUP of *translator pairs* as shown:

EXAMPLE:

```
MyTranslations  GROUP      !declare local translations
Items          USHORT(4)    !4 translations pairs
                PSTRING('Hello') ! item 1 text (macro)
                PSTRING('Hola')  ! item 1 replacement text
                PSTRING('&Where') ! item 2 text
                PSTRING('&Donde') ! item 2 replacement text
                PSTRING('&Dog')    ! item 3 text
                PSTRING('&Perro') ! item 3 replacement text
                PSTRING('Goodbye')! item 4 text
                PSTRING(' Adios ')! item 4 replacement text
                END
```

Of course, declaring an entire lexicon in local data would be tedious and cluttered.

- ◆ For large pairs of translations, an ASCII file that holds the translation pairs is more preferable. These files normally can be identified with a .TRN extension, but you are not limited to that naming convention.

The implementation of the TRN file is accomplished by executing the *ExtractText* method as shown:

```
Translator.ExtractText='.\\MyApp.trn'
```

The translator object scans your active window and controls for matching text of the first item of the defined pair. When you apply the *TranslateWindow* method, the text of the second item of the defined pair is automatically applied.

A slightly different variation of the Translator Class implementation is discussed in the next section.

Popup Menus

Popup menus in the ABC templates are almost overlooked and an after thought due to the limited control required by the developer to implement them. However, much power and control of popup menus is available through the use of the ABC templates, objects, and selected embeds.

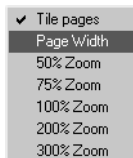
Default Popup Menu Configuration in ABC

The ABC template generated code does not reference the *PopupClass* objects encapsulated within the *ASCIIViewerClass*, *BrowseClass*, and *PrintPreviewClass*. The extent of user control is check boxes which activate or deactivate specific menu items related to the specific template you are using.

In the *Browse Class*, popup menu options are available for standard data manipulation (Insert, Change, Delete), record selection (Select), and query options.



The *Print Preview* procedure (*Previewer Class*) add popup menu support for a variety of Zoom and Tile options.



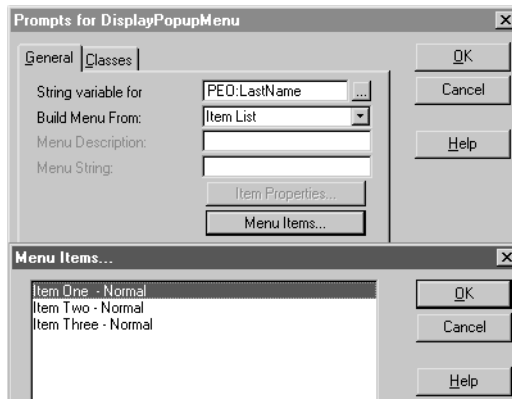
The ASCII Viewer procedure offers popup support for printing, find, and line selection.



The ABC Popup Menu Code Template

The ABC Templates declare a local `PopupClass` class and object for each instance of a new `Popup` code template (named *DisplayPopupMenu* in the Template Registry).

The naming convention used by the ABC template is *PopupMgr#* where # is the instance number of the `Popup` code template. The templates provide the derived class so you can use the `Popup` code template Classes tab to easily modify the popup menu behavior on an instance-by-instance basis. However, you also have an extensive array of options available in the code template interface to allow your specific style of popup menu implementation and control.



Creating Your Own Popup

Try this example with popup menus to help you get more familiar with “the ABC way”.

Create a custom popup menu in a form procedure that appears when the user RIGHT-CLICKS over a selected entry field.

1. Declare your custom popup object

In this first step, we are declaring a new object, a popup object, that will later inherit specific properties and methods. Take it another step. You declare a string variable, and later assign it a value. An object's values are the properties and methods contained within the base class we are deriving the object from.

So, for now, we are simply declaring an object that we will use. The `&` indicates a reference to the `PopupClass`, meaning "the memory that my object will need is the same as the `PopupClass` allocation". Use the *Local Data, Other Declarations* embed point and enter the following declaration:

```
MyPopup &PopupClass
```

2. Instantiate (activate) the new popup object.

Now its time to bring our object to life. Use the *ThisWindow(Window Manager) Init PROCEDURE(Method)*, *Initialize the procedure* embed, and enter the following:

```
MyPopup &= New PopupClass  
MyPopup.Init()
```

The first statement activates our object, and the `Init` method handles any initialization necessary for the object. In the first statement, parentheses are optional after the `NEW` clause.

3. Add items to your popup to call a procedure or post an event using the following methods:

Two other methods are demonstrated here. One method allows us to mimic the behavior of a button that we have hidden. Whatever the button is programmed to do, our popup menu item will also do. The second method simply allows us to post an event that we can trap (detect) in another embed. Use the *ThisWindow(Window Manager) Init PROCEDURE(Method)*, *Open the window* embed, and enter the following:

```
MyPopup.AddItemMimic('Call MyProc',?MyProc)  
MyPopup.AddItemEvent('Maximize',Event:Maximize)
```

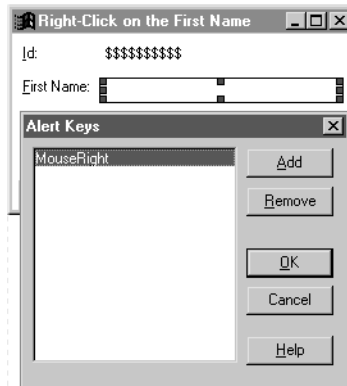
To add an icon to the popup menu, use the following text:

```
MyPopup.AddItem([' & PROP:Icon & '(computer.ico)]MenuChoice')
```

After adding our menu items, let's write some code to detect for the mouse `RIGHT-CLICK`, normally the standard action for popup display.

4. Detect that the user RIGHTCLICKED over your selected field, and display the popup.

First, alert the mouse right key by right-clicking over the desired control, and add the Mouse right key in the IDE interface:



Next, add the following source code to the *ControlEvents, ?PEO:Firstname,Alert Key* embed point:

```
If KeyCode() = MouseRight
    MyPopup.Ask()
END
```

Of course, additional code may be required, depending on the menu action that you wish to perform, or event that you may need to trap.

5. When the window is closed, kill (deactivate) your custom popup object.

This is another standard when you instantiate your own objects. As important as it is to Initialize (Init), we use the appropriate embed point, *ThisWindow Window Manager, Kill PROCEDURE()*, Leave the procedure scope to free memory and effectively kill our object:

```
MyPopup.Kill
```

Popup Text Translation

The ABC libraries provide support for custom text configuration using the Translator Class. The configuration rules are similar as in other static window controls. However, because a popup menu is not a static window element, you must remember to invoke the *TranslateString* Translator Class methods *before* you have created the elements for your popup menu (or invoked your Popup Class methods). Here's a small example:

```
MyTranslations GROUP           !declare local translations of buttons
Items           USHORT(1)      !6 translations pairs
                PSTRING('Maximize') ! item 1 text
                PSTRING('Biggie')  ! item 1 replacement text
                END

Translator      TranslatorClass !declare Translator object
strvar         string('Maximize')
```

We wish to translate the text of a popup menu item, *Maximize*. After defining the translator pair (translate *Maximize* to *Biggie*), we define a local string (*strvar*) with the matching popup default value.

Next, initialization and kill is the same as in other Translator Object implementation, but the method for popup menus is slightly different:

```
Translator.Init           !initialize Translator object
Translator.AddTranslation(MyTranslations) ! add default translation pair
strvar = Translator.TranslateString(strvar)
```

The third line of code above is significant. Instead of using the *TranslateWindow* method, we opt for the *TranslateString*, since a popup is not considered to be part of a static window structure.

Our popup menu now displays as follows:



6 - DATA AND FILE ACCESS TECHNIQUES

Data and File Access

Introduction

In this chapter we'll review the ABC Data and File Handling capabilities, and share some tips and techniques to assist you in migrating File Handling and/or Data Handling code you have hand-coded (embedded) within your Legacy applications. For those applications that you have created without the use of Legacy File or Data Handling hand-coded (embedded) code, the Application Converter tool is probably all you need to migrate your application to ABC.

The easiest way to become familiar with the ABC library is to read the Application Handbook--specifically the overview section for each Class. In addition to the Application Handbook, Clarion also has a Class Viewer to show you the ABC Library properties and methods in a tree view. On any Classes tab, just press the button labeled "**Application Builder Class Viewer**" to view the ABC Library structure.

Additionally the ABC Template set contains two Code Templates, which help you use the ABC Library: *CallABCMethod* and *SetABCProperty*. These were created to "walk you through" writing ABC Library code in any executable code embed point. These two code templates will write your method calls and object property assignment for you!

Although you may have already determined that the Application Converter tool will automatically handle the conversion of your application, reviewing the previously referenced material along with the information that follows, will give you insight regarding the significant improvements made to Clarion and techniques when using the ABC Classes.

Chapter Organization

Overview Presents New (not available in Legacy) ABC File Handling capability.

ABC FileManager, RelationManager, ErrorClass

Covers a high-level overview of the **main ABC Classes associated with File and Data handling.**

Files, ABCs, and Legacy Applications

Identifies specific Legacy hand-coded (embedded) code that **you must manually migrate to conform to ABC Library.**

New ABC File Handling Capability

Identifies and explains new functionality available with the ABC Library.

Advanced References

Supplies additional information and reference material for the new functionality. Topics like how do I implement Referential Integrity within Clarion's IDE to how do I find conceptual and overview material about "Lazy Open" are covered in this section.

Overview

Along with and primarily because of the software development options available with the OOP paradigm, Clarion's ABC classes make your development job even easier than it was using the Legacy tools.

As an example, Clarion now relieves you of the tedious and error prone tasks of explicitly handling many of the code steps related to the enforcement of Referential Integrity (RI). Additionally, the ABC File methods automatically perform all the error checking, data validation, and data auto-incrementing (*see following examples*). These are just a few of the many innovative time and resource saving features now available with the Clarion ABC Libraries.

New ABC File Handling Capability

- New lazy open capability
- Support for on-server Referential Integrity
- Improved record buffer integrity during RI updates or deletes.
- All priming / validation done in one place (localized in an object)
- Improved integrity when using field validation.
- Support for INLIST validation.
- Improved recovery from a sequential read of a locked record
- Handling of the "between procedure" alias problem

ABC FileManager, RelationManager, BufferedPairsClass, ErrorClass

In this section we'll briefly cover some of the ABC Classes. See the Application Handbook for complete information about the following classes.

FileManager

The FileManager relies on the ErrorClass for most of its error handling. Therefore, if your program instantiates the FileManager it must also instantiate the ErrorClass. See *Error Class* for more information.

Perhaps more significantly, the FileManager serves as the foundation or “errand boy” of the RelationManager. If your program instantiates the RelationManager it must also instantiate the FileManager. See *Relation Manager Class* for more information.

RelationManager

The RelationManager class declares a relation manager object that does the following:

- Consistently and flexibly defines relationships between files—the relationships need not be defined in a data dictionary; they may be defined directly (dynamically) to the relation manager object.
- Reliably enforces discrete specified levels of referential integrity (RI) constraints between the related files—the RI constraints need not be defined in a data dictionary; they may be defined directly (dynamically) to the relation manager object.
- Conveniently forwards appropriate file commands to related files—for example, when a relation manager object opens its primary file, it also opens any related files.

The RelationManager class provides “setup” methods that let you describe the file relationships, their linking fields, and their associated RI constraints; plus other methods to perform the cascadable or constrainable database operations such as open, change, delete, and close.

Relationship to Other Application Builder Classes

FileManager and BufferedPairsClass

The RelationManager relies on both the FileManager and the BufferedPairsClass to do much of its work. Therefore, if your program instantiates the RelationManager it must also instantiate the FileManager and the BufferedPairsClass. Much of this is automatic when you INCLUDE the RelationManager header (ABFILE.INC) in your program's data section. See the *Conceptual Example* and see *File Manager Class* and *Field Pairs Classes* for more information.

ViewManager

Perhaps more significantly, the RelationManager serves as the foundation or “errand boy” of the ViewManager. If your program instantiates the ViewManager it must also instantiate the RelationManager. See *View Manager Class* for more information.

