# A Temporary Subrecord Method for Entering Related-many Records

**Lincoln Stoller, Ph.D**

**3/8/97**



## Overview

**This article describes a temporary subrecord method for data entry into a related-many file. The method is more powerful than using arrays (the AreaList™ array management external not withstanding) and less problematic than using extended transaction processing. The method is described for one-to-many but can be extended to support many-to-many structures.**

**Subfiles provide one of the best tools for entering related records because they are memory resident and they have a record structure. In addition, 4D gives us powerful tools for subrecord entry and display that are not available for arrays. On the other hand, subrecords are usually a poor method of storing data because of the many constraints on searching, printing displaying, importing and exporting subrecords. Since subrecords are good for data entry but poor for data storage this method uses them for data entry and then copies their contents to a separate file for permanent storage — in other words subrecords are never saved to disk.**

**The temporary subrecord method requires the parent file support a subfile structure that contains a mirror image of the related-many file. When you enter the parent record input layout you create a mirror subrecord for each related-many record. Your user then performs all modifications on the memory-resident subrecords. If the parent layout is accepted the subrecords are copied to the related-many file and deleted before the parent record is saved. If the parent layout is canceled the subrecords are automatically erased when the parent is unloaded.**

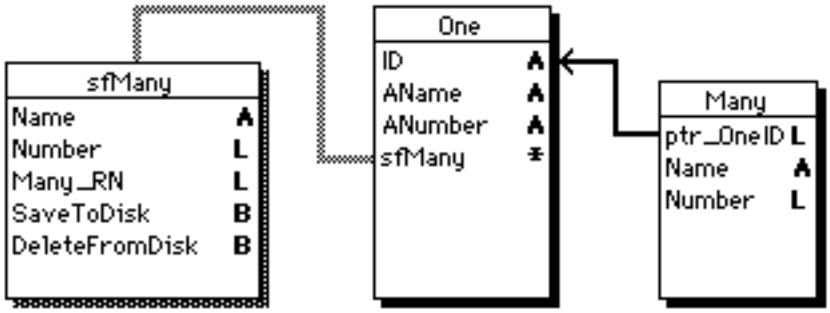


**Figure 1: Many-to-One file structure with subfile for data entry.**

**In this example we'll use the one-to-many structure shown in figure 1. Managing subrecords and their related-many counterparts requires three additional system subfields. These subfields lie at the heart of the temporary subrecord method and are described below.**

| subfield name | type | description |
|---|---|---|
| Many_RN | longint | Record number of the preexisting Many record. Set to -3 when a subrecord is created and no Many record yet exists |
| SaveToDisk | boolean | Set to FALSE whenever a preexisting Many record is first copied to subrecord. Reset to TRUE when a subrecord is modified or created. |
| DeleteFromDisk | boolean | Set to TRUE to mark the related Many record for deletion. |

**Figure 2: System subfields for subrecord management.**

**Use of System Subfields:**

**Many_RN:**

**When a One record is modified all of its related Many records are loaded into the sfMany subfile. The Many record numbers, stored with their mirror subrecords, allow you to recover the record using the Goto Record command. Record numbers are unique within each file but they can not be used for permanent identification because they can be reassigned if a record is deleted or the data file compacted.**

**SaveToDisk:**

**Few related Many records are usually changed when a parent record is modified. Consequently it is most efficient only to update Many records when their subrecord image is modified. The subfield SaveToDisk, which is set to True whenever a subrecord is modified, allows us to update only those Many records that have been modified.**

**DeleteFromDisk:**

**Handling Many record deletions can be a problem if you allow the mirroring subrecords to be erased. So instead of erasing a subrecord when the user requests it be deleted, just set its DeleteFromDisk field to True. Then search for and display only those subrecords that have DeleteFromDisk equal to false. This allows us to retain all the mirroring subrecords throughout the data entry process. When it's time to save the One record we determine which Many records to erase by searching for subrecords with DeleteFromDisk equals true.**

## Layout and Procedural Control

**Most of the code needed for handling subfiles is generic — the nongeneric aspects relate to the subrecord structure and must be hard-coded through the structure editor. The code flow is sketched in figure 3. This consists of the One record's Before phase where related Many records are copied to subrecords; the During phase where changes are made to subrecords; and the After phase where changes to the related records are written to disk.**

