Search

Clarion MAGAZINE

November 2000 Index

This time of year usually means DevCon, and that means a DevCon subscription special. We're not going to let a little thing like the lack of a Florida conference slow us down, so we're announcing the No DevCon 2000 Subscription Special. Purchase a one or two year subscription and buy all 20 back issues of Clarion Magazine for only $82 more! That's over 30% off the regular back issue price! We'll also take an additional $10 off if you buy a two year subscription..
(Nov 28,2000)

Making Sense of ABC's ErrorClass - Part 2

If there's one part of ABC that consistently draws fire, it's ErrorClass. In this three part series, Russ Eggen puts on his flak jacket and steps out into no man's land to explain what ErrorClass is all about, why it's good, and what you can do with it.
(Nov 28,2000)

Profiler/Debugger Tools Update

The CCI Profiling/Debugging tools have been updated to version 1.2, with an easy to use template. Just drop in the global extension and go!
(Nov 28,2000)

November 2000 News

Clarion news, notes, and happenings from around the globe.
(Nov 28,2000)

Clarion MAGAZINE

Conditional Sort Orders and Page Breaks in Reports: Part 1

by Steve Parker

Recently I needed to do a "receiving report," a listing of items received on a given date. Initially, I thought this would be very straightforward: group by purchase order (P.O.), set up a group header for P.O. control data, a footer for group totals, filter on date received and off to the races.

It’s *never* that easy. I should have known better.

Some of our customers don’t cut purchase orders but still want a report of what they’ve purchased and received. Some make spur of the moment purchasing decisions when the sales rep visits and, of course, revising a completed P.O. makes no sense (and is bad accounting practice too).

So, I have to be able to receive and report with or without a P.O.

Some customers want to attach a copy of the receiving report to the P.O. They therefore think that they require one P.O. per page. Some are environmentally aware and don’t want the page breaks (i.e., they’ve figured out how to use scissors to separate the sections for multiple purchase orders).

Some want grand totals. Some don’t.

Within a P.O., some want the items sorted by PLU (Price Lookup Unit or UPC code), some prefer part number.

Subtotals by P.O. are, of course, a given. But because I may be using one of two sort orders, I can’t set a key (obviously, I’m going to use an ABC method that sets the sort order and none of those accept keys or use them if entered, only variables or lists of fields). Without a key, how do I do the required group breaks?

I have one filter (receive date), one printing option (grand totals, yes or no), one layout option (one P.O. per page, yes or no), P.O. subtotals and two possible sort orders within a given P.O. I have to accommodate all of these within a single report procedure (unless I absolutely must, I do not want to create multiple report procedures – maintenance nightmare in the making, you understand) and I have to do these subtotals without a matching key.

And, just for fun, a little twist: once a user makes a set of selections, I want to remember and re-use them. I certainly don't need users complaining about having to re-enter their standard choices.

The secret to most of what I need lies in knowing how a Clarion report is structured and the judicious use of a few embeds.

The nature of a Clarion report

On its face, a Clarion report is a very complex thing. Fixing appropriate positions for the various possible bands is sometimes an exercise in … well, let's just say that it isn't always a pleasant experience. For example, setting the report's position from the Report Properties worksheet actually sets the area of the page used for the detail area. And the detail area includes group headers, actual details and group footers but not the page header or footer. It's somewhat counter-intuitive.

The developer has no control over page overflow in the sense that there are no embeds (as there were in DOS) for group headers/footers and the page footer. There are also no reliable ways of knowing when overflow is going to occur. *That's* an important point.

In fact, most of a report seems to lay inside this enormous black box commonly known as "the report engine," with which one or two Clarion developers have been know to take exception on occasion.

Thinking of a report as a black box, however, actually depends on another way of describing Clarion reports. This other view may be described as the "code view" of reports and, in the code view, a report is really an extraordinarily simple entity: A report is a Process with a `Print()` in the middle. Or,

> **A report is a Print() in a loop.**

Put this way, I find reports are much less intimidating. This might be oversimplifying a bit, but it's a useful concept.

`Print()` simply sends a report structure to the active Window's printer device, which may be a named printer, a file or even a modem. Valid structures which can be arguments in the `Print()` statement are detail bands, group headers/footers and page headers/footers. In short, you can `Print()` almost any report entity that has a Label.

So, it *is* entirely possible to force a new page *whenever* desired by embedding `Print(RPT:Footer)`.

> **NOTE:** The `Print()` statement takes the structure's Label with prefix, not its Use variable. However report structures still require a use variable.

Because the page footer is part of the overflow calculation in that great, huge black box, page overflow *will* occur when the footer is printed.

The important point is that this illustrates my ability to force something to print even though I do not know what is going on inside the print engine. This is important enough to repeat: I can place a band on a report, a band of any type whatsoever, and I can make it print when I want it to, under conditions I set and check and control.

This notion leaves but one minor issue: Where do I determine what conditions are being met?

Report embeds

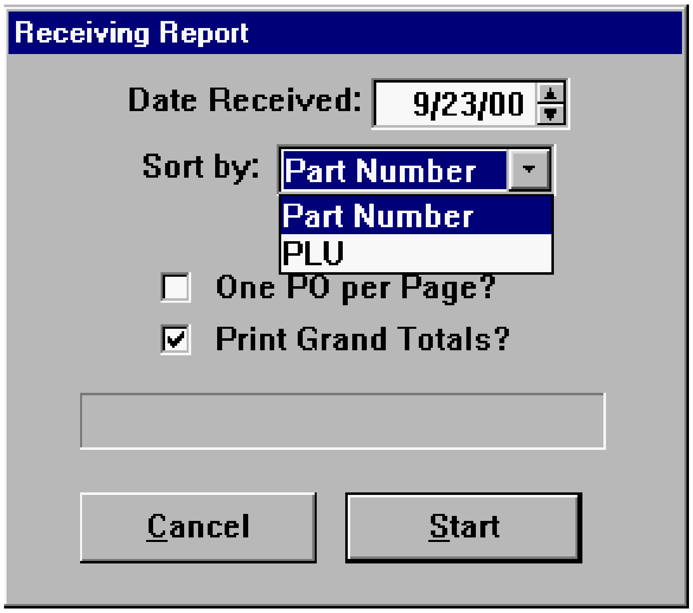
If a report is a Process wrapped around a `Print()` then the important embeds are almost exactly the same as in a Process type procedure.

The embeds I find myself using most often are few in number.

`ThisWindow.Init`: This embed is used in reports in exactly the same way as in any other procedure. Any setup or preliminary work I need to do is done here. However, because the progress window is available and because it is opened before the process/report, if I need to capture variables, for example to prime a filter, I *can* use the progress window (combined with the *Pause the Process* code template).

While I still have the option of creating a separate procedure to capture variables, I also have the option of using the progress window:

Figure 1. Using the Progress Window to capture variables


Receiving Report

Date Received: 9/23/00

Sort by: Part Number

Part Number
PLU

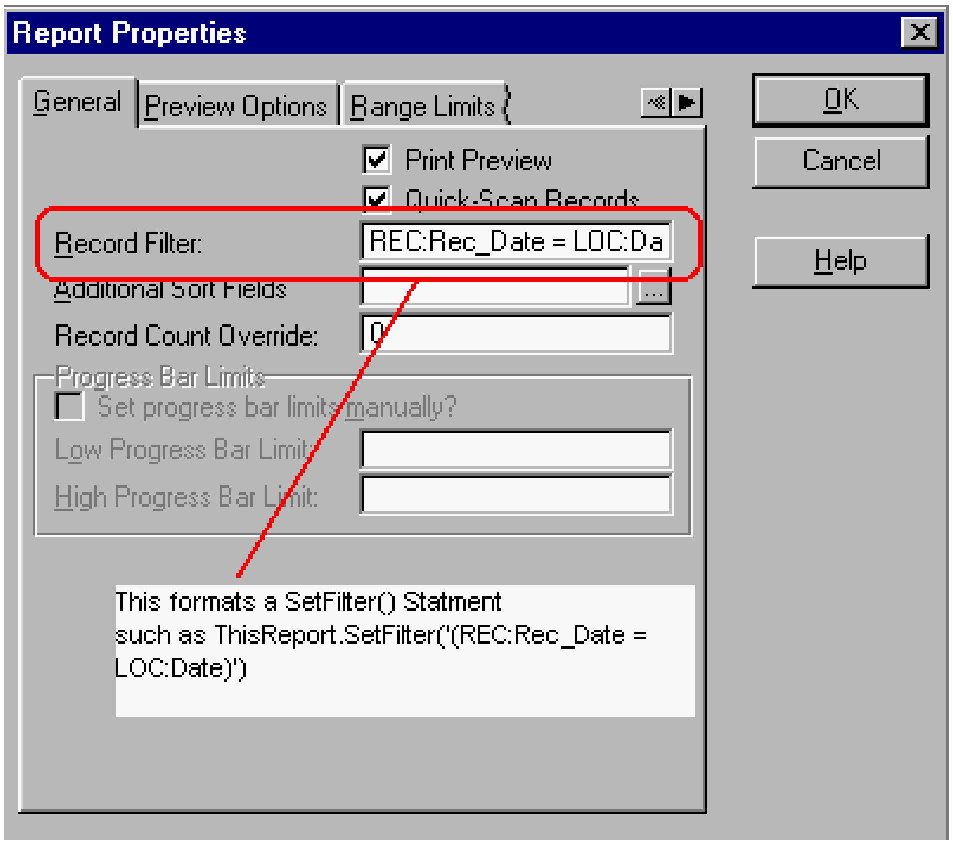
☐ One PO per Page?

☑ Print Grand Totals?

One virtue of using the progress window is that the variables entered by the user inherently remain visible. So if the user gets a report based on incorrect input, there’s no excuse for not having cancelled.

`ThisProcess.ValidateRecord`: As in any ABC procedure, I can test records for inclusion/exclusion. Conditions tested here are in addition to any range or filter conditions entered in the Report Properties worksheet:

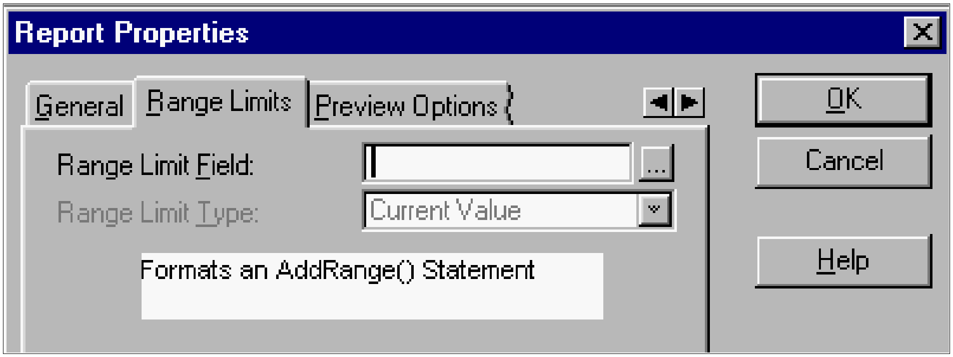
Figure 2. Standard report filters



☑ Print Preview
☑ Quick-Scan Records
Record Filter: REC:Rec_Date = LOC:Da
Additional Sort Fields
Record Count Override: 0
☐ Set progress bar limits manually?
Low Progress Bar Limit:
High Progress Bar Limit:

This formats a SetFilter() Statment such as ThisReport.SetFilter('(REC:Rec_Date = LOC:Date)')

Figure 3. Standard report ranges



Range Limit Field:
Range Limit Type: Current Value

Formats an AddRange() Statement

The typical type of code used in ValidateRecord, before Parent Call looks like this:

```
If <expression>
  Return Record:Filtered
End
```

This code excludes a record if `<expression>` is true.

Validation code can be quite elaborate, as shown in Figure 4:

Figure 4. Sample complex validation code

```
 If ~PRO:NoMoreChecks
   If PRO:ContactDate and (PRO:ContactDate <= Today())
     Return Record:OK
   End
 End
 If LEA:ExpDate
```

```
    If ~LEA:EDateCheck and (LEA:ExpDate - Today() <= 120)
      Return Record:OK
    End
  End
  If LEA:NoticeDate
    If ~LEA:NDateCheck and (LEA:NoticeDate - Today() <= 30)
      Return Record:OK
    End
  End
  Return Record:Filtered
```

This code will include a record if any one of three conditions are met. Otherwise, the record is excluded. In this case, I've made a sort of reverse filter (filters normally exclude records; this code includes them).

`ThisWindow.TakeNoRecords`: Many Clarion developers want to present their own "No Records" message, instead of the generic one integrated into the Process Class (see Template Writing Made Easier) or no message at all. `TakeNoRecords`, before Parent Call is where this is done.

If a report is not to be previewed, but sent directly to a printer or file, `TakeNoRecords` can be used to print a no-records band (details in a moment) and can be setup to so that its message is not presented (with a simple `Return` before the Parent Call).

`ThisWindow.AskPreview`: This embed is unique to Report procedures and it is quite important.

The `AskPreview` method is called even if the preview option is *not* selected (don't ask). What is even more important about this method is that it is called after all records have been processed (i.e., after the loop has terminated) but before the report is closed.

This means that anything I decide to print from this embed will print after all other bands, details, records, etc. This explains why a detail band reserved for grand totals or report summaries should be called from this embed.

