Chapter 5
Rules, Triggers and Referential Integrity

Developers have gotten along just fine for many years without having rules, triggers and referential integrity code built into their databases. As a result, it's easy to think that this set of Visual FoxPro features can be ignored. Even though we've done without these particular tools, we *have* been enforcing rules and referential integrity all along. In the past we had no choice but to enforce rules and protect the integrity of our data in our forms or procedural code. Now, we have some options that are worth considering; when judiciously applied, they can save us some work and improve the quality of our applications.

Rule and trigger functions

Many lines of code have been written to protect the integrity of our databases. Consider the following issues that are probably representative of issues you've encountered:

- Deletion of the last line-item of an invoice
- Entering a client without a case number
- Entering a line-item on an invoice for an item that is only returned, never sold (like an empty acetylene tank)
- Entering a purchase order with a specified shipping method of "UPS" when only "Motor Carrier" and "Air Freight" are valid shipping methods for this vendor
- Changing an invoice number
- Deletion of a customer when there is active or inactive historical information related to this customer
- Entering a quantity of 2 items that sell for $5 each, and showing the extension correctly as $10
- Deletion of an invoice with 112 line items
- Entering a birth date of 12/15/1928 for a 6-year-old child

Much of what we do to protect the integrity of our systems' data falls into one of three categories:

- Domain constraints—Limiting field values to those permitted for the type of data that the field represents.
- Internal consistency—Ensuring that no two fields contain logically inconsistent data.
- Referential integrity—Limiting the occurrence of "orphan" records.

Let's consider a few examples based on the preceding list.

If the user chooses to delete an invoice, it would be nice if all we had to do was to issue the DELETE command on the invoice record and commit the change. However, in the real world, we should also delete those 112 line items. This is an example of a referential integrity function. We are eliminating the possibility of "orphan" records in the line-items table.

A customer ordered three $5 widgets for a total of $15, but because the stock was short, one of the three items was backordered and only two were shipped. The user edits the order prior to invoicing, changing the quantity shipped to 2. The total needs to be changed from $15 to $10. This is an example of an internal consistency function. The unit price, the quantity sold, and the extended amount must be logically consistent and mathematically correct.

If the user enters a birth date of 12/15/1928 into a patient record in a pediatric medical practice management program, we might have some code to check the age of the patient. If the calculation results in an age greater than 18 (or whatever criteria the practice has for its clientele), we could present the user with a message indicating that this value is not appropriate, and require that it be corrected before committing changes to the record. This represents a domain constraint function.

Thus, the question is not *whether* our applications need to deal with these situations; it's *how*. The rules, triggers and referential integrity code that we build into the database can be part of your strategy for addressing these needs.

In addition to the basic requirements with regard to data integrity, we can use rules and triggers in other ways to enhance our applications or make them easier to implement.

What are rules and triggers?

Visual FoxPro internally detects when changes are made to the data contained in tables in a database. The developer has the option of evaluating a logical expression in response to these changes. The expression can be any native Visual FoxPro function, like EMPTY() or ISNULL(), or a user-defined function (UDF). Usually any UDF called by a rule or trigger is stored in the .DBC as a stored procedure. Because the .DBC is automatically opened when one of its tables is opened, this ensures that the UDF will always be available when called by a rule or trigger. However, if it's necessary to share UDFs across multiple databases, you can store them in a shared procedure file or as independent .PRGs. Note that if you intend to access your Visual FoxPro database via ODBC that the ODBC driver has certain limitations on what can and cannot be accomplished in a stored procedure.

Any change to existing data will execute a field-rule code associated with the modified field, the table or row rule associated with the table, and the update trigger. When a record is deleted, the delete trigger is fired. If a new record is inserted into the table, or if a deleted record is recalled, the insert trigger fires.

Let's look at some of the basic behaviors of rules and triggers.

Rule behavior

There are two types of rules in Visual FoxPro, and they can be applied to either tables or views. There are field rules and table rules. The table rules are also commonly referred to as "row" rules. Both types of rules display certain behaviors:

- They fire when data is changed. They're optionally non-retroactive (can be added to an existing table without being applied to existing records).
- They prevent shifting focus to another row or field if the rule is violated; that is, if the rule evaluates to .F.
- They fire when a new row is inserted. A rule will fail if default field values violate the rules, preventing the insertion of a new record.
- They aren't affected by buffering; that is, they cannot be "turned off."
- They cannot move the record pointer *for the current table* during rule-code execution.
- Code executed by a rule or trigger allows use of the OLDVAL() and GET-FLDSTATE() functions but doesn't require buffering to be in effect.
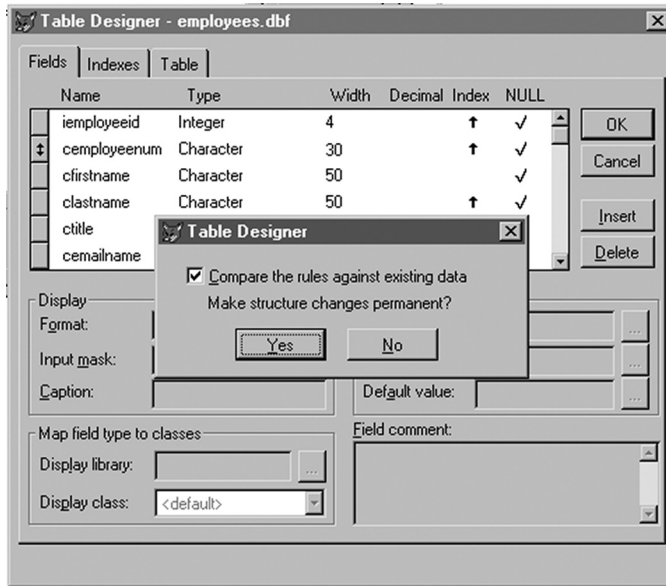
Figure 5.1. The Table Designer prompting to apply rules to existing records.

One limitation of some Visual FoxPro form controls is that there is no native technique for determining if a control's value has been changed. The InteractiveChange event fires for every keystroke, or every increment of a spinner, and as a result is often too "granular" in its response. As a result, many developers perform some evaluation in the LostFocus( ) method of the control to determine if the user has made any changes.

Such an evaluation is not needed when establishing rules. The rule fires only when the value of the field or fields to which the rule is bound changes. This is true of *any* change, including inserting a new row or appending a blank row. Because the failure of a rule (that is, the rule evaluates to .F.) prevents the user from shifting the focus to another field or moving the record pointer, this behavior suggests that it is very important to ensure that the default field values do not violate a rule. Such a situation would forever prevent insertion of a blank row.

