DataDraw 3.0 Manual

By

Bill Cox

Table of Contents

1 Introduction

DataDraw is an ultra-fast persistent database for high performance programs written in C. It's so fast that many programs keep all their data in a DataDraw database, even while being manipulated in inner loops of compute intensive applications. Unlike slow SQL databases, DataDraw databases are compiled, and directly link into your C programs. DataDraw databases are resident in memory, making data manipulation even faster than if they were stored in native C data structures (really). Further, they can automatically support infinite undo/redo, greatly simplifying many applications.

DataDraw databases can be persistent. Modifications to persistent data are written to disk as they are made, which of course dramatically slows write times. However, DataDraw databases can also be volatile. Volatile databases exist only in memory, and only for the duration that your program needs it. Volatile databases can be directly manipulated faster than C structures, since data is better organized in memory to optimize cache performance. DataDraw supports modular design. An application can have one or more common persistent databases, and multiple volatile databases to support various tools' data structures. Classes in a tool's database can extend classes in the common database. DataDraw is also 64-bit optimized, allowing programs to run much faster and in less memory than standard C programs using 64-bit pointers. This is because DataDraw databases supports over 4 billion objects of a given class with 32-bit object references.

DataDraw is released under the GNU Library General Public License, Version 2. It costs you nothing to use, and does not restrict your application in any way. Only the DataDraw program itself is covered by the license.

2 Use Cases

If your application is 99% GUI, and 1% data manipulation, don't use DataDraw, because that 1% isn't worth automating. If you need to write a CGI application for the Apache web server with a MySQL back-end, don't use DataDraw, because the speed DataDraw gives your application will be wasted. If you don't use data structures more complex than a tree, don't use DataDraw, because there will be little for DataDraw to automate. Use DataDraw when you need speed, efficiency, and/or rich data structures. Use it for the simplicity it brings your project, it's automated debugging, persistence, and undo/redo capabilities.

DataDraw is extensively used in EDA tool development, where speed is critical and data structures complex. It has, for example, been used in technology mappers, circuit simulators (both analog and digital), placers, and routers. DataDraw has been in use since in EDA since 1992, and has matured greatly over that time. DataDraw has also been used in compiler development.

Internet servers also benefit from DataDraw. A DataDraw backed application can process 100X to 1000X more transactions per second than a LAMP based application. This makes DataDraw a good choice for SIP servers, BitTorrent, and other applications supporting thousands of simultaneous connections. Embedded web servers could also benefit from DataDraw's small memory footprint, power efficient data manipulation, and ultra-high speed. Telephony applications, and other CPU intensive tasks are potentially a good fit. Editors of all kinds are a good fit with DataDraw, because of it's infinite undo/redo automation.

3 When to use DataDraw vs MySQL and PHP

LAMP is a very powerful combination for creating web applications: Linux, Apache, MySQL, and PHP.

Apache provides an incredibly powerful framework built around a world-class web server. PHP provides a powerful language for developing web applications rapidly. MySQL provides a way for these web applications to manage data. DataDraw is not meant to replace any of this. However, Apache is bloated, PHP is a slow interpreted language, and MySQL interprets ASCII commands that it reads through sockets that communicate with PHP. All this slows the system down 100-1000X, relative to plain old C code. Most applications don't care: if I'm just trying to sell stuff over the Internet, being able to process even one transaction per second is probably fine.

DataDraw is for demanding applications for which LAMP is too slow and/or bloated. While running, a DataDraw application owns the database, and does not share it with others. That makes it well suited for implementing some tasks, and not others. For example, it is well suited for building SQL servers, or BitTorrent trackers, and embedded servers, but not well suited for Apache modules. In these cases, consider embedding both DataDraw, and a free, fast, tiny HTML server, such as the MiniWeb HTTP server, directly in your application. This will allow you to serve many times more requests per second, in far less memory.

4 The DataDraw Language

Think of DataDraw database definition files as being similar to SQL files, but heavily focused on C-style data instead of the cryptic data formats supported by SQL. Like SQL, DataDraw is a language for describing data, not algorithms. You write your algorithms in C, but describe your database in DataDraw.

Here are the basic elements that makeup DataDraw code, and how they correspond to SQL terms:

- Module – similar to SQL databases
- Class – similar to SQL tables
- Relationship – similar to SQL foreign keys and C++ collections
- Typedef – similar to SQL blobs, which allow user-defined binary data to be stored
- Schema – just a logical grouping of classes that would look good together in an entity-relationship diagram
- Class fields – like fields (or "columns") in SQL tables
- Objects – like SQL table rows
- Object reference – similar to an SQL "primary key", or a C pointer

There are also some elements taken from C, which have no equivalent in SQL:

- Enum
- Relationship types: linked lists, hash tables, etc

Like SQL tables, DataDraw classes are made up of fields. Currently supported field types include:

- bool – Boolean type similar to C++ 'bool'
- bit – Exactly like bool, but uses only 1 bit in memory
- int – C integers
- uint – C unsigned integers
- char – C chars
- float – C float
- double – C double
- pointer – Handle to an object
- typedef – User defined data types, typically C structures in their programs
- enum – C enum
- sym – A symbol in a symbol table, with a C string for it's name

Pointers are similar to a C pointer, but in reality is an "object reference" which is a value used to access an object's fields. Pointers can be declared "cascade", which means that when an object is destroyed, the object pointed at should also be destroyed. "sym" is a handy symbol type. There is a global symbol table provided by DataDraw which keeps trackof symbols. A typical use for symbols is naming objects. The "bit" type is different only in that it tells DataDraw to encode the field asa bit in memory, rather than allocating a whole byte. This saves memory, but slows down reading and writing the value slightly.

You can also have unions of fields, just like in C, to save space in the database. Integers are 32-bit by default, but you can be more specific with any of the following:

    int8, int16, int32, int64, uint8, uint16, uint32, uint64

"uint" means unsigned integers. Any field can also be declared as an array, which can be dynamically sized. See the section "Arrays" below for more detail.

Perhaps the most important featureof DataDraw is "relationships". These are similar to "container classes" in C++. A big difference is that relationships are symmetric between a "parent" class and a "child" class. So, for example, if a car has a linked-list of tires, the tires will also have an owning car. Supported relationship types are:

- pointer – the parent and child simply cross point to each other
- linked_list – the parent has a singly linked list of children
- tail_linked – the parent has a linked list of children, and also a pointer to the last child
- doubly_linked – the parent has a doubly linked list of children
- array – The parent has dynamically sized array of the children
- hashed – The parent has a hash table of children, queried by symbol

Note again that there are no executable statements in DataDraw, such as assignment statements, loops, functions and such. You write your algorithms directly in C, as before. DataDraw only supplies the database.

5 A Simple Example

A DataDraw file starts with the module declaration:

```
module database db
```

This declares the "database"module, and says that all generated C functions and macros will be prefixed with "db", to keep them from conflicting with other functions in your application. If you leave out the prefix, all functions will be prefixed with the full module name.

After the module declaration, you can declare enumerated types:

```
enum Day DAY_
    SUNDAY
    MONDAY
    TUESDAY
    WEDNESDAY
    THURSDAY
    FRIDAY
    SATURDAY
```

In the generated code, the prefix "DAY_" will be prepended to your enum values. These constants can be used in directly in your program. For example, you could write code to thank God on Friday like this:

```
if(day == DAY_FRIDAY) {
        thankGod(); /* You have to provide this function yourself */
}
```

You can also declare external types defined in your C program. This is similar to the concept of a binary "blob" in a database. DataDraw will generate code that allows you to store these types in the DataDraw database. Just tell DataDraw that they exist with a "typedef" statement. So, for example, if you have a custom C structure you wrote by hand that keeps track of what's in a funky chicken's gizzard, you can tell DataDraw like this:

```
typedef Gizzard
```

And then you can declare classes that use these types. Lets suppose you wanted a database of funky chickens. Instead of creating a funky chicken table in SQL, you declare a class in DataDraw:

```
class FunkyChicken
        Gizzard gizzard // Who knows what the heck you defined in there...
        Day birthday // Just in case birthdays are very important to funky chickens
        FunkyChicken bestFriend
        array FunkyChicken chicks //Every funky chicken has lots of chicks
```

