Summit '98

The Joy of Triggers

by David Adams and Dan Beckett

Triggers

Introduction

Triggers are table methods used to control and validate actions that affect records or the table as a whole. Triggers provide a mechanism for maintaining data and relational integrity. In this chapter we explain what triggers are, how they're used in 4D, how to write them, and a lot of details you'll need to construct a reliable, high-performance system.

About Triggers

Triggers Enforce Rules

Databases have rules about how data is organized and related:

Whenever an invoice is deleted, all line items must be deleted, too.

Whenever the on-hand supply of a part drops below the reorder point, a purchase order reminder must be created.

Each employee ID number must be unique.

Triggers are the mechanism that enforces the data and relational integrity rules for your system. Triggers provide an opportunity to validate actions—and stop them if necessary—before they are committed. It also gives you a chance to update related data and calculated fields whenever source data changes.

Why Triggers Are Great

Triggers are great because they run *whenever* data changes. There's no way for something to "slip through the cracks". Triggers run when records are changed from any of these sources:

- ❖ A custom method
- ❖ User action in a form
- ❖ Record import
- ❖ A Web connection
- ❖ A 4D Plug-in
- ❖ A 4D Open client

The standard rules you define for your tables are enforced no matter what. The logic resides on the server, and needs to be written only once. You don't need to write different code for different types of client software, and you don't have to worry about forgetting to write the necessary code. Triggers also give you a chance to validate a table's rules before committing the record. If there is any problem, you can halt the operation.

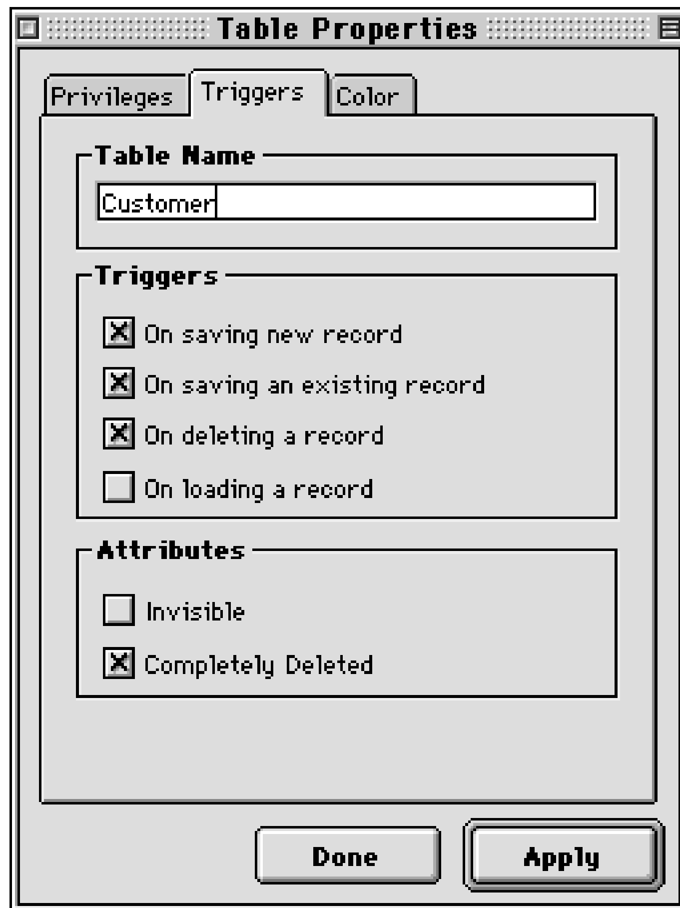
Why Are They Called Triggers?

The term “trigger” is borrowed from the SQL world for a similar mechanism. 4D uses this term so SQL programmers can understand what 4D is doing. In 4D the term “trigger” refers to two related concepts:

1) Database events: Special record-related events that invoke trigger code.
2) Table methods: The code that runs in response to database events.

Database Events

You use the Table Properties dialog to selectively activate the database events a particular table responds to.



The trigger tab of the Table Properties dialog.