BufferedPairsClass

The BufferedPairsClass is a FieldPairs class with a third buffer area (a “save” area). The BufferedPairsClass can compare the save area with the primary buffers, and can restore data from the save area to the primary buffers (to implement a standard “cancel” operation).

The BufferedPairsClass lets you move data between field pairs, and lets you compare the field pairs to detect whether any changes occurred since the last operation.

This class provides methods that let you identify or “set up” the targeted field pairs.

Note: The paired fields need not be contiguous in memory, nor do they need to be part of a structure. You can build a virtual structure simply by adding a series of otherwise unrelated fields to a BufferedPairsClass object. The BufferedPairsClass methods then operate on this virtual structure.

Once the field pairs are identified, you call a single method to move all the fields in one direction (left to right), and others single methods to move all the fields in the other directions (right to left, left to buffer, etc.). You simply have to remember which entity (set of fields) you described as “left” and which entity you described as “right.” Other methods compares the sets of fields and return a value to indicate whether or not they are equivalent.

Relationship to Other Application Builder Classes

The BufferedPairsClass is derived from the FieldPairsClass. The BrowseClass, ViewManager, and RelationManager use the FieldPairsClass and BufferedPairsClass to accomplish various tasks.

FieldPairsClass

In database oriented programs there are some fundamental operations that occur over and over again. Among these repetitive operations is the saving

and restoring of field values, and comparing current field values against previous values.

The ABC Library provides two classes (FieldPairsClass and BufferedPairsClass) that supply this basic buffer management. These classes are completely generic so that they may apply to any pairs of fields, regardless of the fields' origins.

Tip: The fundamental benefit of these classes is their generality; that is, they let you *move* data between pairs of structures such as FILE or QUEUE buffers, and *compare* the data, without knowing in advance what the buffer structures look like or, for that matter, without requiring that the fields even reside in conventional buffer structures.

In some ways the FieldPairsClass is similar to Clarion's deep assignment operator (`:=`: see the *Language Reference* for a description of this operator). However, the FieldPairsClass has the following advantages over deep assignment:

- Field pair labels need not be an exact match
- Field pairs are not limited to GROUPs, RECORDs, and QUEUEs
- Field pairs are not restricted to a single source and a single destination
- You can compare the sets of fields for equivalence
- You can mimic a data structure where no structure exists

The FieldPairsClass has the disadvantage of not handling arrays (because the FieldPairsClass relies on the ANY data type which only accepts references to simple data types). See the *Language Reference* for more information on the ANY data type.

FieldPairsClass Concepts

The FieldPairsClass lets you move data between field pairs, and lets you compare the field pairs to detect whether any changes occurred since the last operation.

This class provides methods that let you identify or “set up” the targeted field pairs.

Once the field pairs are identified, you call a single method to move all the fields in one direction (left to right), and another method to move all the fields in the other direction (right to left). You simply have to remember

which entity (set of fields) you described as “left” and which entity you described as “right.” A third method compares the two sets of fields and returns a value to indicate whether or not they are equivalent.

Note: The paired fields need not be contiguous in memory, nor do they need to be part of a structure. You can build a virtual structure simply by adding a series of otherwise unrelated fields to a FieldPairs object. The other FieldPairs methods then operate on this virtual structure.

Relationship to Other Application Builder Classes

The ViewManager and the BrowseClass use the FieldPairsClass and BufferedPairsClass to accomplish various tasks.

The BufferedPairsClass is derived from the FieldPairs class, so it provides all the functionality of the FieldPairsClass; however, this class also provides a third buffer area (a “save” area), plus the ability to compare the save area with the primary buffers, and the ability to restore data from the save area to the primary buffers (to implement a standard “cancel” operation).

Error Class

The ErrorClass declares an error manager which consistently and flexibly handles any errors. That is, for a given program scope, you define all possible errors by ID number, severity, and message text, then when an error or other notable condition occurs, you simply pass the appropriate ID to the error manager which processes it appropriately based on its severity level.

The defined “errors” may actually include questions, warnings, notifications, messages, benign tracing calls, as well as true errors. The ErrorClass comes with about forty general purpose database errors already defined. You can expand this list to include additional general purpose errors, your own application-specific errors, or even field specific data validation errors. Your expansion of the errors list may be “permanent” or may be done dynamically at run-time.

Files, ABCs, and Legacy Applications

In most cases, Clarion's RAD (Rapid Application Development) tools automatically generate the File Handling code for you. If you are interested in understanding what is required to "port" your legacy hand-coded (embedded) code or just learning about Clarion's ABC File Handling methods, the following code examples will give you a brief overview of some of the important concepts.

Note: If you have hand coded (embedded) file handling Clarion 2.003 code as illustrated below, you will need to manually convert the code to the Clarion ABC equivalent code.

Clarion 2.003 code Clarion ABC Library equivalent

OPEN(File)	Relate:File.Open()	!This ensures all related files are opened,
SHARE(File)	Relate:File.Open()	! as well as the named file, so Referential
CheckOpen(File)	Relate:File.Open()	! Integrity constraints can be enforced.
PUT(File)	Relate:File.Update()	! The Relate: object enforces RI constraints
CLOSE(File)	Relate:File.Close()	! This ensures all related files are closed.
ADD(File)	Access:File.Insert()	!These ABC methods perform error handling
IF ERRORCODE() THEN STOP(ERROR()).		! so the error check is unnecessary. Insert
		! also handles auto-inc and data validation.
PUT(File)	Relate:File.Update()	!The Relate: object enforces RI constraints
IF ERRORCODE THEN STOP(ERROR()).		! in Update() and Delete() methods.
DELETE(File)	Relate:File.Delete(0)	!Parameter suppresses the default confirm
IF ERRORCODE THEN STOP(ERROR()).		! dialog when 0.

Another common file handling situation is the simple file processing LOOP. In 2.003, you would write code like this:

```

SET(key, key)
LOOP
  NEXT(File)
  IF ERRORCODE() THEN BREAK.      !Break at end of file
  !Check range limits here
  !Process the record here
END

```

And here is the equivalent ABC code:

```

SaveState = Access:File.SaveFile() !Tell ABC to "bookmark" where it's at
                                        ! (just in case)
SET(key,key)                            !Note there's no change here
LOOP UNTIL Access:File.Next()           !Breaks when it tries to read past end
                                        ! of file

        !Check range limits here
        !Process the record here
END
Access:File.RestoreFile(SaveState) !Undo the "bookmark" (SaveState must
                                        ! be a USHORT)

```

Another common code construct is getting a record from a file. In 2.003, you might write code like this:

```

IF File::used = 0
    CheckOpen(File)
END
File::used += 1
CLEAR(FIL:record)
FIL:Code = 123
GET(File,FIL:CodeKey)
IF ERRORCODE() THEN CLEAR(FIL:Record).
File::Used -= 1
IF File::used = 0
    CLOSE(file)
END

```

And here is the equivalent ABC code:

```

Relate:File.Open()                        !This handles all error conditions
CLEAR(FIL:record)
FIL:Code = 123
Access:File.Fetch(FIL:CodeKey)           !Fetch clears the record on errors
Relate:File.Close()

```

And of course, the file Open and Close method calls can be generated for you if you just add the file to the procedure's File Schematic. The ABC Library is smart enough to only open a file if it really needs to, making your program more efficient. Using Clarion's ABC Library methods you write less code to accomplish the same (or more) functionality.

New ABC File Handling Capability

The following is a brief overview of the major File Handling capability available with the ABCs.

In addition to the following information, it is recommended that you spend time becoming familiar with the FileManager and RelationManager Classes. See the appropriate chapters of the Application Handbook.

- New lazy open capability
- Support for on-server Referential Integrity

- Improved record buffer integrity during RI updates or deletes.
- All priming / validation done in one place (localized in an object)
- Improved integrity when using field validation.
- Support for INLIST validation.
- Improved recovery from a sequential read of a locked record
- Handling of the “between procedure” alias problem

Lazy Open

The **LazyOpen** property indicates whether to open the managed file immediately when a related file is opened, or to delay opening the file until it is actually accessed. A value of one (1 or True) delays the opening; a value of zero (0 or False) immediately opens the file.

Delaying the open can improve performance when accessing only one of a series of related files.

Implementation:

The Init method sets the LazyOpen property to True. The ABC Templates override this default if instructed. See *Template Overview—File Handling* for more information.

The various file access methods (Open, TryOpen, Fetch, TryFetch, Next, TryNext, Insert, TryInsert, etc.) use the UseFile method to implement the action specified by the LazyOpen property

See Also:

Init, Open, TryOpen, Fetch, TryFetch, Next, TryNext, Insert, TryInsert, UseFile

Referential Integrity Handling

With the introduction of the ABC Library and the ABC Templates, the complex task of maintaining RI efficiently in a Client/Server environment is written for you by the Application Generator.

Maintaining the Referential Integrity of a database is a key element to Relational Database design. Referential Integrity means that, for every One-to-Many (Parent-Child) relationship between tables in the database there exists a Parent record for every Child record (no “orphan” records). To put it in more formal terms, there must be a valid Primary Key value for every existing Foreign Key in the database.

“Orphan” records can occur when the Parent record is deleted, or the Primary Key value (which provides the link to the Foreign Key in the Child record) is changed. Preventing these “orphan” records requires that the

database contain rules stating what action will occur when the end-user attempts to delete the Parent record, or change the Primary Key value.

The most common RI rules are “restrict” (do not allow the delete or change) and “cascade” (delete the related Child records or change their Foreign Key values to match the new Primary Key value in the Parent record). A rarely used rule is “clear” (change the Foreign Key values to NULL when the Parent record is deleted or the Primary Key value in the Parent record changes).

RI constraint enforcement is best handled in Client/Server applications by specifying the RI rules on the back-end database server, usually by defining Triggers, Stored Procedures, or Declarative RI statements. By doing this, the database server can automatically handle RI enforcement without sending any Child records across the network to the Client application for processing. For example, if the rule for Delete is “cascade,” the database server can simply perform all the required Child record deletions when deleting the Parent record from the database—without sending anything back across the network to the Client application.

In the Clarion Dictionary Editor, when you establish a relationship between two files (tables), you can also specify the RI rules for that relationship. Since the database server will actually be handling the RI functionality, the most appropriate way to specify the RI rules in the Clarion Data Dictionary would be to specify “no action” so the Client does nothing.

Data Validation

Data validation means the enforcement of business rules (that you specify) as to what values are valid for any particular field in the database. Typical data validation rules enforce such things as: a field may not be left empty (blank or zero), or the field’s value must be either one or zero (true or false) or within a certain specified numeric range, or the field’s value must exist as a Primary Key value in another file (table).

These data validation rules can be specified either in the Client application or in the back-end database server. The best way to handle implementing data validation rules in your Client/Server applications, so as to generate minimal network traffic, is to specify the business rules in both the Client application *and* the database server:

- By enforcing data validation rules in the Client application you ensure that all data sent to the back-end is already valid. By always receiving valid data the database server will not generate error messages back to the Client application. The net effect of this is to reduce the network traffic back from the database server.

- By enforcing data validation rules on the back-end database server you ensure that the data is always valid, no matter what application is used to update the database—even updating the data with interactive SQL cannot corrupt the data. Therefore, you are covered from both directions.

Enforcing these rules in both your Clarion applications and the database server may seem like a lot of work. However, the Clarion Data Dictionary Editor allows you to specify the most common rules by simply selecting a radio button on the Validity Checks tab of the affected field's definition. By doing this, the actual code to perform the data validation is written for you by the Application Generator's Templates.

The BUFFER Statement

The Clarion BUFFER statement can have a tremendous impact on Client/Server application performance. BUFFER tells the file driver to set up a buffer to hold previously read records and a read-ahead buffer for anticipated record fetches. It also specifies a time period during which the buffered data is considered to be valid (after which the data is re-read from the back-end database server).

