Data Base Basics: Getting Started in Paradox

by Dennis Santoro

Last Revised on January 15, 2002

The following document is a compendium of basic information that one should know about and consider when using Paradox as a database development tool. This is geared at a very basic level and is most useful for those just starting out with Paradox. Others may find it helpful however. This is not designed as a replacement for the help or the manuals (should you have them) but as a useful adjunct to them.

This is very much a work in progress. Any feedback regarding topics to cover, improving clarity, etc. are welcome.

General Considerations

The basic component of a database is data. Data is distinct from information. Data can be defined as collected or available input. As a general rule, databases are designed for the purpose of assisting with making decisions. Data are generally are not useful for decision making without further processing. Information is directly useful in decision making. It is based on processed data and therefore is the output of a data processing system. Databases underlie many data processing systems .

It is also useful to understand the distinction between databases and applications. An organization has numerous data resources. The combination of all the data which are useful as resources to the organization represent the potential scope of the database. The actual database is the combination of all the data which is actually collected and managed. While many data resources that might be useful could be included in a database, in all practicality that data will not all be available (e.g. proprietary information from competitors), will not all be manageable and will not all be within the scope of the development resources that a company can devote to database development. For these reasons it is important to define the database's scope. The scope of decisions which you want the database to assist with and the scope of data and processes which will be needed to assist with those decisions are the key components here. Because of this most databases are organic and grow in scope and features over time. Therefore, the importance of proper system design and database normalization can not really be understated. The more you do right in the planning process the less you will have to fix and the fewer compromises you will have to make later as you extend your system.

Databases are composed of data sets. Data sets are organized subsets of data which can be appropriately represented in two dimensions. These two dimensional sets are referred to as tables in Paradox. They will be described more below and the process of defining data sets is more fully described in the paper Data Normalization: A Primer by Dennis Santoro which is also available at http://www.RDAWorldWide.com.

A structure which exists between these two, data bases and data sets, is what users commonly

refer to as databases. These are, in fact data based applications. Applications are functional programs which make use of the data in one or more data sets to provide organizational information for some purpose. It is useful to understand these distinctions. The focus of database development should generally be on the purpose for which we are managing the information.

Data Sets and Their Components

A collection of properly normalized data sets, together comprise a database. Any data base may have one or more applications. These applications would be designed to manipulate the data and/or create information with the database. One database may, for example, have a customer contact tracking application and an order entry component as separate applications which meet different organizational needs.

Data sets in Paradox are referred to as tables. Tables are two dimensional representations of sets of data. They are, similarly to a spreadsheet, made up of rows of data items, referred to as records and columns of data items, referred to as fields.

Each record in a data set is descriptive of one single entity. Suppose, for example, that we have a list of people for which we want to create a table. Each person would be the entity described in a single record. Each field in a set describes a single attribute of all the records in the table. In our list of people, for example, one field would probably be first name and another would be last name which would describe two attributes of each person in the table.

In the Paradox format, tables consist of a set of files with the same name and a number of extensions. All data sets will contain a file in the format filename.db. This is the data table itself which contains the two dimensional set of records and fields. If you create a key for the table (which you generally should) you will also have a file called filename.px which contains the primary index (the index of the key). Keys are the unique value in a field, or combination of values from multiple fields, which make each record in the data set unique. Keys are also more fully discussed in the paper on data normalization.

If you add validity checks you will also have a file in the format filename.val. Other functions such as adding secondary indices will add other components. Secondary indices add files with the same name as the .db file but will add 2 files for each index. These will have an .X?? and a Y?? extension (where ? stands for a character).

Components of the same table will always share a file name but have these different extensions. When copying or moving Paradox tables you should copy or move all files with the same name regardless of the extension. If you use Paradox's internal copy function this is managed for you. If you use Explorer, File Manager, DOS' copy command etc. you must ensure this yourself. See the help under file names for a full list of Paradox file extensions and their meaning. These table components must, under most circumstances, be kept together to maintain the integrity of a Paradox table. For example, copying just the db file of a table without the px and val files will generally produce errors in Paradox if you try and open the table.

Beginning Design Considerations

In any database there are certain things that a developer must consider. First, each table in the database should have a key. The key is a unique identifier for each record. For example, in a Customer Tracking system the customer table would include a CustomerID which would be the appropriate key for the table. You have to explicitly identify it as a key when you structure the table in Paradox. In each table the key is the first field (or fields).