If you haven't noticed yet, there are no semicolons at the end of lines. In DataDraw code, elements are grouped by indentation, as in Python. Here's a simple DataDraw file describing a basic poker game database.

```
module poker pk persistent // "Poker" is the module name, "pk" is it's prefix

enum cardValue // Enumerated type of card values.
        CARD_2 = 2
        CARD_3 = 3
        CARD_4 = 4
        CARD_5 = 5
        CARD_6 = 6
        CARD_7 = 7
        CARD_8 = 8
        CARD_9 = 9
        CARD_10 = 10
        CARD_J = 11
        CARD_Q = 12
        CARD_K = 13
        CARD_A = 14

class Root // The root of the database – a good place for global data
        uint pot // Money in the middle – not the kind you smoke
        uint antiUp

class Deck //A deck of cards

class Card // One card in a deck
        cardValue value
        bool shown
```

```
class Player // A player in the card game
    uint cash

relationship Root Player hashed //This also gives the player a'Sym' field containing his name
relationship Root Player:dealer // By default relationships are 'pointer', or one-to-one
relationship Root Card linked_list
relationship Deck Card doubly_linked
relationship Player Card doubly_linked
```

Hopefully, two things about this format grabbed you're attention. First, the classes don't seem to have many fields. Deck doesn't have any! Second, there are a lot of relationships. This is fairly typical of DataDraw applications: heavy into relationships. Also, the “persistent” keyword causes DataDraw to generate a persistent database that keeps data mirrored in real-time on disk, and which loads at start-up.

Now let's suppose that you want to write an AI for playing poker. The AI will of course have all kinds of additional data, classes, and relationships. Further, it will want to attach additional data to the cards and players. DataDraw makes this easy to do. You just create an additional DataDraw file that might look a bit like:

```
module ai volatile // the prefix is the same as the module name in this case

import poker // This module runs off the 'poker' module

class Card:poker
    int scoreIfPlayed

class Player:poker
    double score
```

The “volatile” keyword is optional, since databases are volatile by default. What we've introduced here is dynamic extension, similar to what you can do in Python, but without any execution time penalty. The line “class card:poker” indicates that the local class card is a dynamic extension of the class of the same name in the poker module. Normally, C/C++ programmers have to put void pointers in their database classes as hooks to dangle additional tool specific data. DataDraw not only automates the extension, it does so without adding any void pointers to anything. This is one of the coolest features of DataDraw. See the Dynamic Extension below for more detail.

For a more detailed example, download the DataDraw source code. DataDraw uses a DataDraw generated database! See it's definition in Appendix A.

6 Installing and Running DataDraw

6.1 System requirements

DataDraw is very light weight, and can be used on Windows, Linux, Solaris, or even embedded platforms. The earliest versions ran on DOS, on IBM machines with 640K of memory and 12MHz 286 processors.

6.2 Compiling from Source

Until DataDraw is more widely adopted, you will likely need to compile it from source to use it. On a

Linux machine, do the following:

```
$ tar -xvzf datadraw-3.x.x.tar.gz
$ cd datadraw-3.x.x
$ ./configure
$ make
$ su
$ make install
$ exit
```

Note that 'x.x' should be replaced with the version number of your copy of DataDraw. This should create the 'datadraw' executable and copy it to /usr/local/bin/datadraw. If you would like to install it elsewhere, pass the "–prefix=<dir>" flag to the configure script.

Alternatively, you can check out the most recent source code using;

```
$ svn co http://datadraw.svn.sourceforge.net/svnroot/datadraw/trunk/datadraw3.0
```

This will create a datadraw3.0 directory, and you can then cd into it and compile as above. You also need to install DataDraw's utility library. This is done in a similar way:

```
$ cd util
$ ./configure
$ make
$ su
$ make install
$ exit
```

6.3 Command Line Arguments

DataDraw's command line has the following format:

```
datadraw [options]... module
```

Module files must end with a '.dd' suffix. The '.dd' suffix will be supplied if not given on the command line.

The 'datadraw' executable accepts the following command line arguments:

-a	-- Generate the database administration tool. Implies -p
-h file	-- Use file as the output header file
-I path	-- Add a directory to the module search path
-m	-- Start the database manager to examine datadraw's database
-p	-- Set the module as persistent. Implies -b
-s file	-- Use file as the output for the source file
-u	-- Set the module as undo_redo

DataDraw will create two files: dbdatabase.c, dbdatabase.h, where 'db' is replaced with the module prefix you defined in your database defintion file.

6.4 Module Path

DataDraw applications can be very large, with multiple projects exceeding 600K lines of C code. Such projects are built in a very modular way. There are common databases, persistent or not, and volatile databases for each tool that runs off the common databases.

Each common database and each tool has it's own database.dd file in it's source directory. Since a tool's database description file typically extensively depends on the common databases, DataDraw must be able to find them to generate code. By default, DataDraw looks only in the current directory. There are two ways to help DataDraw find imported modules.

First, you can use the '-I directory' option. However, if you want DataDraw to know this information in a more automatic way, consider setting the DD_MODPATH environment variable. Directories in this variable are separated with ':' characters. For example, in your .bash_profile (if you use bash), you could add:

```
EXPORT DD_MODPATH=source/maindatabase source/addtionaldatabase
```

7 Linking to Your Application

Building a DataDraw application requires the following steps:

- Include dbdatabase.c, dbdatabase.h, and util.a in your project
- Add DataDraw's 'util' directory to your include path for compilation
- For volatile databases call dbDatabaseStart(), where 'db' is your module prefix
- For persistent databases, see below for more detail.
- When exiting, call dbDatabaseStop(), especially if you have a persistent database

If you have multiple modules in your application, start any persistent database when your application starts, and stop them when it stops. For any volatile tool data, start their databases when the tool starts, and stop them when they are done.

Since DataDraw requires the 'util' module, you you will automatically have it ready for other uses. It has many helper utility functions found useful over many years of development. Check out the “Utility Library” section below for an in-depth description.

7.1 Additional Steps for Persistent Databases

If all you want is a way to save your program's data to disk, ignore this section, and just call the Load/Save functions provided in the C API. This provides a very fast binary read/write to disk of the entire database in one file. See Binary Load/Save later on in this document for how to do this.

First, instead of linking with util.a, link with utilp.a. Then, to use a persistent database, your application needs to either initialize it, or load it from disk. Assuming “pr” is your module prefix, and “graph_database” is the path to your database, you should load or initialize your database with code like this:

```
utStart();
prDatabaseStart();
prTheRoot = prRootAlloc();
utStartPersistence("graph_database", true, true);
```

This also assumes you have a root object in your database that you use to find all the other objects, and to keep track of your global data. The first parameter to utStartPersistence tells DataDraw where to save your data. The second says whether you want it saved in binary or ASCII. The binary form is compatible with utLoadBinaryDatabase, and the ASCII form is compatible with utLoadTextDatabase. The binary version is much faster, but the text version can be more convenient. The third parameter says whether or not you want to automatically keep a backup copy of the database.

You need to occasionally tell DataDraw when the database is at a stable point, such as when a transaction has been completed. Call the "utTransactionComplete" to indicate this. All database modifications made after the last call to utTransactionsComplete will be discarded the next time your application starts. See Transaction Processing below for more details on this. This function takes one parameter, "flushToDisk". If true, recent writes are flushed to disk right away. Otherwise, they are buffered in memory to improve disk write speeds.

To further speed up writing changes to disk, DataDraw only writes changes to one file, "recent_changes", which grows so long as you continue making changes. When you call utTransactionComplete, if the recent_changes file has become greater than 25% of the total size of the database, then the changes will be applied to disk, and the recent_changes file deleted.

Finally, when your application is shutting down, be sure to call "utStopPersistence()". This causes all recent writes to the database to be flushed to the recent_changes file, and closes all open database files.

7.2 Additional Steps for Infinite Undo/Redo

Whether or not your database is persistent, you can use DataDraw's infinite undo/redo feature. Instead of linking with util.a, link with utilu.a (for non-persistent) or utilup.a (for persistent). Also add the "undo_redo" keyword on the end of your module declaration, like this:

```
module db Database undo_redo
```

This will cause DataDraw to generate the undo/redo API calls you need. Be sure to also specify the "persistent" keyword if you want a persistent database with undo/redo.