It is also important to note that `AskPreview` will be called even if the report is aborted (the Cancel button is pressed) or if there are no records. For this reason, I always declare a local variable,

```
Aborted Byte
```

and set it to "1" in the Cancel button, Accepted embed. Then, any code in `AskPreview` is wrapped in this code:

```
If ~Aborted
  Print(RPT:GrandTotals)
End
```

to protect myself from unforeseen printing and user confusion.

In a report that goes straight to a printer, it is valuable to indicate that the user cancelled on the printout or there were no qualifying records, as in Figure 5.

Figure 5. Printing either totals or a cancelled message

```
If ~Aborted
  If Counter               !Qualifying records were found
```

```
    Print(RPT:GrandTotals)
  Else                       !No records
    Print(RPT:NoRecords)
  End
Else                         !Cancelled
  Print(RPT:CancelledBand)
End
```

In this case no one can think the report is complete or empty when it isn't, as the case may be.

This embed, as discussed in "Template Writing Made Easier," is also appropriate for conditionally enabling or disabling print preview.

`ThisProcess.TakeRecord`: This is the *really* important embed.

In a report, `TakeRecord` is a little bit different than in other procedures. In other procedures, `TakeRecord` is where the record is read. In other procedures, `TakeRecord`, before Parent call is immediately before the record is read; after Parent Call is immediately after the record is read but before anything further is done with it.

In a report, `TakeRecord` behaves in just this way but has some additional functionality, very important additional functionality. `TakeRecord` is immediately before the `Print()` statement or, more accurately, the `Print()` statement is just after the Parent Call in `TakeRecord`. TakeRecord, Priority 5001 is immediately after the record read and immediately before the printing of the detail band(s). Priority 8000 is immediately after the detail band is printed.

When you think about it, this makes sense: first read the record, then give the developer a chance to work with it (calculate an extended amount from a cost or price and a quantity or concatenate a name or address, for example) and, finally, print it.

In Part 2 of this article, I'll show you how to put all of this code together.

Steve Parker started his professional life as a Philosopher but now tries to imitate a Clarion developer. A former SCCA competitor, he has been known to adjust other competitors' right side mirrors - while on the track (but only while accelerating). Steve has been writing on Clarion since 1993.

Main Menu | Log In | Subscribe | Links | FAQ | Advertising

Reborn Free CLARION online

published by CoveComm Inc.

Search

Clarion MAGAZINE

Main Page

COL Archive

Log In
Subscribe
Renewals

Frequently Asked
Questions

Site Index
Article Index
Author Index
Links To
Other Sites

Downloads
Open Source
Project
Issues in
PDF Format
Free Software

Advertising

Contact Us

The Clarion Advisor: Keep Those PDFs Handy!

Conditional Sort Orders and Page Breaks in Reports: Part 1
(Nov 7,2000)

The Clarion Advisor: Keep Those PDFs Handy!
(Nov 7,2000)

The Clarion Challenge: Editor Macros
(Nov 7,2000)

Not everyone is happy with the decision to make the printed docs an extra cost item in Clarion 5.5, but if you've decided to stick with the PDF documentation, you might as well keep it close to hand. It's easy to add the PDFs to a user menu in the Clarion IDE, as demonstrated in this tip from Andrew Guidroz.

Make a copy of your c55ee.ini file before making any changes. In case something goes wrong, you want to be able to set things back the way they were.

You'll need to make two changes to c55ee.ini in order to open the PDFs from inside the Clarion IDE. I'll summarize the changes here, but if you want an overview of how to go about customizing the IDE, see John Morter's article on the subject. Although that article was written for C5, the technique applies to C5.5 also.

Inside c55ee.ini you'll need to made additions to two sections, [User Applications] and [User Menus]. Those changes are shown below in bold. Where the line wraps on this page I've used a line continuation character, but you want each entry on its own line.

```
[User Menus]
_version=41
1=Data Modeller|DIC5
2=&Docs/&Handbook|AH
```

```
[User Applications]
_version=41
CWRW=c55rw %f %a
DIC5=DM5.EXE
AH=c:\Progra~1\Adobe\Acroba~1.0\Reader\AcroRd32.exe↵
 c:\C55\Doc\55abc1.pdf c:\c55\doc\55abc2.pdf
```

The **AH** is the key that links the application entry with the menu entry. You may need to edit the AH= line to suit the path to acrord32.exe. Note also that you need to use the 8.3 version of the file path because the environment is 16 bit.

Search

Main Page

COL Archive

Log In
Subscribe
Renewals

Frequently Asked
Questions

Site Index
Article Index
Author Index
Links To
Other Sites

Downloads
Open Source
Project
Issues in
PDF Format
Free Software

Advertising

Contact Us

The Clarion Challenge: Editor Macros

Conditional Sort Orders and Page Breaks in Reports: Part 1
(Nov 7,2000)

The Clarion Advisor: Keep Those PDFs Handy!
(Nov 7,2000)

The Clarion Challenge: Editor Macros
(Nov 7,2000)

Okay, so the Clarion editor doesn't have the most sophisticated macro handling on the planet. But at least in C5.5 you can finally save macros between sessions, and that makes the macro feature a lot more useful. So this month we're asking for your macros.

Punch them in, write them up, and send them down the 'pike to editor@clarionmag.com. We'll feature the best submissions in a December issue.

Looking for inspiration? Check out James Cooke's excellent article on the subject. Even before you could save macros, James found ways to use them productively.

Read the Clarion Challenge results.

Main Menu | Log In | Subscribe | Links | FAQ | Advertising

Main Page

COL Archive

Log In
Subscribe
Renewals

Frequently Asked Questions

Site Index
Article Index
Author Index
Links To Other Sites

Downloads
Open Source Project
Issues in PDF Format
Free Software

Advertising

Contact Us

Conditional Sort Orders and Page Breaks in Reports: Part 2

by Steve Parker

In Part 1 of this article, I discussed some strategies for using report embeds to conditionally control printing. Now it’s time to put these into practice, beginning with grand totals.

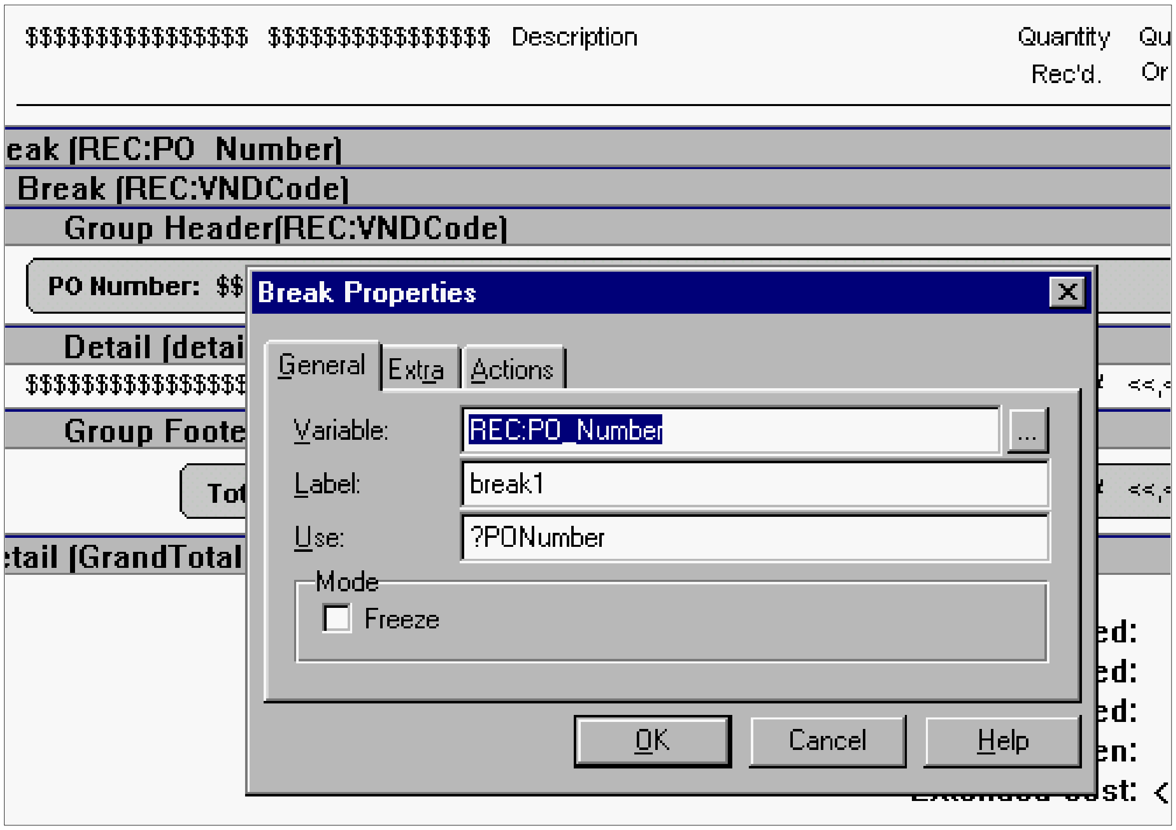
Grand totals

Printing grand totals, a case in point, is not especially difficult when these embeds are understood.

First, create a band in which the total fields can be populated. Press "Bands" on the Report Formatter’s Main menu, then click "Detail." Place the cursor on an existing detail band and click. A new band appears below the existing detail. (If the order of the bands is not to your liking, use the ellipsis on the main Properties worksheet, the page layout view or F12 in the formatter to rearrange them.)

Right click on the new band to get its Properties Worksheet and make sure it has a Label (preferably one that is easily remembered) and Use variable:

Figure 6. Band properties



Break Properties

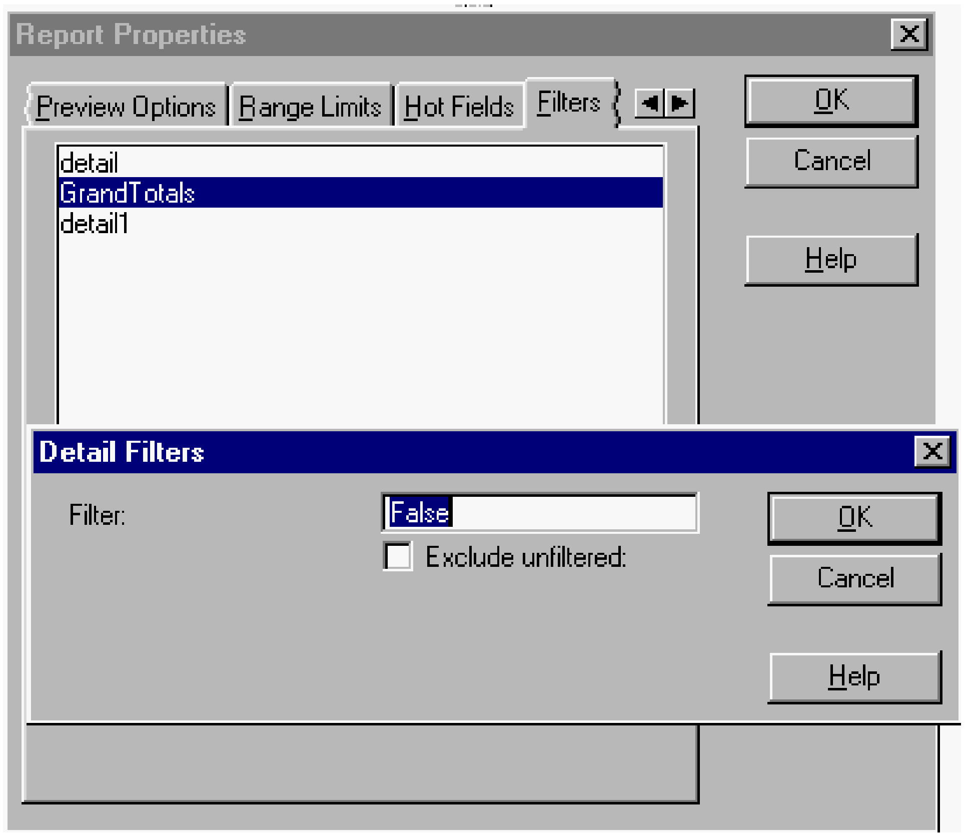
General | Extra | Actions

Variable: REC:PO_Number
Label: break1
Use: ?PONumber
Mode
[ ] Freeze

At this point, if you populate variables in the new band and run the report, the report will print both bands sequentially. This is not what is needed. So it is imperative to stop the new band from printing until I want it to print, as I explained in Part 1.

The easiest way to accomplish this is from the main Procedure Properties worksheet: click Report Properties and scroll to the Filters tab. Select the total band, press "Properties" and set its filter to "False:"

Figure 7. Preventing a band from printing



Filter: False
Exclude unfiltered: [ ]

Now the band will not print unless and until I explicitly execute code to tell it to print. Since this particular band is intended to display report totals, ThisWindow.AskPreview, before Parent Call is the appropriate place, after all the records have been read but before the report is closed:

```
If ~Aborted
  Print(RPT:GrandTotals)
End
```

or

```
If ~Aborted
  Print(RPT:GrandTotals)
Else
  Print(RPT:NoRecords)
End
```

as necessary. Notice that I have not discussed whether this total band should be populated with Total Fields or manually accumulated fields. This is not material.