In this context “trigger” means the database events that cause your trigger code to run. For example:

> *When the customer delete record trigger runs, we need it to find and delete all related call history records.*

The events listed in the triggers tab of the Table Properties dialog are called “database events” in the 4D language. You can think of them as “table events” or “record modification events”. The idea is that you have a chance to run code whenever changes to the table are committed.

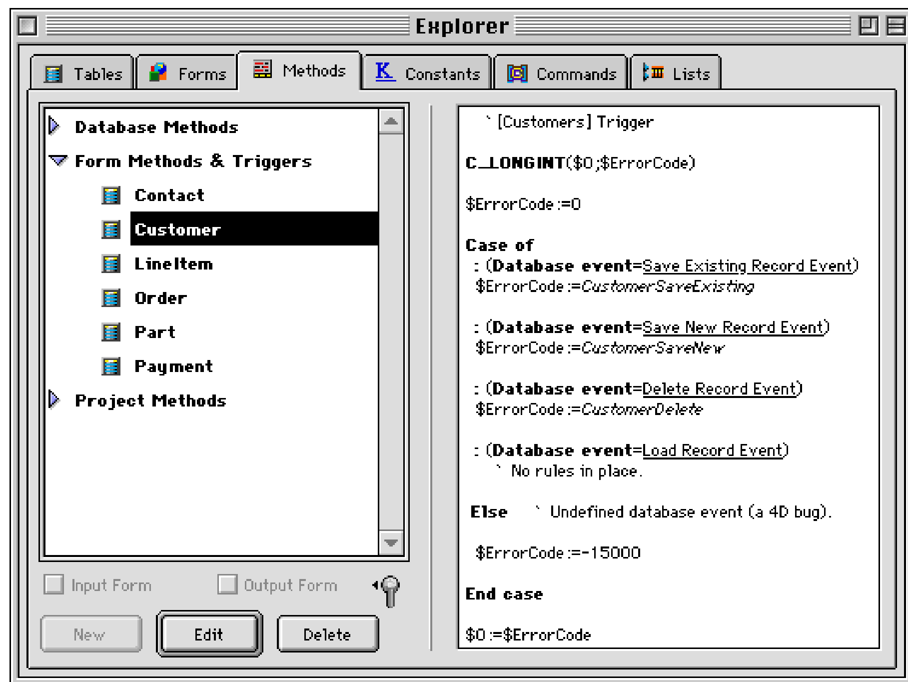
You can trap four different events:

1) A new record is being saved.
2) An existing record is being saved.
3) An existing record is being deleted.
4) An existing record is being loaded.

The loading record event is rarely useful, as we’ll discuss later on. People often ask about a “creating new record event.” This would be a useful place to put code that initializes fields and assigns sequence numbers. 4D does not include such an event because creating a record does not actually change the data. It is only when you save a new record that the data file is updated. 4D provides distinct events for saving new records and saving existing records for optimization and convenience.

Triggers are Table Methods

Triggers are really another name for “table method” in 4D. You can find a table’s trigger in the Explorer just where you would expect to find a “table method”:



Triggers are table methods.

Triggers Execute On the Machine with the Database Engine

The biggest gotcha with 4D triggers is that they execute on the machine where the database engine runs. This means that under 4D they run on the current machine, and under 4D Server they run on the server machine. The effect of this is that your code can behave *very* differently under 4D Server than under 4D. Here are the key differences:

- ❖ If you display a window from inside of a trigger under 4D, the user sees the window. Under 4D Server it appears on the server machine!
- ❖ Under 4D Server the trigger has *no* access to the variables on the client machine.
- ❖ Under 4D Server the trigger does not have access to its own table of process variables.

Before you get too alarmed, you should know that a trigger shares several things with the client:

The current record in each table.

The current selection in each table.

The read/write state for each table.

The locked state for each record.

Process sets.

Process named selections.

Variables are the biggest area where there is a difference. Under 4D each trigger shares the process variables with the current process. Under 4D Server all triggers share a single set of process variables on the Server machine.