Using the API is simple. Use the utTransactionComplete() command to indicate undo/redo stable points in the database. Then, to undo the last change, just call

```
utUndo(numChanges)
```

where "numChanges" is the number of undo commands you want to execute (typically just 1). To redo the changes after an undo, just call

```
utRedo(numChanges)
```

Be sure to only call utUndo after completing a transaction. The database is considered in an erroneous state otherwise, and datadraw exits - your database gets fixed the next time your application runs by dropping modifications beyond the last complete transaction.

With a persistent database, your undo/redo changes will be written to the recentChanges file. The database will not have a chance to be compacted until you tell DataDraw that you no longer need the undo buffer. Do this with:

```
utCompactDatabase();
```

This will compact the database, and reset the undo/redo buffer.

8 Database Administration, Backups and Viewing

DataDraw provides a simple database administration utility for managing your data. To invoke it, your program simply needs to call:

```
utManageDatabase();
```

It using a command-line interface to view and backup data. Commands supported are

- create <module> <class> – allocate a new object, and return it's object number
- compact – Compact the database, and delete the recent_changes file
- destroy <module><class> <object number> – Destroy an object
- help - this command
- list – list the modules in the database,their object counts and memory usage
- list <module> – list classes in the module, their object counts and memory usage
- list <module> <class> – list fields of a class
- quit - quit the database manager
- set <module> <class> <object number> = comma separated values – set all fields of an object
- set <module> <class> <object number> <field> = <value> – set a value
- show <module><class> – show all field values of all objects of the class
- show <module><class> <object number> – show an object's fields
- show_hidden <true or false> - Enable/disable listing of DataDraw internal fields
- load_binary <file> - Read the data from the binary database file into the database
- save_binary <file> - Write out the database in binary format to the file
- load_text <file> - Read the data from the text database file into the database
- save_text <file> - Write out the database in text format to the file

There are also two utility functions that allow your application to read/write ASCII database backups:

To backup:

```
utSaveTextDatabase(FILE *file);
utLoadTextDatabase(FILE *file);
```

Your application may benefit by command-line switches to access these features directly. Consider passing stdin and stdout for the FILE parameters.

Another way to do a backup is simply to make a copy of the database. However, if you use binary format, any change to your database defintion will cause your binary backups to fail to load. With text backups, load_text warns when you are missing fields, and sets them to a reasonable default value. It also warns if fields in your database no longer exist, and it drops such data.

8.1 The admindata Migration Utility (Preliminary)

This tool is under development, but should be available before you ever need it. Here is it's preliminary documentation.

Sometimes you may wish to modify your database definition and recompile your application. Unfortunately, this makes your application incompatible with any existing databases you have. The recommended way to migrate data is to write out an ASCII file in a custom data format, and to write a parser for that format which remains backwards-compatible.

However, in the real world, many programs simply don't have that nice backwardscompatible custom data format. If you need to munge your data to make it compatible with a new version of a DataDraw based application, use the 'admindata' utility. It reads in your ASCII backup file, and runs a TCL shell which lets you modify the data. When you're done, you can write the modified database back out in ASCII. The TCL commands you can use include:

- add <module>
- add <module> <class>
- add <module> <class><field> <type>
- drop <module>

- drop <module> <class>
- drop <module> <class> <field>
- delete <module> <class> <object number>
- get <module> <class>
- get <module> <class> <object number>
- get <module> <class> <object number> <field>
- set <module> <class> <class table>
- set <module> <class> <object number> <object value list>
- set <module> <class> <field> <object number> <field value>
- get_type <module> <class> <field>
- set_type <module> <class> <field>

With admindata, you can write TCL scripts to migrate your data to the new format. The “set_type” function tries to be smart about converting data. It uses standard C cast conversions to switch between types. If you change an object number size, every reference of that type in the database is also converted.

Note that the “delete” command does not cascade-delete any other objects. However, it does find all occurrences of pointers to the object and fixes them. For example, it will remove the object from linked lists and hash tables.

9 Arrays

DataDraw provides dynamically sized arrays. Just put the “array” keyword before a field declaration in a class. In particular, strings are represented with arrays of char. For example, you could create a string class:

```
class string
    array char value
```

This would create a database class with a variable sized string... kind of silly, so I don't actually recommend doing it. If you want to write a killer string class that takes over 1000 lines of code, go learn C++. The language was practically designed to build string classes.

When you declare an array, DataDraw automatically inserts some helper fields into your class. Consider the following example:

```
class Card

class Deck // As in a deck of cards
    array Card cards
```

In this example, class Deck will get an additional field, just as if you had defined it yourself, like this:

```
class Deck
    uint cardIndex // This is the index into the card heap of the first card
    uint numCards // This is the number of cards allocated in the heap
```

In the C API, additional functions will be generated for accessing the array, and resizing it. For more information, check the arrays section in th C API chapter.

10 Relationships

Relationships in DataDraw are highly efficient, and DataDraw's real strength. Compared to plain old C programming, the convenience of automatic support for things like doubly-linked lists is huge. Compared to C++ collections, DataDraw relationship are more efficient, both in speed and memory, safer, and automatically maintained.

Relationships in DataDraw incorporate one of the most powerful and important capabilties found in most SQL engines: the ability to cascade delete objects automatically. In fact, there is exact zero reason ever to write a destructor function ever again, for any application based on DataDraw. DataDraw automatically destroys dependent objects, and fixes up all the object references. If for some reason you need to do special processing when an object is destroyed, you can register a destruction hook with the database which will be called just before the object is removed from the database.

Relationships are between two classes: a “parent” and a “child”. Consider the following example:

```
relationship Company Employee linked_list
```

If we were to model a company with employees in C++, we'd be tempted to use a linked_list container class. These classes typically allocate objects containing a next pointer, and a void pointer to point to elements in the list. DataDraw, in contrast, directly embeds the next pointer in the child class, eliminating the void pointer. In addition, it embeds a back-pointer from the child object to the parent, so that when the child is destroyed, it can remove itself from the parent. You don't have to do it manually. It's done automatically when you call the function to destroy the child object.

In some cases, you know that a child object will never be destroyed while it is still in a relationship with a parent object. In this case, you may not want the back-pointer field. You can specify this with the 'child_only' relationship attribute. For example:

```
relationship Company Employee linked_list child_only
```

Though rare, in some cases you only want the back pointer, since the parent class never needs access to it's children. In that case, specify the “parent_only” relationship attribute.

In some cases, you will need named relationships. This is true in the case of a directed graph data structure. Both the parent and child classes can be labeled in a relationship. Here's the graph example:

```
class graph
class node
class edge
relationship graph node doubly_linked mandatory
relationship node:from edge:out doubly_linked mandatory
relationship node:to edge:in doubly_linked mandatory
```

Note the “:” labels. These labels say that one list contains all out edges from a node, and the other contains all in edges to a node. Whenever multiple relationships exist between the same two classes, they must have different labels.

Also note the “mandatory” relationship attributes. Mandatory relationships are almost the same as “cascade” relationships. Cascade relationships cause all child objects to be destroyed when the parent is destroyed. Mandatory only differs in debug mode. If a child is destroyed that does not have a parent, an error condition will be flagged. Semantically, mandatory relationships are those which the children must be in to be valid objects. Nodes, for example, exist in graphs, and not otherwise. Edges are always between two nodes. An edge without nodes makes no sense.

The relationship types are:

- pointer – the parent and child simply cross point to each other
- linked_list – the parent has a singly linked list of children

- tail_linked – the parent has a linked list of children, and also a pointer to the last child
- doubly_linked – the parent has a doubly linked list of children
- array – The parent has dynamically sized array of children
- hashed – The parent has a hash table of children, queried by symbol

We'll cover them one at a time.

