

[Home](#) [Web Design](#) [Programming](#) [Fairlight CMI](#) [Soap Box](#) [Downloads](#) [Links](#) [Biography](#) [About...](#) [Site Map](#)



Record Recycling

[CA-Clipper](#) [Opportunities](#) [Tips and Tricks](#) [Networking](#) [Internal Errors](#) [Source Code](#) [CA-VO](#)

[BACK TO
CA-CLIPPER
TIPS & TRICKS](#)

Because the `pack` command is a problem (see [The Pack Command](#)), a replacement needs to be found. Record recycling is the answer.

Regardless of the dangers of `pack`, its purpose is to remove deleted records. However, most data files grow over time and eventually stabilize at a certain size as records are added and removed throughout the course of normal activity.

The principle behind record recycling is to re-use the deleted records, thus eliminating the need to `pack` them out.

The Method

Instead of using the `append` command to add another record to a table, the program should search first for a deleted record and present it as the "new" record. This approach saves time because the application does not need to ask the operating system to allocate more space to the table.

Record recycling requires two main functions, one to add records (here it's called `AddRec()`) and one to remove records (`EraseRec()`).

In this implementation, the `EraseRec()` function not only deletes the record, but also blanks out all fields. This provides a bit of security, but also makes it easier for the `AddRec()` function to determine if a record is a candidate for recycling. Thus, records which are only deleted (but not blanked out with `EraseRec()`) are not going to be recycled.

The `EraseRec()` function is called wherever `delete` is now used. The difference is that `EraseRec()` locks and unlocks the record for you.

The `AddRec()` function is called anytime you want a new record:

```
if AddRec()
    // Display input screen, etc...
    // ...
    dbunlock()
else
    ErrorMessage("Unable to add a new record!")
endif
```

Although `AddRec()` will work most of the time, you need to decide what to do if it fails. This depends on your application.

The Code

Let's look at the source code for the record recycling functions. In the following discussion, the program file (**RECYCLE.PRG**) is split up so that it can be examined in sections, but you can paste it back together before using it.

Possible enhancements to the source code are given at the end.

```
=====
* File Name      | RECYCLE.PRG
* Description    | Code for recycling records.
* Author        | Greg Holmes, greg@ghservices.com
* -----
* Routines      | AddRec ( )
*               | EraseRec ( )
*               | RecIsEmpty ( )
*               | EmptyType ( cType )
* =====

#include 'dbstruct.ch'
```

This section of the program file contains the descriptive header, which indicates the

names of the functions that follow. Also, the **dbstruct.ch** header file is included. It defines constants which are used as offsets into the table structure array (which is used in the `EraseRec()` function).

```

*-----
* Function      | AddRec
* Purpose       | Add a record by finding one or appending.
* Parameters    |
* Returns       | .T./.F. - whether successful.
* Assumes       | This function performs fastest if an index
*               | is open (empty records float to the top).
*               | If there is no index open, then 'append
*               | blank' is performed.
* Side Effect   | The record pointer is left pointing at the
*               | new record (if successful), or at
*               | end-of-file (if not successful).
*               | If successful, then new record is LOCKED.
*-----

function AddRec ( )
    local lResult:=.F., lDeleted

    lDeleted := set(_SET_DELETED, .F.)

    if .not. empty(ordkey()) // There is an index.
        dbgotop()           // Empty recs 'float' to top.
        if rlock()          // Lock record for integrity.
            if deleted()
                if RecIsEmpty()
                    dbrecall()
                    dbcommit()
                    lResult := .T.
                else
                    dbunlock()
                endif
            else
                dbunlock()
            endif
        endif
    endif

    if .not. lResult         // Not able to recycle.
        dbappend()          // Standard 'append blank'.
        if neterr()
            dbgobottom()
            dbskip()         // Go to end-of-file.
        else
            lResult := .T.
        endif
    endif

    set(_SET_DELETED, lDeleted)

return lResult

```

The `AddRec()` function is one of the two primary functions for record recycling, and it is responsible for finding available records. A record is a candidate if it is deleted and all of the fields are empty.

First, the function makes deleted records visible with `set(_SET_DELETED, .F.)` and saves the previous state of the flag.

Because blank values always appear first in an indexed table (unless the index is descending), the code checks to see if there is an active index. If so, the code moves to the top of the index where blank records are most likely to appear. To ensure that the record data remains consistent for the next few lines of code, the record is locked.