Triggers are Functions

Triggers are *functions* that return an error code, or 0 if there was no problem. You can return 4D error codes, or your own custom error codes. If you can't perform a necessary action inside a trigger—perhaps you can't delete a related record—then returning an error code halts the operation. You can use numbers from -15,000 to -32,000 for your custom error codes. This result is returned to the client machine for handling. We'll explain how to handle errors shortly.

Only One Trigger Runs at a Time

Triggers run at a very low level of the database engine. Only one trigger cascade executes at a time in the entire system. (We look at trigger cascades in detail next.) Under 4D Server only one trigger cascade runs at a time for all processes and clients. This means that in a 50-user system, when a trigger is running based on one client action, no other trigger can start executing. (Imagine what happens when one of your triggers stops because of a coding error!) In real-world systems triggers perform well when used appropriately. The one-trigger-at-a-time rule does not prevent other processes from running simultaneously. Stored procedures, Web connections, and other client processes continue to execute unless they are trying to run a trigger.

The one trigger at a time rule is not documented by ACI because they may implement simultaneous triggers in the future. The defensive programming strategy is to assume that ACI will change the one-trigger-at-a-time rule in a future version. Code your system to respect today's one trigger at a time behavior, but don't rely on it. Keep your triggers small, fast, and error free. Don't program triggers that check for uniqueness assuming that no other trigger could be running simultaneously. If you need to ensure that only one trigger runs at a time, use a semaphore to "lock" the operation.

Note *If you're verifying uniqueness then the "unique" field attribute is the most reliable method, even today. If you use triggers to verify that a combination of fields is unique it works today because of the one-trigger-at-a-time rule. Or does it? If your code runs inside of a transaction then it is possible for multiple processes to save the same combination of "unique" fields. The only 100% reliable way to insure uniqueness is to use a field with the unique attribute set to true.*

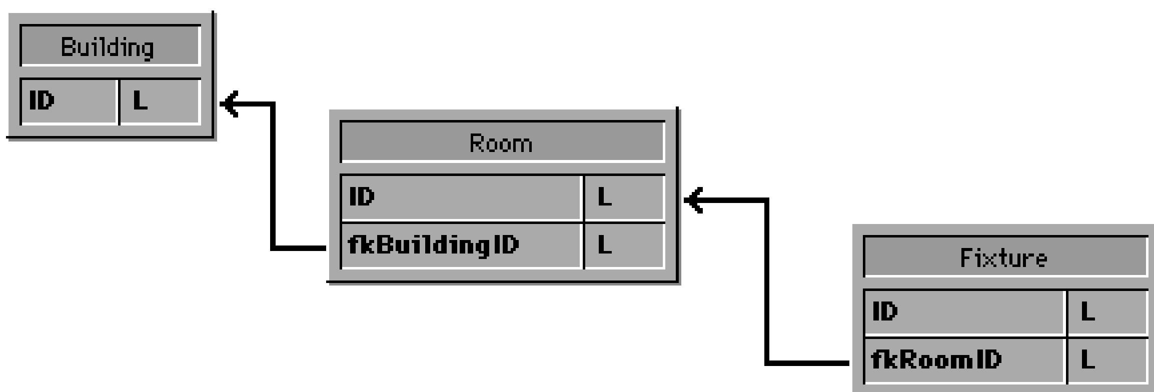
Trigger Cascades

Introduction

When a trigger perform an action that invokes another table's trigger you have what is called a *trigger cascade*. In other words, if a trigger changes data in another table then that other table's trigger may also run. This is exactly how triggers should work. If a trigger could change records in another table without invoking that other tables' triggers, you could not protect your data and relational integrity, which is the entire point of triggers in the first place. Triggers may cascade to related or unrelated tables; the key is that one trigger has invoked another.