There are two approaches to creating keys. They can be based on actual data in the table which would create a unique identifier for the record. For example, in a phone number table, phone number would make a good key since no two phone numbers would be the same. Be careful using this approach, however. Some people will assume, for example, that Social Security Numbers in the US would make a good key for a table of people. Unfortunately, however, since a SSN can change or be duplicated (as in the case of identity theft and having your SSN changed if yours is stolen) SSN does not make a good key. Keys should not change once they are assigned to a record except in very rare circumstances.

The other school of thought here is that keys should simply be meaningless integers. These can just be assigned sequentially and, since they are meaningless but unique, they will never need to change nor will gaps in the sequence from later deletions be a problem. Note that a table's keys and its record numbers are in no way the same thing.

In most databases, multiple tables will be related to each other. These relations are made based on the record keys in the tables. (Again, see the previously mentioned paper on normalization for a full discussion). Continuing with our Customer Tracking example, we would probably want an associated table of phone numbers for these customers. Since any customer could have multiple phone numbers and we would not know how many they might have, we would want a separate phone table. As mentioned above, in this table an appropriate key would probably be the phone number itself. But to link it to the Customer table we would need to have the CustomerID to which this phone number belongs in the record as well. This fleld would provide the link. Since this is not the key nor part of the key for the phone table we refer to this as a foreign key in the phone table.

When linking tables we refer to the table which contains the primary entity as the parent table and the table which contains subsets which are related to that parent table as the child table. In the above example, the Customer table will be the parent and the phones table will be the child. The set of relationships amongst tables is referred to as the data model and the highest level table in a data model is referred to as the master table.

Tables can be related to each other in a one to one (1:1) relationship in which each record in a related table has one, and only one, possible related record in the other related table. It is possible for there to be no matching record to the parent table in the child table in this instance but there can not be more than one matching record between the parent and child. Tables can also be related to each other in a one to many (1:M) relationship. In this case any record in the parent can relate to any number of records in the child table but records in the child table can relate to only

one record in the parent table. Here also it is possible for the parent table to contain records with no related records in the child table. It is not acceptable for a child table to have records which are not related to one record in the parent table, however. These are referred to as orphan records and should not be allowed as they cause data validity problems. Tables can also be related many to many (M:M) in which case there can be any number of record matches between parent and child tables. Again it could be possible to have a parent record with no corresponding child record in a set of records related M:M.Since in the case of a M:M relationship either side of the relationship can legitimately be considered the parent or the child, records can exist in either table which have no corresponding match in the other table.

In Paradox, links between tables can only be made based on indices. An index in a paradox table is a listing of the unique values in a field or field combination and a pointer back to the record or records that contain that value. Indices can significantly speed up operations in Paradox which use them. If the link can not be made between the primary indices of two tables then secondary indices will be needed.

In our customer tracking example our customer table is keyed on CustomerID. This is an integer field and the table's primary index. Each CustomerID can exist only once in the table. In the phones table the key is the phone number so that each phone number can exist only once. But several phone numbers can be related to the same CustomerID so CustomerID can not be a key for the phones table. It can, however, be a secondary index. Then, to link the customer table to the phone table you link the CustomerID from the customer table to the secondary index on the phone table's CustomerID thereby creating a 1:M link. To link tables on a secondary index in Paradox and have the child table be able to be edited the secondary index must be specified as a maintained index.

As an alternative you could create multi part keys in the child tables but this has the drawback of increasing redundancy, increasing table size and slowing down key index access to the data set. In the above example the phone table would be keyed on CustomerID and phone number. Then any unique combination of the two fields would define a unique phone record. At this level we have not increased redundancy. But once we move to a third level of relationship we would since we would need the CustomerID, phone number and at minimum one other key value to create a unique record in a table which was subsidiary to the phone table. If we used the unique integer approach all we would need is the integer key for the new table and the foreign key phone number. This redundancy or reduction of it increases as our data model increases the number of levels it contains.

Additional Paradox Related Considerations

In addition to proper normalization and appropriate design of keys, indices and links there are several other considerations one should take into account when designing Paradox tables.