10.1 Pointer Relationships

Pointer relationships simply embed a single field in the parent and child classes. These fields are simply references to each other. The field name in the parent is:

    LabelChild

Where "Label" is the optional label you specified in the relationship, and "Child" is the name of the child class. The field name in the child is:

    LabelParent

Which is similarly derived. So, if an Employee class is assigned to a Cubicle in a pointer relationship, it would be declared like:

    relationship Cubicle:assigned Employee:trapped

This would create two fields, just as if they had been declared like:

```
Class Cubicle
    Employee TrappedEmployee

Class Employee
    Cubicle AssignedCubicle
```

The difference, however, is that the destructor generated for Cubicle knows to clear it's employee's cubicle pointer, and vise versa. Had these fields simply been directly embedded in the classes, the destructors would not have know what to do with them! This makes manually embedding pointers in classes very dangerous. Extra care is required when using pointers in classes.

10.2 Linked List Relationships

Linked list relationships are quite simple. If a Node has a relationship to outgoing Edges, it might be declared like this:

    relationship Node:from Edge:to linked_list mandatory

In this case, the following fields are inserted into the classes:

```
class Node
    Edge FirstOutEdge

class Edge
    Node FromNode
    Edge NextNodeOutEdge
```

These fields can be accessed just like any other field you define in your class. In addition, for all relationships that allow multiple children, an iterator is automatically generated, so you don't have to access the "Next" fields manually in your Ccode. Also, functions to insert and remove objects to and from the list are generated. This is described in the "C API" section later.

Use linked list relationships when you know that objects will not often be removed from them, and when you don't mind always adding objects at the head of the list. Basic linked lists use the least memory of one-to-many relationships, but removing an object from a linked list runs very slowly: the entire list may have to scanned in order to find the previous object in the list.

10.3 Tail Linked List Relationships

Tail linked lists embed the same fields as linked lists, with one additional field. Again, using the Node/edge example, we would also have:

```
class Node
        Edge LastOutEdge
```

Use tail linked lists when you want to be able to append objects to the list. This is commonly done to preserve the order objects are processed. However, removal from a tail linked list is just as slow as removal from a linked list.

10.4 Doubly Linked List Relationships

Doubly linked lists use a bit more memory than linked lists, because every child has an additional pointer in it. However, removal of an object runs very quickly, in constant time.

The added fields are identical to the tail linked list, except that one more has been added to the child class. Again, with the Node/Edge example, we would have:

```
class Edge
        Edge PrevOutEdge
```

Use doubly-linked lists when removal speed counts. Doubly linked lists are heavily used in EDA applications.

10.5 Array Relationships

Array relationships are quite different than linked lists. If we have a Node class with an array of outgoing Edge objects, we might declare the relationship like:

```
relationship Node:From Edge:to array mandatory
```

This creates the following fields:

```
class Node
        array Edge ToEdges
        uint UsedEdge // This is used by the append function to automatically resize your array

class Edge
        Node FromNode
        uint FromNodeIndex
```

Note that "FromNodeIndex" is set automatically when you insert an edge into the array. It is used to help the destructor remove the Edge from the relationship.

Also note that all array properties also generate additional fields you can access. This is discussed above in the “Arrays” section.

10.6 Hashed Relationships

Hashed relationships are the most interesting of the lot. You get a bunch of fields added to your classes. Suppose we have the following relationship:

```
relationship Graph Node hashed mandatory
```

This gives Nodes names, and creates a symbol-based hash table of Nodes on Graph. The fields look like:

```
Class Graph
    array Node nodeTable
    uint numNodes
    Node firstNode
    Node lastNode

Class Node
    sym Sym
    Graph graph
    Node nextTableNode // Pointer to next node in sameposition in nodeTable
    Node nextNode
    Node prevNode
```

Hash tables have a super-set of the fields used by doubly-linked lists. This allows objects to be added to a hash table relationship in order, and even if they don't havean assigned symbols. Unnamed objects are added to the doubly linked list, but not put into the table. In the generated C files, code will be created to find a Node object, given it's owning Graph and it's name. Use hash table relationships when you have objects identified by unique names.

11 Unions

DataDraw supports C-style unions, with a catch. DataDraw needs to know what kind of values you have in your class, so it can do things like print it out in ASCII. To use a union, you must specify a property in your class that determines the type ofthe union.

For example, a generic drawn object might need a union of pointers to provide access to various kinds of shapes:

```
enum ShapeType
    LINE
    CIRCLE
    RECTANGLE

class Shape
    ShapeType type
    union type
        LINE: Line line cascade
        CIRCLE: Circle circle cascade
        RECTANGLE: Rectangle rectangle cascade
```

This declares that the shape object has apointer to a specific type of drawn object, and that when the shape is destroyed, so should the child object.

Array properties are not allowed in unions, since their size varies. Bit types are converted to bool. Unions are initialized based on their first property, which should be correct, since the enumerated type they depend on are initialized to zero.

12 Dynamic Extension

This feature is one of the main reasons DataDraw is used in EDAapplications, rather than some sort of C++ "persistent object base".

Consider a typical EDA system: we will have a common persistent database to model a netlist. This data structure will virtually define the architecture of the entire system. A simplified netlist database could look like:

```
module Netlist nl
class Netlist
class Instance
class Port
class Net
relationship Netlist Instance hashed mandatory
relationship Netlist Net hashed mandatory
relationship Instance port hashed mandatory
relationship Net Port doubly_linked
```

This simple database allows me to create netlists, and manipulate them. However, when my application starts, the netlist is loaded into the main database. All the tools that work off this database will need to attach additional fields to nets, ports, and instances.

If the main database was built using C++ classes, I'd face some difficulties. In particular, the main method of adding fields to a class in C++ is inheritance. But my objects already exist in the database. C++ programmers are reduced to using one of two methods to extend objects:

- Don't use inheritance. Emulate dynamic extension manually with void pointers in the database.
- Use inheritance, and copy the database into a local database that includes the extensions needed

Both techniques are commonly used. Both suck. Niether simplifies life by simply allowing you to extend objects that already exist.

Dynamic extension in DataDraw is trivial. Simply specify in your local database what class you are extending. For example, if I have a tool that needs to add a "level" attribute to instances, and a "delay" attribute to nets, it can be done like this:

```
module Mytool tl
import Netlist

class Instance:Netlist
        uint level
class Net:Netlist
        double delay
```

These class declarations to not create any new classes or objects! All they do is add new fields to the classes already defined in the database. When you call tlDatabaseStart(), the new fields will be allocated in the Netlist database automatically, and the will be freed when you call tlDatabaseStop(). In your code, you will simply continue to use Netlist Inst and Net classes and objects, but you will also have the additional fields.

Classes with local extensions can be used in relationships just as before. For example, if you have a Path class, it could be related to Nets:

```
class Path // Keep track of a singal path through a netlist
relationship Path Net linked_list
```

This allows your tool to build paths that include nets without worrying about converting data back and forth from the database. DataDraw makes where the data is defined nearly invisible. In fact, the only clue in the C API is the module prefix: functions are always prefixed with the local tool prefix. This means that functions that access local fields on Inst and Net will be prefixed with “tl” in this example.

Not only is dynamic extension convenient, its fast. There is no difference in speed when accessing extended fields vs main database fields. It's even faster than accessing fields in traditional C structures.