Example

Imagine this three-table relationship:

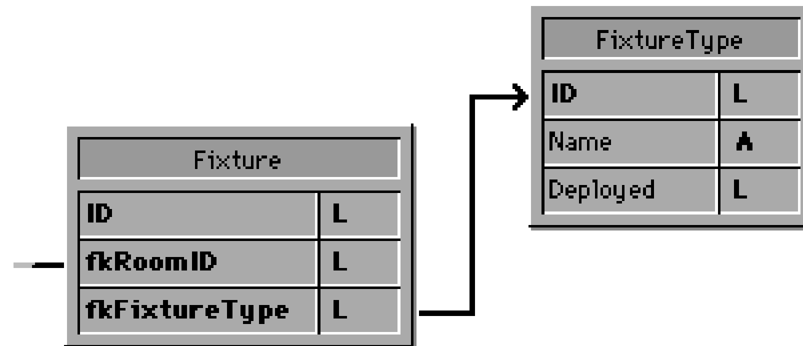


When you delete a [Building] record its trigger deletes all related [Room] records. The [Room] trigger in turn deletes all related [Fixture] records. This cascade of triggers and deletions is exactly what you want to avoid orphaned records in your system. This is how cascading triggers are designed to work.

 © David Adams and Dan Beckett

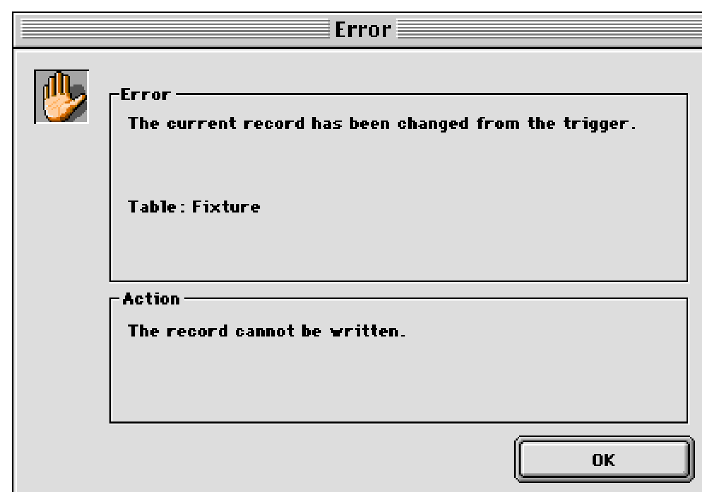
Gotcha

Consider, however, what happens when we modify the example structure with a fourth table:



The FixtureType is now tracked in a new table.

The [FixtureType]Deployed field includes an up-to-the-moment sum of how many of each fixture type are deployed in the field. This kind of summary field can deliver enormous practical benefits, and is relationally defensible. But where does the field get updated? One thought would be to change the count whenever a [FixtureType] record is changed. In our example this would mean that the count would get out of date when the [Fixture] record is deleted. The [Fixture] trigger should update the value because it is the source of the change. This works perfectly. The hazard to watch out for is putting the update code in two locations. If you also have the [FixtureType] trigger update the current count, what happens? When the [Fixture] changes the [FixtureType] record and saves it the [FixtureType] trigger runs and changes the selection in the [Fixture] table. Since 4D is currently deleting a selection of [Fixture] records you've got a problem. Changing the selection of a table used in an earlier trigger is like pulling the rug out from under your feet.



4D often warns you when you've changed the selection illegally.

Avoiding Trouble In Cascading Triggers

The 4D language includes commands to help you avoid trouble in cascading triggers. The **Trigger level** function tells you what level you're at. The first trigger executed is level 1, the second is level 2, and so on. The **TRIGGER PROPERTIES** lets you find out the current record, table, and database event for any current trigger level. This allows your triggers to behave differently depending on how they are invoked. If, for example, you wanted to say that [Fixture] records can only be deleted by the [Room] trigger then the [Fixture] trigger would include code like this:

```
C_LONGINT($0;$errorCode)

Case of
 : (Database event=Delete Record Event)

   ` Deletion is allowed only from the [Room] trigger.
   ` [Fixtures] cannot be deleted directly.

  If (Trigger level<2)

    ` Level is less than 2? We don't allow direct deletion of [Fixture] records.
    ` Return an error code:

   $errorCode:=-16000

  Else

    ` Get the properties of the invoking trigger (Trigger level - 1).
   TRIGGER PROPERTIES(Trigger level-1;$invokingEvent;$invokingTableNum;$recordNum)

    ` What we want to see is that this was invoked by the [Room]
    ` delete record trigger

   $roomTableNum:=Table(->[Room])

   If ($invokingEvent # Delete Record Event) | ($roomTableNum # $invokingTableNum)
      ` Wrong event or table, return an error.

    $errorCode:=-16000

   Else
      ` It's OK to proceed with the deletion.
    $errorCode:=0

   End if   ` ($invokingEvent # Delete Record Event) | ($roomTableNum # $invokingTable-
    Num)

  End if   ` (Trigger level<2)

End case

$0:=$errorCode
```