When establishing a rule for an existing table, the rule is applied by default to all existing records. However, you have the option of adding a new rule, *without* applying the rule to the existing records. If modifying the table using the Data Designer, this can be accomplished by clearing the check box labeled "Compare the rules against existing data" in the Save dialog. If establishing the rule in code using the ALTER TABLE command, use the NOVALIDATE clause. Note that this prevents applying the rule to existing records *at the time the rule is established*, but if the user subsequently modifies an existing field or record, the rule will be applied.

When working directly against tables, there is no way of temporarily "turning off" rules during data entry. Buffering has no effect. A row or table-buffered table will not prevent a field-level rule from firing as soon as the field loses focus, and a table-buffered table will not prevent a row-level rule from firing when the user moves the record pointer.

If you execute a UDF from a rule, you cannot move the record pointer in the table executing the UDF as a rule, but you can change work areas and navigate through another table.

Here is one of the more interesting (and surprising) behaviors of code that is called from a rule or trigger: Two functions that normally require table buffering to be in effect when called, GETFLDSTATE() and OLDVAL(), work just fine within rule and trigger code. This implies that there is a level of internal buffering to which we don't normally have access, and which allows us to determine the state of the "dirty buffer" flags and the value of each field prior to modification in rule and trigger code, even if buffering isn't in effect.

Field and row rule behavior

In addition to all the behaviors mentioned above, field rules display the following properties:
- They fire as soon as the modified field loses focus.
- Failure (returning .F.) absolutely prevents moving off the field, or any control bound to the field.
- Field-rule code can modify fields on the current record, except the one firing the trigger.

By contrast, table or "row-level" rules include these properties:
- They fire only when the record pointer is moved.
- Row-rule code can modify any field in the modified record.

Data validation

Because rules can be used to validate user-entered data, we should examine data-validation techniques in general for a moment, and then assess rules as a tool to use for data validation in the larger context.

Data validation is usually of concern in the context of end-user data entry. This is where the developer loses some control over the application. As a result, the developer must program defensively, but in a user-friendly manner that facilitates data entry and protects the integrity of the data. In this context, data validation can be handled in one or a combination of several ways. Deciding how to handle data-entry data validation requires answering two questions: First, when is the user informed about unacceptable data? Second, how is the user prevented from putting the unacceptable data into the database? Unfortunately, there are no hard and fast rules that can be used to answer either question.

As a general rule of thumb, it's best to inform a user as soon as possible that some piece of data is unacceptable. On the other hand, you want users to be free to enter data in any order they find convenient, and not interrupt the flow of their work with message boxes and beeps. There is also the matter of performance and efficiency. If we call some kind of validation routine or method for every keystroke, this could have a noticeable impact on the application's performance. It is possible to eliminate the need for data validation in the first place by limiting the user's ability to enter data to only those values that are permissible, via some kind of picklist control, or by enabling or disabling, as appropriate, various options on a form.

A very basic validation technique is to use the Valid() event of a control, returning a value of 0 if the data entered is not acceptable. This prevents the user from shifting focus from the control until he enters an acceptable value. To many developers, this is one of the worst ways to enforce data validation. In the worst case, it forces the user to enter an acceptable value even if all he wants to do is click the Cancel button.

Another basic technique, but one which works at the extreme opposite end of the spectrum, is one in which the form's Save() method is executed conditionally, depending on the validity of all data entered. This confronts the user with an informational message after he completes data entry and asks to save all entries.

A middle-ground approach taken by some developers is that the validation is done periodically during data entry, possibly triggered by the LostFocus() events of the various controls, or via a Timer object, which enables or disables the Save button depending on the validity of the user's entries. This technique can be very effective, especially if there is some mechanism (via the status bar, for instance) to inform the user as to why the Save button is disabled.

One inherent weakness of performing data validation at the user-interface level of an application is that the rules are often hard-coded. This might be perfectly acceptable in some situations, and less so in others. Consider the rule, used as an example above that considers a patient's birth date to be invalid if it yields an age over 18. What if the pediatrician decides that she will no longer treat adolescents, and the age limit needs to be lowered to 12? Or perhaps the age limit varies by the patient's insurance plan? A data-driven rule would be preferable in this case. In fact, this particular rule is really a *business* rule, and some would argue that such a rule should *not* be enforced in the database itself.

Rules can play a role in all three approaches. The problem with allowing the user to actually violate a rule on a table interactively is that a FoxPro-generated dialog is presented to the user, displaying the message text that is stored in the RuleText property associated with the rule. While this behavior by itself does not present a problem, the ultimate effect on the user is the same as returning a value of 0 from a Valid() event; they are forced to satisfy the rule before they are allowed to shift focus to any other control, including the close box or "Exit" command button.

A Valid method can return a logical or numeric value. If a logical .F. is returned, the effect is to produce an audible "beep" and a wait window that says "Invalid input," and if the user then presses the <Enter> key, the control's previous value is restored. This behavior can be avoided if desired, giving the developer more control over the response to the invalid input by returning a value of 0. This simply prevents the control from losing focus; no automatic error indicator is triggered. A value other than 0 moves focus to a subsequent or prior control on the form, depending on the value returned. A 1 moves focus to the next control, while a –2 moves focus back two controls in the tab order.

Performing data validation by table and field rules represents a rather draconian extreme. However, when table or field rules are in place on a table, they aren't triggered by data entered into an updateable view, at least not until the data modifications are committed using TableUpdate(). If the TableUpdate() fails due to a rule violation, the value of the RuleText property gets stored in the array created with the AERROR() function, and can be used to present the user with an informative message box.

When using views, it is even possible to employ the database rules without ever triggering them, using them instead to enable or disable a Save button, and displaying useful information to the user. The following lines of code will retrieve the field-rule expression and the field-rule text from an open database:

```
DBGETPROP("<Table.FieldName>","Field","RuleExpression")
DBGETPROP("<Table.FieldName>","Field","RuleText")
```

Given that it's easy to determine the ControlSource property of any given control, the name of the underlying table's field can be determined by the following line of code:

```
DBGETPROP("<View.FieldName>","Field","UpdateName")
```

Another way of giving the user immediate feedback about a domain constraint violation, but absolutely preventing the violation in the database, involves using a single stored procedure for both views and tables, but responding differently depending on whether the rule is being called from the table or the view. The following stored procedure is used to validate the tDateWorked field of both the v_Time_Card_Hours view and the Time_Card_Hours table. The code in **Listing 5.1** gives the user an error message that doesn't interrupt the flow of data entry when using a view, but will not allow the invalid data to be inserted into the database when the TableUpdate() is called.