1. Avoid spaces, -,_ and other odd characters and reserved words such as Date and Time in field names, table names, directory names and aliases. While you might get away with it generally they will trip you up when you start coding. It is best to use only the characters

a-z, A-Z and 0-9 in filed names. Use mixed capitalization to maintain readability. For example, rather than use First Name as a field name use FirstName. Rather than use Item# use ItemNum. These characters are not a good idea in variables in code either but you will generally find out immediately if a variable which uses one of these will work or not which can not be said for table names, etc. For a list of reserved words see reserved words in the Paradox help. For a suggested naming convention see the faq TIP:PdoxWin:Suggested ObjectPAL Naming Convention:2000.03.25 at the coreldevelopers.faqs news group at the cnews.corel.ca or cnews.corel.com public news servers.

2. Long File names (any file name longer than 8 characters before the period) should generally be avoided. Microsoft has recently changed the algorithm which it uses to convert long file names to short ones for internal program use. Windows 2000, (and probably Windows XP) use a different algorithm than Windows 95, 98, Mellenium and NT 4.0. Also certain versions of Novell networks can have problems with long file names. This means if you use long file names that, because of the file name shortening algorithm, are incorrectly converted, you can have table read and data integrity problems. This is especially true if the files start with matching characters in the first 6 places. It is critical if you now, or potentially will, use a mix of operating systems using the different algorithms. Stick to 8 character names before the extension (.db, .px etc.) to avoid this problem. Since users are unlikely to see the table names and should they need to, you would have documentation of what is in each table anyway this should not be a problem for a developer. Users generally interact with a database through forms and reports. The forms and reports and your code are all that really need to know the table names.

3. Avoid auto increment fields (+) as key fields. There are numerous problems in using them, especially if the key is also used for a link. Primarily, under certain circumstances such as table corruption and rebuilds they can become resequenced and if you add tables together then numbers from the table being added from will become resequenced to the numbers after the ones in the table being added to. While newer versions of Paradox have improved this situation the fact that if a table is so badly damaged that you have to recover the data and add it to an empty copy of the table or if you add tables together the added data will become resequenced are sufficient reasons not to use Paradox autoincrement fields as keys or parts of keys. (This problem effects other databases too so if you are using other databases you should investigate this as well. Resequencing of your key values would break links and quite likely also link records inappropriately. Both of these situations would cause data integrity problems which would be unacceptable. Instead you can either write code to create the incrementing integers or you can check out Resource Development Associates' AutoKey product on our web site (http://www.RDAWorldWide.com) which can do this for you. If you chose to write the code yourself you will need to use a control table approach to key creation. See the next point in this paper for more on this issue. For more information on the control table approach see the AutoKey documentation. You can download the documentation for free.

4. To avoid auto increment fields many beginning users attempt to use Paradox's cMax function to get the highest number in the table and then increment it by one. If your application will

only be used in a single user environment then this is acceptable although the performance will not be great as the tables get longer. If, however, your application will be used in a multiuser environment then this is an unacceptable approach for several reasons. First, this approach is likely to fail in a multiuser environment since it may be the case that one user acquires the maximum value and begins editing a new record with the key set to the maximum plus 1. While this user is editing and the record is still uncommitted (posted) to the table another user may perform the same function. Now 2 uncommitted records exist with the same potential key value. When the first user puts the record back that will be fine. When the second user puts their record back the key will now already exist in the table thus causing their record to produce a key violation. This record will not be able to be committed. Second, a cAnything function such as a cMax requires a full lock be placed on the table. If another user is editing a record in the table or has a lock on any record or the table for any reason then the cMax or any other cAnything function will fail. In this case you will not be able to create key values and therefore will be unable to add records to the table. If you can get a full lock, no other user will be able to access the table for any purpose until your lock is released. This can cause performance problems and couse other operations to fail.

Aliases in Paradox

A major advantage of Paradox is the ability to use aliases. Aliases are names you provide that are mapped to directory locations (Public Aliases or Project Aliases) or that provide a different name for a table object (Table Aliases). You define aliases in the Paradox Alias manager, the Idapi configuration utility or BDE configuration utility or the Odapi configuration utility depending on your version of Paradox. You can also define aliases through Object PAL code. Table aliases can be defined in the data model designer in the form or report design tools. The specifics of defining aliases can be found in the help but the utility of aliases and some potential pitfalls will be discussed here.

Use aliases religiously. Do not use paths directly. When designing databases and applications separate your data from your forms, queries, reports, etc. Use the aliases whenever you reference the data, forms etc. whether building queries, forms, reports or writing code. This will make your code and application more portable and functional. This way if you need to move your application basically all you will need to do is redefine your paths that are defined by your aliases. You will not need to change any code as you would if you were coding actual paths into your calls to data, forms and other program components.