13 Schemas

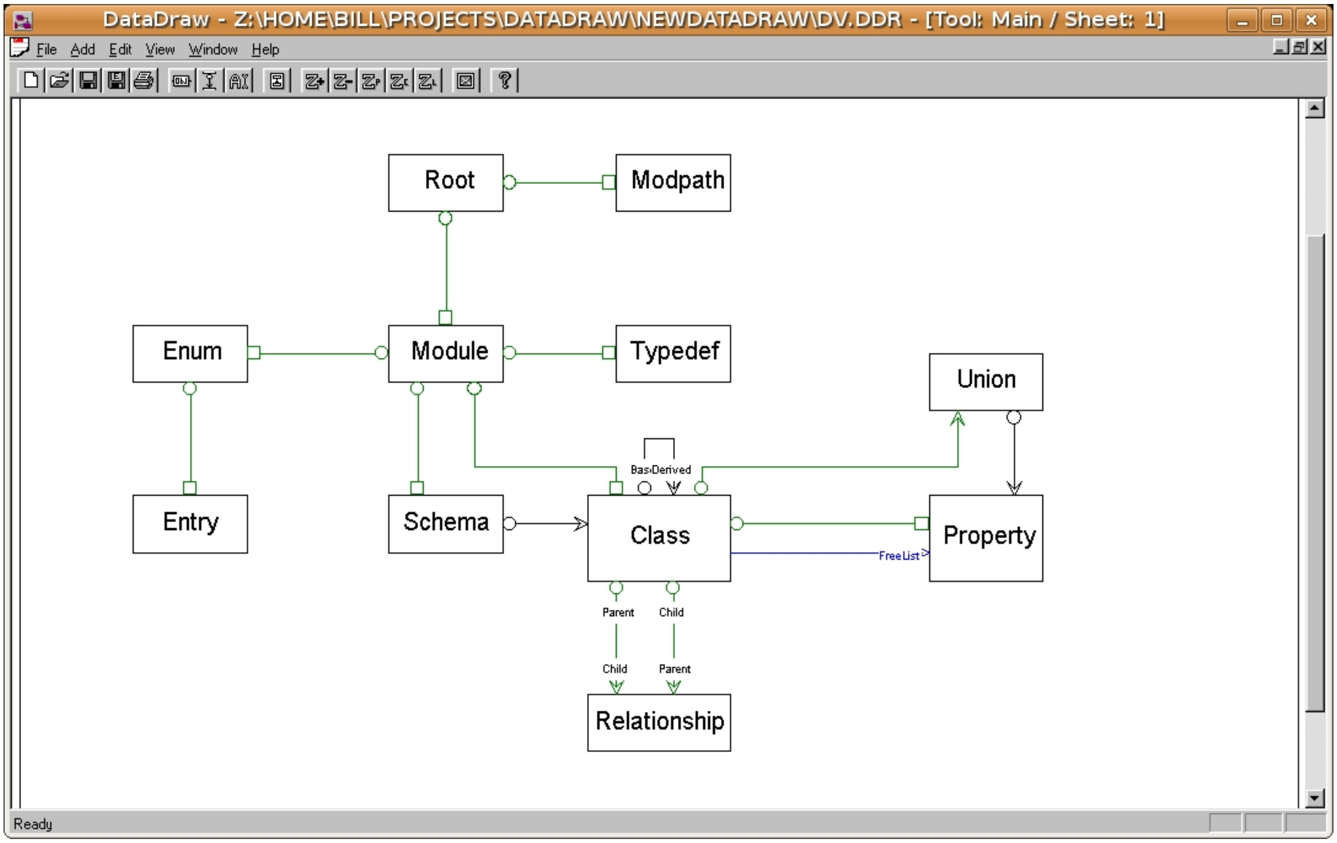
When programs get very large, there can be hundreds of classes in the common database. There needs to be a way to organize them into logical sections. Schemas serve this purpose. To declare a new schema within a module, simply use

```
schema Name
```

Where “Name” is whatever you want to call the schema. Class and relationship declarations following this will be part of this schema. By default, all classes not in any declared schema belong to a schema called “Main”. For example in an EDA database, you might have a “Netlist” schema, which describes netlists, and an “Attribute” schema, which shows how you attach attributes to various netlist objects. If you mixed these classes together, it would reduce readability.

Of course, you don't need a special statement to group your classes logically together. Why does DataDraw need to know about the groupings? The reason is for viewing. The administration tool generated by DataDraw allows you to show classes in any given schema Even more importantly, a graphical viewer for DataDraw schemas is in the works. When complete, it will automatically create entity-relationship diagrams for you, one schema at a time.

Versions 1 and 2 of DataDraw used schema editors, which allowed you to draw the classes of your database. For example, here is DataDraw's own database schema, drawn with DataDraw 2.0:



The boxes represent classes, and the arrows are relationships. The ends contain both the relationship labels and shapes that indicate the type of relationship. The bubbles are parent pointers, the light arrows are simple pointers, the full arrows are linked lists of some sort, and the empty boxes are hashed relationships. Arrays would be draw as boxes with an 'X' in them. Light green relationships are cascade delete, and dark green ones are mandatory.

We found a funny thing when trying to teach people to use DataDraw 1.0 and 2.0. Programmers get use to a way of coding and prefer to continue working this way. Possibly more than with any other profession, programmers are use to editing text files, rather than drawing pictures. We've changed DataDraw 3.0 to reflect this, and now generate databases from database description files, rather than schema diagrams.

14 Cache Efficiency

You may have read several times by now that DataDraw claims to be faster than native C code using tradition pointers to structures. This is true. In tests on various compute intensive EDA applications, DataDraw sped up tools anywhere from 10-20%, averaging about 15%.

The reason for this is vastly improved cache efficiency. When a typical inner loop goes through lists of objects, the fields needed are loaded into the cache, along with the ones that are not needed. Usually, most of the fields ending up in the cache are never accessed. With DataDraw, like data is stored together in memory. For example, if a Net class has a float called Delay, in memory the Delay fields are all stored together.

So, in a typical inner loop, if you access a field like Delay, you'll fill up your cache with Delay data, rather than useless bits your loop doesn't use. This makes DataDraw applications faster than traditional C applications.

15 64-Bit Performance

Can you tell that the authors of DataDraw are performance freaks? They are, and they ran into 64 bit computing when many people were still transitioning from 16-bits to 32.

Here's the problem they ran into: Everyone was expecting to be able to run bigger designs more quickly using 64-bit machines. It didn't happen. Instead, the programs used more than 50% more memory and ran 20% slower. So for example, an engineer might estimate he needs 5 gig of memory based on his 32-bit experience, but he'll soon find out that he actually needs 8.

Here's why: In typical large EDA databases, most data is in the form of pointers, not data fields. If 80% of your data was in terms of 32-bit pointers, and you move to a 64-bit machine, your data structures grow in size 60%. That means you need to buy a lot more memory to do the same job your old 32-bit computer could do. It also means that your cache will be even less useful, because your data no longer fits into it well. This is the main reason for the slow-down in run-times on 64-bit machines.

DataDraw solves these problems. 64-bit pointers are only used to point to the beginning of data arrays. From there, 8, 16, 32, or 64 bit integers are used to index into the arrays to access the data. The index is the object's “reference”. Most objects simply use 32 bit references (the default). In fact, no one to date has needed a 64 bit object reference, since no DataDraw application has ever had to load more than 4 billion objects of any given class, though it will surely happen one day.

To specify the size of an object reference, use the “reference_size” class attribute. For example

```
class BoardLayer reference_size 8 // Who needs more than 255 layers on a PC board?
```

This would allow layers in a PC board database to be referenced with 8-bit values. That can save a lot of memory.

16 Debugging DataDraw Applications

Since DataDraw is typically used for all of a program's data structures, rather than just persistent data, it is important to have debugging aids.

When you compile your application with the “DD_DEBUG” flag defined (pass -DDD_DEBUG to gcc) DataDraw does extra checking. In particular, it checks that all object references are valid before used, and that objects are not deleted twice nor accessed after deleted. Array bounds are also checked. Programmers who use DataDraw's debugging capabilities find little need for tools like Purify, since memory rarely gets corrupted.

An important is included in the C API when you compile in debug mode. You can call this from your debugger:

```
dbShow<class>(object)
```

So, for example, if you have an object reference of type Person stored in a local variable named “person”, you could view it's fields using the gdb debugger with:

```
call dbShowPerson(person)
```

If you want to access fields directly, it's fairly simple. Each class generates a global variable of the form <prefix><class>s. For example, a “Net” class in a module with a “db” prefix generates a global “dbNets” variable. It's value is the structure containing arrays for all properties of the class. So, for example if the Net class has a "Delay" field, and you want to know the delay on the netwith object reference 1234, you can see it in gdb with:

```
print dbNets.Delay[1234]
```

To see all the fields available, just print dbNets. The DataDraw API also declares one global variable per tool: <prefix>RootData. It has all the pointers to arrays of field data, and some bookkeeping variables. Usually, it's not used in debugging, but if for some reason you need to know the number of allocated or used objects of various classes, just print it out.

You will probably notice that references are integers, rather than pointers. This is different than what you are probably use to, but one nice thing is that they do not change from run to run. This greatly simplifies debugging. They are also smaller, and easier to remember or write down than pointers.

Another handy utility is the built-in database manager, which you can invoke in gdb with:

```
call utManageDatabase()
```

This lets you examine and modify your data, or even dump it all out as ASCII to a file.