Listing 5.1. A dual-purpose, field-validation stored procedure.

```
FUNCTION ValidateDateWorked()
   LOCAL llValid
   #DEFINE LOCALVIEW 1
```

```
   IF EMPTY(tDateWorked) OR tDateWorked > DATE()
      llValid = .F.
   ELSE
      llValid = .T.
   ENDIF
   IF CURSORGETPROP("SourceType") = LOCALVIEW ;
         AND ! llValid
      ?? CHR(7)
      SET MESSAGE TO "Invalid Date"
      WAIT "Invalid date" WINDOW NOWAIT
      llValid = .T.
   ENDIF
   RETURN llValid
ENDFUNC
```

This function uses the CURSORGETPROP() function to determine whether the field is being changed in a view or in the table directly. If it's in a view, it simply beeps, displays a WAIT WINDOW NOWAIT and puts a message on the status bar to alert the user that something needs to be corrected before she can commit her changes. If the user is updating the table directly (or initiates a Save() despite the error), the function returns a value of .F. and the update is rejected.

Note that this function is triggered (as are all rules) when a new record is appended, and the rule is triggered subsequently only when the user makes a change to the tDateWorked field. Establishing an appropriate default value could address this issue.

Data modification using rules

If you're familiar with the various normalization rules for database design, you're probably familiar with the most common rules corresponding to what is known as first, second, and third normal forms. Part of the third normal form specifies that there shouldn't be any redundant calculated data in a record; that is, you shouldn't include a column that contains values that can be calculated from the values contained in two or more other fields. For instance, third normal form argues against having an "extension" column when you have quantity and price fields; the extension can be determined from these two fields and doesn't need to be stored in the table.

However, in even the most carefully designed and rigorously implemented database, this is the rule that is most commonly, deliberately violated. Sometimes it allows certain application features to be implemented more easily, or perhaps the clients or users prefer that such calculated fields be included. Once this decision is made, it then becomes necessary to accurately maintain the values stored in these fields. If the calculated fields are not displayed during data entry, they can be "batch" updated prior to committing the user's modifications. However, if the calculated fields need to be displayed and updated in "real time" during data entry, rules can come to the rescue here, simply replacing the calculated value based on the contents of other fields.

You can see an example of this application in the Time_Card_Hours_Rule() that is fired in response to a change to either the tStart or tEnd fields of the v_Time_Card_ Hours view. Changing either value results in a recalculation of the value stored in the bBillableHours field. Note that because it was preferred that this synchronization be "real-time" and visible to the user, it was implemented in the view. It could also be implemented in the table itself. This would guarantee that no further code would be required in the application to ensure that bBillableHours was always consistent with the start and end times entered.

Another self-modification rule is evident in the v_Time_Card_Hours view. Note that there are *two* functions called in the rule for the tDateWorked field. One is the one discussed earlier, validating this field, the other is to establish default values for the tStart and tEnd fields, making the data-entry process a little more efficient.

Trigger behavior

Triggers display somewhat different behaviors than do rules, owing to their different function and usage within an application. As with rules, triggers fire in response to changes made to the tables in our database. We can then decide to evaluate one or more expressions in response to these events. The following points summarize trigger behaviors:

- Triggers fire on TABLEUPDATE(), or on modification if the table is not buffered. Thus, you can delay evaluation of any trigger expression or execution of any trigger code by using buffering.
- UPDATE occurs when any field is changed.
- INSERT occurs when a new record is added, *or* when a deleted record is recalled.
- DELETE occurs when the delete flag is set.
- Delete code can change work areas and modify other tables.
- Code executed in response to a trigger cannot modify the record that is firing the trigger.
- OLDVAL() and GETFLDSTATE() can be used even if the table is not buffered.

As with table and view rules, we use properties for each table in the database to specify what action is to be performed when any of these three triggers are fired. Note that, unlike rules, which can work on fields or tables, there exist *only* table triggers. Modifying a view alone will not fire a trigger; a trigger will be fired only when the changes in a view are used to modify the underlying table by calling the TableUpdate() function.

While table and field rules often can be a single expression (like NOT EMPTY(cCase_ No)), the expressions evaluated in response to a trigger are usually user-defined functions. Such functions are usually maintained in the database's stored procedures.

There is no requirement that trigger code be kept in the database's stored procedures. As with any other UDF, as long as Visual FoxPro can find the code, it'll be executed. However, unlike a key-value-generating function, trigger functions are most often very database-specific, and therefore less likely to be shared between databases. Thus, there usually isn't much to be gained by storing trigger functions in a separate procedure file.

When putting together a comprehensive system that employs rules and triggers, it's important to understand the normal sequence of events between the user changing a field value in a form and the change being recorded on disk:

1.   Data is modified in a view or buffered table.
2.   We attempt to commit the modifications by issuing a TableUpdate().
3.   Rules (if any) are fired.
4.   If a rule does not fail, any applicable triggers are fired.
5.   Modifications are accepted/rejected (equivalent to TableRevert() at the table level).

So what can you do with triggers? The most common use for triggers, in part because Microsoft supplies a spiffy tool that supports it, is building referential integrity rules into our databases. Later we'll discuss other interesting things that can be done with triggers, but we'll start with a discussion of referential integrity.

Referential integrity

Referential integrity refers to protecting the references between the tables based on primary keys and foreign keys in order to avoid child records with no corresponding parent records (orphaned records). Orphaned records are created by:

*   Inserting a child record when no corresponding parent record exists
*   Deleting a parent record, leaving the corresponding child records intact
*   Changing a parent record's primary key value, so that the child record's foreign key value no longer points to a valid parent record

As you can see, each of these three actions corresponds to one of the three available triggers. In the case of an INSERT, we can respond in one of two ways: we can either ignore the insertion of an orphan record, or we can *restrict* (prohibit) the insertion if no parent record exists. In the case of a DELETE, we can ignore the deletion, allowing the orphan records; we can *cascade* the deletion to any child records, deleting them along with their parent; or we can restrict the deletion, prohibiting it if any child records exist. Similarly, we have the same options when the primary key value of a parent record is changed; we can ignore the change, cascade the change to the child records, or restrict the change if child records exist.

Referential integrity rules apply to *relations*, not to tables. As a result, a delete or update rule applies to the deletion or modification of the *parent* table, and an insertion to the *child* table. The role of a particular table changes depending on which relationship we're referring to. Thus, while we assign code to be executed in response to a trigger for a particular table, the action of that code will vary depending on the nature of the relationship. We may require deletion of a record in table A to cascade to table B, but be restricted if there are child records in table C. If you're unfamiliar with referential integrity rules and this seems a bit unclear, it should click into place as this chapter progresses. See **Table 5.1**.

Table 5.1. Available referential integrity options.