That looks like a lot of code just to avoid trouble! Whenever possible avoid intricate interdependencies among triggers. The simpler the design, the easier it is to implement, test, and maintain. Now let's take a look at how to write triggers that do what you need, work reliably, and are readable.

How to Write Triggers

Cutting Edge Technique: Paper and Pen

The easiest way to write triggers is to start with a piece of paper. Write down the rules for each table in your database narratively. Before you write any code answer these questions for each table:

1) What has to happen each time a new record is saved?
2) What has to happen each time an existing record is saved?
3) What has to happen each time an existing record is deleted?
4) What has to happen each time an existing record is loaded?

If you built your database from a formal design you already have these rules documented. If you built your database without documenting these rules, it's not too late. Putting these rules on paper costs nothing and provides these benefits:

1) It makes it simple to implement the code.
2) It makes it possible to verify that your triggers are complete.
3) It makes it possible for another programmer to work with your system.

We've included trigger development worksheet at the end of this chapter as a reminder of what you need to know before writing a trigger.

Trigger Format

Here's a sample of what a trigger looks like:

```
   ` [Customers] Trigger

C_LONGINT($0;$ErrorCode)

$ErrorCode:=0

Case of
   : (Database event=Save Existing Record Event)
      $ErrorCode:=CustomerSaveExisting

   : (Database event=Save New Record Event)
      $ErrorCode:=CustomerSaveNew

   : (Database event=Delete Record Event)
      $ErrorCode:=CustomerDelete

   : (Database event=Load Record Event)
      `   No rules in place.

   Else`   Undefined database event (a 4D bug).

      $ErrorCode:=-15000

End case

$0:=$ErrorCode
```

We've selected this format for several reasons. The qualities to notice are:

> The trigger always return a result through $0 even if no error is expected. This emphasizes that the trigger is a function, and makes it easy to add error results later.
>
> All possible and *impossible* database events are tested. This makes it explicit that each possibility has been considered, not neglected.
>
> Each event calls a subroutines instead of putting the functional code directly in the trigger.

The way we call subroutines surprises some people, so we'll add a few comments. We believe that using project methods for the "guts" of your triggers provides these benefits:

1) The trigger is small and loads quickly. Only the code for the current database event is loaded.
2) It makes it easy for two events to call the same routine, like both of the save events above.
3) It makes it easy for different tables to share code.

Error Handling

Overview

If you're using triggers, you need to include error handlers. Triggers give you a chance to stop operations that should not take place. Triggers return a longint result in $0. If you return a 0 this says "the operation was fine" and the requested operation (save or delete) is committed. If you return an error code then the requested operation does not take place. In order for this to work correctly you should always install a custom error handler with **ON ERR CALL** to trap and handle trigger errors. When you receive one of your custom errors you can give the user guidance on how to correct the problem.

4D Versus 4D Server

In every version of 4D we've tested error handling for the same trigger differs when deployed under 4D and 4D Server. If you are going to deploy under 4D Server then you should do the following:

- ❖ Test under 4D Server.
- ❖ Test under 4D Server.
- ❖ Test under 4D Server.
- ❖ Install a custom error handler on the client side with **ON ERR CALL**.
- ❖ Install a custom error handler on the server side with **ON ERR CALL**.