One pitfall users often stumble upon is defining lookup tables by alias. While this can be done the result will be tables that are only portable if you create exactly the same paths on the machines to which you move the tables. The problem here is that if you navigate by alias to a lookup table the path will replace the alias in the table structure. This makes sense as lookup tables control the allowable values in the field that does the looking up. Because of this, if you could simply realias the lookup to a different location the lookup table may not exist or may contain different data than the original. Either of these situations can cause key violations in the table that does the lookup. The best way to manage this is to keep lookup tables in the same directory as the tables that use

them for lookup.

If you have created lookups that have hard coded paths and you want to fix them so that they do not have hard coded paths, follow these steps.

1. Remove or rename the lookup table.
2. In interactive paradox, try and open the table that has the lookup.
3. When you get the error, lookup table not found or corrupt select the remove lookup option.
4. When the table opens, go straight to the restructure dialog box. Select save.

If the file saves your lookup is gone and the rest of the table is fine.

5. Now put the lookup table the same directory as the table that will do the looking up.
6. Make the directory with the tables the Working directory (this is one of the few cases you would ever want the working directory to be where the tables are).
7. Open the table that will have the lookup, go in to restructure and redefine the lookup. Just pick the lookup, do not use aliases or path navigation.
8. Save the file. The lookup should no longer have a path coded.

Steps 5 to 8 are the way to create lookups without hardcoded paths from the beginning, as well.

Another potential problem with lookups, even with ones that have no hard coded path data, is having the path to the folder they are actually located in be to long. Certain operations of the Borland Database Engine (BDE) are sensitive to paths being too many characters. This is usually around 64 characters. Lookup operations are one such circumstance where this is a problem. While you might think this will not affect you if you have not hardcoded a path in to your lookup in fact it does. Here is why. When Paradox attempts to use a lookup table it defers to the BDE. If the BDE does not find a path to the lookup hardcoded it assumes, correctly, that the lookup is in the same directory as the table that is doing the looking. To locate that file, it gets the path to the file doing the looking and uses it to find the lookup table. If the path the files are in is too long the operation will fail, therefore, because the BDE uses the relative path. So, it is advisable to keep your data folders near, but not at, the root of you drive or share that is housing the data.

Another pitfall is using the :Work: alias. While this alias is a default location to which Paradox looks for many operations it is mobile and can shift directories. Aliases you specify directly will always point to the directory you specified. It is a good idea to start the users up in a directory (which will end up being the :Work: directory) which is a neutral location so that users have to navigate via alias to any data, forms, reports, etc. This is good from a user training perspective. This also eliminates potential problems since if a user saves things they will not save them into your data or forms aliases but the neutral :Work: alias unless they explicitly tell Paradox to do otherwise. Users can be forced into a specific :Work: alias on startup either through code or through the -w command line switch on the startup icon. There are many useful command line switches. Unfortunately they are not well documented until the helps with the later versions of Paradox (beginning with Version 8). Forcing the :Work: alias to be a neutral location is also a good practice when designing an application as well. If you design with no parts of your ap in the

actual working directory you will immediately notice when you inadvertently omit an alias in a call to any data or application part.

Table Aliases

Table aliases are also useful. They are particularly helpful if a form or report requires a second instance of a table in the data model. This is often the case when placing lookup tables on a form for example since lookups are sometimes used by more than one main table. If you don't have the navigation by alias and then the tables themselves aliased you will have problems on ANY form where you try and place the same table twice. But with the table aliases it should work just fine.

To add table aliases step by step in Paradox 9 or 10.
1. Open a new BLANK form.
2. Select Format|Data model
3. Select the alias your tables are in.
4. One by One select the tables you want. As you select each one and it appears in the model, right click on it's blue title bar, select table alias and type in a name. Make certain no 2 names are the same. It is generally best to make the name different at the beginning of the table by adding a letter (e.g. ANames for the first, BNames for the second) or a number.
5. In the master of the first set select the link field(s) and drag them to the first child. Create the link or accept it. Once the link arrow exists you can right click on it and modify it as needed.
6. Continue for the subsequent tables.
7. When done, click OK.
8. Save.

It is a little simpler in Paradox 7 or 8 but the basics above will get you through it however you enter a specific form's data model.

Fixing Forms and Reports That Were Created Without Aliases.