When the file driver knows it has buffers to hold multiple records it can optimize the SQL statements it generates to the back-end database server. This allows the back-end database server to return a set of records instead of a single record at a time (also called "fat fetches"). The net effect of this is to change the pattern of network traffic from many small pieces of data to fewer but larger chunks of data, making for more efficient overall network utilization. The most common use of BUFFER would probably be in procedures which allow the end-user to browse through the database.

By setting up buffers to hold already read records, the Client machine fetches records from the local buffer when the user has paged ahead then returns to a previous page of records, instead of generating another request to the back-end database server for the same page of records. This eliminates the network traffic normally generated for subsequent requests for the same set of records.

Setting up read-ahead buffers enables the Client application to anticipate the user's request for the next page of records and receive them from the back-end database server while the user is still examining the first page. Therefore, when the user finally does request the next page, those records are also fetched from the local buffer on the Client machine, giving the end-user apparently instantaneous database retrieval.

Advanced References

Now that you have a basic understanding of the Clarion ABCs, let's take a look at how easy it is to implement a few of these "built-in" objects, methods and properties into your application. You'll soon discover that incorporating complex functionality like Referential Data Integrity within your application is as easy as selecting a radio button or checking a check box within Clarion's IDE.

Let's start with Referential Integrity (RI). For those of you that have been developing software for a few years, you'll immediately appreciate the ease at which you can include this Clarion supplied, highly complex functionality within your application. Much of the elegance, effectiveness, and efficiency of Clarion's built-in RI functionality is only achievable through the incorporation of the OOP paradigm within the ABC libraries.

Note: In this section we'll only cover those IDE items relevant to the subject matter presented in this chapter.

Becoming familiar with concepts described in the Application Handbook, especially the following chapters, will quickly orient you to features available within the IDE.

Application Handbook Chapters:

Template Overview
Wizards and Utility Templates
Procedure Templates
Control Templates
Code and Extension Templates

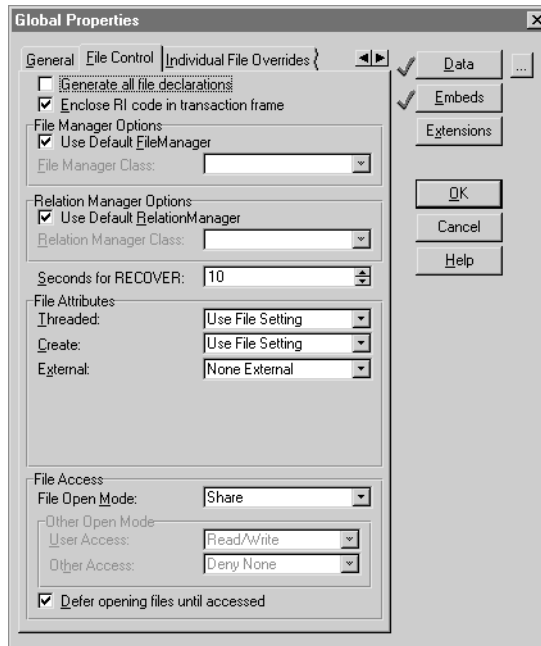
Referential Integrity

To gain a complete working knowledge of RI as it applies to Clarion ABCs refer to the following material:

"Global ABC Template Settings - File Control Tab Options"	Application Handbook
"Procedure Templates - Process Template"	Application Handbook
"Database Design and Network Traffic"	Programmer's Guide
"Dictionary Editor"	User's Guide
"Part IV ISAM Database Drivers"	User's Guide
"Part V SQL Accelerator Drivers"	User's Guide
"FileManager"	Application Handbook
"RelationManager"	Application Handbook

Referential Integrity and the IDE

There are two main IDE Dialogs used to easily establish/modify RI:

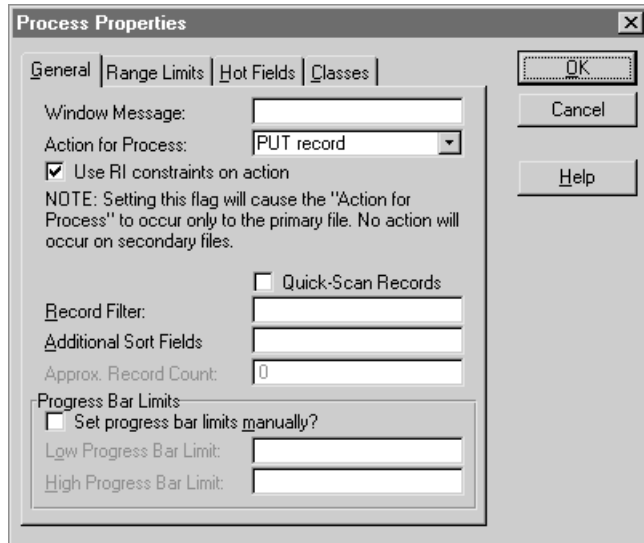


Global ABC Template Settings - File Control Tab Options

Enclose RI code in transaction frame

Check this box to ROLLBACK changes if an update fails during a Referential Integrity maintenance operation (transaction). You should clear this box for file systems that do not support transaction frames such as Clipper, dBase, and FoxPro. See *Database Drivers* for information on individual file systems. See *LOGOUT*, *COMMIT*, and *ROLLBACK* in the *Language Reference*.

Tip: If all files in a relation chain are using the same file system, and the file system supports transaction framing, and you do not want transaction framing around the RI code, you must clear the check box for each file in Individual File Overrides and in Global Settings.



Procedure Templates - Process Template

Use RI constraints on action

Check this box to enforce the RI constraints defined in your data dictionary. Clear this box to generate a simple PUT or DELETE depending on the **Action for Process** chosen.

Lazy Open

To gain a complete working knowledge of Lazy Open as it applies to Clarion ABCs refer to the following material:

“Global ABC Template Settings - File Control Tab Options”

Application Handbook

“Global ABC Template Settings - Individual File Overrides”

Application Handbook

“UseFile (use LazyOpen file)

Application Handbook

“UseView (use LazyOpen file)

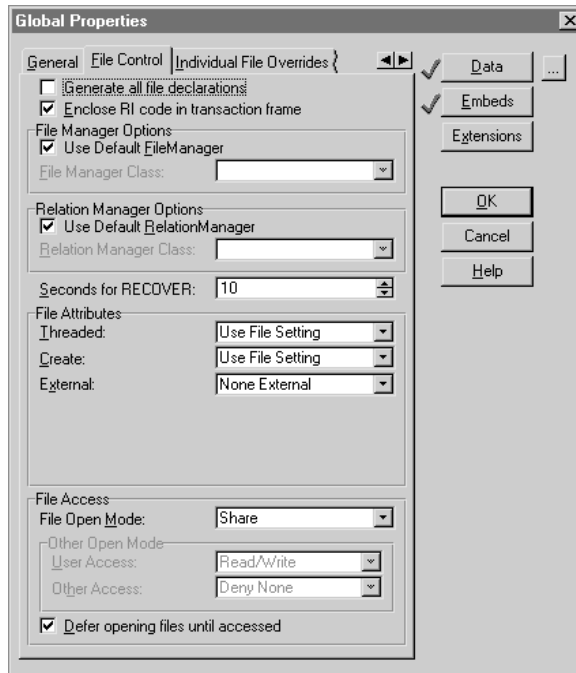
Application Handbook

“PROP:LazyOpen”

Language Reference

Lazy Open and the IDE

There are two main IDE Dialogs used to easily establish/modify Lazy Open:

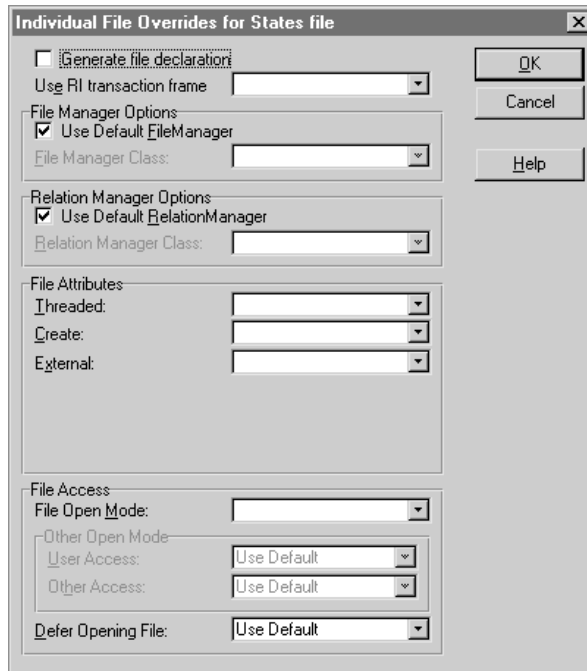


Global ABC Template Settings -File Control Tab Options

Defer opening files until accessed

Specifies when your application opens related files. Check the box to delay opening the file until it is actually accessed.

Delaying the open can improve performance when accessing only one of a series of related files. Clear the box to open the file immediately whenever a related file is opened. See *File Manager Class—LazyOpen* and *UseFile* for more information.



Global ABC Template Settings -Individual File Overrides

Defer Opening File

Specifies when your application opens a file. Select an open from the dropdown list . In most cases you will not have to modify this setting. “Yes” can improve performance since it causes the selected file to only open when needed. “No” will cause the file to open immediately whenever a related file is opened. See *File Manager Class— LazyOpen* and *UseFile* for more information.

Buffer Handling

To gain a complete working knowledge of Buffer Handling as it applies to Clarion ABCs refer to the following material:

“**BufferedPairsClass**”

“**FieldPairsClass**”

“**Buffer statement**”

“**Part IV ISAM Database Drivers**”

“**Part V SQL Accelerator Drivers**”

“**BUFFER (set record paging)**”

“**FLUSH (flush buffers)**”

“**PRESS (put characters in the buffer)**”

“**PRESSKEY (put a keystroke in the buffer)**”

“**STREAM (enable opening stream buffering)**”

“**PROP:Buffer**”

Application Handbook

Application Handbook

User’s Guide

Programmer’s Guide

Programmer’s Guide

Language Reference

Language Reference

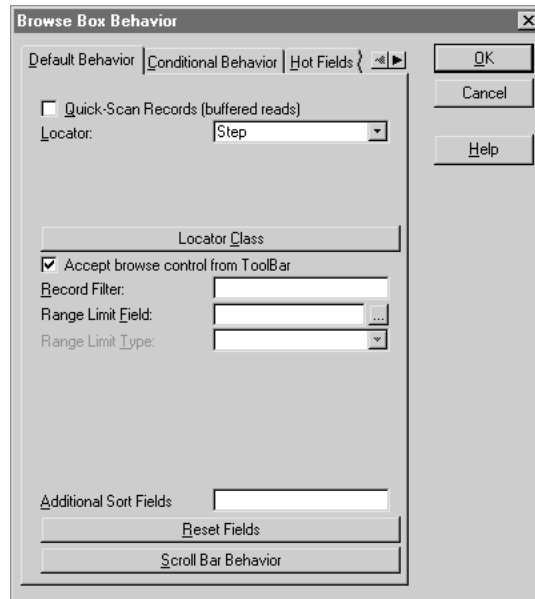
Language Reference

Language Reference

Language Reference

Language Reference

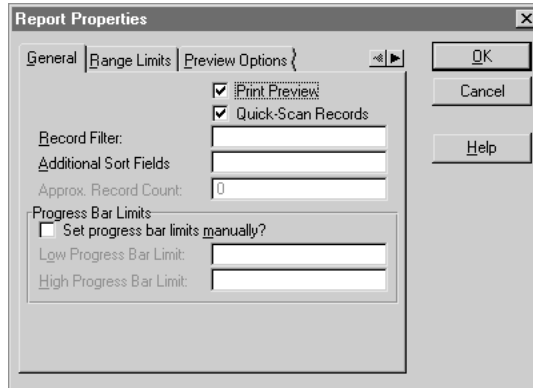
Buffer Handling and the IDE



Quick-Scan Records (buffered reads)

Specifies buffered access behavior for file systems that use multi-record buffers (primarily ASCII, BASIC, and DOS). See *Part III - Database Drivers* for more information. These file drivers read a buffer at a time, allowing for fast access. In a multi-user environment these buffers are not 100% trustworthy, because another user may change a record between accesses. Without quick-scan, the driver refills the buffers before each record access as a safeguard.