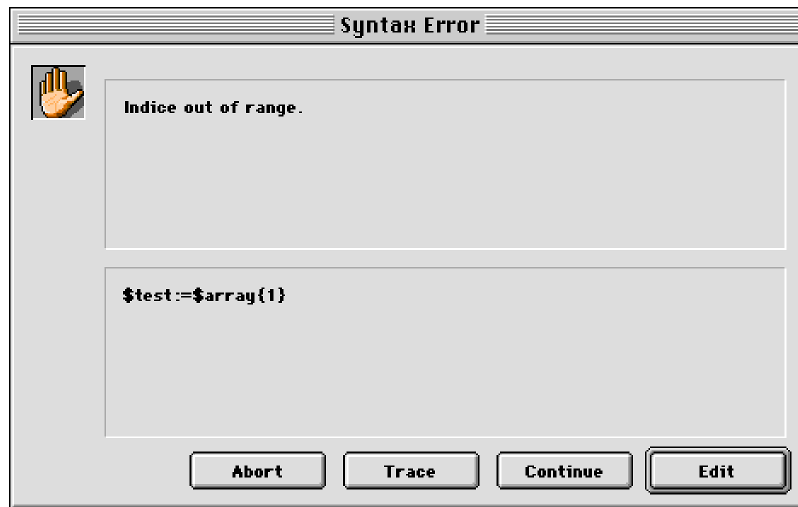
Do not consider trying to save time by avoiding any of these steps as you will almost certainly end up with errors which take even more time to correct.

Why You Need a Server-Side ON ERR CALL Method

A custom error method installed on the server machine from inside of a trigger traps code errors, and allows you to detect errors in a trigger cascade. Imagine, for example, that you have a trigger that includes this code:

```
ARRAY LONGINT($array;0)
$test:=$array{1}
```

This code produces a range error because it's asking 4D to read an array element that does not exist:



Not what you want to see on the server machine.

Think about what happens if this dialog appears on the server machine. It means that the current trigger is halted, and that the requesting process is halted waiting for the dialog to be dismissed. The problem is not limited to the current user: it spreads to any other process that invokes a trigger. Only one trigger executes at a time globally, so an error like this stops more and more clients as they perform work that invokes triggers. Since their triggers are waiting in line behind the stalled trigger their process appears frozen. This looks particularly odd because other process continue to run…until they invoke a trigger. Ideally you should remove all such errors from your code, but an **ON ERR CALL** routine lets you trap and handle the error without delaying the trigger indefinitely.

You also need to use a custom error handler inside of a server-side trigger to detect errors from a trigger cascade.

> [Building] trigger invokes [Room] trigger.
>
> [Room] trigger returns an error.

How does the [Building] trigger read the error returned from the [Room] trigger? It reads the 4D system variable called Error. In our tests the Error system variable is populated reliably only if you have a custom error handler installed.

 © David Adams and Dan Beckett

Installing Server-Side Error Methods

All triggers and several database methods share a process variable table, and a current error method. The most straightforward way to handle errors on the server is to install one error handling method in **On Server Startup**. If you are performing interpreted testing you should also initialize the Error system variable:

```
Error:=0
ON ERR CALL("trapErrorsOnServer")
```

Trigger Optimization

Keep Triggers Short and Fast

It is rarely sensible to worry about small speed differences. Triggers are one of the times when it is sensible to worry about the small things. Your trigger code should include whatever is required, and nothing else. The following phrase should be your guide for trigger construction:

Honor sufficiency.

Because triggers run often—and only one runs at a time—a slow trigger can cause noticeable performance problems. Do not read this to mean “triggers are slow” or “triggers make your system slow.” Triggers are a very powerful feature, and execute quickly. Our point is to emphasize that triggers are designed for certain tasks and that you should use them appropriately. If you use them as designed and as intended then you will be happy with them.

Avoid The On Load Trigger

There are three reasons that you should avoid the Load Record Event database event: first, it slows everything down, second, you probably don’t need it, and third, it doesn’t work.

The Load Record Event database runs once for *each* record loaded in a selection. This means it runs once for each record displayed in an output form. Under 4D Server, the trigger runs on the server machine. So a long trigger, executed for each record at an output form on a single client machine, reduces the server processing time available to *all* other processes and clients. Because triggers can run frequently, it is essential that you make sure that only the events you use are turned on in the table properties dialog, and that your trigger code executes quickly and is error free.