A couple of handy functions to place break-points in are utExit_ and utError. When compiled with the DD_DEBUG flag, the DataDraw database calls these functions when it finds problems.

17 Transaction Processing

Some systems perform high value transactions, such as taking orders for products bought on-line. It is rather inconvenient when such a database gets hosed. DataDraw provides a way to help prevent this.

Users may call the dbTransactionComplete function to indicate that the database has reached a stable point (assuming your module prefix is "db"). When dbDDRStart is called for a persistent database, pass "true" for the "deleteIncompleteTransactions" parameter. When loading the database from disk, DataDraw reads the recent history of changes. Any changes past the last time dbTransactionComplete was called will be ignored.

To do proper transaction based processing, you need to be able to deal with multiple simultaneous transactions. Here's the DataDraw recommended way:

Create a volatile database for gathering transaction data. For clarity, let's assume the transaction is an order for some product. Gather all of the order information needed in this volatile database (name, credit card, etc). Then, when the transaction is ready to be committed, write all the databack to the persistent database. Finally, call dbTransactionComplete.

Assuming your running single threaded, like any true speed-freak hacker should, all database modifications between dbTransactionComplete calls will represent individual transactions. Well, that's not quite true... now that we have Core Duo processors, maybe you should have two threads... In this case, you'll need to lock the database beforewriting to it. DataDraw does not support this directly: you'll need to deal with mutexs on your own, which you should already know how to do,since you're writing a multi-threaded application.

18 Persistent Database Format

DataDraw conveniently saves your database in the exact same binary format used in binary load/save. It is in the file called "database". Changes to the database arerecorded in the file "recentChanges". Whenever "recentChanges" is large enough (currently 25% of the size of the database), "database" is overwritten, and “recentChanges” emptied.

This is done for two reasons. First, it greatly speeds up the database,since only one file is accessed to commit any changes, and all writes are done at the end of this file. Second, it provides a simple way to deal with transaction processing, since recent changes that are not committed can be ignored the next time the database is loaded.

Arrays are a bit different than other fields. Arrays are actually little heaps, similar to the ones supported by malloc. A file containing array data is a concatenation of blocks. Blocks are either allocated to an object, or they are free memory. each block contains:

A reference to the object owning the block, or null if freed
The data elements if not freed, or the length of free space followed by space if freed

To help with data alignment in memory, the field is padded with 0's to make it's size a multiple of the data element size.

New data for arrays is always allocated on the end of the heap. If there is no room, and the heap has less than 25% free memory, then the heap is made bigger, and the array is allocated on the end. Otherwise, the heap first compacted, so that all free memory is on the end in one large free block. This method results in O(C + N) average allocation time, where C is a constant, and N is the length of data being allocated. This is possible because the allocator is able to compact data on-the-fly, unlike data in a C heap.

For multi-byte fields, data is written to the database in the orderused by the host machine. This means that binary databases may not be compatible between CPU architectures.

19 The C API

The C API is simple and straightforward. All data is accessed through functions and macros generated by DataDraw.

To simplify discussion of examples, this section assumes we have defined the following netlist database:

```
module Netlist nl
class Netlist
class Instance
class Port
class Net
relationship Netlist Instance hashed mandatory
relationship Netlist Net hashed mandatory
relationship Instance Port doubly_linked mandatory
relationship Net Port doubly_linked
```

19.1 Object References

Objects are manipulated by passing their “references” to functions and macros that make up the C API. References are much like structure pointers, but what they actually are is burried under the API. In reality, they're integers. An object reference variable is declared with types like:

```
<prefix><Class>
example: nlNetlist netlist; /* This would be a reference to a Netlist object in module “nl” */
```

19.2 Null References

Every class has a unique NULL value declared for it, of the form <prefix><class>Null. For example, in module “nl”, a class named “Net” would have a NULL value called “nlNetNull”.

If you need to test to see if an object exists, compare it's reference to the NULL value for the class. For example:

```
if(nlPortGetNet(port) !=nlNetNull) {
    ...
}
```

19.3 Accessing Properties of Objects

Fields of objects are read with macros of this form:

```
<prefix><class>Get<field>(object)
```

For example, nlNetGetDelay(net) returns the delay field associated with a net. Setting has a similar format:

```
<prefix><class>Set<field>(object, value)
```

For example, nlNetSetDelay(net, 10.0) sets the delay of a net to 10.0.

This syntax is also used to get/set pointer fields in the database. For example, to set the next net port pointer of a port directly, use:

```
/* Remember, the NextNetPort field is automatically created. See Linked Lists for details. */
nlPortSetNextNetPort(port,nextPort);
```

There is one exceptions to how data is accessed. If the field is Boolean, the access format is simpler:

```
<prefix><class><field>(object)
```

However, the Set format is the same as for other types. For example, nlNetVisited(net) would return true if the Visited flag on the net were set. To set it, use nlNetSetVisited(net, value).

19.4 Enumerated Types

This couldn't be simpler... Enumerated types declared in a database description file are simply declared in the generated header file. For example, if the a nldabase.dd description says

```
enum InstanceType INST_
    NAND
    NOR
    INV
    FLOP
```

Then the header file will declare:

```
enum nlInstanceType {
    INST_NAND,
    INST_NOR,
    INST_INV,
    INST_FLOP
}
```

19.5 Symbols

Symbol fields are automatically added to your classes when you use hashed relationships. This function is used to create symbols:

```
utSym utSymCreate(char *name);
```

If you pass the same name to this function twice, the second call returns the symbol created the first time. Symbols are used so commonly as names in hashed relationships that if you use a hashed relationship, DataDraw automatically generates macros of the form:

```
<prefix><class>GetName(object)
<prefix><class>SetName(object, name)
```

For example, you can call nlNetGetName(net), and the value of the net's symbol will be returned as a char *. You can set the name with nlNetSetName(net, “N1”)

19.6 Typedefs

Typedefs declared in a database description are provided by you, not DataDraw. To enable DataDraw to manage fields of types you declare, you must create a special header file, called “dbtypedef.h” (assuming “db” is your module prefix).

That's all it takes. Once declared, custom fields types can be used just like any built-in type.

19.7 Default Constructors

To create objects, call their default constructors. Their format is <prefix><class>Alloc(). For example, nlNetAlloc() creates a new Net object in the “nl” database. Default constructors initialize all fields to 0, or equivalent (false for bool, etc). Typedef fields are set to 0, so be sure to initialize them in your constructor to reasonable values! To speed up allocation, you can use <prefix><Class>AllocateRaw(), however, since it does not initialize your object, it's use should be limited to speed critical loops. To make class specific constructors, you will write code that calls the default constructor, and then initializes the objects fields. By convention, these constructors you write should have the form <prefix><class>Create(...). For example,

```
/* Create a new net object */
nlNet nlNetCreate(
        nlNetlist netlist,
        utSym sym)
{
        nlNet = nlNetAlloc();

        nlNetSetSym(net, sym);
        nlNetlistAppendNet(netlist, net);
        return net;
}
```

19.8 Destructors

If you use DataDraw properly, you will never have to write another destructor function. DataDraw writes them for you. Their syntax is:

void <prefix><class>Destroy(object)

For example, nlNetDestroy(net) destroys the net, and any cascade or mandatory children objects, and patches up all the object references in all the relationships.

If you are a total speed demon, you may bypass DataDraws elegant destructors, and simply free an object:

void <prefix><class>Free(object)

It is rarely wise, but in some critical loops it can be useful.

If you use the "create_only" memory management style for a class, it turns out you can actually free your objects, but only all at once. The following macro frees them all:

void <prefix><class>FreeAll()

You can then start over creating them. By the way, default constructors for create_only objects are super-fast.