What *is* material is the technique of creating a band and preventing it from printing except under developer control:

1. Create a new detail
2. Set its filter to "False"
3. When the time is right, print it.

In other words, the technique to print grand totals isn't limited to grand totals, not at all. Any band can be printed when conditions occur that I, as the developer, decide should cause it to print.

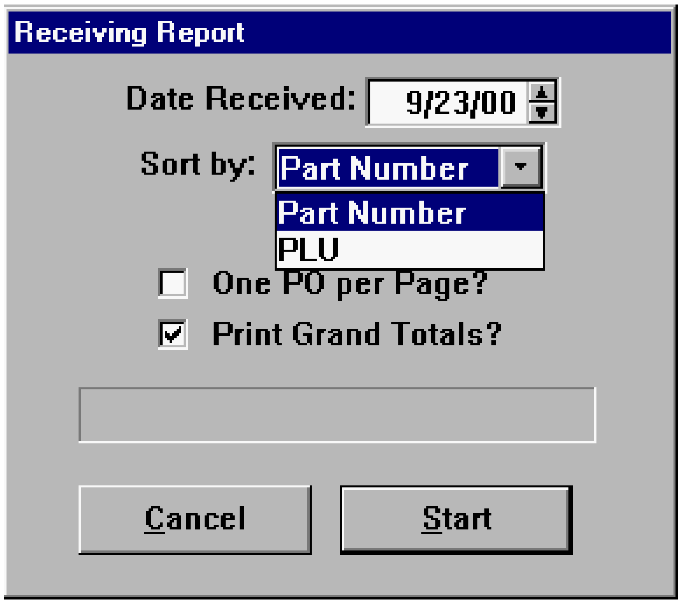
This is the secret of using the print engine without further challenging your hairline.

The receiving report

With this background, the receiving report turns out to be, if not 100% straightforward, at least not quite as difficult as it first appeared to be.

I know that I need to get the date and the desired sort order, whether or not the user wants one P.O. per page and whether or not grand totals are wanted.

Figure 8. Using the Progress Window to capture variables



As shown in Figure 8, I use the Progress Window (with the Pause template) to capture these. And, because I want to save and restore previous choices, I use an INI file. In `ThisWindow.Init`, after Open Files, I add the code shown in Figure 9.

Figure 9. Initializing report selections

```
LOC:Date = Today()
OnePOPerPage = GetINI('RecRpt','PageBreak',,↵
  Clip(gFil:abDataPath) & 'abrec.ini')
DoGrandTotals = GetINI('RecRpt','Totals',,↵
  Clip(gFil:abDataPath) & 'abrec.ini')
?List1{PROP:SelStart} = 1
```

This code primes the filter (`REC:Rec_Date = LOC:Date`), restores previous choices, if any, for one per page and grand totals and sets the first item in the sort order drop list to "Part Number."

If the user does not abort the report, in the `AskPreview` method, right after the grand totals are printed (or not), I save the current selections:

```
If ~Aborted
 PutINI('RecRpt','PageBreak',OnePOPerPage,↵
   Clip(gFil:abDataPath) & 'abrec.ini')
 PutINI('RecRpt','Totals',DoGrandTotals,↵
   Clip(gFil:abDataPath) & 'abrec.ini')
End
```

In the receiving report, printing grand totals depends on whether or not the user cancelled the report *and* whether or not grand totals were requested, so:

```
If DoGrandTotals and ~Aborted
  Print(RPT:GrandTotals)
End
```

So far, not too difficult and only two things are left to do: set the report's order and handle one/page vs. tree-hugging.

One per page, or not

Carl Broll, in a FAQ in my knowledge base, says:

> Assuming that you have a group break for the department, you can set or clear PROP:PageAfter of the group footer.
>
> For example:
>
> ```
> SETTARGET(Report)
> IF UserWantsDepartmentsSeparated
>   ?DepartmentBreakFooter{PROP:PageAfter} = True
> ELSE
>   ?DepartmentBreakFooter{PROP:PageAfter} = False
> END
> SETTARGET
> ```
>
> Actually, if you leave the group footer's "Page after" checkbox unchecked in the AppGen report formatter you should only need:
>
> ```
> SETTARGET(Report)
> IF UserWantsDepartmentsSeparated
>   ?DepartmentBreakFooter{PROP:PageAfter} = True
> END
> SETTARGET
> ```

In turn, this leaves two issues. First, it seems I need a break to do this. Second, where do I put the code?

Since the first issue involves the sort order and I haven't looked at that yet, let's consider the "where" issue.

"Where?" turns out to be simply solved based on the fact that a Clarion report is actually a data structure, just like a Clarion window. Therefore, no changes made before the structure is open will be effective (nor will changes made after the structure is first accessed). So WindowManager.OpenReport, after Parent Call is the logical place for the code in Figure 10.

Figure 10. Final page break code, after Open Report

```
If ~Aborted
  SetTarget(Report)
  If OnePOPerPage
    ?PO_Nbr{Prop:PageAfter} = True
  Else
    ?PO_Nbr{Prop:PageAfter} = False
  End
  SetTarget
End
```

An alternate

I don't necessarily have to set a property to trigger a new page. The previous discussion should make this clear.

If I know that the user wants a new page for each P.O. and I know that the P.O. number has changed, I know I can use `Print(RPT:Footer)` to force page overflow. All I need to know is when the P.O. number changes. Simple.

Actually, this time, it *is* simple.

Since I know that `TakeRecord`, Priority 5001 is after reading the next record, it follows that the code in Figure 11 should do the job.

Figure 11. Manually checking for breaks

```
If OnePOPerPage              !Wants one per page
  If FirstLoop               !if first record, prime SaveNumber
    SaveNumber  = REC:PO_Number
    FirstLoop = 0
  End
  If SaveNumber <> REC:PO_Number  !if number has changed
    Print(RPT:Footer)             !Print footer and overflow
  End
  SaveNumber = REC:PO_Number
End
```

In this particular report, the first two nodes of the sort order are fixed and the same field is always used for the break. Either method for forcing a new page should work perfectly well. Your choice.

On the other hand, if the nodes of the order where variable, if different fields could be at the top nodes, both methods would still work but a lot more code would be required. I suspect that the method shown in Figure 11 will tend to be a bit more comfortable to those of us not yet fully accommodated to our ABCs. If a band is simply to be printed and a new page is not desired, Figure 11's code is the way to go.

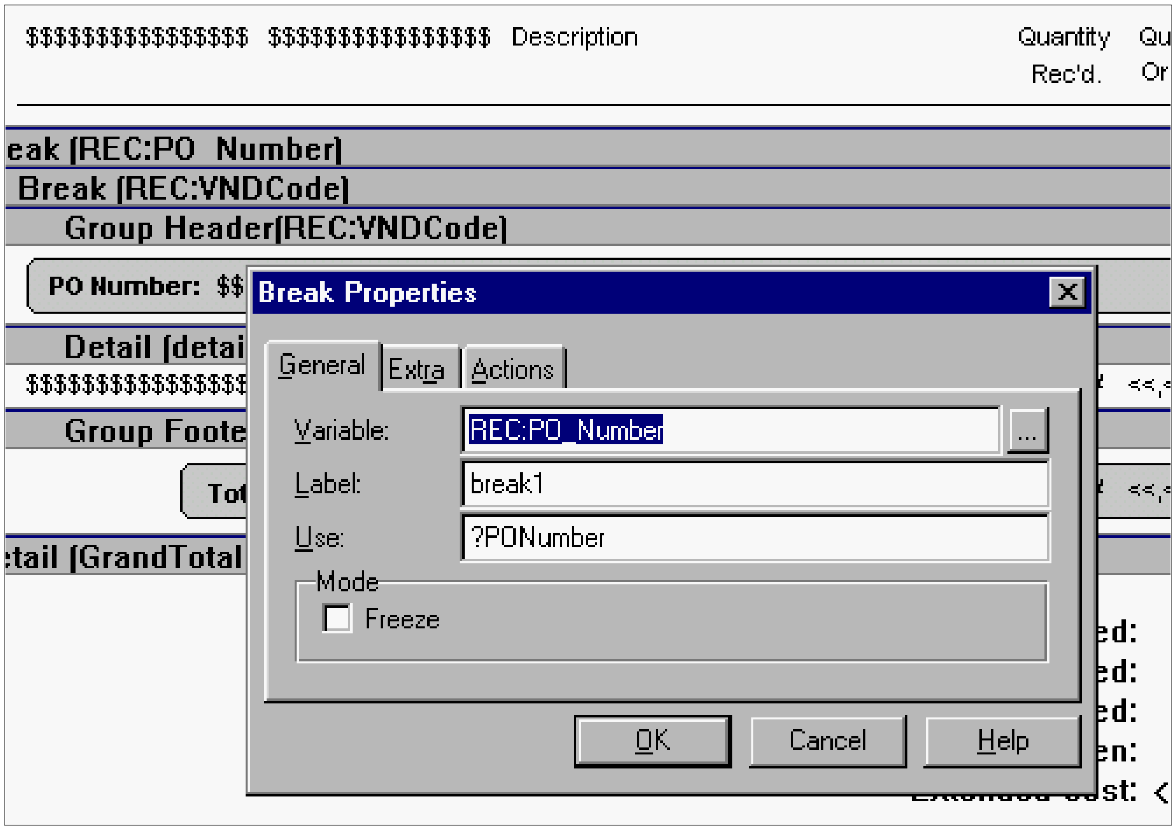
Setting the report order

What started out as a simple choice between Part Number order and PLU (UPC Code) order turns out not to be so simple after all. If I want to have conditional page breaks, I also need a break on P.O. Number. Whether or not I use the `PageAfter` property, I still have to sort on P.O. Number in order to know when there's been a change in that datum.

Either way, I have to sort on P.O. Number.

On reflection, that's not such a hard thing after all. It means that while the user chooses between Part Number and PLU (UPC code) in the drop down, my code will choose between `'REC:PO_Number,REC:PartNumber'` and `'REC:PO_Number,REC:PLU'` strings for sort orders. This gives me the break I need on the P.O. Number and, fortunately, when populating break bands, the report formatter no longer insists on a key, as it did in the DOS version.

Figure 12. Set a break on any variable



Variable: REC:PO_Number
Label: break1
Use: ?PONumber
Freeze [ ]

Because I am controlling the sort order, I am in control over when the break variable changes. Declaring a break field in the absence of a key in the file schematic is not, *in this case*, a potential cause of error. (If you cannot or do not reliably predict when a variable may change, you invite disaster.)

Actually, since it seems I will use the drop list selection simply as a trigger then I can refine the sorting even further, as deep as necessary for my purposes.

And, in fact, on analysis I do want another sort node, on vendor.

While a P.O. cannot have more than one vendor, I have to report on non-purchase order receipts. Non-PO receiving could involve multiple vendors. If I add a vendor node beneath the P.O. Number node and use the vendor as the break for printing subtotals, nothing changes as far as standard purchase orders are concerned; the Number-Vendor combination will print its subtotal break at the same point as P.O. Number-only break. But, if I break on vendor and there is no purchase order, non-PO receiving breaks on each vendor.

Now each non-PO vendor acts like a purchase order.

First, however, I need to ensure that all non-PO's (which, by default, have `REC:PO_Number = 0`) display acceptably. In `TakeRecord`, after Parent Call, 5001, I add the code from Figure 13.

Figure 13. Displaying P.O. numbers

```
If REC:PO_Number = 0
  LOC:POString = 'Non-PO'
Else
  LOC:POString = REC:PO_Number
End
```

In Figure 13, `LOC:POString` is what is actually populated on the report.

Aesthetics taken care of, I can setup the desired order in Open Report, before Parent Call:

```
ThisReport.AppendOrder(Choose(Choice(?List1), |
'+REC:Rec_Date,+REC:PO_Number,+REC:VNDCode,+REC:PartNumber', |
'+REC:Rec_Date,+REC:PO_Number,+REC:VNDCode,+REC:PLU'))
```

(This code is inspired by the People example distributed with Clarion 5.5.)

The `SetOrder` method will also work correctly here.