Users often find out about Aliases and their advantages after having made a significant investment in designing forms and reports. They are then reluctant to go back and fix the older forms and reports thinking this will require a substantial effort. But, as it turns out, Paradox makes this quite easy. To Add aliases to previously unaliased forms or reports.:

1. Move the form/report or the tables so they are not in the same folder.
2. Give the folders aliases
3. Open the form in design mode.
4. You will get a message saying "Table (name) not found. Do you want to use a different table?"
5. Say yes and navigate to the table by using the alias (in the dialog that pops up).
6. This will continue for every table in the DM. Repeat as needed.
7. When complete, save and redeliver your form. (Note: you may have to fix code that does not use aliases such as calculated fields etc. but the basic code will work, you just need to add in the alias calls.)

Referential Integrity

Another area where users often get tripped up in designing Paradox databases is in setting referential integrity (RI). On the surface, the ability to have Paradox keep you from creating orphan records (records in a child table with no corresponding record in a parent table, which should not be allowed) or to have the change of a key value pass to child records that also use that key seems wonderful. The implementation of this in Paradox, unfortunately, is not really worth the effort; especially if you take the potential drawbacks into account.

To begin with, RI can only be one level deep in Paradox. If your data model is deeper than that you will want to write referential integrity code yourself to manage the additional levels. Since few data models will only be 2 levels deep this in itself should be sufficient reason not to use RI. If you already have to write the code for levels deeper than parent to first generation child then why not simply apply it to that level as well.

But a further consideration with referential integrity is that the paths for the tables are hard coded into the RI component of the table itself (the val file). This means that for your tables to be portable once you have applied referential integrity you will need to create exactly the same paths, including the drive letter, for any tables you wish to move. Also, in cases of table corruption, which does happen at times, you will likely not be able to recover a file with RI without deleting the val file which contains all your other validity checks. If you have to delete the val file you will have to recreate all the validity checks yourself.

Additionally, if you follow the keying and relationship concepts described above, where each table has an integer key and relationships are defined between the integer key and that value as a foreign key in the related child tables you will not need RI nor code to effectively implement cascading update RI. Since your key values will not change you will never inadvertently unlink a record or set of records by using RI cascading update between higher level tables. Writing RI code is the subject of two excellent papers by Liz Woodhouse on thedbcommunity.com.

Backup Table Structures

While it is important to have good backups for your systems, backups in Paradox require a little additional consideration. First, since a Paradox table is made up of multiple files, not all of which will change as records are updated, incremental backups are generally ineffective for Paradox backups. If, for example, you do incremental backup based on last change date, the val files will rarely be backed up. If you attempt to recover a database with a new db file and an old val file you will get a fatal mismatch. Therefore it is recommended that you do full backups of your data directories so as to get consistent copies of all the associated files.

Another important strategy is to keep empty copies of your tables. Should a file become so damaged that it requires you to delete your indices or your val files you would then be able to make a copy of your empty table and add the records form the damaged table to the known good empty copy of the table rather than rebuild validity checks, indices, passwords, etc. manually.

Local vs. Remote Paradox Installations

Paradox can be installed on a network in 2 possible configurations if your users need to share data and applications. The specifics of installing Paradox properly is covered in the faqs on the Corel news server (cnews.corel.com or cnews.corel.ca) and will not be covered again here. But the possible configurations can make a difference in your applications' perfomance.

Paradox (or Paradox Runtime) can be installed on the server in your network and run from the server by the workstations. Alternatively, Paradox can be installed locally on each machine in the network. Your applications (forms, reports, queries, libraries, etc.) can also be installed on the server or locally to each workstation. To properly share data it must be installed on a server. This is a prime reason for developing your applications with separate locations for the data and the other application components.

While data will have to move across the network to be shared, your applications and the Paradox program components will too if they are installed on the server. This can cause substantial network traffic. The greater the network traffic the slower each request will be served and the greater the likelihood of collisions of data packets. This performance hit can easily become unacceptable. It is also easy to avoid.

Installing Paradox locally to the user machine will allow all the Paradox program component requests to happen locally and generate little or no network traffic. This can be a huge performance improvement. While this can cause some difficulties in cases where you have to apply a patch to Paradox this can be mitigated by programs that manage network installations. Several exist on the market and many networks have and use them already.