19.8.1 Destructor Hooks

Though rare, sometimes our application has to do special processing when an object is destroyed. For example, if you do reference counting, you will want to decrement a reference counter when a parent object is destroyed. This can be done with constructor hooks:

<prefix><class>SetDestructorCallback(function)
<prefix><class>GetDestructorCallback()

19.9 Manipulating Relationships

Relationships are manipulated with the following commands:

<prefix><parent>Insert<child label><child>(parentObject, childObject)
<prefix><parent>Append<child label><child>(parentObject, childObject)
<prefix><parent>InsertAfter<child label><child>(parentObject, prevChildObject, childObject)
<prefix><parent>Remove<child label><child>(parentObject,childObject)

For example:

```
nlInstInsertPort(inst, port); /* Insert aport on the head of the list */
nlInstAppendPort(inst, port); /* Append a port to the end of the list */
nlInstInsertAfter(inst, prevPort,port); /* Insert the port afterprevPort */
nlInstRemovePort(inst, port); /* Remove the port from the list */
```

Linked_list relationships do not have an "Append" function, otherwise, all these functions are available for linked_list, doubly_linked, tail_linked, and hashed relationships.

Hashed relationships also provide two more functions:

<prefix><parent>Find<child label><child>(parentObject, sym)
<prefix><child label><child>Rename(childObject, sym)

Normally, you set the symbol for a child object, and simply insert it into the hashed relationship. To find an object given it's symbol, you can use the Find lookup function. For example,

```
inst = nlNetlistFindInst(netlist, utSymCreate("U1"));
```

This will find the inst named “U1” in the netlist. To rename it, you could use:

```
nlInstRename(inst, utSymCreate(“U2”));
```

For array relationships, the following is used instead:

```
<prefix><parent>Insert<child label><child>(parentObject, index,childObject);
```

Note that the append and remove functions are the sames as for other relationships. For example, if in the “nl” module, Bus has and array of Net, we would have:

```
nlBusInsertNet(bus, index, net);
nlBusAppendNet(bus, net);
nlBusRemoveNet(bus, net);
```

In this case, the nlBusGetNumNet(net) function tells you how many nets were allocated on the bus. The nlBusGetUsedNet(net) function tells you “end” of the array, in the sense that an append function will add the next net there.

The Append function for array relationships is particularly useful. It automatically increases the size of the array as you append objects, so no manual sizing of the array is required.

19.10 Iterators

DataDraw provides simple iterators foryour relationships. The form is:

```
<prefix>Foreach<parent><child label><child>(parent,child) {
        /* ... your loop body here */
} <prefix>EndForeach<parent><child label><child>;
```

In practice it looks like:

```
nlForeachNetPort(net, port) {
        /* do something with port here */
} nlEndForeachNetPort
```

It is not wise to modify the list while iterating over it. If you need to remove an element from a list, use the SafeForeach iterators:

```
<prefix>SafeForeach<parent><child label><child>(parent,child) {

        /* ... your loop body here */
} <prefix>EndSafeForeach<parent><child label><child>;
```

which look like:

```
nlSafeForeachNetPort(net, port) {
        if(needToRemovePort(port)) {
                nlNetRemovePort(net, port);
        }
} nlEndSafeForeachNetPort
```

If your class uses the create_only memory management style, then an iterator is created that allows you to traverse all the objects of that class, in the order in which they werecreated:

```
<prefix>Foreach<class>(object) {
        /* ... your loop body here */
} <prefix>EndForeach<class>;
```

In practice it looks like:

```
nlForeachNetlist(netlist) {
    /* do something with netlist here */
} nlEndForeachNetlist
```

19.11 Array Manipulation

Array properties are get and set and allocated with these functions:

```
<prefix><class>Geti<field>(object, index)
<prefix><class>Seti<field>(object, index, value)
<prefix><class>GetNum<field>s(object)
<prefix><class>Alloc<field>s(object, numValues)
<prefix><class>Resize<field>s(object, numValues)
<prefix><class>Free<field>s(object)
```

For example, if an instance has an array of ports, it could be manipulated like:

```
nlInstanceAllocPorts(instance, numPorts);
nlInstanceGetNumPorts(instance); /* Returns number of allocated ports */
nlInstanceResizePorts(instance, newNumPorts);
nlInstSetiPort(instance, index, port);
nlInstGetiPort(instance, index);
nlInstFreePorts(instance);
```

There's one more way to manipulate arrays, but be careful with it. The macro:

```
<prefix><class>Get<field>s(object)
```

returns a pointer to the data in the array directly in memory. This is useful when calling qsort, for example. However, be aware that DataDraw moves this data around on it's own without telling you. If you use this macro to directly access data, be sure not to allocate or free any array data until your done using the pointer.

19.12 Persistence, and Undo/Redo

The following commands are used to control persistence and undo/redo. See the "Additional Steps" sections above for a more detailed description.

```
bool utStartPersistence(char*databaseDirectory);
void utLoadDatabase(void);
void utSaveDatabase(void);
uint32 utUndo(uint32 numChanges);
uint32 utRedo(uint32 numChanges);
```

The values returned by utUndo and utRedo are the number of transactions undone or redone.

19.13 Miscellaneous

There some additional useful macros:

```
<prefix>Allocated<class>()
<prefix>Used<class>()
```

These are used to find out how muchmemory has been allocated for a class, and how much of that memory has been used so far. DataDraw keeps a pool of free objects that it allocates from, so that it wont have to call malloc every time you create an object. There is also a pair of macros for each class for converting object references to integers and back:

```
<prefix><class>2Index(object)
<prefix>Index2<class>(index)
```

For example, to get an integer for a “Net” object in the “nl” module, use nlNet2Index(net). To convert the other way, use nlIndex2Net(index).

19.14 Array Class Types

It is very common to need dynamic arrays for holding object references. DataDraw automatically creates these little helper classes for you if you specify the array_class attribute in the class definition. For example, to have a helper array class generated for class Net, use:

```
class Net array
        ...
```

The effect is to automatically generate a helper class for manipulating dynamic arrays of nets. The class generated has this form:

```
<prefix><class>Array
example: nlNetArray
```

They are exactly like classes that had been declared like

```
class NetArray
relationship NetArray Net array
```

To use them, you write code like:

```
nlNetArray netArray = nlNetArrayAlloc(numNets);
nlNetArrayAppendNet(netArray, net);
nlNetArrayAllocNets(netArray, numNets);
nlNetArraySetiNet(netArray, index, net);
nlNetArrayGetiNet(netArray, index);
nlNet *nets = nlNetArrayGetNets(netArray); /* For use in qsort, for example */
nlNetArrayFree(netArray); /* Don't forget to free them! */
```

Most commonly, the append function is used to automatically grow the size of the array, so no allocation or resizing needs to be done manually.

19.15 Binary Load/Save

Many C++ programmers waste time overloading the >> and << operators so they can do binary load/save to disk. DataDraw not only automates this 100%, it makes it much faster, and requires no ongoing maintenance. That said, simple binary dumps make a very poor save format. If you change any fields in your database, it will no longer be backwards compatible. If you take the easy way out and use a simple binary dump, you may spend much of the rest of your life writing file format conversion utilities. A better way is to write a custom reader and writer, with a documented file format that can be supported well into the future. However, most projects like to start out using the free load/save format :-)

All you is call:

```
bool utSaveDatabase(char*fileName);
bool utLoadDatabase(char*fileName);
```

When you just need it quick, it can't be beat.

Be sure that you keep all your global values in the database, rather than global variables, so that they will be saved and loaded with your database. It is common to create a “Root” class for global data, as well as top-level relationships. If you only create one “Root” object, and if you declare the Root class “create_only”, then it can always be accessed with dbFirstRoot(), assuming your module prefix is “db”.