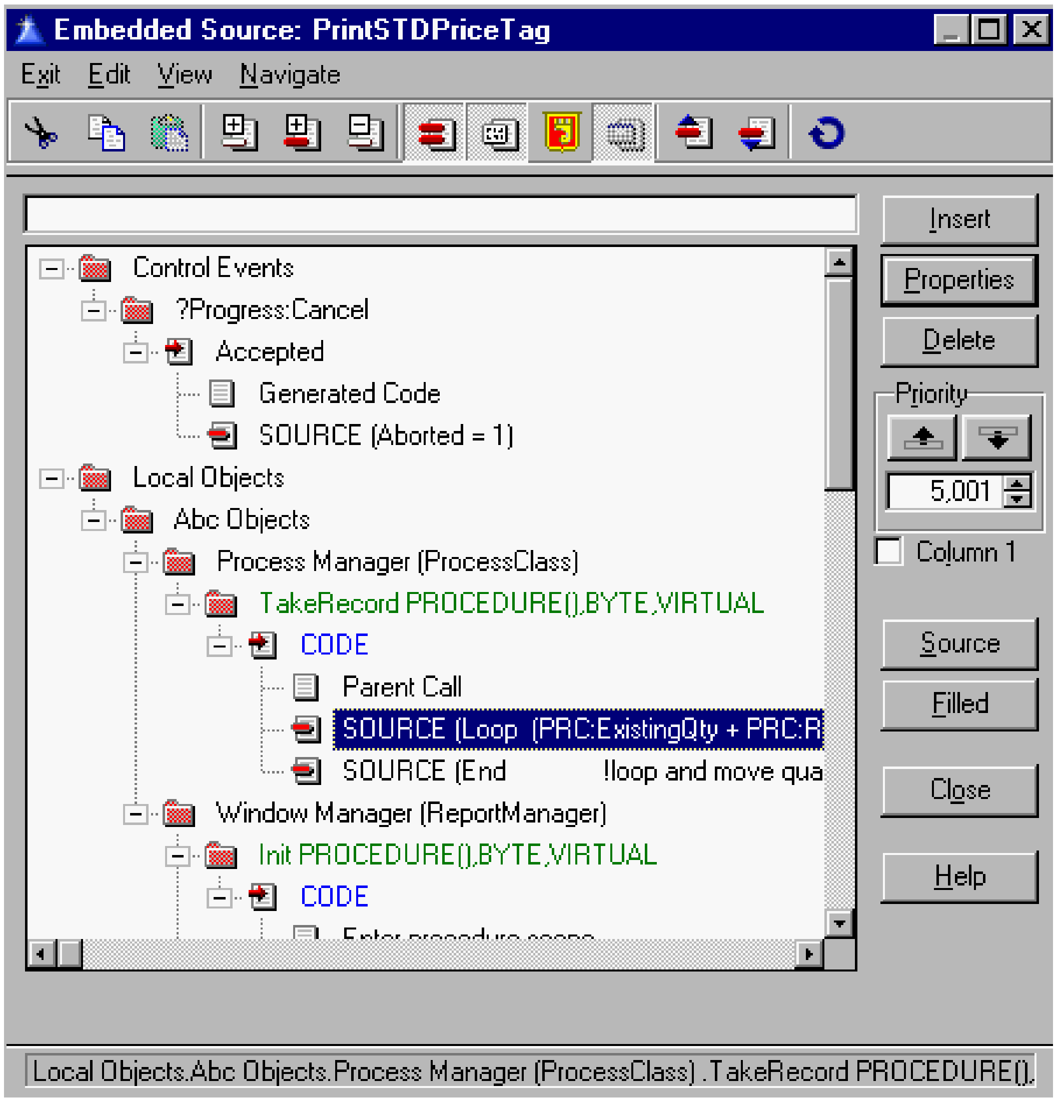
Note that I added the date as the top node. Because the `AppendOrder` and `SetOrder` methods set the sorting on the underlying View and I need to filter by date, setting the date as the top node will speed record processing. While using either of these methods, even with strings exactly matching declared keys, isn't as fast as using a key, it is much faster than using a brute force, a.k.a. simple, filter (see "Order! Order! Order in the Files," Clarion Online, 2, 10, May 1999).

Digression

The following digression has nothing to do with the receiving report, but it's too good not to mention, you see.

I have an embed immediately before the detail is printed and another embed immediately after. That means that I can put the `Print()` in a loop.

Figure 17. Print() in a loop



```
Control Events
  ?Progress:Cancel
    Accepted
      Generated Code
      SOURCE (Aborted = 1)
Local Objects
  Abc Objects
    Process Manager (ProcessClass)
      TakeRecord PROCEDURE(),BYTE,VIRTUAL
        CODE
          Parent Call
          SOURCE (Loop  (PRC:ExistingQty + PRC:R
          SOURCE (End           !loop and move qua
    Window Manager (ReportManager)
      Init PROCEDURE(),BYTE,VIRTUAL
        CODE
          Enter procedure scope
```

Local Objects.Abc Objects.Process Manager (ProcessClass) .TakeRecord PROCEDURE(),

Why ever would I want to do this? How about to print enough labels for all my inventory items?

This is a really useful consequence of the report-as-`Print()`-in-a-loop point of view.

Summary

The report engine may be a black box and this may cause some aggravation. By and large, that impacts aesthetics. Function, the *sine qua non* of application programming, is still very much code controlled.

With a few simple embeds and a firm control over your data, you can break where you want, start new pages when you want and, generally, have a grand time wowing users. By extending the technique discussed here, it is entirely possible to set any number of breaks on any number of fields even when those fields and the desired order are not known until runtime.

How? That's another story, another article.

Steve Parker started his professional life as a Philosopher but now tries to imitate a Clarion developer. A former SCCA competitor, he has been known to adjust other competitors' right side mirrors - while on the track (but only while accelerating). Steve has been writing on Clarion since 1993.

Reborn Free CLARION online
published by CoveComm Inc.

Search

Clarion MAGAZINE

The Clarion Advisor: List Totaling

Conditional Sort Orders and Page Breaks in Reports: Part 2
(Nov 14,2000)

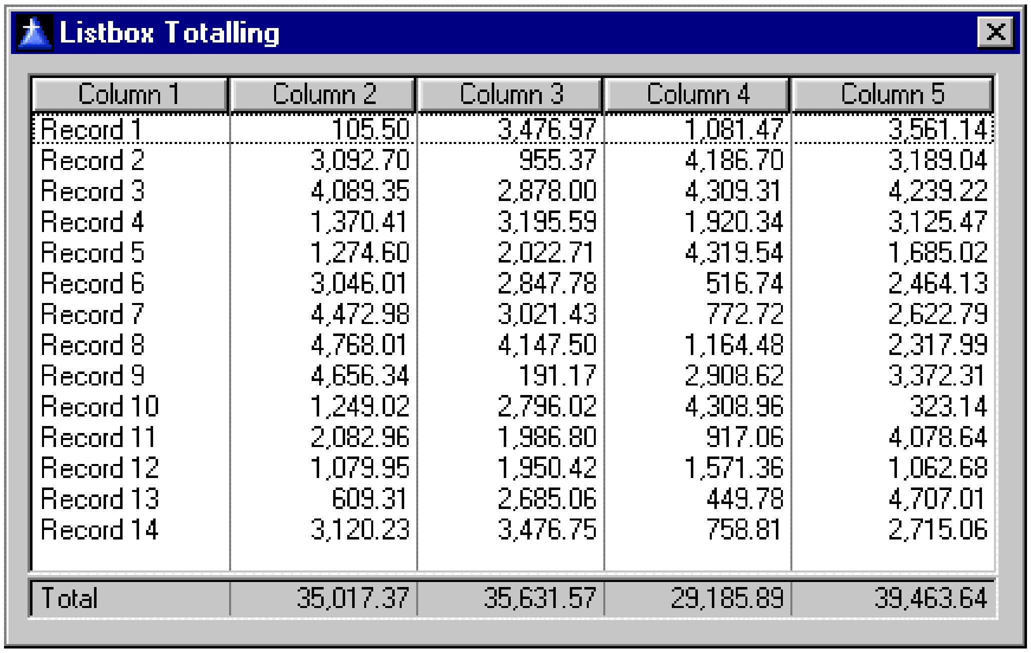
The Clarion Advisor: Displaying List Totals
(Nov 14,2000)

by Dave Harms and Jeff Slarve

One of the tricks of productive programming is to not let yourself be limited by the standard ways of solving a particular problem. Consider a situation where you want to display column totals underneath a list. If you’re like me, you think about doing this with strings displayed below the list. If you’re like Jeff Slarve, you think "Why not use another list?" That seems like a lot less hassle than having to deal with a bunch of strings.

Figure 1 shows a small program that does this (with data stored in a simple queue, not a browse box). The interesting point about this application is that if you resize the columns, the totals display automatically lines up with the new column positions.

Figure 1. Automatic column totals



Column 1	Column 2	Column 3	Column 4	Column 5
Record 1	105.50	3,476.97	1,081.47	3,561.14
Record 2	3,092.70	955.37	4,186.70	3,189.04
Record 3	4,089.35	2,878.00	4,309.31	4,239.22
Record 4	1,370.41	3,195.59	1,920.34	3,125.47
Record 5	1,274.60	2,022.71	4,319.54	1,685.02
Record 6	3,046.01	2,847.78	516.74	2,464.13
Record 7	4,472.98	3,021.43	772.72	2,622.79
Record 8	4,768.01	4,147.50	1,164.48	2,317.99
Record 9	4,656.34	191.17	2,908.62	3,372.31
Record 10	1,249.02	2,796.02	4,308.96	323.14
Record 11	2,082.96	1,986.80	917.06	4,078.64
Record 12	1,079.95	1,950.42	1,571.36	1,062.68
Record 13	609.31	2,685.06	449.78	4,707.01
Record 14	3,120.23	3,476.75	758.81	2,715.06

Total	35,017.37	35,631.57	29,185.89	39,463.64

How much code does it take? Aside from declaring an instance of the class that manages this behavior, and a duplicate of the original list queue, two lines:

```
LT.Init(?List1,?List2)
```

and

```
LT.TakeEvent
```

where `LT` is the instance of the list totaling class. Oh, and you also have to create a list box for display of the totals.

You can download the code at the end of this article. It consists of sample application (one clw file and a prj) which contains the list totaling class and a demo procedure.

To initialize the list box totaling class, just pass in the FEQ for the list with the columns to be totaled, and the FEQ for the list to hold the totals. That second list will store just a single record with the column totals, and it gets its appearance from the data list via the property syntax:

```
Self.TotalList{PROP:Format} = Self.InList{PROP:Format}
```

There's some other minor copying of attributes like length and x position, as well as the total list's distance below the data list. That saves fine tuning at design time and ensures consistency of appearance.

The `TakeEvent` method is placed in the Accept loop, where it's called each time an event happens. If the event is `EVENT:ColumnResize`, `TakeEvent` will call `SyncColumns` to adjust the column widths.

```
Loop Ndx = 1 to 255
If NOT Self.TotalList{PROPList:Exists,Ndx} then break. If
RightToLeft Self.TotalList{PROPList:Width,Ndx} | =
Self.InList{PROPList:Width,Ndx} else
Self.InList{PROPList:Width,Ndx} | =
Self.TotalList{PROPList:Width,Ndx} end end
```

There are a few things this demonstration app could use. If you have enough items in the list to cause the scroll bar to appear, the rightmost total column should be indented enough to compensate for the scroll bar width. You can get that information from an API call. And this code could also be made into a template, and perhaps even be adapted to work with ABC browse boxes and total fields. This is, as the saying goes, an exercise left to the reader.

Download the source

Clarion MAGAZINE

Making Sense of ABC's ErrorClass

by Russ Eggen

Part 1 of 3

Recently there was a discussion on the newsgroups about ABC's `ErrorClass`. The discussion centered on confusion about what `ErrorClass` really does, or more precisely, what it does not appear to do.

The real bone of contention is that in any application you have to test for the specific error condition and if an error is detected, know what to do about it. When designing an application, you must take this into account. ABC does this with `ErrorClass`, but not in a way most developers are used to trapping errors.

The Application Handbook describes what `ErrorClass` is and what it does quite well. And David Bayliss wrote an article in Clarion Magazine about it. However, neither the Handbook nor David's article gives a short, simple explanation. So let me get this out of the way now.

> The purpose of `ErrorClass` is to report an error condition and indicate the severity of the error.

You cannot make it any simpler than that. Notice the definition does not say it handles errors; it *reports* them.

I know you want to trap for specific errors, not get the severity level. You also want to find out what triggered the error. I will get to that shortly.

If you are of the mindset that you must grab the error condition *and* use `ErrorClass`, you are not using ErrorClass correctly. I will get to that issue shortly as well. Before I do, a visit to `ErrorClass` is in order.

Detecting errors

`ErrorClass` does what its name implies -- it detects errors. That is oversimplifying its duties, however.

`ErrorClass` must be generic to use anywhere, so it has to be a bit "flat." In other words, without knowing ahead of time what would cause an error, how does `ErrorClass` report it? And more importantly, who gets this report? In Legacy applications the error handling code is generated along with the code to manage browses, forms, and everything else. In ABC, most of the logic that manages forms, browses, and other aspects of a business application is contained within the ABC library. That means that most of the errors that happen are going to happen inside ABC, not inside the generated code. Any error handling solution has to be designed in such a way that it can be plugged into that generic ABC code.

Clarion developers write business applications for the most part. After all, Clarion is a business language, with strong data file handling. Things go wrong with data files. You may try to OPEN a file that does not exist. Or what if you are looking for a specific record and it does not exist? There are many other types of errors; not all are fatal. Also, these types of errors could happen in any application that uses a data file.

And since we all want to write rock-solid applications, detecting error conditions is merely the first step.

There are two major tasks that need addressing. The first is error detection. The second is determining if something needs to be done.

Errors and Exceptions

There are two ways of dealing with problems that occur in a program, error handling and exception handling. Error handling is what Clarion programmers know. You write code after an operation to determine if something bad happened, and then you take an appropriate action. In exception handling you have a block of code that can cause a problem, and it's marked as such. If an error occurs, the code "throws" an exception and the block of code is exited. After that block of code is some code to handle any exceptions that may have occurred. This code can take action, or can "throw" the exception again, back to the calling code. ABC's method of error handling is a lot more like the exception model than the error model, but it's not easy to implement because the Clarion language doesn't have built-in support for exceptions. Exception handling is often better than error handling because it reduces the number of places where you need to write code to handle errors.