	Cascade	Restrict	Ignore
Delete	●	●	●
Update	●	●	●
Insert		●	●

Keep in mind that these options are not exhaustive. They are simply the most common way of enforcing referential integrity, and they are the options provided by the Visual FoxPro Referential Integrity builder. You could respond to a deletion, for instance, by changing the foreign key value of the child records to a default value; preserving their contents, and changing their reference to another "utility" parent record that allows them to be accessed, but in a different context. For example, imagine that a salesperson is leaving a company, and the sales manager wants to remove his record from the salesperson table but reassign his accounts to another salesperson (or perhaps, by default, to the sales manager). Another alternative is to change the foreign key to NULL or some other value that indicates that the parent records have been removed. Finally, there is one issue that referential integrity and orphan records fail to address: the case in which a parent without children is meaningless—a purchase order or invoice without line items is a good example. All of these issues can be addressed through properly crafted code that is executed in response to a trigger.

Remember that no rule says that there is anything inherently wrong with orphan records. Whether orphan records are acceptable or not can be decided only within the context of a particular database design. For instance, Microsoft SQL Server has a feature called "Declarative Referential Integrity," in which each declared relation cascades deletions and updates, and restricts insertions. This is a shotgun approach, and *does* imply that orphan records are to be avoided. This is not necessarily the case. The best example of the permissibility of orphan records is a codes table. A codes table is a parent table, and the tables that use those codes are the child tables. If a code field is a required field in the child table, then putting a restriction on the insertion of the child records will enforce this. However, if the code field is not required, then there should be no restriction on the insertion; orphan child records are permitted.

Now that we have some idea of what referential integrity is all about, let's take a look at the sample database in the \SAS\LocalData\ folder of the download files. **Figure 5.2** shows all of the table relations along with their associated referential integrity rules.
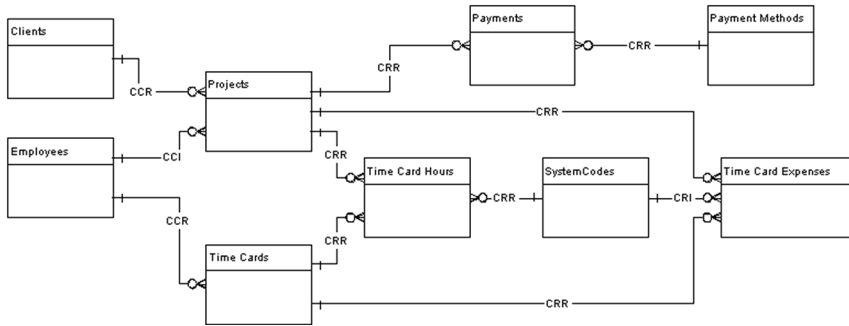

Figure 5.2. Relations and referential integrity rules from \SAS\LocalData\Time and Billing. DBC. Rules are stated in Update/Delete/Insert order and indicate Cascade, Restrict or Ignore.

The referential integrity rules are shown in Figure 5.2 for each relation. The first letter indicates the rule for updates, the second for deletions, and the third for insertions. For example, the rules for the relation between employees and time_cards is "CCR", indicating that changes (updates) to the primary key of the employee record are cascaded to the time_cards table, preserving the link. Deletion of an employee record is cascaded to related records in the time_cards table, and inserts to the time_cards table are restricted to only records that have a corresponding employee record.

Contrast the deletion rule between the employees table and the time_cards table, with that between the time_cards table and the time_card_hours table. Deletions in employees are cascaded to time_cards, but deletions in time_cards are *restricted* if corresponding records exist in time_card_hours. Note that insertions to the time_card_hours table are restricted and prohibited unless a parent record exists in the projects table. Thus, the existence of a time_card_hours record implies that some work has been done on a project, and deletion of a record in the time_card_hours table could result in the loss of a record of billable hours on a current project. Thus, the rule permits removal of an employee, and will automatically delete the employee's time_card records. However, time_card records cannot be deleted if billable hours are associated with the time_card. The end result of this set of rules is that the delete trigger on the employee table should succeed as long as there are no billable hours on one of the employee's time cards, but fail if there are billable hours associated with one of his time cards.

Consider one more example:

The rule on the relation between systemCodes and time_card_expenses prohibits deletion of a systemCode record if its code is in use in the time_card_expenses table.

However, an expense may be incurred for which there exists (as yet) no expense code, so insertion of a time_card_expenses record is permitted, even though there may be no corresponding record in the systemCodes table. Thus, the user can enter an expense record without having to provide an expense code. The presence of a memo field would allow the user to log an unusual or one-time expense in a timely fashion without having to create a new expense code.

Implementing referential integrity rules

In the Visual FoxPro Database Container (the .DBC file), the referential integrity rules are stored in a field called "RIINFO", using the same three-character code used in Figure 5.2, which indicates the rules to be enforced on updates, deletions and insertions. If you open the sample \SAS\LocalData\Time and Billing.DBC as a table and browse it, you will see that all of the RIINFO fields are empty except for those records whose object type is "Relation." It is possible to manipulate this field directly, but it isn't convenient. Note that the name of the child table in the relation is found by referring to the record with the objectid indicated by the parentid field of the relation record. Even worse, the parent table's name is buried in the binary data in the Property field of the .DBC! Not really optimal for setting these values.

Note, too, that setting these values only indicates the rules that you would *like* to have enforced, but does nothing (by themselves) to *enforce* these rules. The RIINFO field in the .DBC is simply a convenient repository for the rules that we want to enforce.

Because this sample database uses surrogate primary and foreign keys, we could easily indicate "Ignore" for all updates. This is because the user never sees the primary key field in the application, and therefore has no opportunity to change it; nor is there any reason to want to change any of the primary keys. They have no meaning in and of themselves, which is why they're called "surrogate." The user can change the employee number (which is not a primary key) without having to worry about changing its value in any child tables, because the employee number (not the employee ID) doesn't appear in any child tables. However, I've included both cascade and restrict rules on updates for demonstration purposes. It's kind of neat to browse the employee and time_cards tables at the same time, and watch the employee ID change in the time_cards table in response to changing it in the employee table!

The VFP Referential Integrity Builder

Microsoft includes, as part of the Visual FoxPro package, a Referential Integrity Builder that both allows you to manipulate the values in the RIINFO field, and from these values, generate referential integrity code that is stored in the stored procedures of the database. The good news: the RI code generated by this utility works pretty well. The bad news is that there's no way to set the RIINFO field values *without* generating the RI code, and the code that the RI builder generates is so voluminous and verbose that it's very difficult to trace or understand. This code can (in fairly com-

plex systems) generate an object file that exceeds the Visual FoxPro limit of 64K on compiled program modules. A project I'm currently working on has more than 270 relations; as a result of this and the fact that RI rules were established on deletions and insertions of all tables, the builder-generated RI code did indeed hit this limit.