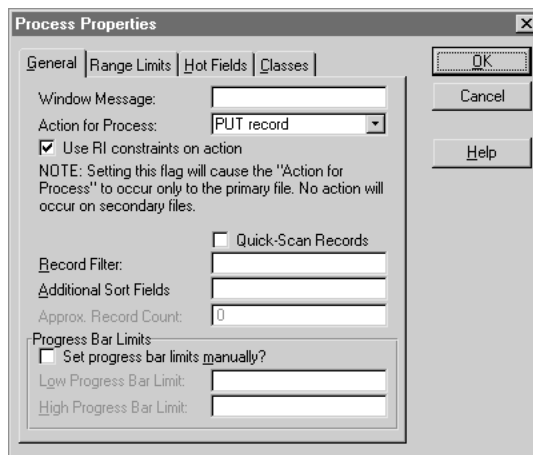
Quick-scanning is the normal way to read records for browsing. However, rereading the buffer may provide slightly improved data integrity in some multi-user circumstances at the cost of substantially slower processing.



Quick-Scan Records (Reports)

Specifies buffered access behavior for file systems that use multi-record buffers (primarily ASCII, BASIC, and DOS). See *Database Drivers* for more information. These file drivers read several records at a time. In a multi-user environment these buffers are not 100% trustworthy because another user may change a record between accesses. As a safeguard, the driver refills the buffers before each record access.

Quick scanning is the normal way to read records for batch processing. However, rereading the buffer may provide slightly improved data integrity in some multi-user circumstances at the cost of substantially slower processing.

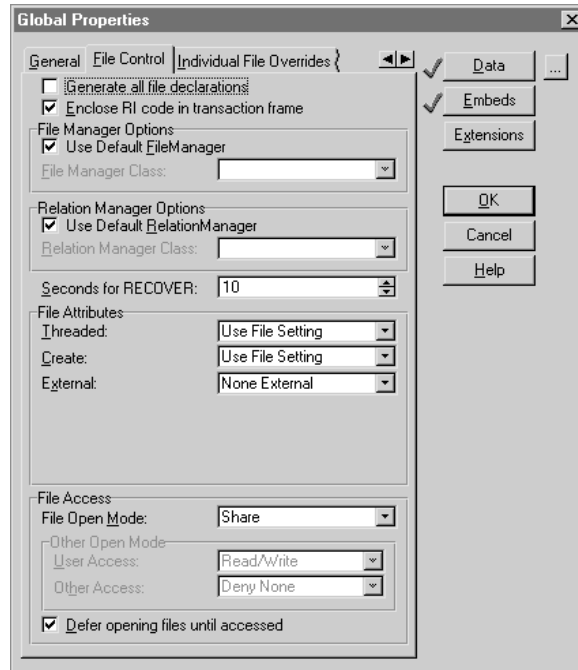


Quick-Scan Records

Specifies buffered access behavior for file systems that use multi-record buffers (primarily ASCII, BASIC, and DOS). See

Database Drivers for more information. These file drivers read several records at a time. In a multi-user environment these buffers are not 100% trustworthy, because another user may change a record between accesses. As a safeguard, the driver refills the buffers before each record access.

Quick scanning is the normal way to read records for batch processing. However, rereading the buffer may provide slightly improved data integrity in some multi-user circumstances at the cost of substantially slower processing.



Threaded

Specifies whether the application generator adds the `THREAD` attribute to `FILE` structures. `THREAD` is needed for MDI browse and form procedures to prevent record buffer conflicts when the end user changes focus from one thread to another.

Use File Setting Sets the `THREAD` attribute according to the setting in the data dictionary. See the *User's Guide—Dictionary Editor—File Properties*.

All Threaded Adds the `THREAD` attribute to each `FILE`.

External

Specifies whether the application generator adds the **EXTERNAL** attribute to **FILE** structures. **EXTERNAL** specifies the memory for the **FILE**'s record buffer is allocated by an external library. See the *Language Reference* for more information.

Note: When using **EXTERNAL** to declare a **FILE** shared by multiple libraries (.LIBs, or .DLLs and .EXE), only one library should define the **FILE** without the **EXTERNAL** attribute. This ensures that there is only one record buffer allocated for the **FILE** and all the libraries and the .EXE will reference the same memory when referring to data elements from that **FILE**.

One Threaded Omits the **THREAD** attribute for each **FILE**.

None External Omits the **EXTERNAL** attribute from all file declarations and enables the **Export All File Declarations** prompt.

Export All File Declarations

Check this box to export file declarations (see *Module Definition Files* in the *Programmer's Guide*). This prompt is only available when you specify *Dynamic Link Library (.DLL)* as the **Destination Type** in the **Application Properties** dialog.

All External Adds the **EXTERNAL** attribute to all file declarations *and* lets you specify the **Declaring Module** and whether **All files are declared in another .APP**.

Declaring Module

The file name (without extension) of the **MEMBER** module containing the **FILE** definition without the **EXTERNAL** attribute. If the **FILE** is defined in a **PROGRAM** module, leave this field blank.

All files are declared in another .APP

Check this box to ensure that files are opened and closed at the right time, thereby preserving the integrity of the file buffers, when the files are declared in another application (rather than hand code).

7 - WINDOW AND CONTROL TECHNIQUES

Introduction

In this chapter we'll review two major functional enhancements to Clarion. They are directly related to the incorporation of the OOP paradigm and only available with the ABC Class Libraries. They are:

- Window Resizer - Major Enhancement.
(ABC WindowResizeClass)
- Multi-Language Support. - New.
(ABC TranslatorClass)

Note: The Application Handbook has complete information and example code for both of the above ABC Classes.

Overview

Window Resizer

The WindowResizeClass lets the end user resize windows that have traditionally been fixed in size due to the controls they contain (List boxes, entry controls, buttons, nested controls, etc.). The WindowResizeClass *intelligently* repositions the controls, resizes the controls, or both, when the end user resizes the window.

Multi-Language Support

By default, the ABC Templates, the ABC Library, and the Clarion visual source code formatters generate American English user interfaces. However, Clarion makes it very easy to efficiently produce non-English user interfaces for your application programs.

The TranslatorClass provides very fast runtime translation of user interface text. The TranslatorClass lets you deploy a single application that serves all your customers, regardless of their language preference. That is, you can use the TranslatorClass to display several different user interface languages based on end user input or some other runtime criteria, such as INI file or control file contents.

Alternatively, you can use the Clarion translation files (*.TRN) to implement a single non-English user interface at compile time.

Resizer

The Resizer - Overview

The intelligent repositioning is accomplished by recognizing there are many different types of controls that each have unique repositioning *and* resizing requirements. The `WindowResizeClass` also recognizes that controls are often nested, and considers whether a given control's coordinates are more closely related to the window's coordinates or to another control's coordinates. That is, intelligent repositioning correctly identifies each control's parent. See *SetParentControl* for more information on the parent concept.

The intelligent repositioning includes several overall strategies that apply to all window controls, as well as custom per-control strategies for resizing and repositioning individual controls. The overall strategies include:

See *SetStrategy* for more information on resizing strategies for individual controls.

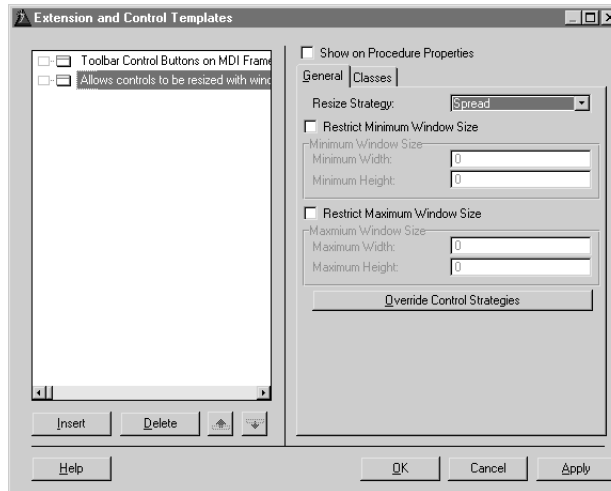
Note: To allow window resizing you must set the WINDOW's frame type to `Resizable` and you must check the immediate box to add the `IMM` attribute to the WINDOW. We also recommend adding the `MAX` attribute. See *The Window Formatter—The Window Properties Dialog* in the *User's Guide* for more information on these settings.

WindowResize

The `WindowResize` template lets the end user resize windows that have traditionally been fixed in size due to the controls they contain (List boxes, entry controls, buttons, etc.).

Tip: The `WindowResize` code repositions and resizes each control relative to its parent. This approach provides attractive, rational resizing of virtually any window, regardless of the controls it contains.

The template generates code to reposition the controls, resize the controls, or both, when the end user resizes the window.



Resize Strategy

Specifies the method for resizing and repositioning the controls to fit within the new window size. Choose from:

Resize Scales all window coordinates by the same amount, thus preserving the relative sizes and positions of all controls. That is, all controls, including buttons and entry fields get taller and wider as the window gets taller and wider. Window fonts are unchanged.

Spread Maintains the design-time look and feel of the window by applying a strategy specific to each control type. For example, **BUTTON** sizes are not changed but their positions are tied to the nearest window edge. In contrast, **LIST** sizes *and* positions are scaled in proportion to the window.

Surface Makes the most of the available pixels by positioning other controls to maximize the size of **LIST**, **SHEET**, **PANEL**, and **IMAGE** controls. We recommend this strategy for Wizard generated windows.

Don't Alter Controls

Controls are not resized when the window is resized.

Tip: Even though list boxes may be resized, the column widths within the list box are not resized. However, the right-most column does expand or contract depending on the available space.

Restrict Minimum Window Size

Check this box to specify a minimum window height and width. This lets you enforce a minimum reasonable size of the window based on the size and number of controls on the window. In other words, you can keep your end user from shrinking the window so much that its controls become invisible or unrecognizable.

Minimum Width Specify the minimum width of the window in dialog units. Dialog units are based on the window's font and are 1/4 of the average character width.

Zero sets the window minimum to the size at which the window opens (not necessarily the design time size). In other words, it takes into account any .INI setting plus any runtime Property syntax. Thus, we allow the developer to open the window, perform any dynamic control production (including resizing the window) before the minimum restriction takes effect.

Minimum Height Specify the minimum height of the window in dialog units. Dialog units are based on the window's font and are 1/8 of the character height.

Zero sets the window minimum to the size at which the window opens (not necessarily the design time size). In other words, it takes into account any .INI setting plus any runtime Property syntax. Thus, we allow the developer to open the window, perform any dynamic control production (including resizing the window) before the minimum restriction takes effect.

Restrict Maximum Window Size

Check this box to specify a maximum window height and width. This lets you enforce a maximum reasonable size of the window.

Maximum Width Specify the maximum width of the window in dialog units. Dialog units are based on the window's font and are 1/4 of the average character width.

Zero sets the window maximum to the size at which the window opens (not necessarily the design time size). In other words, it takes into account any .INI setting plus any runtime Property syntax. Thus, we allow the developer to open the window, perform any dynamic control production (including resizing the window) before the maximum restriction takes effect.

Maximum Height

Specify the maximum height of the window in dialog units. Dialog units are based on the window's font and are 1/8 of the character height.

Zero sets the window maximum to the size at which the window opens (not necessarily the design time size). In other words, it takes into account any .INI setting plus any runtime Property syntax. Thus, we allow the developer to open the window, perform any dynamic control production (including resizing the window) before the maximum restriction takes effect.