Ingredients For Errors

The first step to detecting errors is to have a list of all possible errors. These are documented and you can determine what the error is by using the `ERROR()` and `ErrorCode()` functions.

Sometimes it is necessary to inform the user that something abnormal happened. And to assist your customer (and yourself), a description of what happened can provide clues on how to remedy the condition. So you need to have some descriptions for the error conditions too.

Not all errors are fatal, nor do they signal the end of the world. Sometimes, they are informative, instructing a user what the next step is. There is a need to detect the severity level of an error.

One other nice touch would be to report who triggered the error. Was it ABC? A third party template? One of your own classes? So you need an error category. More on this later.

And finally, error handling should have something meaningful to display to the user about the error condition, perhaps instructions on the next step to take, or if the error was fatal, who to contact for help.

Since this information has to be somewhat configurable, ABC keeps it in a queue:

```
ErrorEntry    QUEUE,TYPE ! List of all translated error messages
Id              USHORT   ! Error message identifier
Message         &STRING  ! Message text
Title           &STRING  ! Error window caption bar text
Fatality        BYTE     ! Severity of error
Category        ASTRING  ! Optional category for this error,
                        ! if blank uses CurrentCategory
           END
```

The other items needed are the severity levels. This is the area that some Clarion developers have a problem with as a call to the error handler only returns the severity of the error. This is quite deliberate, and I'll explain why later. Here are the definitions of the severity levels:

```
! Severity of error
Level:Benign                  EQUATE(0)
Level:User                    EQUATE(1)
Level:Program                 EQUATE(2)
Level:Fatal                   EQUATE(3)
Level:Cancel                  EQUATE(4)
Level:Notify                  EQUATE(5)
```

Putting a degree of severity to an error handler signals to the caller the type of action to take, if any.

The next ingredient is a list of all errors that could conceivably happen in a business application, preferably with the option to translate those errors into other languages.

ABC stores error messages using a combination of equates and a group structure:

```
! Message numbers for predefined
! error messages in ABERROR.TRN
                        ITEMIZE(0),PRE(Msg)
None                      EQUATE
AbortReading              EQUATE
AccessDenied              EQUATE
AddAnother                EQUATE
AddFailed                 EQUATE
AddNewRecord              EQUATE
!Edited for brevity
                        END
```

`Itemize` is a Clarion statement that is just a fast way of defining constants with sequential values. The Itemize statement starts numbering at 0 so the first item in the list is equated to zero, the next to 1, and so on. The `Pre()` attribute attaches the prefix to the label. The above is the same as this Clarion code:

```
MGS:None                        EQUATE(0)
MSG:AbortReading                EQUATE(1)
...
```

With `Itemize` you don't have to worry about inadvertently assigning duplicate numbers to equates. In your code you always refer to the equate, which makes your code readable than if you use a raw number.

`DefaultErrors` is the group structure that holds the definitions of the errors:

```
DefaultErrors GROUP
Number USHORT(44)
       USHORT(Msg:RebuildKey)
       BYTE(Level:Notify)
       PSTRING('Invalid Key')
       PSTRING('%File key file is invalid. Key must be ↵
              rebuilt.')
       USHORT(Msg:RebuildFailed)
       BYTE(Level:Fatal)
       PSTRING('Key was not built')
       PSTRING('Error: (%ErrorText) repairing key for ↵
              %File.')
       USHORT(Msg:CreateFailed)
       BYTE(Level:Fatal)
       PSTRING('File Creation Error')
       PSTRING('Error: (%ErrorText) creating %File.')
!Remaining group edited for brevity
END
```

The first item in `DefaultErrors`, `USHORT(44)`, indicates how many errors are defined in this group. You can expect this group to grow (and therefore number to increase) as more features are added. Open the ABError.TRN file in the libsrc folder to see all of these messages.

Each message is comprised of four items.

- `USHORT(MSG:RebuildKey)` is the error number. Remember that this is defined in the Itemize group explained previously.
- `BYTE(Level:Notify)` is the level of severity. In this case, it is not fatal, but the user should be made aware of this condition.
- `PSTRING('Invalid Key')` is the text on the window title bar of the message presented to the user.
- `PSTRING('%File key file is invalid. Key must be rebuilt.')` is the message text. Notice the %File? This is an expandable macro. It is replaced with the name of the file that has an invalid key. This is so the file with the invalid key is identified.

The next `USHORT` starts the next error ID. Each error has four elements.

Why all the fuss?

This seems like a lot of work just to have simple error reporting. After all, you can do this far easier with a simple `IF ErrorCode()` test. Why is all this necessary?

Remember the design elements here.

1. File errors often happen inside the ABC class libraries, not inside the generated code, so error handling has to be generic enough to work with ABC.
2. You should write any custom error handling code only once, and use it everywhere.

Note that these two elements align with each other. This is an indication that the design is a good one.

ErrorClass

The primary objective of `ErrorClass` definition is to report error conditions to a calling procedure, along with all necessary information about the error.

For example, in ABC, there is a `FileManager` class. It is has a reference to `ErrorClass`. This makes sense, as when dealing with files, you have to be able to detect errors.

Each file in your application gets its own instance of the `FileManager` class, and actually these instances are derived from `FileManager`, so they have all of the default `FileManager` capabilities but can also have custom code which applies just to this particular file.

NOTE: The `FileManager` instances are declared in the AppNbc0.clw module. I mention this only so you can see the details of how this is done. All the business rules defined in the dictionary are generated into this module.

In some of the ABC `FileManager` methods you will find calls to `ErrorClass` like this:

```
SELF.Throw
```

`SELF` refers to the current object's name (whatever it may be, the actual name is not important). So even though you're looking at code in the ABC `FileManager` class, there will be one instance of this code running for each file. Thus write once, use everywhere.

OK, so what does `Throw` do? Perhaps a picture would help. Consider Figure 1.

Figure 1. Message flow in ErrorClass

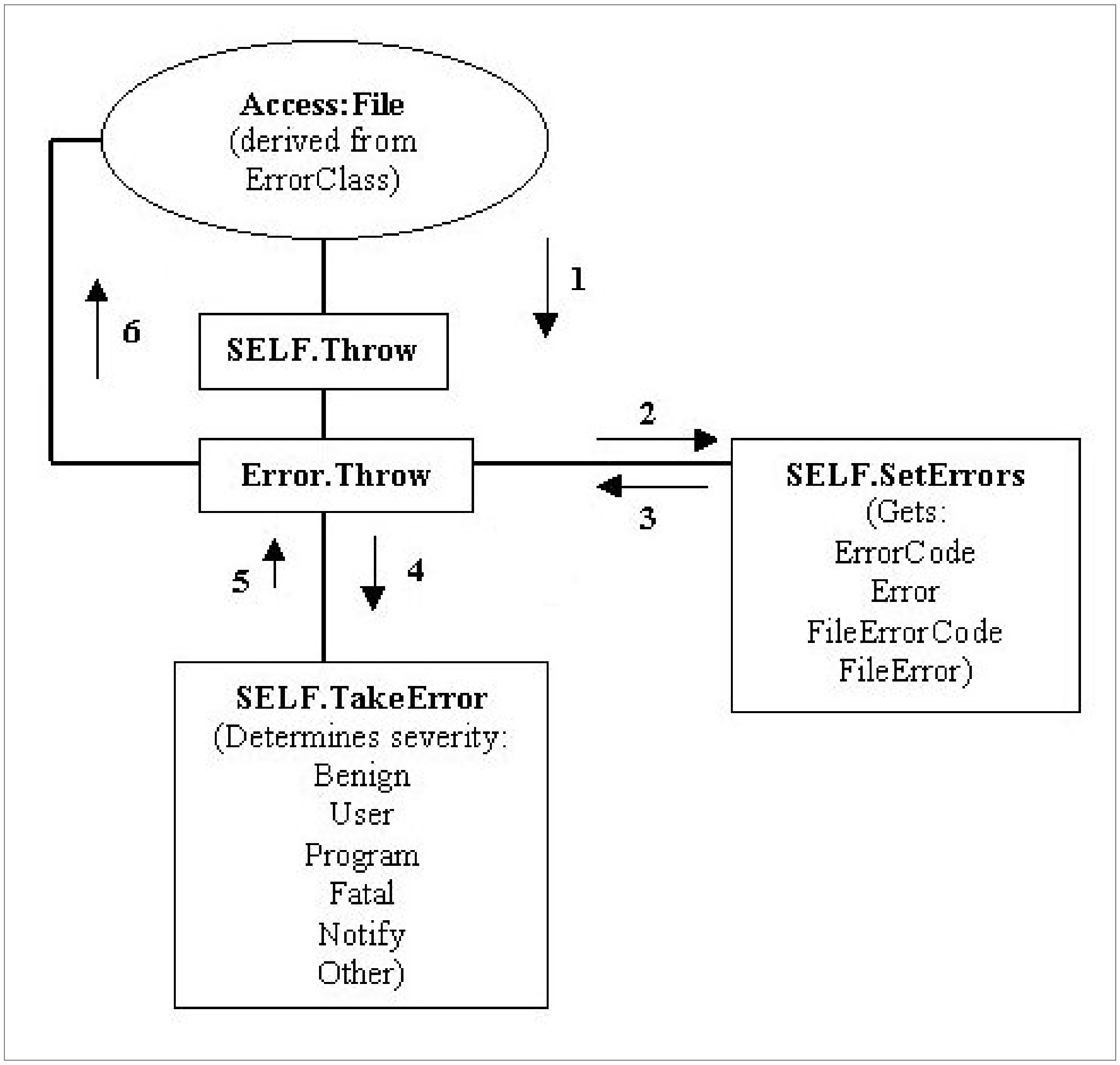


Figure 1 shows how the methods work with each other. The file's `FileManager (Access:File)` calls `SELF.Throw` when an error condition is detected.

`Access:File` is derived from `ErrorClass`, it calls the `ErrorClass's Throw` method (1). The Throw method calls `SetErrors (`2) and gets any error from the Clarion library. It then returns to Throw and the next line is executed, which is a call to `TakeError` (3). `TakeError` checks the severity level (4) and returns it to `Throw` (5). Throw then returns the ID of the error and its fatality level (6).

The above is really more than most developers need to know, but it does expose how this object gathers information about all potential errors as each call is made.

But where does it get all this information and how does it return the correct error strings, display the messages and so forth? Those are different methods and a queue is used to store data.

In an ABC object, a method may call other methods that could call others. The reason why they do this is so that bits of information are gathered and accumulated as the flow goes through the various methods. One seldom needs to know every little line of code that is run.

The method calls are like a building a car on an assembly line. You do not need to see the assembly line that builds the engine, but you may be interested in how the engine is mounted in the frame or what type of engine is used. Even at a high level, you don't care about the engine. All you do know is that it has sufficient power to get a car moving. If it doesn't, then you are interested in going a bit deeper.

That is a pretty decent metaphor for how ABC does its work. Too many procedural coders get caught up in the insignificant bits of ABC. Like the assembly line, as a car purchaser or even a mechanic you don't need to know the voltage for the spot welders any more than you need to know the thickness of the metal or what type of metal is welded.

For the same reason, when someone tells me they need to get the error code from a file operation, I say they don't. What you should be asking is why ABC did not do this for you? That is its job. Not yours.

Let's say you're testing for a record not found or end of file conditions in some embed code that uses ABC methods for file access. Again, why are you doing this? It's too low level. You are making extra work for yourself! Let ABC handle it.

For example, consider finding a specific record by key value. This is a common task in business applications. In straight Clarion code, it would look like this:

```
FIL:KeyField = ‘SomeValue’
SET(FIL:KeyLabel,FIL:KeyLabel)
GET(File)
IF ErrorCode() = 33
  MESSAGE(FIL:KeyField & ‘ is a value not on file.’)
ELSE
  MESSAGE(‘Success! I found a record with ‘ |
    & FIL:KeyField)
END
```

Or perhaps in a loop:

```
FIL:KeyField = ‘SomeValue’
SET(FIL:KeyLabel,FIL:KeyLabel)
LOOP
  NEXT(File)
    IF ErrorCode() = 33
      !No more records with this value
    ELSE
      !Do something with each record successfully read
  END
END
```

Either way, it's pretty straightforward. In ABC, the same functionality would look like this:

```
FIL:KeyField = ‘SomeValue’
LOOP UNTIL Access:File.Fetch(FIL:KeyLabel)
```

```
  !Do something with each record successfully read
ELSE
  !No more records with this value
END
```

Whoa! Where is the error checking? As it turns out. The `Fetch` method will take the steps to check for an error. If the error is severe enough (like corrupted keys, record mismatch, or other fatal errors) then it `Fetch` throws a message. An error of "not found" is not a fatal error, so it comes back with a different severity level. In this instance the code uses `Fetch` like a Boolean (true or false). If Fetch comes back with zero (`LEVEL:Benign`), then there was no error. In other words, it found a match. Anything else, it did not find a match. So the only thing that matters is that the return value was non-zero.

Moral of the story

The bottom line is that if you can expect to find typical error checking from file operations in any procedure, do not code it, as ABC can handle it. If you need special error handling that is unique to this application or procedure, then you must step in and change the way ABC handles errors. Next week I'll show you how to do that.