For speed, the code then determines if the top-most record is deleted. This check is performed before the slower `RecIsEmpty()` function is called. The `RecIsEmpty()` function is described below.

If a deleted, blank record is found, then it is recalled and the change is committed to disk. The result flag is then set to indicate success.

The last part of the function performs a standard `append blank` if the record recycling was unsuccessful. If the `append blank` fails, the record pointer is moved to end-of-file, otherwise the result flag is set to indicate success.

Finally, the deleted flag is set back to the state it had upon entry to the function. Note that the new record is left locked if the function was successful in finding or adding a new record. This copies the behaviour of the standard `append blank` command which *adds* and *locks* a record.

```
*-----
* Function      | EraseRec
* Purpose       | Blank all fields and mark record as deleted.
* Parameters    |
* Returns       | .T./.F. - whether successful.
*-----

function EraseRec ( )
    local lResult:=.F., nLen, nL, aStruct

    if rlock()
        aStruct := dbstruct()
        nLen := len(aStruct)
        for nL := 1 to nLen
            fieldput(nL, EmptyType(aStruct[nL][DBS_TYPE]))
        next
        dbdelete()
        dbcommit()
        dbunlock()
        lResult := .T.
    endif

    return lResult
```

The `EraseRec()` function blanks out every field in the record, using a `for..next` loop to visit each field. The `EmptyType()` function is described further below.

```
*-----
* Function      | RecIsEmpty
* Purpose       | Find if this record's fields are all empty.
* Parameters    |
* Returns       | .T./.F.
*-----

function RecIsEmpty ( )
    local lResult:=.T., nLen, nL

    nLen := fcount()
    for nL := 1 to nLen
        lResult := lResult .and. empty(fieldget(nL))
        if .not. lResult
            exit
        endif
    next nL

    return lResult
```

This function visits each field in a record to determine if all of the fields are empty. An optimization has been added which checks if `lResult` is false each time through the loop. This causes the `for..next` loop to stop as soon as a non-empty field is encountered.

```
*-----
* Function      | EmptyType
* Purpose       | Return an empty value of the desired type.
* Parameters    | pcType - the type to create.
* Returns       | xResult - an empty value.
*-----

function EmptyType ( pcType )
    local xResult
```

```

do case
  case pcType == 'A'
    xResult := {}
  case pcType == 'B'
    xResult := {|| NIL }
  case pcType == 'C'
    xResult := ''
  case pcType == 'D'
    xResult := ctod('')
  case pcType == 'M'
    xResult := ''
  case pcType == 'N'
    xResult := 0
  case pcType == 'L'
    xResult := .F.
  otherwise
    xResult := NIL
endcase

return xResult

```

This last function simply returns an empty value of the specified type. Most of the possible CA-Clipper data types are represented in the `do case` statement.

Enhancements

There are several alterations that can be made to the previous functions.

- *Do not blank records, just delete*

This is actually a simplification of the code that would speed up the `AddRec()` function because it would not have to check for empty fields. Also, the `aStruct` array and the `for..next` loop could be removed from the `EraseRec()` function.

- *Look harder for deleted records*

The source code above does not try very hard to find deleted records that are candidates for record recycling. Code could be added to check "several" records at the top of the file for blank fields. It might be reasonable to skip through the table for up to 2 seconds while looking for candidates. Also, the `rlock()` and `dbappend()` function calls should be attempted more than once to increase the possibility of success.

- *Use a dedicated index for deleted records*

The most elegant enhancement would include a dedicated index (for each table) that only stores deleted records. The index would be created with any key, but with a condition:

```
index on str(recno()) for deleted() to FILENAME.NTX
```

Then the `AddRec()` function would set the order to the special index and call

`dbgotop()`. If `eof()` is encountered, then there are no deleted records in the table.

This method is easier to manage if you use an RDD that supports multiple-key index files, such as the DBFCDX driver. In the DBFCDX file format, each index order is given a name, or *tag*, which is used when selecting the index order. Thus each table can have an index order called "DELETED" and there would be no conflict because each table would have its own CDX index file.

[Home](#) [Web Design](#) [Programming](#) [Fairlight CMI](#) [Soap Box](#) [Downloads](#) [Links](#) [Biography](#) [About...](#) [Site Map](#)



Send comments about this site to Greg at gregh@ghservices.com

All pages copyright © 1996-1999 [GH Services](#)™ Created 1997/08/21 Last updated 1999/09/30

All trademarks contained herein are the property of their respective owners