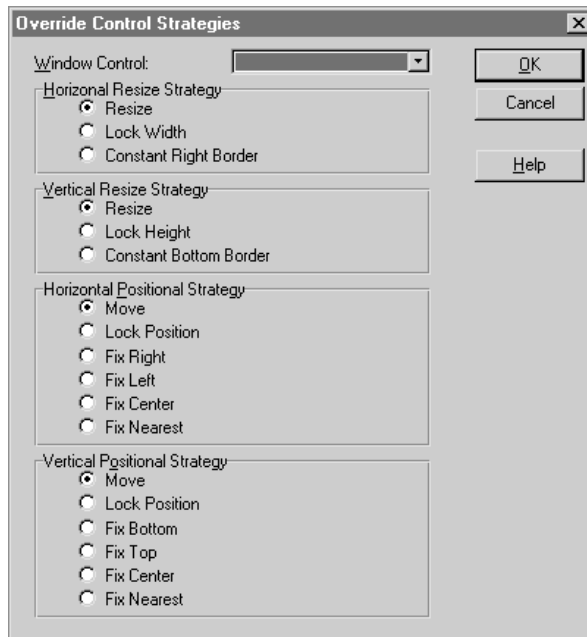
Override Control Strategies

Press this button to override the default resize strategy for individual controls. This opens the **Override Control Strategies** dialog.

Override Control Strategies

The **Override Control Strategies** dialog lets you override the default resize strategy for individual controls. For example, by default, buttons are “fixed” to the nearest window borders and are not repositioned like most other controls. However, if you want your procedure to reposition the button like other controls, you may specify this here. See also *Window Resize Class—SetStrategy*.

Press the **Insert** button to select the control for which to set the resize strategy. Then choose from the following sizing and positioning options:



Horizontal Resize Strategy

Specify how the control’s width is determined when the end user resizes the window. Choose from:

Lock Width The control’s design time width does not change.

Constant Right Border
Locks right edge, moves left.

Vertical Resize Strategy

Specify how the control’s height is determined when the end user resizes the window. Choose from:

Lock Height The control’s design time height does not change.

Constant Bottom Border
Locks bottom edge, moves top.

Horizontal Positional Strategy

Specify how the control's horizontal position is determined when the end user resizes the window. Choose from:

Lock Position The control's left edge maintains a fixed distance (the design time distance) from parent's left edge.

Fix Right The control's right edge maintains a proportional distance from parent's right edge.

Fix Left The control's left edge maintains a proportional distance from parent's left edge.

Fix Center The control's center maintains a proportional distance from parent's center.

Fix Nearest Applies Fix Right or Fix Left, whichever is appropriate.

Vertical Positional Strategy

Specify how the control's vertical position is determined when the end user resizes the window. Choose from:

Lock Position The control's top edge maintains a fixed distance (the design time distance) from parent's top edge.

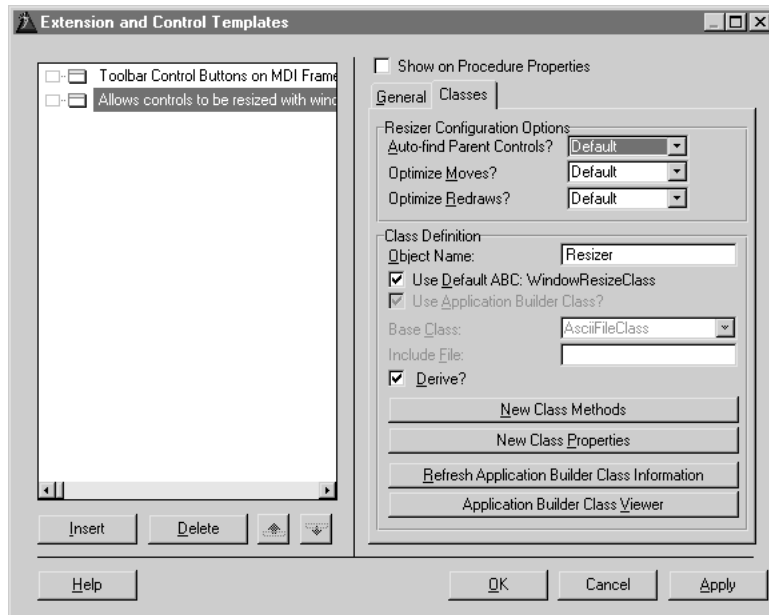
Fix Bottom The control's bottom edge maintains a proportional distance from parent's bottom edge.

Fix Top The control's top edge maintains a proportional distance from parent's top edge.

Fix Center The control's center maintains a proportional distance from parent's center.

Fix Nearest Applies Fix Top or Fix Bottom, whichever is appropriate.

Resizer Configuration Options



Automatically find parent controls

Check this box to set parent/child relationships among window controls. Clearing the box makes the WINDOW the parent of all its controls. Setting parent/child relationships lets any special scaling cascade from parent to child. See *WindowResizeClass Methods—SetParentDefaults* for more information.

Optimize Moves

Check this box to move all controls at once during the resize operation, producing a snappier resize and avoiding bugs on some windows. See *WindowResizeClass Properties—DeferMoves* for more information.

Optimize Redraws

Check this box to make controls transparent (TRN attribute) during the resize operation, producing a smoother redraw and avoiding bugs on some windows. See *WindowResizeClass Properties—AutoTransparent* for more information.

Classes Tab

Use the Classes tab to override the global Resizer setting. See *Template Overview—Classes Tab Options—Global and Local*.

Translator

Multi-Language Support - Overview

TranslatorClass Concepts

The TranslatorClass and the ABUTIL.TRN file provide a way to perform language translation at runtime. That is, you can make your program display one or more non-English user interfaces based on end user input or some other runtime criteria such as INI file or control file contents. You can also use the TranslatorClass to customize a single application for multiple customers. The TranslatorClass operates on all user interface elements including window controls, window titlebars, tooltips, list box headers, and static report controls.

The ABUTIL.TRN File

The ABUTIL.TRN file contains translation pairs for all the user interface text generated by the ABC Templates and the ABC Library. A translation pair is simply two text strings: one text string for which to search and another text string to replace the searched-for text. At runtime, the TranslatorClass applies the translation pairs to each user interface element.

You can directly edit the ABUTIL.TRN file to add additional translation items. We recommend this method for translated text common to several applications. The translation pairs you add to the Translator GROUP declared in ABUTIL.TRN are automatically shared by any application relying on the ABC Library and the ABC Templates.

Note: Save this file in a separate directory to avoid it being overwritten.

Translating Custom Text

The default ABUTIL.TRN translation pairs do not include custom text that you apply to your windows and menus. To translate custom text, you simply add translation pairs to the translation process, either at a global level or at a local level according to your requirements. To help identify custom text, the TranslatorClass automatically identifies any untranslated text for you; you need only supply the translation. See *ExtractText in the Application Handbook* for more information.

Macro Substitution

The TranslatorClass defines and translates macro strings. A TranslatorClass macro is simply text delimited by percent signs (%), such as %mymacro%.

You may use a macro within the text on an APPLICATION, WINDOW, or REPORT control or title bar, or you may use a macro within TranslatorClass translation pairs text.

You define the macro with surrounding percent signs (%), and you define its substitution value with a TranslatorClass translation pair (without percent signs).

This macro substitution capability lets you

- translate a small portion (the macro) of a larger text string
- do multiple levels of translation (a macro substitution value may also contain a macro)

Relation to Other Application Builder Classes

The TranslatorClass object is called Translator, and each template-generated procedure calls on the Translator object to translate all text for its APPLICATION, WINDOW or REPORT. Additionally, the template-generated PopupClass objects (ASCIIViewer and BrowseBox templates) and PrintPreviewClass objects (Report template) use the Translator to translate menu text.

Note: The ABC Templates use the TranslatorClass to apply user interface text defined at compile time. The templates do not provide a runtime switch between user interface languages.

Conceptual Example

The following example shows a typical sequence of statements to declare, instantiate, initialize, use, and terminate a TranslatorClass object.

This example applies both default and custom translations to a “preferences” window. It also collects and stores untranslated text in a file so you don’t have to manually collect the text to translate.

Example:

```

PROGRAM
  _ABCD11Mode_ EQUATE(0)
  _ABCLinkMode_ EQUATE(1)
  INCLUDE('ABUTIL.INC')           !declare TranslatorClass
  MAP
  END

MyTranslations GROUP              !declare local translations
Items              USHORT(4)      !4 translations pairs
                  PSTRING('Company') ! item 1 text (macro)
                  PSTRING('Widget %CoType%') ! item 1 replacement text
                  PSTRING('&Sound') ! item 2 text
                  PSTRING('&xSoundx') ! item 2 replacement text

```

```

                                PSTRING('&Volume')           ! item 3 text
                                PSTRING('&xVolumex')         ! item 3 replacement text
                                PSTRING('OK')              ! item 4 text
                                PSTRING('xOKx')            ! item 4 replacement text
                                END
INIMgr      INIClass                !declare INIMgr object
Translator  TranslatorClass         !declare Translator object
CoType      STRING('Inc.')          !default company type
Sound       STRING('ON ')           !default preference value
Volume      BYTE(3)                 !default preference value

PWindow WINDOW('%Company% Preferences'),AT(, ,100,35),IMM,SYSTEM,GRAY
CHECK('&Sound'),AT(8,6),USE(Sound),VALUE('ON','OFF')
PROMPT('&Volume'),AT(31,19),USE(?VolumePrompt)
SPIN(@s20),AT(8,20,21,7),USE(Volume),HVSCROLL,RANGE(0,9),STEP(1)
BUTTON('OK'),AT(57,3,30,10),USE(?OK)
END

CODE
INIMgr.Init('.\MyApp.INI')           !initialize INIMgr object
INIMgr.Fetch('Preferences','CoType',CoType) !get company type, default
! Inc.

Translator.Init                      !initialize Translator object:
! add default translation pairs
Translator.AddTranslation(MyTranslations) !add local translation pairs
Translator.AddTranslation('CoType',CoType) !add translation pair from
! INI
Translator.ExtractText='. \MyApp.trn' !collect user interface text
OPEN(PWindow)
Translator.TranslateWindow           !translate controls & titlebar
ACCEPT
  IF EVENT() = EVENT:Accepted
    IF FIELD() = ?OK
      INIMgr.Update('Preferences','Sound',Sound)
      INIMgr.Update('Preferences','Volume',Volume)
      POST(EVENT:CloseWindow)
    . . .
  Translator.Kill                    !write user interface text

```


8 - BROWSE PROCEDURE TECHNIQUES

Introduction

This chapter explores the enhancements to the Browse box template and procedures that use it. There are several classes that work closely together to produce this functionality.

Note: The Application Handbook has complete information and example code for the ABC Classes used in a Browse procedure.

Overview

A “Browse Procedure” can be made in several ways. You can create one using the Browse Wizard, the Browse wizard, the Browse Procedure template, or by populating a Browse Box control template on a Window procedure. These all produce similar results--a window with a LIST control that displays data. All other functionality (update capability, selection capability, sorting, totaling, etc.) is optional and generally set at design time using the template prompts.

The ABC template prompts are very similar, although there are some additional prompts to extend customization capabilities. In addition, the code generated uses the ABC library, allowing more customization with less effort on the developer’s part. Many new features are now directly supported when using the ABC templates, and a multitude of additional features are easily implemented by setting ABC properties or calling ABC methods.

Here are some of the features made possible by the ABC templates and libraries:

- ◆ Improved edit-in-place support
- ◆ Increased efficiency (especially under SQL)
- ◆ Ability to avoid loading browse until visible (vital on forms with many children)
- ◆ File loaded browses
- ◆ Filtered locators
- ◆ Filtered locators using “Contains”
- ◆ Print button
- ◆ Non-keyed sort orders

- ◆ Ability to switch between multiple update forms (including edit-in-place)
- ◆ Selected bar stays in same position when returning from update and selected bar stays in same position when switching tabs
- ◆ Ability to support 'partially filled' browses (vital for SQL)
- ◆ Ability to locate to 'current' location when entering browse
- ◆ Significant reduction of 'browse flicker'
- ◆ Query by Example (QBE)

In the rest of this chapter, we examine function points (application features) and manners of implementation. Keep in mind that these techniques are not the *only* way to implement the functionality, but in most cases is the easiest and most efficient way. In addition, this chapter cannot cover every possible application feature and is not intended as a primer for application development. Instead, it is meant to help developers solve problems using ABC paradigm to its fullest potential.

Browse Box Template Features

ABC Edit-in-place

The ABC templates' Browse Box has built-in support for edit-in-place (actually this is in the Browse Update buttons, but since they are only useful for a Browse Box control template, we will consider them as part of the Browse).