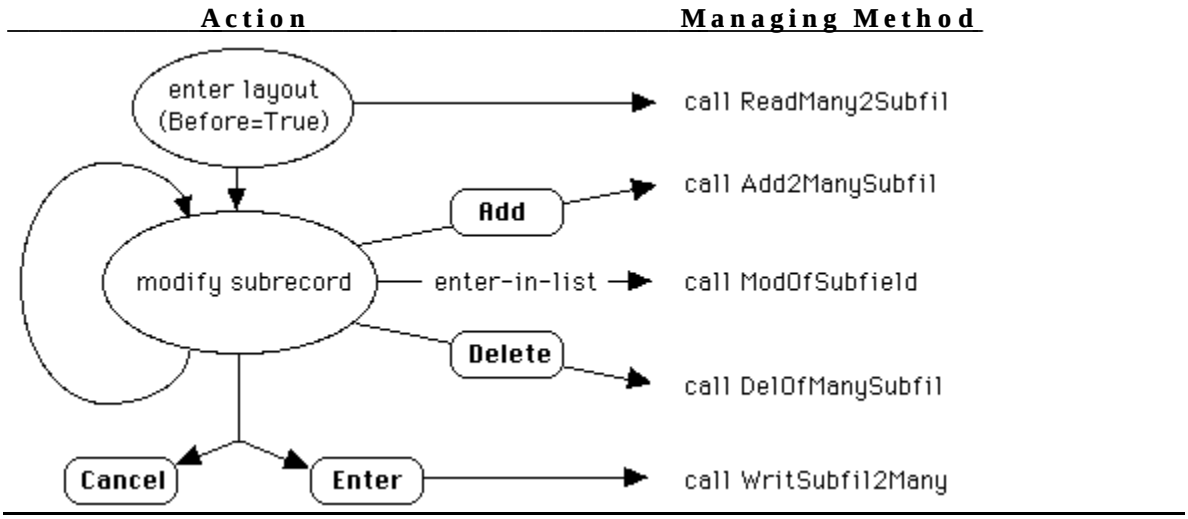


**Figure 3: Code flow for setting up and managing the input layout.**

**In the Before phase you locate the related Many records and copy them into newly created subrecords for display in an included layout. Subrecord additions and deletions are effected using an "Add to included" automatic action button and a "No action" button labeled "delete." Modifications are done using 4D's enter-in-list feature (this could be easily modified to support double clicking to open an included-record full-screen display).**

**If the layout is canceled the subrecord changes have no effect on the records in the Many file — they are automatically purged from memory when the current One record is unloaded. This automatic clearing of unneeded memory, called automatic garbage collection, is a benefit of using subrecords.**

**The Many records are only updated when the "Enter" button is pressed. The Enter button should be a "No action" type button so that you can run a related record update method before issuing the "Accept" command to save the One record. The update method processes the subrecords in a FOR loop according to the following rules.**

| If a subrecord represents a ... | then... |
|---|---|
| new Many record, | create and save a new Many record; |
| new Many record that's deleted, | skip to the next subrecord; |
| preexisting Many that's modified, | save changes; |
| preexisting Many that's deleted, | delete the Many record; |
| preexisting Many that's unchanged, | skip to the next subrecord. |

**Figure 4: Rules for updating the related Many file.**

**These rules are supported by the system subfields in figure 2 above and the "Case of" statement in figure 5.**

```
`Decision tree code section that is used in WritSubfil2Many method.
c_Longint  ($Many_RN)
$Many_RN:=[One]sfMany'Many_RN
$DeletFromDisk:=[One]sfMany'DeleteFromDisk
$SaveToDisk:=[One]sfMany'SaveToDisk
Case of
:(($Many_RN=-3) & ($DeletFromDisk=True ))
```

```
      `a new subrecord was subsequently deleted;
      `do nothing with this subrecord, go to the next.

   :(($Many_RN=-3) & ($DeletFromDisk=False ) & ($SaveToDisk=True ))
      `a new subrecord is to be saved to disk;
      `create a new Many record, copy the subrecord information to it and save to disk.
      Create Record   ([Many])
      [Many]ptr_OneID:=[One]ID
      [Many]Name:=[One]sfMany'Name
      Save Record   ([Many])

   :(($Many_RN>=0) & ($DeletFromDisk=True ))
      `a preexisting Many record has been marked for deletion;
      `locate the Many record and delete it.
      Goto Record   ([Many];$Many_RN)
      Delete Record   ([Many])

   :(($Many_RN>=0) & ($DeletFromDisk=False ) & ($SaveToDisk=True ))
      `A preexisting Many record that has been modified;
      `locate the Many record, copy subrecord values to it and save.
      Goto Record   ([Many];$Many_RN)
      [Many]Name:=[One]sfMany'Name
      Save Record   ([Many])

   :(($Many_RN>=0) & ($DeletFromDisk=False ) & ($SaveToDisk=False ))
      `This represents a preexisting Many record that has not been changed;
      `do nothing with this subrecord, go to the next.
   End case
```