The only remaining problem with the RI builder-generated code, as far as I'm concerned, is that many developers are forced to accept its efficacy on faith, or if they have the patience, they must test the code repeatedly, and verify empirically that it indeed works as it's supposed to. Given the complexity that can exist in even a fairly simple relational database model, it isn't unusual to be confronted unexpectedly with a failed trigger. Until you become comfortable with your own RI rules and the code that implements them, you'll find yourself tracing the RI code to find out what trigger is failing and why. If you do this repeatedly (as I have) with the RI builder-generated code, you'll eventually come to have a high degree of confidence in its operation.

However, many developers will take one look at the RI builder-generated code and say (with a great deal of justification), "I don't have time to figure out how this works, and determine if it works, and I'm not going to make my users/clients test this stuff!"

Sadly, referential integrity code does *not* have to be this ugly.

Referential integrity logic

If we are to consider only the five different situations that basic RI logic needs to handle, the logic is extremely simple, as shown in **Table 5.2.**

Table 5.2. Basic referential integrity logic.

Case	Action
Cascaded Update	Change the foreign key value in the child table from its present value to the new value just established for the parent table's primary key
Restricted Update	Check to see if there are any child records, and if so, prohibit the update
Cascaded Delete	Delete all child records with a foreign key value matching the primary key value of the parent record
Restricted Delete	Check to see if there are any child records, and if so, prohibit the deletion
Restricted Insert	Check to see if there is a record in the parent table whose primary key value is the same as the foreign key value for the record being inserted. If there is, allow the insert, if not, prohibit the insert

Using a product like Visual FoxPro, we have two basic choices on how to implement this logic.

We can (as the RI builder-generated code does) rely on Xbase commands and procedural coding to set the appropriate indexes, establish the key values of the tables being modified, SEEK the values of interest in the related tables, and take the appropriate actions. As an alternative, we can take advantage of the more powerful and concise SQL techniques that we have at our disposal.

While Microsoft SQL Server, as mentioned previously, provides a facility for "Declarative" referential integrity, if you need more control over how RI rules are applied, you must write trigger code that looks suspiciously like what you'll see in this chapter. Because the only way to modify the data in a SQL database is via SQL commands, there is no option to use Xbase syntax. However, as you can see from the code shown in this chapter, SQL makes this type of code much simpler, easier to follow and debug, and requires a lot less typing! I've often wondered why the RI builder uses procedural Xbase code to do the job.

Before looking at how we can accomplish this, let's examine a couple of issues that the RI builder-generated code deals with, and see if we can't simplify this part of our implementation.

Of the five different types of procedures that we'll need to perform out of the five different cases listed above, only two of them involve modifying data; the operations that implement cascading rules. Operations that implement restriction rules do not modify any data outside the table that is currently firing a trigger.

When considering the cascading operations, there are two important things that need to be accomplished. We need to be able to "undo" anything that is done. A cascaded change implies the possibility of a change in table A cascading to table B. The change in B cascades to table C. However, there is no guarantee that *every* relation in such a chain has a cascading rule. If table C has a child table D, and the relation between tables C and D carries a restrictive rule, then updating or deleting records in table C would fail (it has child records in D and therefore prohibits the change). At this point, related records in table B have already been changed to reflect the change made to table A. What if the cascaded change between B and C fails because of the restrictive rule between C and D? Another possible scenario is if table A is related to two child tables, B1 and B2. There is a cascading rule between A and B1, but a restrictive rule between A and B2. It can happen that the cascade between A and B1 occurs first, but table B2 has child records that cause the trigger to fail.

In either of these scenarios, do we go back and "undo" the change made to the tables already modified as a result of a cascading rule? We sure do. In fact, Visual FoxPro has this neat thing called a TRANSACTION that allows us to do just that very easily.

A table is often a child of some tables and a parent to others. Because of this, we can never be certain whether the trigger is firing because the user is making a change to a table, or whether a table is being modified in response to a trigger firing on a parent

table. Because it's important for all cascading operations to be wrapped in a single transaction, only the code fired by the first trigger in a chain reaction of triggers needs to BEGIN TRANSACTION, END TRANSACTION if the cascades are successful, and ROLLBACK if any of the triggers fail. Fortunately, someone at Microsoft anticipated this need and provided the _TRIGGERLEVEL system memory variable that can be used to determine when we are in a "top-level" trigger, and not some place further down the chain of triggers. _TRIGGERLEVEL is 0 when trigger code is not executing, 1 when the first level trigger is executing, and increments of 1 for each subsequent trigger that is fired.

The other, related issue that we need to address with regard to cascading RI functions is that our top-level function needs to determine whether some other RI function caused a trigger to fail, so it knows whether to END TRANSACTION or ROLLBACK. The way in which a UDF called by a trigger "fails" is to return a value of .F. However, the RI code we write doesn't call a UDF (which would allow it to check the value returned by the UDF); it simply performs the appropriate modifications to a child table. As you might expect, the failure of a trigger puts Visual FoxPro into an error condition. Thus, by simply executing a command that will cause a private memory variable to be set in the event of an error, we can detect a failure of another trigger, or indeed, any other type of error that occurs as a result of the execution of our trigger code.

This issue of a failed trigger triggering ON ERROR brings up a very interesting point. As soon as any trigger-initiated (or rule-initiated) function returns a value of .F., Visual FoxPro is in an error condition. This means that if you have code in the Error method of a form or some other control that is issuing a TableUpdate() command, the error condition will immediately trigger the execution of the object error-method code. Thus, if you have carefully written code that detects a failed TableUpdate(), and checks for the reason for the failure and presents some user-friendly dialogs to explain the problem to the user, this entire process will be short-circuited by the form or object Error method.

If you examine the RI builder-generated code, you'll see that this business of setting the ON ERROR and beginning a transaction is performed for all RI procedures, not just those involving cascading changes. This isn't necessary for restricted changes. A change is either restricted at the top-level trigger code, in which case there is no need to detect an error at a subsequent trigger level; or the current trigger was initiated by a cascading change at a higher trigger level. Therefore, only cascading code needs to take responsibility for wrapping things in a transaction, and checking for errors further up the line.

Algorithms for cascading and restricting changes

Let's put together all of the ideas we've discussed so far, using some pseudo-code to get a feel for how we will implement our referential integrity rules. **Listing 5.2** shows

the pseudo-code for a cascading change, and illustrates how the top-level call wraps subsequent triggers in a transaction and can detect if some other trigger fails. **Listing 5.3** shows the steps in restricting a change, depending on the presence or absence of related records.