The BrowseUpdateButtons template provides three buttons for managing file I/O for a BrowseBox: Insert, Change, and Delete. These three button controls act on the records in a browse box. When pressed, the button retrieves the selected record and invokes the respective database action for that record.

The BrowseUpdateButtons template provide a choice of an update procedure (recommended for files with two-way relationships) or edit-in-place updates (recommended for lookup files--files with one-way relationships).

To implement Edit-in-place, you need only to check the **Use Edit in Place** check box on the **Actions** tab of the Browse Update Buttons control template.

This lets the end user update the browsed file by typing directly into the BrowseBox list and provides a very direct, intuitive spreadsheet style of update. You can also configure the Edit in place behavior with the Configure Edit in place button.

The Save option

The *Configure Edit in place* dialog offers the Save option for four different keyboard actions. These options determine whether changes to an edited record are saved or abandoned upon the following keyboard actions: TAB key at end of row, ENTER key, up or down arrow key, focus loss (changing focus to another control or window, typically with a mouse-click). Choose from:

<i>Default</i>	Save the record as defined in the <code>BrowseClass.Ask</code> method.
<i>Always</i>	Always save the record.
<i>Never</i>	Never save the record, abandon the changes.
<i>Prompted</i>	Ask the end user whether to save or cancel the changes.

Remain editing options

The *Configure Edit in place* dialog offers the Remain editing options for three different keyboard actions. Check these boxes to continue editing upon the following keyboard actions: TAB key at end of row, ENTER key, up or down arrow key. Clear the boxes to stop editing.

Retain column option

The *Configure Edit in place* dialog offers the Retain column option for the up and down arrow keys only. Check this box to continue editing within the same list box column in the new row. Clear to continue editing within the left most editable column in the new row.

Insertion Point option

The *Configure Edit in place* dialog offers the Insertion Point option for initial new record placement in the list. The droplist choices— before, after, and append— indicate where the edit-in-place row will appear in the list when inserting a record. Before and after indicate placement in relation to the highlighted record, and append places the edit-in-place row at bottom of the list.

This does not change the sort order. After insertion, the list is resorted and the new record appears in the proper position within the sort sequence.

Column Specific options

Press this button, then press the Insert button to specify the CLASS of object to use when editing a specific list box column. The Column Specific dialog lets you control the class (and object) the procedure uses to edit a specific Browsebox column. You may specify your own or a third party class.

By default, the BrowseUpdateButton template generates code to use the EditEntryClass in the ABC Library. You can also use the other edit classes or derive your own. The Application Generator must know about the CLASS you specify--see the Template Overview--ABC Compliant Classes section in the Application Handbook for more information.

Adding column Specific options to a column, also adds important embed points you can use during Edit-in-place. In the following section, we'll use these embed points to add hot key lookup to an entry control during Edit-in-place.

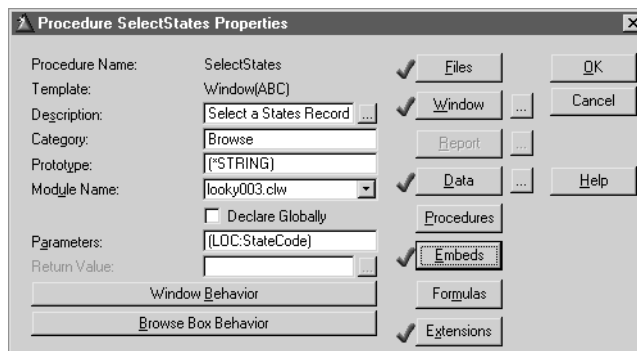
Calling a Lookup from Edit-in-place

The EditClass (or the derived class for a column) creates a control for data entry. In this example, we will assume that it is an ENTRY control (using the EditEntryClass) and that the application requires lookup capabilities for that control.

Create the Function to return the data for the lookup

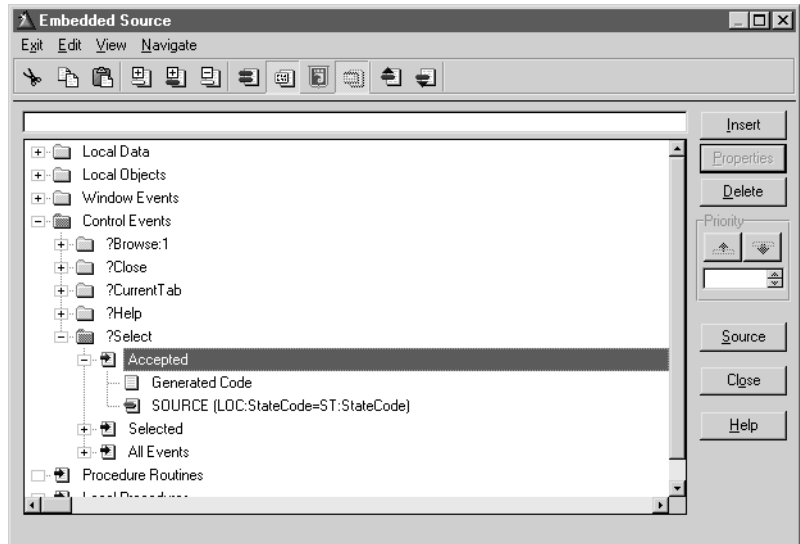
First, we create a SelectState procedure (a simple browse with a Select button) and make it a Function by adding these three things:

1. Add the Prototype and parameter as shown below:



- In the Control Events, ?Select, Accepted embed point, add the following code:

```
LOC:StateCode=ST:StateCode
```



With the function created, next we'll add the functionality to call the lookup and save the returned value.

Implementing the lookup

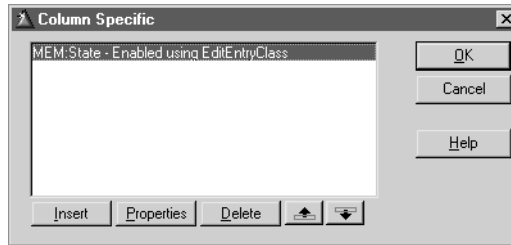
The key point to remember is that the control created for edit-in-place can be referred to as SELF.FEQ inside its object. This allows you to set its properties without needing to know anything else about it. In this example, we will use a Browse Box in the PEOPLE.APP example. The browse is on a Members file and contains a State field.

By adding this column to the Column Specific list in the Edit in place options, we get the EditInPlace::MEM:State CLASS, which is derived from the EditEntryClass and automatically contains the following methods:

```
CreateControl  
Init  
Kill  
SetAlerts and  
TakeEvent
```

In addition, we also get embed points within all of the methods. We'll use these for the embedded code needed for the functionality that we want.

- In the BrowseMembers procedure, add the MEM:State field to the Column Specific list.



Next, we'll add the code to alert the F10 key to perform the lookup.

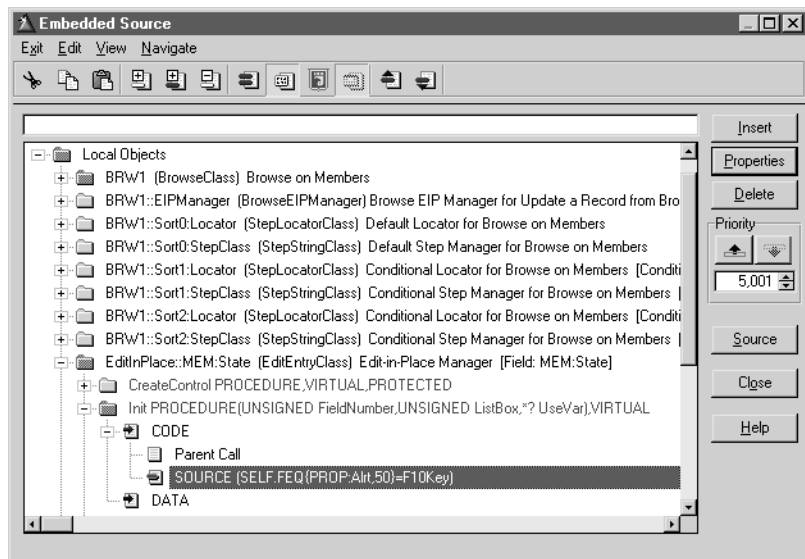
2. In the Local Objects, EditInPlace::MEM:State, Init embed (after the parent call) embed point, add the following code:

```
SELF.FEQ{PROP:Alrt,50}=F10Key
```

Notice the second parameter (50). The PROP:Alrt property is an array and the second parameter specifies the offset. The ABC Library automatically adds alerts for the following (in ABEIP.CLW):

```
SELF.Feq{PROP:Alrt,1} = TabKey
SELF.Feq{PROP:Alrt,2} = ShiftTab
SELF.Feq{PROP:Alrt,3} = EnterKey
SELF.Feq{PROP:Alrt,4} = EscKey
SELF.Feq{PROP:Alrt,5} = DownKey
SELF.Feq{PROP:Alrt,6} = UpKey
```

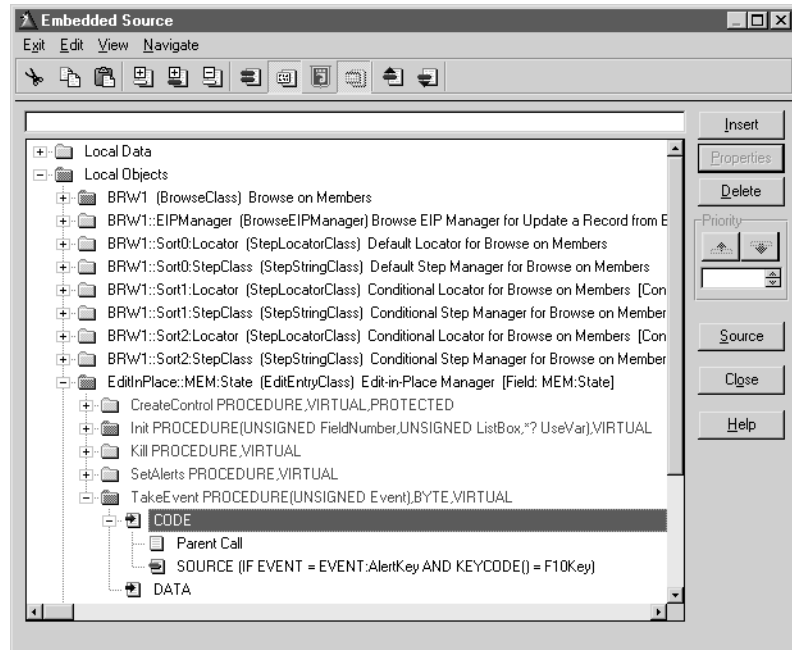
These are added to allow navigation between columns in edit-in-place mode. If we don't provide the second parameter, the alert is added to the first offset, overwriting the alert for the TabKey. We'll use a high number of 50 to allow for any additional alerts that may appear in an interim update to Clarion.



Next, we add the code to call the lookup procedure when the F10 key is pressed.

2. In the Local Objects, `EditInPlace::MEM:State`, `Init` embed (after the parent call) embed point add the following code:

```
IF EVENT = EVENT:AlertKey AND KEYCODE() = F10Key
  GlobalRequest=SelectRecord           !to enable the Select button
  SelectStates(Queue:Browse:1.MEM:State) !call the lookup
  PUT(Queue:Browse:1)                 !put to the queue
  DISPLAY()
END
```



Combining Edit-in-place and an Update procedure

At times, an application might want two (or more) methods to update a record in a browse. For example, you might want edit-in-place for simple edits and for adds (and more detailed updates), you would want an update form.