**Figure 5: "Case of" statement in the WritSubfil2Many method.**

**What is described so far is incomplete because it does not check the read/write status of the Many records before updating. Since each process in 4D version 3 can establish its own record locking, write access privelages must be confirmed even in single user applications. Making this scheme multi-process aware ( or multi-user aware) requires only the addition of transaction processing while Many records are being modified and canceling the transaction if any of them are found to be locked.**

### Adding multi-user/ multi-process compatibility

**The only time we need read/write access to the Many file is when changes are saved through the WritSubfil2Many method. But it is possible that one or more of the Many file records will be locked, and if this occurs changes need to be**

**rolled back and the update canceled. This is achieved by starting a transaction before looping through each subrecord and either validating or canceling the transaction after exiting the loop. The final version of the WritSubfil2Many method is shown in figure 6.**

```
`method WritSubfil2Many
`final version.
c_Longint ($j)
$SomeLocked:=False
Read Write ([Many])
Unload Record ([Many])
All Subrecords ([One]sfMany)
Start Transaction
For ($j;1;Records in Subselection ([One]sfMany))
   -- "Case of" statement from figure 5 --
   If (Locked ([Many]))
      $j:=$j+Records in Subselection ([One]sfMany)
      $SomeLocked:=True
   End if
   Next Subrecord ([One]sfMany)
End For

If ($SomeLocked)
   Cancel Transaction
   Alert ("One of the related Many records was in use and could not be updated. "+
   "Try entering your changes at a later time.")
Else
   Repeat      `Delete all subrecords, subrecords are never saved.
      Delete Subrecord ([One]sfMany)
      All Subrecords ([One]sfMany)
   Until (Records in subselection ([One]sfMany)=0)

   Validate Transaction
End if
```

**Figure 6: Code changes needed to make the method multi-user/multi-process compatible.**

---

**-- SIDEBAR --**

**The Difficulty of Using Transactions**

**An alternative to using subfiles as described above is to start a transaction in the Before phase of the one record, make modifications directly to the related Many records through an included layout, and then validate or cancel the transaction according to whether the layout is accepted or canceled. I'll call this "extended transaction processing," and there are several problems make this a poor alternative.**

**The first problem is that every time the user makes and saves a change to a Many record, that record is locked and remains locked to every other person and process until the transaction is completed. Both the One and Many records will be locked for an unpredictable length of time which can have adverse effects on other users or processes.**

**The second problem relates to the global scope of transactions. This means that once a transaction is started it effects all actions, through any layouts, until it is completed. If a transaction is started, the user makes changes and then switches to another file, all of the changes will be saved or rejected depending on how transactions is completed. As the developer you probably won't control what files the user visits in the course of the transaction and the final result may be unexpected, especially if the actions appeared independent to the user.**

**Since only one transaction is supported per process — calls to start subsequent transactions will have no effect, while calls to complete a transaction will act to prematurely complete whatever transaction is in progress. Consequently, if you use a transaction for the duration of the input layout, it is essential to insure that operations outside the input layout do not begin and complete transactions of their own.**

**In order for your code to know whether or not a transaction is in progress your code must be "transaction-aware." To make your code transaction-aware you need to manage your own global variable that counts how many times your code has attempted to start a transaction and how many times its attempted to complete one. Once a transaction is in begun it should only be completed by the**

same section of code that initiated it. You can do this by replacing all or your calls to Start Transaction, Validate and Cancel Transaction with the appropriate call to the SysTransaction method shown below.

```
`method SysTransaction
`$1 = action to perform.
`Longint global variable TransLevel must be initialized to 0 at startup.
Case of
:($2="Start")
    if (TransLevel=0)
        Start Transaction
    end if
    TransLevel:=TransLevel+1

:($2="Validate")
    if (TransLevel=1)
        Validate
    end if
    TransLevel:=TransLevel-1

:($2="Cancel")
    if (TransLevel=1)
        Cancel Transaction
    end if
    TransLevel:=TransLevel-1
End case
```

**Figure 7: Custom transaction processing method that tracks the number of times transaction processing is initiated and completed.**

While the SysTransaction method is necessary for extended transaction management it does not solve all the problems inherent in this technique. For instance, consider what happens if a transaction is in progress and the user enters information into another record through a second layout that's also controlled by an extended transaction. Imagine that the user cancels the second layout, returns to the first layout and accepts it. In this case it is unclear what will happen with the data entered through the second layout.