Listing 5.2. Pseudo-code for a cascading RI function.

```
Check if _TRIGGERLEVEL is 1
   Save old ON ERROR setting
   Tell ON ERROR to set a private memvar .T.
   BEGIN TRANSACTION
Determine primary key value for record being changed
Perform necessary action on child table
Check again if _TRIGGERLEVEL is 1
   Check to see if our private error memvar is .T.
      ROLLBACK
   Else
      END TRANSACTION
   Restore old ON ERROR
RETURN .T. as long as no error occurred
```

Listing 5.3. Pseudo-code for a restricting RI function.

```
Determine value of key field for modified record
Check for matching records in the related table
Set the return value depending on the presence or absence of
related records
RETURN the return value
```

The "perform necessary action" referred to in Listing 5.2 is either a DELETE-SQL command or an UPDATE-SQL command, depending on whether we are cascading a delete or an update.

In Listing 5.3 the "matching records" are either child records in the case of a restricted DELETE or UPDATE, or parent records in the case of a restricted INSERT. The presence or absence of matching records is accomplished by issuing a SELECT-SQL command that counts the related records. The return value is .T. if no related records are found on a restricted DELETE or UPDATE, but .F. if no related records are found on a restricted insert.

Before proceeding with closer examination of the actual code, let's look at the information that the RI functions need to perform. In the RI builder-generated code, this information is hard-coded, which requires each RI rule to be implemented as a separate code block. However, you'll note that the RI builder code repeats the same pattern of commands over and over, and that the information needed to create each code block is:

- The name of the parent table
- The name of the child table
- The name of the parent's primary key field
- The name of the child's foreign key field

Could we write a single reusable block of code for each of the five RI situations, and simply pass these four pieces of information as arguments? Indeed, we could. Imagine for a moment that we have an array that contains not only the four pieces of information listed above for each relation in the database, but also the type of rules to apply to each of the three different types of triggers. If we know what table is being modified, and how it's being modified, we could use this array as a lookup to determine which function to call and what arguments to pass.

The whole enchilada—a universal referential integrity function

As mentioned in the previous section, we need several pieces of information before we can hope to act appropriately in response to a trigger.

It's easy to determine the table being modified, because we know that a trigger is being executed, so the table being modified is open in the currently selected work area. We can determine the table name by using CURSORGETPROP("SourceName") so we don't need to be confused by aliases. However, it's trickier to determine what kind of modification is being made.

I am grateful to Jim Duffy of TakeNote Computer Consulting for asking a very interesting question a while back on CompuServe's VFOX forum. Jim asked if there was any way to determine, within a piece of trigger code, what trigger was being fired. After a bunch of us dummies replied to Jim with, "Duh, I don't think you can do that, Jim," Michael Colbert of Intelligent Computer Solutions (to whom I'm *extremely* grateful), came back with a supremely elegant solution to the problem. Michael figured out that because GETFLDSTATE() works within trigger code even if the table isn't buffered, you can use this to figure out what trigger is being fired. If the record is DELETED() and GETFLDSTATE() indicates that the deletion flag has changed, a delete trigger is firing. If GETFLDSTATE() indicates that the deletion flag has changed, but the record is *not* DELETED(), then an insert trigger is firing. If there are one or more 2s in the string returned by GETFLDSTATE(-1), then the record is being modified and an update trigger is firing. If there are 3s and/or 4s in the string returned by GETFLDSTATE(-1), then we're dealing with a new record, and an insert trigger is being fired. What could be simpler?

So, one last piece of pseudo-code and we'll take a look at the real thing. **Listing 5.4** represents the function (yes, a single function) that is specified for every update, delete, and insert trigger for every table in the system.

Listing 5.4. Pseudo-code for a universal RI function.

```
Determine whose trigger code is being fired
Determine what kind of trigger is being fired
Establish a lookup array with referential integrity specifications
Based on the table, and the type of trigger, search the lookup
array
```

```
Determine from the lookup array if an RI rule is to be enforced
Call the appropriate RI function, passing the necessary 4 values
from the lookup array
Return the value returned by the RI function
```

Examination of this pseudo-code should make it obvious that the only thing that's going to change in this function is the array referred to in the third line. If we can establish an easy way to set up this array, we'll be much closer to having a reliable, maintainable, and most of all, *understandable* and *verifiable* referential integrity system. There are many times when a trigger will fail unexpectedly, and you'll find yourself tracing this code to make sure it's working right. There are so few lines involved that doing so won't be much of a chore. The RI builder-generated code is such a convoluted mess that you need the patience of Job to trace it through much more than a single level of triggers.

Listing 5.5 shows the NewRI() function as implemented in the sample database for this chapter, with an abbreviated version of the lookup array.