Russ Eggen has been using Clarion since 1986. Before joining Topspeed as a consultant in 1996, he was an independent contractor. Currently, Russ is an instructor at SoftVelocity and is writing the curriculum for the classes. His main goal in life is to get a Clarion program to star in a Tom Clancy movie where the program helps the hero save the world.

The Clarion Challenge: Hey, I'm talking to you!

by Dave Harms

I've been underwelmed by the response to the Clarion Macro Challenge I issued two weeks ago. I started asking around. Is anybody using using these things? Andrew Guidroz II said yeah, he is, but he just writes them as he needs them, as opposed to keeping some standard macros on hand. For instance, he'll write a macro to CLIP() a bunch of variables, or wrap them in some other method call. Very handy when writing XML code, as in AddToXMLDataStructure(). And that's just some of Andrew's code - don't go looking for it in Clarion.

I decided I'd begin with something simple like a CLIP() macro, and maybe that would kick start some reader responses. But there's a problem.

Figure this one out

As it turns out, writing a macro to wrap a variable in a CLIP() function isn't nearly as simple as it would seem. For a variable at the end of a line of code, the macro keystrokes are as follows:

CLIP(<end>)

But what about a variable in the middle of a line of code? You don't want to clip everything, just the variable. In that case you'd probably want something like

CLIP(<ctrl-rightarrow><leftarrow>)

Only you can't use this second approach for a variable at the end of a line of code because the <ctrl-rightarrow> keystroke will take you to the next line.

The real challenge

So here's the real challenge. Write a macro that can wrap a CLIP() statement around a variable whether that variable is in the middle of the line or at the end of the line.

I dare you.

Write up your macro and send it to editor@clarionmag.com. I'll feature the responses in an upcoming issue.

Looking for inspiration? Check out James Cooke's excellent article on Clarion macros.

Reborn Free CLARION online

published by CoveComm Inc.

Search

Clarion MAGAZINE

Holiday Special
25% Discount
on
APM & AFE

This Offer Has Now Expired.

This time of year is traditionally the time for the Florida DevCon, and DevCon is traditionally a time for Clarion Magazine to offer a special deal to tempt new subscribers. There's only one problem: This year, because of the change in Clarion's ownership, there is no Florida DevCon.

So use your imagination!

Announcing the No DevCon 2000 Subscription Special!

If you're not yet a Clarion Magazine subscriber, here's a great way to get started. For the rest of October, purchase a one or two year subscription and **buy all 20 back issues** of Clarion Magazine for **only $82 more!** That's over **30% off** the regular back issue price! We'll also give you an additional **$10 off** if you buy a two year subscription.

Why buy the back issues? Because you get over 200 articles on every aspect of Clarion development. Clarion developers often bemoan the lack of books about Clarion. Buy the back issues and get as much material as you would in *three Clarion books*. Topics include the following:

- Learning the ABC templates
- Reporting
- Debugging
- Legacy programming tricks
- Product reviews
- ABC library secrets
- Writing hand code/embed code
- Database design
- Understanding SQL
- Multi-DLL development
- Modifying the Clarion IDE
- Conference coverage
- Using OLE/COM
- Application design

Clarion Magazine also features interviews with Clarion movers and shakers, regular news about and for the Clarion community, free software, and much more. And Clarion Magazine sponsors free public access to all two years worth of articles originally published in Clarion Online. That adds another two or three volumes to your virtual Clarion bookshelf!

What If I Just Want The Back Issues?

The back issues aren't available separately, but Clarion Magazine does have a pro-rated refund policy for current subscriptions. Just tell us you want to cancel your subscription, and we'll refund you for any future months remaining on your subscription.

Subscribe And Save!

After reading the subscription agreement, choose one of the following:

Two year subscription plus back issues	$232
One year subscription plus back issues	$162

If you have any questions, please email subscriptions@clarionmag.com.

Main Menu | Log In | Subscribe | Links | FAQ | Advertising

Search

Clarion MAGAZINE

Main Page

COL Archive

Log In
Subscribe
Renewals

Frequently Asked Questions

Site Index
Article Index
Author Index
Links To Other Sites

Downloads
Open Source Project
Issues in PDF Format
Free Software

Advertising

Contact Us

Last Week To Save On Your Subscription! (Nov 28,2000)

Making Sense of ABC's ErrorClass - Part 2 (Nov 28,2000)

Profiler/Debugger Tools Update (Nov 28,2000)

November 2000 News (Nov 28,2000)

Making Sense of ABC’s ErrorClass

by Russ Eggen

Part 2 of 3

Last week I explained how ABC’s `ErrorClass` handles file operation errors. Maybe you don’t believe that `ErrorClass` can work in your situation because you have a special circumstance and you think you need the exact error code. I don’t agree with that, but if you must, you can always make the call to the `FileManager’s Throw` method.

If you are not yet convinced, then look at this class from the perspective of custom error handling

Custom error handling

I must admit, adding custom error messages is my favorite part of `ErrorClass`. ABC only supplies the common errors that one could encounter in any given application. And this is fine as long as common errors are all you wish to check for. In this case, you don’t do a thing; ABC does all the work. Nothing could be sweeter.

But what about error conditions that are unique to a particular circumstance? For example, what about someone who does not have the authority to close an invoice? What about not having enough quantity on hand to fill the order? What about someone who gives a wrong password when logging in? What if you wish to have your own message, replacing ABC’s message?

Those are pretty specific conditions, and out-of-the-box ABC isn’t going to handle them. But can ABC do anything for those circumstances? Well, obviously, the answer is yes or I would not have brought it up.

The test application

Lets start with a very simple, one procedure application. It is only a window procedure that simulates a login window. Since the design is security conscious, allow one bad attempt to login. On two bad attempts, shut down the application. In either case, the app needs to display a message to inform the user what happened.

For demonstration purposes, the password is hard coded. Never do that in real life! However, this is a demonstration of `ErrorClass’s` flexibility, not of good security procedures.

ABC has no "login failed" messages. This should not be a surprise as security is application specific, and is something you would not expect to find in all business applications. However, if you wish to add security messages to your arsenal, then adding your own messages as described below is a good start.

Define the errors

The first step is to define your errors, just as ABC does. You need two error ID numbers. One for the innocent goof and the other, more harsh, hacker detection. You do this as follows:

```
!!Declare the error numbers
    ITEMIZE(2000)           !Start with this number
TryAgain    EQUATE
Rejected    EQUATE
    END
```

Put this code in `Local Data.Other Declarations` in your embed tree. You can start with any number you wish; it really does not matter. Even "1" would work, despite ABC already having an error ID with the value of "1". In other words, duplicates are allowed. More on this shortly. One additional thought on this before I move on; you do not actually have to define labels for error numbers. However, if you don’t then it will be harder to recognize which error you wish to throw. Another benefit is that if the number changes for whatever reason, you make the changes in only one place. And this is the one place.

Declare the error group

The next step is to define the error group in the same format `ErrorClass` expects. In this example, it looks like this:

```
!!Declare the error group
PasswordGroup GROUP
      USHORT(2)             !Two errors defined in this group
      USHORT(TryAgain)      !Invite user to try again
      BYTE(LEVEL:Notify)    !Severity of error
      PSTRING('Do not recognize entry...') !Window title
      PSTRING('The system did not recognize that password.|
               Please try again.')
      USHORT(Rejected)      !Password rejected
      BYTE(LEVEL:Fatal)     !Severity of error
      PSTRING('Unauthorized access...') !Window title
      PSTRING('Hacker detected -- closing program!')
    END
```

You put this in the same embed, but a later priority. Or you can use the same embed as the previous code, but ensure it comes afterwards. I like using separate embed priorities as this gives each body of code its own embed point. I also think it makes my job easier if I must maintain the code later. And I always use comments on line one, so that I can read down the embed tree (or the right-hand pane of the application tree with embeds expanded) and quickly locate the code I want to look at.

Tell ABC about the custom errors

The next step is to tell ABC you have some error messages and you wish them added to the default list of errors. This is easily done. I could write the code as source, but I do enjoy using the `CallABCMethod` code template. Its best use is to filter out methods you have no interest in, thus giving you a short list of methods to choose from. It also writes the correct code for you.

Placement of the method call is important. You cannot place this call before `ErrorClass` is instantiated. If you do, you will correctly get a GPF. Since objects on a procedural level are instantiated in `ThisWindow.Init`, it would be logical to place the code there. Select the `Set options from global values` embed point under `ThisWindow.Init.` Press the Source button to open the embeditor. This is what you see:

```
! Parent Call
  ReturnValue = PARENT.Init()
  ! [Priority 5050]

  ! Set options from global values
  IF ReturnValue THEN RETURN ReturnValue.
  SELF.FirstField = ?PasswdPrompt
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
! [Priority 5600]
```

You can add the method call by hand, or you can use the `CallABCMethod` code template (return to the embed tree first).

In any event, the object name is `GlobalErrors` (an object derived from `ErrorClass`). The method you want is named `AddErrors (ErrorBlock EB)`. The prototype is merely defining a variable labeled `EB` of a data type `ErrorBlock`. `ErrorBlock` is defined as a `GROUP` in the ABC classes. Thus, the passed parameter is simply `(PasswordGroup)` – you must type the parentheses. The source now looks like this (embeditor view and if you used the code template – emphasis added):

```
! Set options from global values
  IF ReturnValue THEN RETURN ReturnValue.
  SELF.FirstField = ?PasswdPrompt
  SELF.VCRRequest &= VCRRequest
  SELF.Errors &= GlobalErrors
  ! [Priority 5100]

  GlobalErrors.AddErrors(PasswordGroup)
```

Save the edits. Since you changed an ABC object, it is always a good idea to "clean up" after yourself. So when this procedure quits (killed), you should remove your custom error group. This happens in the `ThisWindow.Kill` method. Since there is an `AddErrors` method, it is logical to assume there is a `RemoveErrors` method. Using the `CallABCMethod` code template, you can see this is true. It takes the same parameters as the previous method, entered the same way.

What did I just do?

ErrorClass is now aware of your new errors. All you have done so far is define some error ID numbers, place them in a group that ABC understands, and then used ErrorClass to add them to the known error messages. Also, when the procedure quits, you've removed the extended error group, as you no longer need them.

That does not seem like much, as nothing will happen if you run the program at this stage. Since the job of ErrorClass is to report what error is detected, you must write the logic to test for the error.

Adding the error detection logic

The design specs say that the program allows for one mistake in the login procedure. If a second mistake is made the program shuts down. This means the testing for these conditions is in the LOC:Password entry control. LOC:Password is a local variable (STRING(20)), located under the Data button. The best event to use is EVENT:Accepted on this control, and the easiest way to get to this embed is to open the window formatter, right click on the entry control and choose Embeds. Only the embeds for this control are visible. Highlight Accepted and add this source:

```
!!Check for valid password
  IF LOC:Password <> 'ThisIsYourLife'
    IF ~LOC:FirstAttempt
      LOC:FirstAttempt = True
      GlobalErrors.Throw(TryAgain)
      CLEAR(LOC:Password)
      SELECT(?LOC:Password)
    ELSE
      ThrowAwayValue# = |
        GlobalErrors.Message(Rejected,BUTTON:OK,BUTTON:OK)
      LOC:FirstAttempt = False
      POST(EVENT:CloseDown)
    END
  END
```

The logic is as follows: If this is the first attempt, set the flag to True and then Throw the TryAgain error. Clear the entry and select it. If this is a second attempt, then call the Message method with the Rejected error. In this case, you don't care about the return value. Set the flag to False and then POST the shutdown event.

This example is a bit simplistic and even silly, but it demonstrates two ways to handle errors. The first is to Throw a message to the user. The second is to send a Message to the user, and if I wanted, I could also test the button pressed by the user. In this case, I did not care as there is one button and I am shutting things down.

Click here for the example application.

That code works. Now try this. Change the fatality level of the first message to LEVEL:Benign. Compile and test. What happened? You got a different result. You may now start to see how ErrorClass works.

Why not simply use the Clarion Message statement instead of ErrorClass's Message method? Both are used exactly the same way. The difference is that the Message method will place the error text and error title in the box for you. This makes sense as the exact error is now known by the time the Message method displays.

Twisted fun

OK, that is nice to have the ability, but what if you do not want to display error messages at all? Some users think the end of the world is coming when they get error messages, even helpful ones. And what do you do about users who cannot read the English language, or, like me, are illiterate in non-English languages? Perhaps a good solution would be to use AVI files to show how to remedy the problem.

In this case, `ErrorClass` must know to play AVI files instead of displaying messages. This means you must teach it to do something completely different! Of course, there is a catch (there always is one). I am not extending the class with new methods. After all, that is too much work. Why not get one of the existing methods to play the AVI files, with everything still in place as before?

Oddly enough, `ErrorClass` design has this in mind. But first, let's get the AVI parts handled. I have a small class definition for playing AVI files (don't worry about how it works). This is simply `INCLUDEd` in a global embed. In `After Global Includes` only one line of code is required:

```
INCLUDE('ErrorAvi.inc') !Include the API objects
                        ! for playing AVI movies
```

This is the class definition for playing AVI movies. In the `Global Data` embed point, only one line of code is needed:

```
AVI ErrorAvi !Derive and instantiate AVI class
```

This instantiates the `ErrorAVI` class. I also want to override the default behavior of the Throw method in `ErrorClass`. In the `Global Objects, ABC Objects, Error Manager (ErrorClass), Throw PROCEDURE(SHORTD ID),BYTE` embed:

```
!!Override the global ErrorClass
Avi.PlayMovie(SELF.Errors.Title) !This calls the
                                 ! correct AVI file
CLEAR(Id)                        !Ensure the error ID
                                 ! is cleared out
RETURN LEVEL:Benign              !Return severity of error
```

The `PlayMovie` method uses `ErrorClass` property `Errors.Title` to determine the name of the file to play. After the movie plays, clearing the error ID is required so that `ErrorClass` does not then display an error message. Also, the severity level must change so that `ErrorClass` does not feel it must do something. That is all that is needed to change the behavior of this method. For reasons that should be obvious, this code is placed after the Parent call.