You probably don’t need the Load Record Event database event. Loading a record is not a data modification event. Apart from tables that must have each loading recorded for detailed auditing there is no appropriate use for this event. Interface-related operations and initialization are not appropriate for a trigger. Remember, triggers only enforce rules about data.

The Load Record Event database event does not always run for performance reasons. One of the ways 4D and 4D Server optimize performance is to use field indexes rather than loading entire records. This is a good thing. The Load Record Event database event does not run unless the record is loaded. This means that common

© David Adams and Dan Beckett

commands like **QUERY** do not invoke the Load Record Event database event for each record in the selection. According to ACI's documentation it is difficult to determine when this event will and will not be invoked reliably.

Triggers and Transactions

Rules

You cannot start or stop transactions inside of triggers. Doing so could change the current record, which is a forbidden action during the record commit period. Transactions must be managed by the invoking process. In other words, if a trigger performs actions that may need to be rolled back, the trigger must already be in a transaction stared by the client machine. Your triggers can test if a transaction has already started with the **In transaction** function. Here is an outline of how triggers and transactions fit together:

```
Start transaction on from invoking process.
   Trigger and all cascaded triggers run on database engine machine.
   Trigger returns an error code or 0 for no error.
If (There was no error in the triggers)
   Validate transaction from invoking process.
Else
   Cancel transaction from invoking process.
End if
```

We say "invoking" process instead of "client machine" because the invoking process can legitimately be any of the following:

- A global process under 4D.
- A Web process under 4D.
- A global process under 4D Client.
- A Web process under 4D Client.
- A stored procedure under 4D Server.

The point is that the trigger (or trigger cascade) is wrapped inside of a transaction, but does not manage the transaction itself. The invoking process uses the result returned by the trigger to determine if the transaction should be validated.

Automatic Action Buttons

If you use automatic action buttons that invoke triggers (accept, delete, first, previous, next and last record) they can be used in conjunction with triggers only if their object methods do not contain transaction management code.

When a coded automatic action button is clicked, the object method code executes first, then the trigger is invoked. So, if the input form is executing in a transaction, the transaction must be managed in the button method, which means the transaction is validated/cancelled before the trigger runs. This is a significant problem.

If, for instance, you were to execute **VALIDATE TRANSACTION** in the method of an automatic action next record button, the transaction would be validated before the trigger runs, meaning the trigger's actions take place outside of the transaction and cannot be rolled back. This creates serious data integrity problems if the trigger encounters a locked record it needs to modify or, for any other reason, it cannot execute successfully. The calling process has already validated the transaction before the trigger has determined if the user can leave the form!

In order to avoid this situation you must avoid using automatic action buttons on forms that execute in transactions and invoke triggers. Instead, you should use coded no-action buttons. In the button method, execute **SAVE RECORD** first, which causes the trigger to run. If the trigger returns an error, **SAVE RECORD** is blocked by 4D and the subsequent object method code can react to the error appropriately. Here is an example of a simple no action next record button method that can be used if the form executes in a transaction:

```
` This causes the trigger to run and display an alert if there is an error.
SAVE RECORD([Table1])

If (Error=0)  ` If the trigger succeeds...
   VALIDATE TRANSACTION
   NEXT RECORD([Table1])  ` Go to the next record.
End if
```

Restrictions on Trigger Code

Do Not Change the Current Record Or Current Selection

Triggers run when a data modification is about to happen. It is a delicate moment between the time a change is requested and the database is actually updated. We call this the "record commit phase." At this time 4D has a copy of your record in memory ready for saving or deleting. You can reject an event entirely (like stopping a delete) by returning an error code in $0. You *must not* do anything inside the trigger that changes the current record or the current selection of the current table, or any table earlier in the current trigger cascade. Here is a list of commands to avoid