Listing 5.5. The NewRI() universal referential integrity function.

```
FUNCTION NewRI()
  LOCAL lcRecordState, ;
    lcTriggerType, ;
    lcTable, ;
    llRetVal, ;
    lcParentKey, ;
    lcChildKey, ;
    lnRelations, ;
    i
  LOCAL ARRAY laRelations[1]

  #DEFINE CHILDCOL 1
  #DEFINE PARENTCOL 2
  #DEFINE CHILDKEYCOL 3
  #DEFINE PARENTKEYCOL 4
  #DEFINE UPDATECOL 5
  #DEFINE DELETECOL 6
  #DEFINE INSERTCOL 7

  lcTable = CURSORGETPROP("SourceName")
  * Determine what type of trigger is firing
  lcRecordState = GETFLDSTATE(-1)
  DO CASE
    CASE LEFT(lcRecordState,1) = "2" AND DELETED()
      lcTriggerType = "DELETE"
    CASE LEFT(lcRecordState,1) = "2" AND ! DELETED()
      lcTriggerType = "INSERT"
    CASE "3" $ lcRecordState OR "4" $ lcRecordState
      lcTriggerType = "INSERT"
    CASE "2" $ lcRecordState
      lcTriggerType = "UPDATE"
  ENDCASE
```

```
*** Lookup Array - RI Specifications *******************************
  lnRelations = 1
  DIMENSION laRelations[1,7]
  laRelations[1, CHILDCOL]     = ‹TIME_CARDS›
  laRelations[1, PARENTCOL]    = ‹EMPLOYEES›
  laRelations[1, CHILDKEYCOL]  = ‹IEMPLOYEEID›
  laRelations[1, PARENTKEYCOL] = ‹IEMPLOYEEID›
  laRelations[1, UPDATECOL]    = ‹C›
  laRelations[1, DELETECOL]    = ‹C›
  laRelations[1, INSERTCOL]    = ‹R›
*** Lookup Array - RI Specifications *******************************

*!*        Find the table whose trigger is firing in the
*!*        lookup array, and if there is a rule associated
*!*        with this trigger for this table, call the
*!*        appropriate RI function
*!*        llRetVal = .T.
  DO CASE
    CASE lcTriggerType = "INSERT"
      FOR i = 1 TO lnRelations
        IF laRelations[i,CHILDCOL] = lcTable ;
           AND laRelations[i,INSERTCOL] = "R"
          lcParentKey = laRelations[i,PARENTCOL] + "." + ;
            laRelations[i,PARENTKEYCOL]
          lcChildKey = laRelations[i,CHILDCOL] + "." + ;
            laRelations[i,CHILDKEYCOL]
          llRetVal = ;
            Restrict_Insert(laRelations[i,PARENTCOL], ;
              laRelations[i,CHILDCOL],lcParentKey,lcChildKey)
        ENDIF
        IF ! llRetVal
          EXIT
        ENDIF
      ENDFOR
    CASE lcTriggerType = "DELETE"
      FOR i = 1 TO lnRelations
        DO CASE
          CASE laRelations[i,PARENTCOL] = lcTable ;
             AND laRelations[i,DELETECOL] = "C"
            lcParentKey = laRelations[i,PARENTCOL] + "." + ;
              laRelations[i,PARENTKEYCOL]
            lcChildKey = laRelations[i,CHILDCOL] + "." + ;
              laRelations[i,CHILDKEYCOL]
            llRetVal = ;
              Cascade_Delete(laRelations[i,PARENTCOL], ;
                laRelations[i,CHILDCOL],lcParentKey,lcChildKey)
          CASE laRelations[i,PARENTCOL] = lcTable ;
             AND laRelations[i,DELETECOL] = "R"
            lcParentKey = laRelations[i,PARENTCOL] + "." + ;
              laRelations[i,PARENTKEYCOL]
            lcChildKey = laRelations[i,CHILDCOL] + "." + ;
              laRelations[i,CHILDKEYCOL]
```

```
            llRetVal = ;
              Restrict_Delete(laRelations[i,PARENTCOL], ;
                laRelations[i,CHILDCOL],lcParentKey,lcChildKey)
          ENDCASE
          IF ! llRetVal
            EXIT
          ENDIF
        ENDFOR
      CASE lcTriggerType = "UPDATE"
        FOR i = 1 TO lnRelations
          DO CASE
            CASE laRelations[i,PARENTCOL] = lcTable ;
                AND laRelations[i,UPDATECOL] = "C"
              lcParentKey = laRelations[i,PARENTCOL] + "." + ;
                laRelations[i,PARENTKEYCOL]
              lcChildKey = laRelations[i,CHILDCOL] + "." + ;
                laRelations[i,CHILDKEYCOL]
              llRetVal = ;
                Cascade_Update(laRelations[i,PARENTCOL], ;
                  laRelations[i,CHILDCOL],lcParentKey,lcChildKey)
            CASE laRelations[i,PARENTCOL] = lcTable ;
                AND laRelations[i,UPDATECOL] = "R"
              lcParentKey = laRelations[i,PARENTCOL] + "." + ;
                laRelations[i,PARENTKEYCOL]
              lcChildKey = laRelations[i,CHILDCOL] + "." + ;
                laRelations[i,CHILDKEYCOL]
              llRetVal = ;
                Restrict_Update(laRelations[i,PARENTCOL], ;
                  laRelations[i,CHILDCOL],lcParentKey,lcChildKey)
          ENDCASE
          IF ! llRetVal
            EXIT
          ENDIF
        ENDFOR
    ENDCASE
    RETURN llRetVal
ENDFUNC

* Cascading Update Function
FUNCTION Cascade_Update( ;
      tcParentTable,tcChildTable,tcParentKey,tcChildKey)
  LOCAL llRetVal, ;
    lcOldError, ;
    luKey, ;
    luNewKey
  IF _TRIGGERLEVEL = 1
    RELEASE plError
    PUBLIC plError
    lcOldError = ON("ERROR")
    ON ERROR plError = .T.
    BEGIN TRANSACTION
  ENDIF
```

```
  luKey = OLDVAL(tcParentKey)
  luNewKey = EVALUATE(tcParentKey)
  UPDATE (tcChildTable) SET &tcChildKey = luNewKey ;
       WHERE &tcChildKey = luKey
  IF _TRIGGERLEVEL = 1
    IF plError
      ROLLBACK
    ELSE
      END TRANSACTION
    ENDIF
    ON ERROR &lcOldError
  ENDIF
  llRetVal = ! plError
  RETURN llRetVal
ENDFUNC

* Cascading Delete Function
FUNCTION Cascade_Delete( ;
       tcParentTable,tcChildTable,tcParentKey,tcChildKey)
  LOCAL llRetVal, ;
    lcOldError, ;
    luKey
  IF _TRIGGERLEVEL = 1
    RELEASE plError
    PUBLIC plError
    lcOldError = ON("ERROR")
    ON ERROR plError = .T.
    BEGIN TRANSACTION
  ENDIF
  luKey = EVALUATE(tcParentKey)
  DELETE FROM (tcChildTable) WHERE &tcChildKey = luKey
  IF _TRIGGERLEVEL = 1
    IF plError
      ROLLBACK
    ELSE
      END TRANSACTION
    ENDIF
    ON ERROR &lcOldError
  ENDIF
  llRetVal = ! plError
  RETURN llRetVal
ENDFUNC

* Restricting Delete Function
FUNCTION Restrict_Delete( ;
       tcParentTable,tcChildTable,tcParentKey,tcChildKey)
  LOCAL llRetVal, ;
    luKey
  LOCAL ARRAY laCount[1]
  luKey = EVALUATE(tcParentKey)
  SELECT COUNT(*) ;
    FROM (tcChildTable) ;
    WHERE &tcChildKey == luKey ;
```

```
      AND ! DELETED(tcChildTable) ;
    INTO ARRAY laCount
  IF laCount > 0
    llRetVal = .F.
  ELSE
    llRetVal = .T.
  ENDIF
  RETURN llRetVal
ENDFUNC

* Restricting Update Function
FUNCTION Restrict_Update( ;
        tcParentTable,tcChildTable,tcParentKey,tcChildKey)
  LOCAL llRetVal, ;
    luKey
  LOCAL ARRAY laCount[1]
  luKey = OLDVAL(tcParentKey)
  SELECT COUNT(*) ;
    FROM (tcChildTable) ;
    WHERE &tcChildKey == luKey ;
      AND ! DELETED(tcChildTable) ;
    INTO ARRAY laCount
  IF laCount > 0
    llRetVal = .F.
  ELSE
    llRetVal = .T.
  ENDIF
  RETURN llRetVal
ENDFUNC

* Restricting Insert Function
FUNCTION Restrict_Insert( ;
        tcParentTable,tcChildTable,tcParentKey,tcChildKey)
  LOCAL llRetVal, ;
    luKey
  LOCAL ARRAY laCount[1]
  luKey = EVALUATE(tcChildKey)
  SELECT COUNT(*) ;
    FROM (tcParentTable) ;
    WHERE &tcParentKey == luKey ;
      AND ! DELETED(tcChildTable) ;
    INTO ARRAY laCount
  IF laCount = 0
    llRetVal = .F.
  ELSE
    llRetVal = .T.
  ENDIF
  RETURN llRetVal
ENDFUNC
```

I think that you'll find most of the foregoing code straightforward and self-documenting. I'd like to call your attention to one thing, however: the use of the OLDVAL()

function in the Cascade_Update() function. This allows the field's previous value to be used in the UPDATE…FOR command to find the child records that still have the *old* foreign key value, and change them to the new one.