To get the one procedure to do what we want, similar embeds are required to get the effect needed. For example, in separate embeds in Local data, Other declarations are these two embeds (shown in embeditor mode):

```
! Start of "Local Data After Object Declarations"
! [Priority 4000]
!!Declare the error numbers
    ITEMIZE(1)          !Start with this number
TryAgain    EQUATE
Rejected    EQUATE
    END
```

```
! [Priority 4100]
!!Declare our error group
PasswordGroup   GROUP
     USHORT(2)             !Two errors defined in this group
     USHORT(TryAgain)      !Invite user to try again
     BYTE(LEVEL:Benign)    !Severity of error
     PSTRING('minbari.avi') !Window title
     PSTRING('The system did not recognize that password.')
     USHORT(Rejected)      !Password rejected
     BYTE(LEVEL:Benign)    !Severity of error
     PSTRING('error.avi')  !Window title
     PSTRING('You have just committed a Fatal error!')
   END

! End of "Local Data After Object Declarations"
```

Just to demonstrate that you can embed code in other places besides `ThisWindow.Init`, the `Run` method is overridden with the following code (source comments, `CallABCMethod` code template to add the code following the comments):

```
ThisWindow.Run PROCEDURE

ReturnValue          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]
  !!Add the error messages before the procedure runs
  GlobalErrors.AddErrors(PassWordGroup)
  ! [Priority 4950]

  ! Parent Call
  ReturnValue = PARENT.Run()
  ! [Priority 5050]
  !!Remove the error messages now that the
  !! procedure is no longer running
  GlobalErrors.RemoveErrors(PasswordGroup)
  ! [Priority 7600]

  ! End of "WindowManager Method Executable Code Section"
  RETURN ReturnValue
```

The rest of the code is the same as before. The second example application contains two AVI movies. They are not really accurate for this design, but they are close enough to give you the idea. The important thing is you now know how to show AVI movies in your application based on error conditions. You might want to add an option in the Setup portion of the application to turn this feature on or off.

Next time: Customizing ErrorClass with a template.

Download example 1 (requires C5.5)

Download example 2 with AVI (requires C5.5)

Russ Eggen has been using Clarion since 1986. Before joining Topspeed as a consultant in 1996, he was an independent contractor. Currently, Russ is an instructor at SoftVelocity and is writing the curriculum for the classes. His main goal in life is to get a Clarion program to star in a Tom Clancy movie where the program helps the hero save the world.

Main Menu | Log In | Subscribe | Links | FAQ | Advertising

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

Search

Main Page

COL Archive

Log In
Subscribe
Renewals

Frequently Asked
Questions

Site Index
Article Index
Author Index
Links To
Other Sites

Downloads
Open Source
Project
Issues in
PDF Format
Free Software

Advertising

Contact Us

Last Week To Save On Your Subscription!
(Nov 28,2000)

Making Sense of ABC's ErrorClass - Part 2
(Nov 28,2000)

Profiler/Debugger Tools Update
(Nov 28,2000)

November 2000 News
(Nov 28,2000)

Open Source Update: Easier-to-use Profiler

by Dave Harms

I've made an updated version of the CCI Profiling and Debugging tools available on the Open Source products page. These tools, and the accompanying template, let you create a trace log of your application's procedure and method calls automatically, and also make it easy to add your own messages to the log. See the related ClarionMag article for more details on usage.

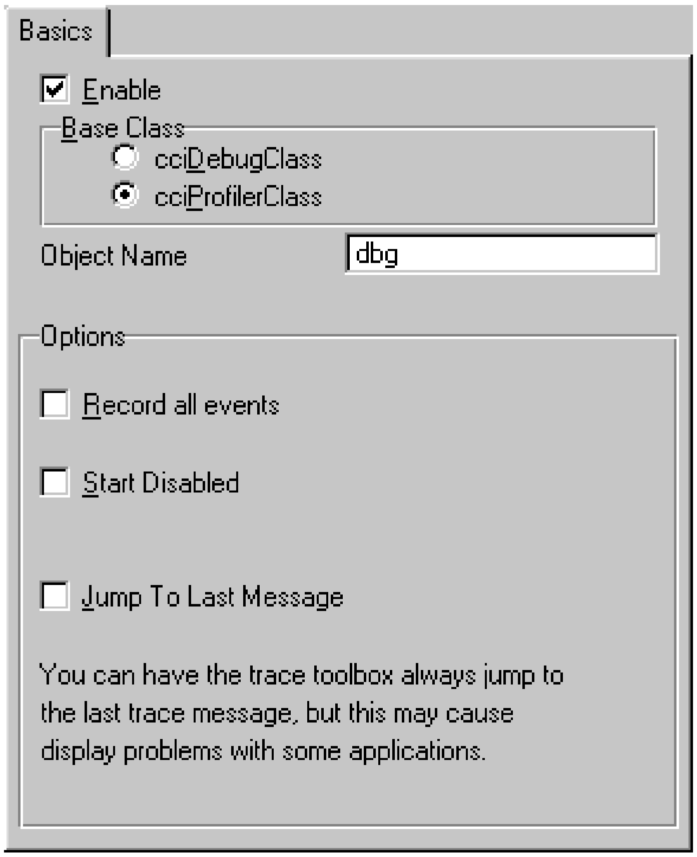
This latest release (1.2) adds a few usability improvements. Following on a suggestion by Gordon Smith, I updated the source files with a compiler pragma to automatically turn off profiling for the profiler and debug classes. This means you no longer have to add project defines for those two source files.

I also added a #PDEFINE to the template to turn on debugging and profiling for the application as a whole, which makes it feasible for the template to add source files to the project automatically without causing any compile errors. All you really have to do now is drop in the template and go!

The new template prompts are shown in Figure 1. If you use cciProfilerClass as the base class you'll get a trace of all procedure calls in your application. If you use cciDebugClass you'll get all of the trace functions, but you won't get the automatically generated procedure calls.

Note that there is now an Enable checkbox. If unchecked, none of the debug/profile code will be generated into your application.

Figure 1. The debug/profile template settings



[✓] Enable

Base Class
- ○ cciDebugClass
- ● cciProfilerClass

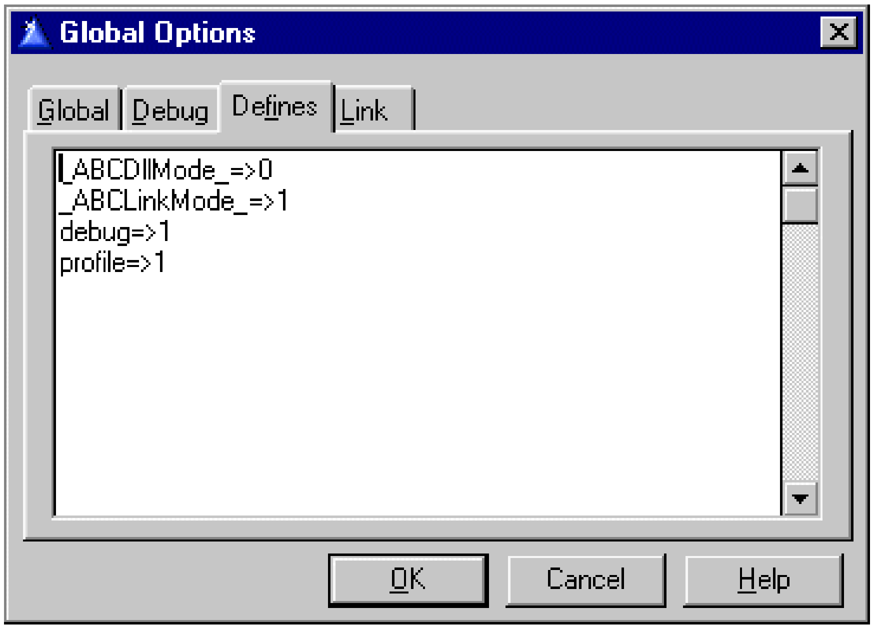
Object Name: dbg

Options
- [ ] Record all events
- [ ] Start Disabled
- [ ] Jump To Last Message

You can have the trace toolbox always jump to the last trace message, but this may cause display problems with some applications.

If you generate your code with the template enabled, then with it disabled, the ccidebug.clw and cciprof.clw source files will still be in your project, but unused. You can leave them there. On the other hand, you may want to remove one of the project defines. Figure 2 shows the defines after generation with the template disabled.

Figure 2. The template-generated defines

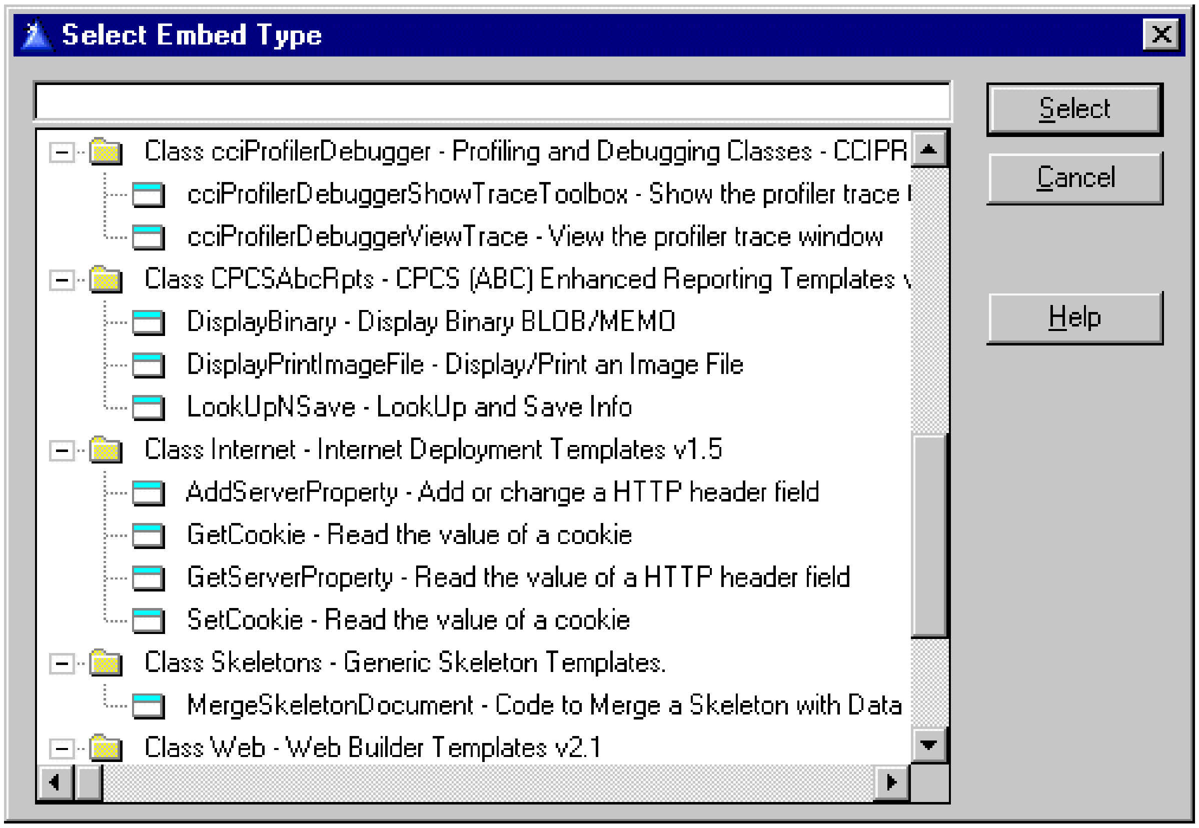


```
_ABCDllMode_=>0
_ABCLinkMode_=>1
debug=>1
profile=>1
```

Important! Because the template generates a debug=>1 setting, debug will stay on even if you turn it off in the project. You'll have to remove this setting here. I could do it automatically when you disable the debugger, but you might want debug on in other situations. Unfortunately there doesn't seem to be a way to remove a define entirely from within the templates.

As before, there are two code templates to call the trace toolbox and the trace window, as shown in Figure 3.