Commands to Avoid in a Trigger Cascade

ALL RECORDS	**ARRAY TO SELECTION**
Before selection	**CREATE RELATED ONE**
DELETE RECORD	**DISTINCT VALUES**
End selection	**EXPORT DIF**
EXPORT TEXT	**GOTO RECORD**
IMPORT DIF	**LAST RECORD**
LOCKED ATTRIBUTES	**Min**
OLD RELATED MANY	**ONE RECORD SELECT**
ORDER BY FORMULA	**PREVIOUS RECORD**
QUERY	**QUERY SELECTION**
READ ONLY	**READ WRITE**
Records in selection	**RELATE MANY**

 © David Adams and Dan Beckett

Commands to Avoid in a Trigger Cascade

RELATE ONE	SAVE OLD RELATED ONE
SAVE RELATED ONE	Std deviation
Sum squares	USE NAMED SELECTION
Variance	APPLY TO SELECTION
Average	CREATE RECORD
CUT NAMED SELECTION	DELETE SELECTION
DUPLICATE RECORD	FIRST RECORD
EXPORT SYLK	IMPORT TEXT
GOTO SELECTED RECORD	IMPORT SYLK
LOAD RECORD	Max
NEXT RECORD	OLD RELATED ONE
ORDER BY	POP RECORD
PUSH RECORD	QUERY BY FORMULA
QUERY SELECTION BY FORMULA	Read only state
RECEIVE RECORD	REDUCE SELECTION
RELATE MANY SELECTION	RELATE ONE SELECTION
SAVE RECORD	SCAN INDEX
Sum	UNLOAD RECORD
USE SET	

To help you avoid calling the wrong commands the sample code includes a function called *tableLevelInCascade*. This function takes a table pointer and returns that table's current trigger level, or 0 if it's not in the current cascade.

More Comments on Not Changing The Current Record or Selection

For years the restriction on what you can do in the **After** phase (4D V3) or a trigger has been described as "you must not change the current record of the current table". If you've heard this rule, and you're like most people, you'll wonder why many of these commands are forbidden. Why can't you execute **SAVE RECORD**, for example? (Well, for one thing, it might cause an endless loop if you used it in a trigger. Saving the record would call the trigger, which would call the trigger, which would call the trigger...) If you watch the current record with the debugger you'll see that the current record does not appear to change when you execute many of the commands in the table of forbidden commands. So why are they forbidden? This is why we spent time earlier explaining about 4D/4D Server record management. The restriction is better described as "you must not change the current record, or force 4D to load a fresh copy of the current record." **SAVE RECORD**, and many of the other commands listed above, reload the current record as part of their behavior. Some commands reload an automatically related one record when applied to a many table, which is why they cannot be used inside of certain trigger cascades.

Other Command Limits

Several commands have limited use within triggers, or potentially create performance problems:

Commands with limited use in triggers.

Command	Limit
BLOB TO DOCUMENT	Usable but to be avoided for performance reasons.
DOCUMENT TO BLOB	Usable but to be avoided for performance reasons.
EXECUTE	Allowed, but you should not use any of the forbidden commands.
PAUSE PROCESS	Do not use on the current process.
SELECTION RANGE TO ARRAY	Usable but to be avoided for performance reasons.
SELECTION TO ARRAY	Usable but to be avoided for performance reasons.
SET CHANNEL	Usable for file operations, but not for modem operations.
Open window	Do not use interface commands on the server machine.
ALERT	Do not use interface commands on the server machine.
CONFIRM	Do not use interface commands on the server machine.
DIALOG	Do not use interface commands on the server machine.
MESSAGE	Do not use interface commands on the server machine.
Request	Do not use interface commands on the server machine.

Remember that only one trigger executes at a time, so a slow operation could be noticed by several users. You should not open windows of any kind as they appear on the server machine where the user may not be able to reach them. You may use the **TRACE** command during development if you have access to the server machine.

Trigger Development Worksheet

Complete one copy of this form for each trigger event in each table.

Database Name	
Table Name	
Trigger (*Check one)*	■ **Loading a record** ■ **Saving a *new* record** ■ **Saving an *existing* record** ■ **Deleting a record**
Method Name	
Action(s)	
Error Codes *Include descriptions*	

 © David Adams and Dan Beckett