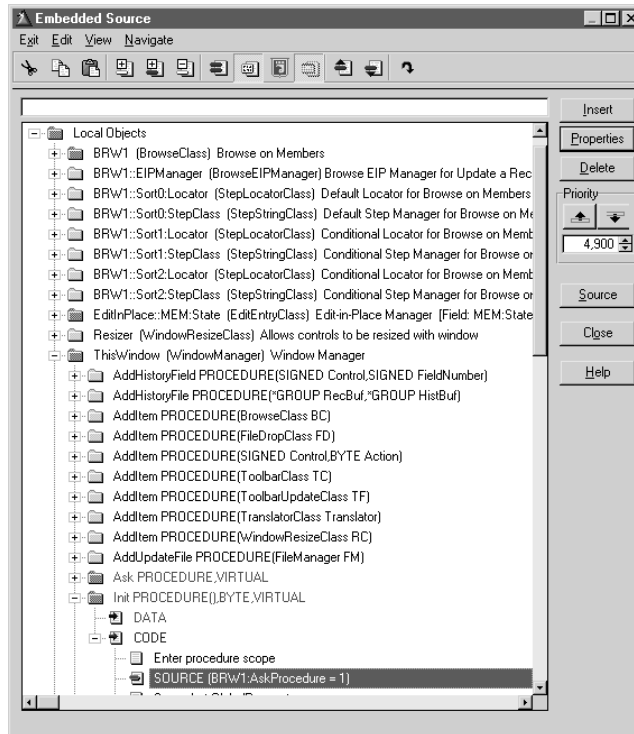
The ABC libraries allow you to add this functionality easily.

First, let's examine what a Browse Procedure does when the user calls for an update. It calls a method named something like `BRW1.ThisWindow.Run`. This method takes a number as a parameter. The number is the value of the `BRW1.AskProcedure` property. In other words, if you enable Edit-In-Place *and* specify an update procedure, you have two-thirds of the work done. Set the `BRW1.AskProcedure` Property to 0 (zero) and you have Edit-in-Place; Set it to 1 (One) and you call your update procedure.

Once again, we'll use the PEOPLE.APP example to demonstrate the technique.

1. Select the BrowsePeople procedure, and press the Properties button.
2. In the UpdateButton section of the Procedure Properties window, check the Use Edit in Place box. **Notice that an update procedure is already specified. Make sure to leave that procedure named!**
3. Embed the following code to set the default update action to call the form. In the *Local Objects, This Window, Init ,Code* embed point after the *Enter Procedure Scope* priority label:

```
BRW1:AskProcedure = 1
```



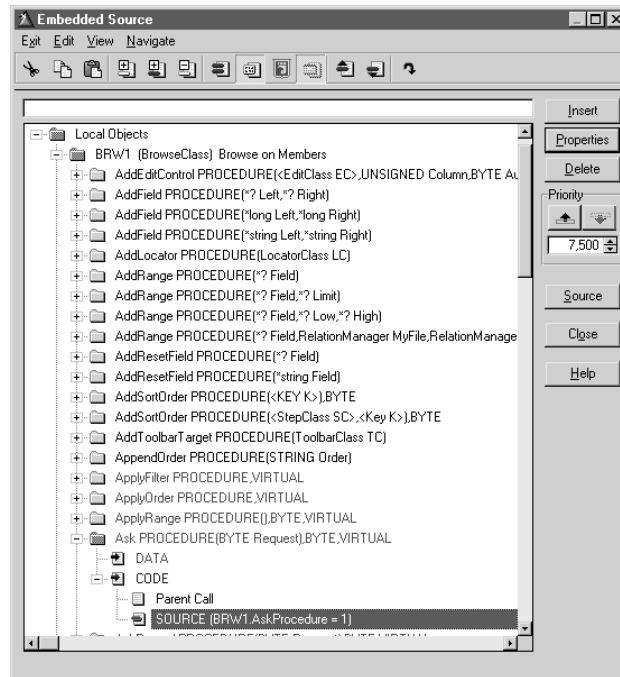
4. Embed the code to set the action of a double-click to use edit-in-place. In the *Local Objects, BRW1, TakeKey , Code* embed point before the Parent Call priority label embed the following code:

```
IF RECORDS(SELF.ListQueue) AND KEYCODE() = MouseLeft2
  BRW1.AskProcedure=0
END
```



- Finally, we will embed the code to set the action back after an edit (either from a form or from an edit-in-place). In the *Local Objects*, *BRW1*, *Ask*, *Code* embed point before the *Parent Call* label, embed the following code:

```
BRW1.AskProcedure = 1
```



This is important. It ensures the value is the same as when you started. I like to think of this type of code as “If you turn the light on when you enter a room, be sure to turn it off when you leave.”

- Compile and run the application. Call the browse procedure and notice the behavior—a double-click gets you edit-in-place and any other method of calling an update gets you the form.

This gives us two alternatives. However, the ABC templates allow us to add more possibilities easily. We know that the *AskProcedure* property can have a value of 0 or 1. What if we want more possibilities?

Let’s look at the second *ThisWindow.Run PROCEDURE* (the one with two parameters). Notice it takes a *USHORT* parameter named *Number* and a *BYTE* named *Request*. The *number* parameter it receives is the value of the *AskProcedure*. With this knowledge, we can easily add a third “update” procedure—this one a record viewer instead of a record update procedure.

- First, create a procedure using the Window Template. Define the *WINDOW* as an *MDI* child, and populate all the fields as *STRING* controls. Let’s call the procedure—*ViewPeople*.

2. Select the BrowsePeople procedure and press Properties.
3. Press the Procedures button and select the ViewPeople procedure. This adds the procedure to the local MAP.
4. Next, we will add a few buttons the window. Press the Window Button to call the Window Formatter.
5. Populate a button with the following properties:
Text: 'View'
USE: ?ViewButton
6. Double-click on the button (&View) and embed the following code in the Accepted Embed point.
BRW1.AskProcedure = 2
POST(Event:Accepted,?Change:2)
7. Now to get the multiple procedures to work, we are going to modify the ThisWindow.Run method.

Let's use the Embeditor for this one.
8. Close the Window Formatter and the Procedure Properties window, right-click on the BrowsePeople procedure and select Source.
9. Locate the second ThisWindow.Run PROCEDURE. This is the one with the parameters.

Let's examine the code:

```
ThisWindow.Run PROCEDURE(USHORT Number,BYTE Request)
ReturnValu     BYTE,AUTO
! Start of "WindowManager Method Data Section"
! [Priority 5000]
! End of "WindowManager Method Data Section"
CODE
! Start of "WindowManager Method Executable Code Section"
! [Priority 2500]
ReturnValu = PARENT.Run(Number,Request)
! [Priority 6000]
GlobalRequest = Request
Updatepeople
ReturnValu = GlobalResponse
! [Priority 8500]
! End of "WindowManager Method Executable Code Section"
RETURN ReturnValu
```

Now let's modify it so it looks like this:

```
ThisWindow.Run PROCEDURE(USHORT Number,BYTE Request)
ReturnValu     BYTE,AUTO
! Start of "WindowManager Method Data Section"
! [Priority 5000]
! End of "WindowManager Method Data Section"
CODE
! Start of "WindowManager Method Executable Code Section"
! [Priority 2500]
ReturnValu = PARENT.Run(Number,Request)
! [Priority 6000]
GlobalRequest = Request
```

```
EXECUTE Number
  Updatepeople
  Viewpeople
END
ReturnValue = GlobalResponse
RETURN ReturnValue
!next 4 lines of code never execute
GlobalRequest = Request
Updatepeople
ReturnValue = GlobalResponse
! [Priority 8500]
! End of "WindowManager Method Executable Code Section"
RETURN ReturnValue
```

This now allows us to have an alternate update procedure! Using this technique, you can have many update procedures. Imagine the possibilities:

- different procedures for different users
- different procedures for different security levels
- different procedures based on whether an application is running locally or over the Internet using Clarion Internet Connect
- the possibilities are endless!

9 - *REPORT AND PROCESS TECHNIQUES*

Introduction

This chapter explores the enhancements to the built-in Report and Process functionality in Clarion. These two classes work closely together to achieve their results.

Note: The Application Handbook has complete information and example code for both of the above ABC Classes.

Overview

Reports and Processes are intimately linked in so far as the Report template uses the process class to provide sequential file reads. This interdependence amongst the classes is the key to the power and versatility of the ABCs.

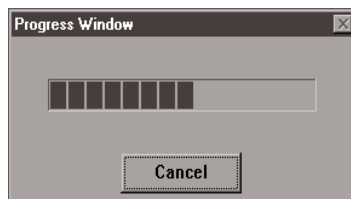
One of the keys to understanding how the new report and process templates work lies in the Progress window. In the Legacy templates the progress window was hard-coded into the template, and therefore not editable from the IDE. This is no longer the case; the developer has full control of the progress window with the ABC templates, and the Progress window now drives the procedure.

Processes & Reports

Because of the incorporation of a process object in the report object, the common functionality will be explored here.

The Progress window

Both the Report template and the Process template utilize a **WindowManager** object to run the **Report** and **Process** objects. The **WindowManager** object also provides user feedback in the form of a progress bar. This Progress window is completely configurable from the IDE.



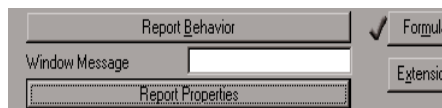
The mechanism employed by the templates to run the process and report objects from the window object is to call the respective Init methods from the WindowManager Init method.

Customization

The two primary areas within the IDE used for modifying the progress window are the Window Formatter and the Report/Process Properties button on the Procedure properties window.

The Progress window is editable like any other procedure window in the Window Formatter.

The Template interface provided by the Report/Process Properties button allows access to specific programmatic and appearance functionality.

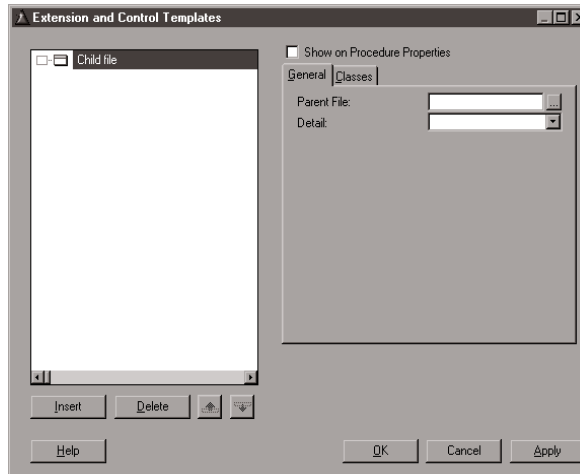


The Procedure properties window also provides Progress window customization in the form of a **Window Message** entry field. The text placed in this entry field is displayed in a **STRING** on the Progress Window above the **Progress bar**.

Progress bar functionality can be manipulated by overriding the record count or manually setting the progress bar limits in the Report/Process Properties dialog. This is usually unnecessary due to the ability of ProcessClass to

Child file processing

Child file processing has been made easier with the addition of the Report Child File extension template. This template allows for reads of parent and child files with a simple Extension template interface.



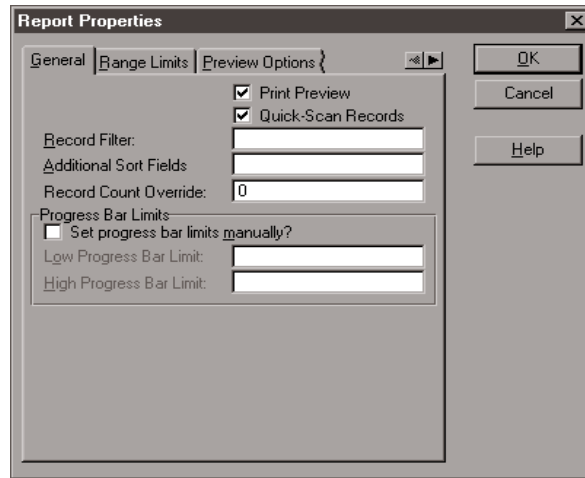
The template handles the retrieval of the child records, and places them into the specified band for a report procedure. The **Detail** dropdown list is disabled if the procedure is just a process.

TakeRecord method

The **TakeRecord** method in the **ThisProcess** object is the key to custom file handling in processes and reports. The **TakeRecord** method is called for every record retrieved, so the programmer has an opportunity to provide any file handling not available through the template interface. Place your custom code before the Parent call for best results.