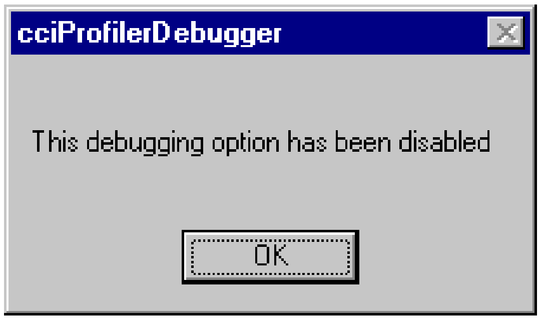
Figure 3. The trace toolbox and trace window code templates



Class cciProfilerDebugger - Profiling and Debugging Classes - CCIPR
cciProfilerDebuggerShowTraceToolbox - Show the profiler trace
cciProfilerDebuggerViewTrace - View the profiler trace window
Class CPCSAbcRpts - CPCS (ABC) Enhanced Reporting Templates
DisplayBinary - Display Binary BLOB/MEMO
DisplayPrintImageFile - Display/Print an Image File
LookUpNSave - LookUp and Save Info
Class Internet - Internet Deployment Templates v1.5
AddServerProperty - Add or change a HTTP header field
GetCookie - Read the value of a cookie
GetServerProperty - Read the value of a HTTP header field
SetCookie - Read the value of a cookie
Class Skeletons - Generic Skeleton Templates.
MergeSkeletonDocument - Code to Merge a Skeleton with Data
Class Web - Web Builder Templates v2.1

I've added code to disable these calls if you use the global disable option. If the user calls these functions when the global template is disabled, the message in Figure 4 displays.

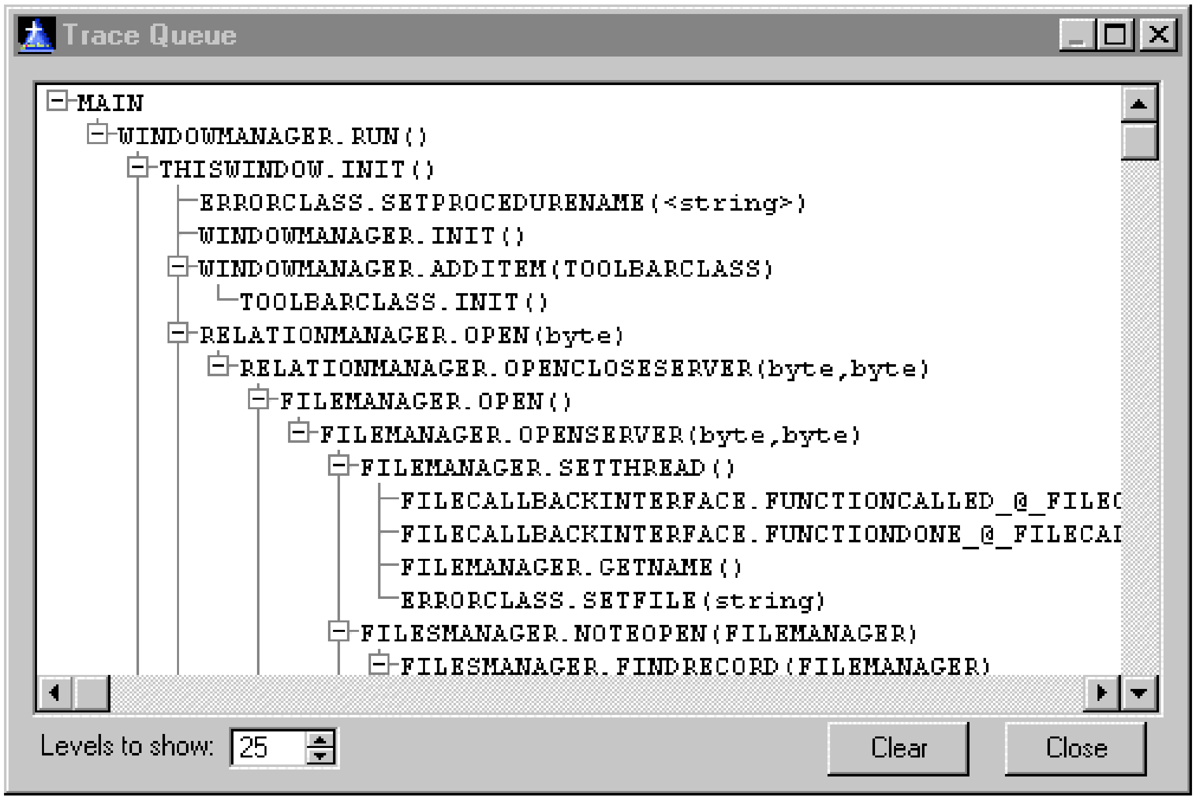
Figure 4. The message displayed on calling the trace toolbox or window when the global template is disabled



This debugging option has been disabled

And, just in case you're wondering what the output is like, Figure 5 shows the trace toolbox with a tree view of the procedure calls at the start of an ABC application.

Figure 5. The trace toolbox in action



```
MAIN
  WINDOWMANAGER.RUN()
    THISWINDOW.INIT()
      ERRORCLASS.SETPROCEDURENAME(<string>)
      WINDOWMANAGER.INIT()
      WINDOWMANAGER.ADDITEM(TOOLBARCLASS)
        TOOLBARCLASS.INIT()
      RELATIONMANAGER.OPEN(byte)
        RELATIONMANAGER.OPENCLOSESERVER(byte,byte)
          FILEMANAGER.OPEN()
            FILEMANAGER.OPENSERVER(byte,byte)
              FILEMANAGER.SETTHREAD()
                FILECALLBACKINTERFACE.FUNCTIONCALLED_@_FILEC
                FILECALLBACKINTERFACE.FUNCTIONDONE_@_FILECAI
                FILEMANAGER.GETNAME()
                ERRORCLASS.SETFILE(string)
              FILESMANAGER.NOTEOPEN(FILEMANAGER)
                FILESMANAGER.FINDRECORD(FILEMANAGER)
```

You can also look in the application's directory for trace.log, a text file version of Figure 5, but missing the tree graphics.

Remember that profiling only works with single EXEs, not DLLs! That's a limitation of the runtime library, since the profiler hooks are not exported.

I hope you find the updated debugging/profiling classes and template useful. These tools have often assisted me in solving difficult bugs, and when used in profiling mode provide an interesting view of what's really happening inside ABC.

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, *published by SAMS (1995). He is also the editor and publisher of Clarion Magazine.*

Clarion MAGAZINE

Clarion News

November 28, 2000

LSZip 2.60 Compression Library Released

The Linder LSZip Compression Library version 2.60 is now available. Because some serial numbers and installation keys are available on various "warez" sites, the installation passwords are no longer valid. If you are interested in a free update please send your registration information to update@lindersoftware.com. This update is free to registered users!

Linder SetupBuilder 3.50 Released

Linder SetupBuilder version 3.50 is now available. Because some serial numbers and installation keys are available on various "warez" sites, the installation passwords are no longer valid. If you are interested in a free update please send your registration information to update@lindersoftware.com. This update is free to registered users!

Sterling Data Templates C5.5 Gold Compatible

All Sterling Data templates are now compatible with C5.5 Gold. These include BackFlash 4.3, IMPEX 4.0, LogFlash 2.3, SearchFlash 1.2, CopyFlash 2.0, RemFlash 1.0, and ZIPFlash 1.1.

Win32 Common Controls On Sale

The SysPack bundle from solid.software is on sale through December 24 for $65, a savings of $14. SysPack contains SysAni, an animation control, SysTrack, a trackbar (slider) control, and SysHotKey, a wrapper class for the hot key control.

UltraTree Platinum 6 Released

UltraTree Platinum 6 is now shipping. New features include virtual trees, multi-root trees, collapsible tree headers, and automatic folder icons. Platinum Premium 5 currently supports only Clarion 5.5. Platinum Standard 6 will be released soon. All current Platinum Premium users are eligible to receive the Version 6 upgrade. All current users and new purchasers of Platinum Standard will automatically receive the Version 6 upgrade.

Solace ReDesigner Beta 2 Released

Demonstration templates are so far available for Clarion 5 and C5.5 Gold. Other

Sterling Data
Components
Freeware
Resources
www.sterlingdata.com

Developer PLUS
Your source for development tools and add-ons.
Your outlet for application sales.

versions can be made available on request. New in this version: a translator module to allow the prompts on Property update screens to be changed for different languages; professional version now available which includes all the source code for the ReDesign engine, including the code for putting grab handles on controls; window and control resets to factory defaults are now instant; only controls that have been altered are now saved rather than whole window; filenames can now be specified to allow each user to have different screen versions of the application. The beta program will finish on 10th December, at which time the half price offer will close.

November 21, 2000

IFT Database Jumpstart Released
Logic Central has released Database Jumpstart (DBJ), an IFT:HTTP Server add-on product. DBJ was developed to make it easy to put a database on the web using the Internet Framework Templates (IFT). DBJ automatically generates tables, forms and update procedures (pretty much inspired by the old Personal Clarion that many may remember). You can develop a database application without any coding. Because DBJ does not use hidden windows, you can develop applications that are not resource intensive. DBJ has over 200 functions in the templates so you can quickly customize your database. However, if you need more control there are several embeds so you can add your own code. DBJ also works with forms created in editors such as Adobe GoLive, DreamWeaver, Composer, MS Notepad, etc. Requires no knowledge of HTML for basic functionality, does require Windows 95 or better, Clarion 5 or better, and Internet Framework Tools: HTTP Server 2.5 or greater. DBJ is available for 50% Discount off Regular Price of $299 until January 10, 2001.

Mail & Fax Templates For C5.5 Gold Released
Stealth Software has released Mail and Fax Templates for Clarion 5.5 Gold. The upgrade is available for US$20 charge. Please email support@stealthsoft.co.za for upgrade details. The upgrade is free to anyone who has purchased within the last three months.

Fomin Report Builder For Clarion 5.5
The Fomin Report Builder for Clarion 5.5 Gold is now available for registered users as free update. Registered users have been notified by email, so if you changed email addresses and haven't received your notice you need to send your new address to Oleg Fomin at fomin@mail.com

November 14, 2000

New RARS Version For C55
Craig Ransom has posted a C55 release of RARSFUNC.app and a provisional release of the C55 Document Send Templates version of RAREMAIL.app (ramail55.app).

CapeSoft File Explorer Introductory Price
CapeSoft has released File Explorer, an OCX wrapper around the IE OCX that ships with Windows. Included in the product are the Adobe PDF viewer template and one for Microsoft Media Player. You can view or play any of the following in your application: HTM, HTML, PDF, AVI, MPG, WAV, SND, MP3 plus a whole lot of other obscure formats. This template allows you to simply drop the document viewer, or media control, directly into your application. The OCXs are already installed on your Windows system. Template includes some standard Clarion buttons to Play, Stop, Next Page, Last Page and so on. The normal price for File Explorer is $99, but it's currently on a Special Price

of $69 during the beta program. Beta users will automatically get a free upgrade to the gold release, and beyond.

CapeSoft Insight 1.0 In Beta 4
The CapeSoft Insight Graphing product is now in Beta 4, and is approaching Gold status. Gold is anticipated by year end. Insight already includes bar, pie, donut, line, Pareto, and Gantt charts, with legends, histograms, hi-lo, candle, and mixed charts on the way. Insight will usually cost $299, but will be priced at $199 for the duration of the beta program. Beta users get free upgrades to the gold release, and beyond.

Special Agent 1.27 Released
CapeSoft has released Special Agent 1.27. This version fixes some bugs, and documents the examples in the documentation.

Capesoft Office Messenger
This new product from CapeSoft is a separate instant messaging utility. Office Messenger is currently used to pass messages from one employee to another inside the CapeSoft offices. It's a serverless system, but messages are queued if undeliverable. This system also allows a single administrator to set all options remotely. It also keeps a message history and a simple phone book. A free 30 day demo download is available. Pricing is expected to be $6 per seat.

Bruce Johnson Presents ABC Training Down Under
CapeSoft's Bruce Johnson will be in Australia and New Zealand from November 12th to November 27th, 2000, presenting the "Programming in Clarion using ABC" course in Perth, Adelaide, Melbourne, Brisbane, Sydney and Auckland. There are still a few places open, but you must act quickly! Training details are as follows: 14 Nov Perth - contact David Griffiths davidgriffo@icqmail.com; 16 Nov Adelaide contact Simon Brewer simon@first-ecom.hm; 19 Nov Melbourne (Bendigo) - contact Gary Richards grichards@netspace.net.au; 21 Nov Brisbane - contact Ray Creighton ray@clarion.org.au; 23 Nov Sydney - contact John Thorley dossier@hunterlink.net.au; 25 Nov Auckland - contact Kevin Dunsford kevin@cslnz.co.nz.

NTS Wizard Templates Updated for Clarion 5.5
Nice Touch Solutions, Inc. has released Clarion 5.5 versions of Query Wizard 5, View Wizard 1, Report Wizard 1, Spreadsheet Wizard 1, and CrossTab Wizard 1. These are available for download by existing customers at no charge. Customers with Query Wizard 4 or earlier will need to upgrade to QW5 to get the Clarion 5.5 version.

November 7, 2000

Clarion 5.5 Arrives!
SoftVelocity has released Clarion 5.5, and CDs have begun showing up at shops around the US. International customers should be receiving their copies shortly. For more details see the SoftVelocity newsletter.

RARS Downloads At Mirror Site
If you're having problems downloading Craig Ransom's RARS and NetClip products from , try the mirror site.

Clarion Skill Pool Updated
A new Search Option has been added to the Clarion Skill Pool. This is a free service

from Sterling Data.

BoxSoft SuperTemplates for C55 Gold

BoxSoft has released new versions of SuperQBE, SuperTagging, SuperBrowse, SuperInvoice, SuperImportExport, SuperFieldFiller, and SuperStuff. These solve a problem with the definition of QUEUE fields.