In case you're counting, the above listing accomplishes its task in 219 lines of code, including comments and white space. There is only one relation in the preceding listing for the specifications array, but each additional relation adds only seven lines of code (see the stored procedures for \SAS\LocalData\Time and Billing.DBC). Thus, for the 11 relations in this system, this code would require 289 lines of code. Compare this to the 2,425 lines of code that the RI builder creates to do the same job!

I don't mean to criticize the RI builder. It's very easy to use; the code it produces does the job and executes very quickly. However, if we can make this feature a little more transparent, then we're more confident of being able to understand how it works, and modify or maintain it if and when we need to. To be fair, the code created by the RI builder does something that the NewRI() function does not: it meticulously closes all tables opened in the process of execution. However, considering how many developers follow the practice of opening *all* tables at application startup, and given the reduced need to "clean up" provided by private data sessions, I felt that I could get away with this slight sloppiness.

If you're wondering whether there's an easy way to maintain the RI rules in the DBC without using the RI builder, and if there's an easy way to establish the laRelations[] array used in the NewRI() function, you'll find a form called EditRI.SCX located in \SAS\Tools\ of the download files. This form allows you to select a database, examine and modify the RI rules, and save those rules back to the database. If, before you close the form, you select the check box that reads "Rules to clipboard on close," you'll discover that the "DIMENSION laRelations" command, the "nRelations=" line and the entire block of code that assigns all of the RI rules to the laRelations lookup array has been copied to the clipboard, ready for pasting into the NewRI() function.

Functions that are still lacking from this tool (as of this writing) include the ability to stuff all RI code into the stored procedures, and to set the triggers for all tables to call the NewRI() function. These two tasks must still be performed manually.

All emptors be caveat!!

First, because this code was designed to be used on a database that uses non-compound integer surrogate keys, it won't work on a database that uses compound primary and foreign keys; nor will it work on a database that uses key expressions on anything other than the field name. Therefore, if you have a primary or foreign key based on an expression that uses a function like UPPER(), VAL(), PADR(), and so forth, this RI code won't work. To understand why, consider just the following line of code:

```
UPDATE (tcChildTable) SET &tcChildKey = luNewKey ;
       WHERE &tcChildKey = luKey
```

Note that if the child table's foreign key expression is cOrder_ID + cPart_No, the SET clause will become SET cOrder_ID + cPart_No = luNewKey, which clearly will trigger an error.

Second, please be aware that this code has been developed for a production application that (as of this writing) is in the middle of development, and has been subjected (at best) to only alpha testing. Use at your own risk. No warranties express or implied. No bailment created. Your mileage may vary. Void where prohibited. Don't take with other medications without consulting your doctor or pharmacist.

Other stuff to do with triggers

As with rules, which are primarily intended to enforce domain constraints, your own needs and creativity will determine how many other things you can do with triggers.

One of the neatest things to do with a trigger, other than enforcing RI rules, is to create an audit trail. While enforcing a referential integrity rule in response to an update is intended to deal with changes to the primary key field's value, keep in mind that *any* change to a record causes the update trigger to fire.

Also, no rule says you can't execute more than one function in response to a trigger. You could add a trigger to a table like this:

```
CREATE TRIGGER ON time_cards FOR UPDATE AS NewRI() ;
       AND Audit_Time_Cards()
CREATE TRIGGER ON time_cards FOR INSERT AS NewRI() ;
       AND Audit_Time_Cards()
```

The AS clause specifies a logical expression, and therefore can include a collection of user-defined functions joined by AND and OR operators.

Assuming that the audit_time_cards table has an additional primary key field, the Audit_Time_Cards() function could look something like this:

```
SCATTER MEMVAR MEMO
m.iTCAudit_ID = NewID("AUDIT_TIME_CARDS")
INSERT INTO audit_time_cards FROM MEMVAR
```

You could have timestamp and user ID fields in the time_cards table. If these fields are automatically maintained (perhaps using a row-level rule?), then the audit_time_cards table will contain a complete record of every change that has ever been made to the time_cards table, showing the date and time the change was made, and who made the change! You could get a little fancier, saving instead the name of any fields that were changed, together with their old values and current values (remember OLDVAL() always works in trigger code!). Entire articles and chapters of books have been written about how to implement audit trails. With intelligent use of the tools

now available to us, we can implement a very sophisticated audit trail with just a few lines of code that works every time, no matter how many forms we add to the system, and no matter how many new developers get involved with the project. No one has to remember to implement this function; it happens automatically.

The code fired by delete triggers to enforce referential integrity determines success or failure of the trigger, depending simply on the presence or absence of child records. However, sometimes the issue isn't the *presence* of child records, but their *state*, that determines whether the trigger should succeed or fail. In the preceding example, an update trigger calls two functions, one of which will always return .T. (the audit_time_cards() function). Thus, this function performs some action while the NewRI() function actually determines the success or failure of the trigger. We could also specify two (or more) different functions, all of which can return either a .T. or .F. As a result, if two or more functions joined by AND are specified as the trigger rule, all must "OK" the update, delete or insert, otherwise the trigger fails and the action is prohibited.

Consider a customer table whose relation with the orders table has a cascade delete rule. However, we don't want to delete any orders that are "open"—that is, not cancelled and not paid. We could specify the delete trigger as:

```
CREATE TRIGGER ON orders FOR DELETE AS Closed_Order() AND NewRI()
```

In this example, the NewRI() function would handle the cascading of the deletion to the order detail table, but only if Closed_Order returns .T. Closed_Order() might look something like this:

```
FUNCTION Closed_Order()
   LOCAL llRetVal
   IF INLIST(cStatus,"P","X") && Paid or cancelled
      llRetVal = .T.
   ELSE
      llRetVal = .F.
   ENDIF
   RETURN llRetVal
ENDFUNC
```

Summary

As with any other powerful feature in Visual FoxPro, rules, triggers and referential integrity can hose you pretty quick if you use them indiscriminately or without much forethought. However, they can, if carefully and thoughtfully employed, make the development process easier. Every process implemented at the engine level is one less thing you have to implement elsewhere in the application. Some things are quite gnarly to implement at the UI level, but become a walk in the park if implemented using a rule or trigger.