Sorting

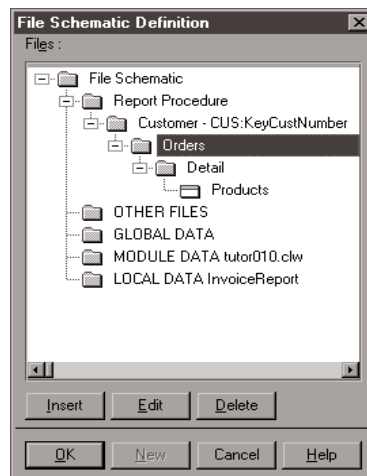
The ad hoc sorting abilities now inherent in Clarion Browse objects extends to processes and reports. To add a sort order that does not exist in a key simply navigate to the Report Properties dialog.



In the **Additional Sort Fields** entry field enter the fields, separated by commas, in the order in which you want the list sorted. The list will be sorted by key (if any) and then the additional sort fields.

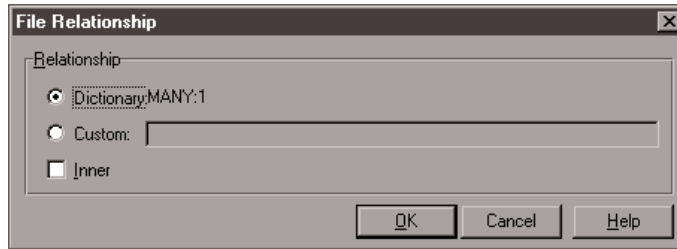
Joins

Inner and custom JOINS are now definable from within the IDE when using the ABC templates. To set the JOIN, highlight the secondary file in the file Schematic of the desired procedure, and press the **Edit** button.



The **VIEW** structure defaults to a "left outer join," where all records for the **VIEW**'s primary file are retrieved whether the secondary file named in the

JOIN structure contains any related records or not. Specifying the INNER attribute on the JOIN creates an "inner join" so that only those primary file records with related secondary file records are retrieved. Inner joins are normally more efficient than outer joins.



Custom and inner JOINS are now easily accomplished without any embedded code.

Just Reports

The following topics are specific to reports.

The ABC Print Previewer

The Print Previewer is now controlled by a Clarion object, The **PrintPreview** Class, which means better functionality and flexibility. Similar in appearance to the Print Previewer found in the Legacy Templates, this Previewer comes with some powerful built-in functionality.

Pages to print

The Previewer provides a menu item under **File** ► **Pages to Print...** that calls the Pages to Print dialog at runtime.



The default value is 1-n, where n is equal to the total number of pages in the report. Individual pages can be printed by separating page numbers by commas. A range of pages to print can be specified by separating the first page number to print and the last page number to print by a dash (-). Combinations of individual pages and ranges of pages are allowed.

PagesToPrint is also the label of the property that holds the value in the **Pages to Print...** dialog. This property can be preset before the Previewer is opened if you know that your end user will want to view a specific page, i.e. the last page of a report. To accomplish this, embed the following line of code in the **Previewer.Open** method of the report procedure.

```
Self.PagesToPrint = RECORDS(ImageQueue)
```

This sets the **Previewer** object to only display and print the last page of the report. The **ImageQueue** property is a reference to a queue containing the file names of the .WMF files created by the report which will be sent to the printer and/or previewed by the end user.

To set the **Previewer** object to display that page when it OPENS you must embed the following line of code in the **Previewer.Display** method before the Parent call:

```
InitCurrentPage = Records(ImageQueue)
```

InitCurrentPage is the label of a parameter passed to the **Display** method that indicates the page to display when the Previewer opens.

Conditional previewer

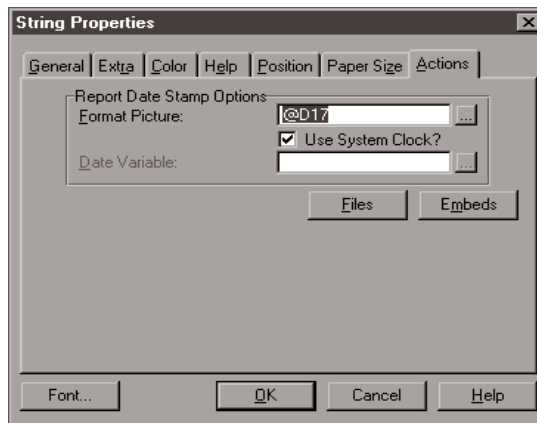
The configurability of objects is the primary benefit of Object Oriented Programming. This is most evident when setting a single property and achieving the desired behavior of the object. If you do not want to provide the Print Previewer to your end users, simply set the **SkipPreview** property to TRUE in the **Open** method of the **ThisReport** object..

```
SELF.SkipPreview = TRUE
```

This can also be combined with a conditional statement allowing for maximum runtime flexibility.

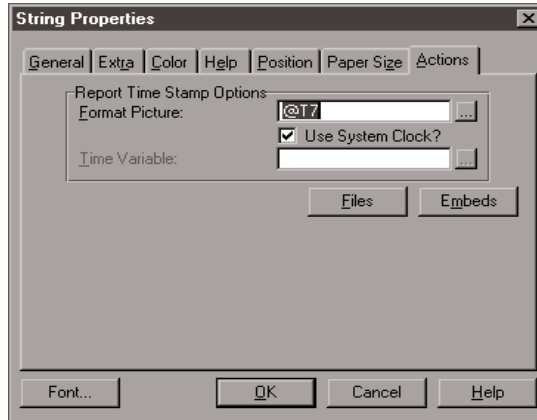
Date and Time

Printing the Date on a report is now as easy as placing a control template. The Report Date Stamp template is available to populate on reports, and is easily configurable through the template interface.



The Format picture is editable, and options are available for utilizing the System clock or a variable containing the desired date.

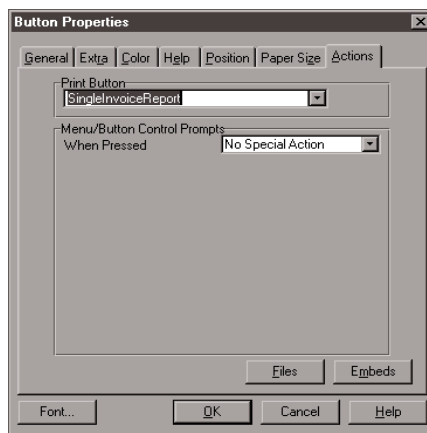
The Report Time Stamp template is also available providing a similar interface for displaying the time the report was created.



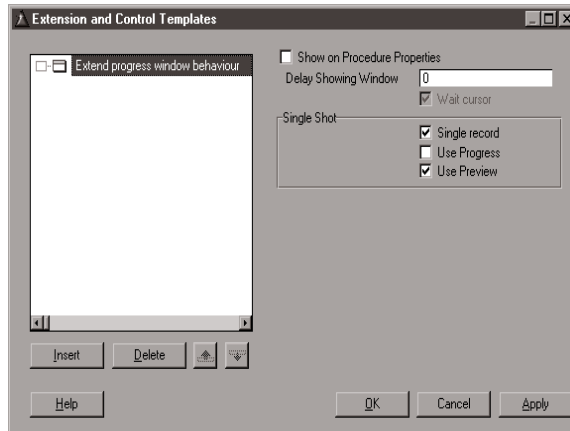
Single record printing

The ABCs contain Templates that enable a button on a Browse procedure to print the highlighted record. To incorporate this functionality, follow these steps:

1. Populate the Browse Print Button control template on the desired browse procedure.
2. Choose the desired report procedure from the **Print Button** Drop List on the Actions tab of the **BrowsePrint** button.



3. Populate the Extend Progress Window extension template on the chosen report procedure.



4. In the **Single Shot** group box make sure the **Single Record** box is checked. This report procedure will now only print a single record.

These four steps are \all that is required to set up a Print Record button.

Index

Symbols

.DLL	108
.WMF	139

A

ABC Development Environment	47
ABC Embed Points	51
ABC Embed Standard	55
ABC Embedding Methodology	51
ABC Translator Class	83
Acknowledgments	11
Adding and using your own error messages	73
Additional Sort Fields	137
Altering an Error Message for a Single Procedure	75
Altering the Severity of an Error Message	73
Altering the Text of an ABC Error Message	72
Application Conversion Process	22
Application Converter	21
Automatically find parent controls	116

B

Benefits of ABC templates	16
Benefits of Clarion Objects	14
Browse Print Button (template)	141
buffer management	93
BUFFER statement	99
BufferedPairsClass	92
business rules	98

C

Changing the Presentation of Error Messages	75
Clarion object, definition	14
Classes, defined	48
Conversion Hints and Messages	27
Conversion Options	25
Conversion, Options after	37
Conversion, reasons for	16
Conversion Rules	24

D

Data Integrity	101
Data validation	98
Defer opening files until accessed	103, 104
Derivation, in IDE	60
DLL	108
DOCKABLE	82
Don't Alter Controls	111

E

Edit-in-place	121
Embed Point, defined	50
Embed point, finding	62
Embed point, need for	62
Embed Standard, ABC	55
Embed Tree	53
Enclose RI code in transaction frame	101
Error Checking While in Stand-Alone Mode	76
Error Handling Techniques	71
ErrorClass	94
Extend Progress Window (template)	142
EXTERNAL	108

F

fat fetches	99
file sharing	103, 104, 107
Foreign Key	97

G

Getting Started Tutorials	13
Getting Started with ABC	19
Global Preservation	81

H

Horizontal Positional Strategy	115
Horizontal Resize Strategy	114
How Template Code is Customized when using the ABC	50
How the ABC Templates generate Clarion Objects	50
How to Use this Handbook	10

I

ImageQueue (property)	139
Inheritance, defined	48
INI File Management	79
Inner JOIN	137

J	
JOINs	137
K	
Key ABC issues covered	7
L	
LazyOpen	103, 104
Legacy Embed Points	51
Legacy Embedding Methodology	51
Legacy Embeds, Mapping	39
Legacy language replacements	37
Legacy Templates, Compatibility	21
M	
Macro Substitution	
TranslatorClass	117
Making Your Own Conversion Rules	29
Mapping Legacy Embeds to the ABC equivalent	39
Maximum Height	113
Maximum Width	112
MEMBER	108
Methods, defined	48
Minimum Height	112
Minimum Width	112
Minimum Window Size	112
N	
Naming Conventions	78
O	
Object, defined	48
Optimize Moves	116
Optimize Redraws	116
Override Control Strategies	113
P	
PagesToPrint (property)	139
Pause Button	135
Primary Key	97
Print Button	141
Print Preview	139
Priorities	59
Priority	53
Process	133
Process template	
RI constraints	102
Programming with Objects in Clarion	50
Progress bar	134
Progress Window	134
Property, defined	48
Q	
Quick-Scan Records	105, 106
R	
read-ahead buffer	99
Referential Integrity	101
referential integrity	
enforcement of	91
Referential Integrity (RI)	97
Relational Database design	97
RelationManagerClass	91
Report Child File (template)	136
Report Date Stamp (template)	140
Report Time Stamp (template)	141
Reports	133
Resize	111
Resize Strategy	111
resize strategy	
for a single control	113, 114
resize windows	110
Resizer Configuration Options	116
RI constraints	
Process template	102
S	
sharing files	103, 104
SkipPreview (property)	140
Spread	111
Stored Procedures	98
Surface	111
WindowResizeClass	110
T	
TakeRecord (method)	136
Techniques	
Adding and using your own error messages	73
Altering an Error Message for a Single Procedure (.....	75
Altering the Severity of an Error Message	73
Altering the Text of an ABC Error Message	72
Changing the Presentation of Error Messages	75
Error Checking While in Stand-Alone Mode	76
Error handling	71
Template	
Browse Print Button	141
Extend Progress Window	142

Pause Button	135
Report Child File	136
Report Date Stamp	140
Report Time Stamp	141
THREAD	107
TranslatorClass	109
macros	117
Triggers	98

U

Use RI constraints on action	102
Using the Embeditor	58

V

validation, data	98
Vertical Positional Strategy	115
Vertical Resize Strategy	114

W

What You'll Find in this Book	9
Why convert to ABC?	16
Window Message	134
WindowResize	110
WindowResizeClass	109
WMF	139