Extended transactions work against a modular code structure. If you allow the user to perform functions in other files while a transaction is in progress then you pass the critical responsibility for transaction management to unspecified extensions and perhaps even to other programmers. Should some section of code

**make too few or too many calls to SysTransaction the ongoing transaction will be completed either too soon or too late. In either case the integrity of your data will be compromised.**

**Using an extended transaction for handling related record input suffers many drawbacks. The subfile method is preferable because it offers more control and supports better design.**

**-- end of sidebar --**

## The Completed Program

**The layout method for the parent record is shown in figure 9. The procedure to copy the related Many records to subrecords is shown in figure 10. And the remaining three methods for data entry are shown in figure 11.**

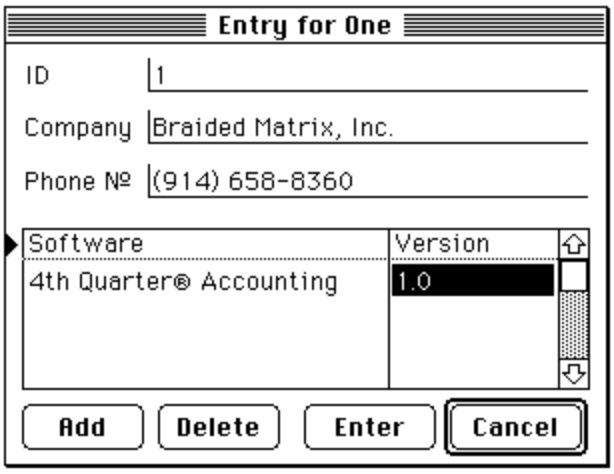


**Figure 8: the One file input layout.**

```
`Layout method for [One];"One_Input"
c_Longint  ($Err)
Case of
: (Form event  =On Load)          `Entering the input layout.
    ReadMany2Subfil

: ((Form event  =On Keystroke) | (Form event  =On Clicked))
    If (bnaEnter=1)          `The ENTER key was pressed.
        $Err:=WritSubfil2Many
        If ($Err=0)          `Related Many records were updated successfully.
            Accept           `Save the One record and exit the layout.
        End if
    End if
End case
```

**Figure 9: the One file input layout method.**

```
`Method ReadMany2Subfil
`called in BEFORE phase to copy Many records to subfile.
c_Longint  ($j)
Relate Many  ([One]ID)
For ($j;1;Records in selection     ([Many]))
    Create Subrecord     ([One]sfMany)
```

```
    [One]sfMany'Name:=[Many]Name
    [One]sfMany'Many_RN:=Record Number    ([Many])
    [One]sfMany'SaveToDisk:=False
    [One]sfMany'DeletFromDisk:=False
    Next Record   ([Many])
End for
```

**Figure 10: Method to copy the contents of the Many records to the subrecords.**

```
`Method Add2ManySubrec
`called by script in ADD button to create a new subrecord.
Create Subrecord    ([One]sfMany)
[One]sfMany'Name:=""
[One]sfMany'Many_RN:=-3
[One]sfMany'SaveToDisk:=True
[One]sfMany'DeletFromDisk:=False

`Method ModOfSubfield
`called by tabbing out of included layout field after a modification.
[One]sfMany'SaveToDisk:=True

`Method DelOfManySubrec
`called by script in DELETE button to mark Many record for deletion.
[One]sfMany'DeletFromDisk:=True
Search Subrecord    ([One]sfMany;[One]sfMany'DeletFromDisk=False )
```

**Figure 11: Button and field triggered methods for subrecord modifications.**

## Summary

**I've detailed a technique that uses temporary subrecords for the display and management of related Many records. The method makes full use of 4D's powerful subrecord features and avoids the limitations of subrecord storage by using only regular files for permanent data storage.**

**The method is easily extendable to accommodate changing file structures and is generic enough to be encapsulated in five global and one layout method (cf. figure 3). The subfile method avoids the pitfalls associated with extended transaction processing, minimizes impact on write access to data, and supports modular program design.**