Installing your application components local to the user can also greatly reduce network traffic and greatly improve application performance. In this case the forms, reports, queries, etc. will not have to be requested and provided by the server but rather will be served up right on request by the local machine from its own hard drive. This also can create some problems when providing updates and changes to your applications. Updates to your applications are certain to be more frequent than program patches. Most database applications undergo frequent revisions and additions as users become more familiar with the possibilities of the applications and as organizational needs change over time.

Managing updates of your applications can, however, be handled directly in Paradox. The general approach is to have a central copy of the application which users have access to but do not actually run. Then their local copy checks against the central copy to see if there are any updates since the local copy last ran. If so, the local copy copies the updates. If not it proceeds normally. You can write such an application yourself or you can buy RDA's AutoReplicator, which is commercially available from Resource Development Associates on our web site (http://www.RDAWorldWide.com).

You may also want to manage aliases and BDE settings centrally. This will not improve performance but it will make managing and modifying public aliases and BDE settings easier. To

do this you need to create a single IDAPI.cfg file with all the settings you wish. Then copy it to a central location on the server where users have sufficient access. Generally locating it in the same location as the .net file on the server is a good choice. You may prefer to put it in a directory which users have only read access to instead however. Make sure you set its read only property on and keep a good copy if you place it in a directory users have any type of modify access in. This will limit users ability to modify any settings. You can force users' Paradox to chose this IDAPI.cfg by using the -o command line switch (e.g. -o P:\netfile\idapi.cfg).

Interacting with Users

It is often the case that you will wish to interact with users; especially to get input values from them for you to manipulate. This will include things like asking the user for a date range, a value to search for, a password, etc. Unfortunately, the Paradox documentation, Help files and Examples in the help do not demonstrate best practices here. They often show user interaction done with View() which produces a dialog box which the user can type in to. This is a terrible way to interact with users and should be avoided. When you use view() you have no control over the formatting of the input, no way to directly check its validity and no ability to tell if the user wanted to continue or cancel the process. Fortunately, Paradox gives you the ability to write your own dialog boxes. All Paradox users wishing to interact with users via code should learn how to do this. It is actually pretty simple to do. There are some examples from a simple Password Dialog box to complex custom search dialogs and a generic use dialog available on the Paradox resources page at www.RDAWorldWide.com.

The basics are that you create a small form, set it as a dialog type in the windows style options and then create fields for the user to enter the info you need. Generally you add OK and Cancel buttons as well. This way you can define validity checks like pictures behind the fields to control data entry (whether the field is bound to a table or unbound) and use the OK and Cancel buttons to determine the user's intent when they close the dialog. Check out the samples on the web site and if you have questions, ask in the Paradox news groups. But do avoid using View().

Disabling Default Behavior

When writing code in Object PAL it is often the case that you want to disable the default behavior of an event or method. There are 2 basic ways to do this. One is to use the command DisableDefault in your code. Another is to use Eventinfo.SetErrorCode(). Eventinfo.SetErrorCode can be called with a specific Paradox error constant or one of your own defined as UserError or UserError plus an integer (e.g. Eventinfo.SetErrorCode(peCannotDepart), Eventinfo.SetErrorCode(UserError) or Eventinfo.SetErrorCode(UserError+1)). If you want to replace the default behavior, use disableDefault. If you want to prevent it, use Eventinfo.SetErrorCode().

Conditions of Use and Reproduction:

accuracy, you are solely responsible for your own use of this information.

Any statements of fact contained in this article should be interpreted as the opinions of the author, which may or may not reflect the opinions of any other entity involved in the transactions that led you to this article.

You may not distribute this information unless you meet the following conditions:

1. You obtain the permission of the author prior to such use including the specifics of what use is requested, any compensation you expect relating to use of this material. Any permissions will be deemed to be granted only for the use specifically agreed to in the document granting permission and only for the time period designated in said grant of permission. (Contact can be made via e-mail at RDAPermissions@RDAWorldWide.com. Please allow sufficient lead time).

2. All content (including this statement of conditions of use and reproduction) is provided completely unchanged.

3. Any additional conditions contained in any grant of permission are met by the user prior to any such use.

Commercial distribution can be arranged by contacting the author.

Feedback is strongly encouraged, especially constructive criticism and/or typographical/syntactical corrections. Response or action is left to the discretion of the author. Flames, abusive language, unsolicited commercial email messages (aka SPAM), and other forms of rudeness will be cheerfully ignored. Professional responses will receive priority attention. Unprofessional contact will be ignored.

All trade names, trademarks, and service marks are acknowledged as the property of their respective owners.