19.16 Watch Out for Side Effects!

Because DataDraw generates so many macros, it's a good idea to never modify data in an expression passed to a DataDraw function. For example, avoid code like:

```
dbNetSetLevel(net, level++);  // Wrong!!! level could get incremented twice!
```

The problem is that you should not count on the parameters being evaluated only once. In debug mode, in particular, it's common to evaluate parameters multiple times. For example, when indexing into arrays, the index is compared to the array size to make sure you're in-bounds. Also, all object references are checked for validity, so they are evaluated more than once. Instead, code like the line above needs to be written like:

```
dbNetSetLevel(net, level);
level++;
```

19.17 A Complete Example

Let's complete the simple netlist database example. Here's the database definition file:

```
module Netlist nl
class Netlist
class Instance
class Port
class Net
relationship Netlist Instance hashed mandatory
relationship Netlist Net hashed mandatory
relationship Instance Port doubly_linked mandatory
relationship Net Port doubly_linked
```

Here's a simple main.c program that uses it:

```
#include “nldatabase.h”

/* We have to write our own constructors :-(  DataDraw can't read our minds */

/* Create a new instance object */
nlInstance nlInstanceCreate(
        nlNetlist netlist,
        utSym sym)
{
```

```
	nlInstance = nlInstanceAlloc();

	nlInstanceSetSym(instance, sym);
	nlNetlistAppendInstance(netlist, instance);
	return instance;
}

/* Create a new net object */
nlNet nlNetCreate(
	nlNetlist netlist,
	utSym sym)
{
	nlNet = nlNetAlloc();

	nlNetSetSym(net, sym);
	nlNetlistAppendNet(netlist, net);
	return net;
}

/* Create a new port object */
nlPort nlPortCreate(
	nlInstance instance
	utSym sym)
{
	nlPort = nlPortAlloc();

	nlPortSetSym(port, sym);
	nlInstanceAppendPort(instance, port);
	return port;
}

int main(void) {
	nlNetlist netlist;
	nlInstance instA, instB;
	nlNet net;
	nlPort portA, portB;

	nlDatabaseStart();
	netlist = nlNetlistAlloc();
	instA = nlInstanceCreate(netlist, utSymCreate(“instA”));
	portA = nlPortCreate(instA, utSymCreate(“portA”));
	instB = nlInstanceCreate(netlist, utSymCreate(“instA”));
	portB = nlPortCreate(instB, utSymCreate(“portB”));
	net = nlNetCreate(netlist, utSymCreate(“net”));
	nlNetAppendPort(net, portA);
	nlNetAppendPort(net, portB);
	/* Do whatever you like with the netlist here... */
	nlNetlistDestroy(netlist); /* This gets rid of everything we just built... just for fun */
```

```
        nlDatabaseStop(); /* This gets rid of everything also */
        return 0;
    }
```

20 The Utility Library

The utilities in util.a have been developed over 16 years to provide basic functions that are not available on all popular computing platforms, and to help take care of some tasks that aren't provided for in the standard C libraries. It provides some basic error handling capability, and fairly powerful memory checking.

To use the utility package, be sure to call utStart() first. If you would like some final error checking to be done on memory, you can call utStop() before exiting.

20.1 Data Types

DataDraw supports several built-in types that aren't exactly like the built-in types in C. These new types are:

```
uint, uint8, uint16, uint32, uint63
int8, int16, int32, int64
bool
utSym
```

The utSym type is the same as "sym" in the database description language. "bool" is actually just char, rather than int, to save memory. This can occasionally force you to cast the result of a relational operator to bool in an assignement. There are also several constants defined for these types:

true, false – These are bool
INT8_MAX, INT16_MAX, INT32_MAX, INT64_MAX – These are maximum values
INT8_MIN, INT16_MIN, INT32_MIN, INT64_MIN – These are minimum values
UINT8_MAX, UINT16_MAX, UINT32_MAX, UINT64_MAX – These are maximum values
UINT8_MIN, UINT16_MIN, UINT32_MIN, UINT64_MIN – These are minimum values

A couple of handy constants are declared for helping parse paths on both Linxu and Windows:

UTDIRSEP – This is \ on Windows, and / otherwise
UTPATHSEP – This is ; on Windows, and : otherwise

20.2 Memory Access

DataDraw programs rarely reallocate memory, which allows them to have full memory debug code on all the time. DataDraw provides wrappers around calloc, malloc, and free for this purpose. What it does is verify that all memory is freed before utClose is called, verify that no pointer is freed twice, and it provides "pickets" at the beginning and and of all memory blocks allocated to help detect when we write past the end or beginning of a memory block. These functions are slow compared to malloc and free, so don't use them to allocate objects one at a time! The functions are:

```
void *utMalloc(int sStruct, int size); /* Allocates memory, but does not clear it */
void *utCalloc (int sStruct, int size); /* Allocates memory, initialized to 0's */
type *utNew(type); /* Allocates a structure large enough to hold the type */
type *utNewA(type); /* Allocates an array of 'type' */
char *utAllocString(char *string); /* Creates a copy of the string using ntNewA */
```

To check for memory leaks, just call utClose() before exiting. Any memory allocated with one of the above functions must be freed before calling utClose(), or a memory leak error will be reported.

20.3 Symbol Tables

DataDraw symbols are stored in symbol tables within the utility library. In your application, the type utSym is implemented for any 'sym' type in a database description file. Functions (methods) for utSym are:

```
utSym utSymCreate(char *name);
utSym utSymCreateFormatted(char *format, ...); /* Takes printf style formats */
```

The value utSymNull is the NULL value for the utSym class.

20.4 Random Numbers

If you are building a portable application, you may need a pseudo random number generator so that your application can use the same pseudo random sequences on any platform. The utility library uses an industry standard random number generator written by Takuji Nishimura. See utrand.c for more details. Functions provided are:

```
void utInitSeed(uint seed);
uint utRand(void); /* Return a random 32-bit unsigned int */
uint utRandN(uint n); /* Return a random 32-bit unsigned int between 0 and n – 1 */
bool utRandBool(); /* Return true or false, randomly */
```

20.5 Message Logging

It is often useful for an application to create a log file while it runs. The following commands support log files:

```
utInitLogFile(char *fileName); /* Call this once to creat the log file */
utLogDebug(char *format, ...); /* Just like fprintf to the log file... used for debug messages */
utLogMessage(char *format, ...); /* Like utLogDebug, but adds '\n' at the end */
utLogTimeStamp(char *format, ...); /* Like utLogMessage, but writes out the time first */
uint32 utStartTimer(char *format, ...); /* Like utLogTimeStamp, but also starts a timer */
utStopTimer(uint32 timerID, char *format, ...); /* Like utStart timer, but reports elapsed time */
utWarning(char *format, ...); /* Like utLogMessage, but prepends "WARNING" */
utError(char *format, ...); /* Like utWarning, but prepends "ERROR" and calls utLongjmp() */
utAssert(bool value); /* Like utError, but exits only if value is false */
```

The utStartTimer function returns a timerID, which needs to be passed to utStopTimer. This is used to verify that start/stop timer calls are nested evenly, which can be hard to track down otherwise.

20.6 Error Handling

The utility library supports hierarchical error handling. Only three functions are needed:

```
bool utSetjmp(void);
void utLongjmp(void);
void utUnsetjmp(void);
```

This greatly simplifies error handling, compared to raw setjmp/longjmp code. To use it, just call utSetjmp() at the start of your tool, and utUnsetjmp before you return. If you encounter an error condition, just call utLongjmp(). Even better, just call utError, which reports an error message, and calls utLongjmp() for you.

A typical tool that needs some error handling would look something like:

```
bool myTool(void)
{
        myDatabaseStart(); /* Your tool uses a DataDraw database, right? */
        if(utSetjmp()) {
                /* If you get here, it was an error condition */
                myDatabaseStop();
                return false;
        }
        /* Your tool does it's thing here */
        utUnsetjmp();
        myDatabaseStop();
        return true;
}
```

20.7 String Manipulation

The utility library provides a set of string buffers that can be used to simplify working with variable sized strings. There are 32 string buffers, which grow if needed. The idea is that often need just a few strings, and it's a pain to allocate buffers for them with malloc. This is one reason why so many programs have problems with buffer over-flows.

These functions are highly recommended, but be careful when using them, since they only return temporary buffers. If you use them, and run into trouble where a returned value gets overwritten, it's probably because you kept it around too long, and the buffer was reused. In such cases, allocate a buffer with malloc, and copy the string to it. Functions provided are:

```
char * utStrcat (char *string1, char *string2); /* Return the concatenation of two strings */
char * utStrncat (char *string1, char *string2, U32 length); /* Similar to strncat */
/* Replace the suffix after the last '.' character with a new suffix */
char *utReplaceSuffix(char *originalName, char *newSuffix);
char *utSuffix(char *name); /* Return the suffix of a string */
char *utBaseName(char *name); /* Return the name without a directory prefix */
char *utDirName(char *name); /* Return the directory name, without the file name */
/* Expand any ${variable} strings with values found in the environment */
char *utExpandEnvVariables(char *string);
char *utSprintf (char *format, ...); /* Create a string from the printf style format */
char *utVsprintf(char *format, va_list ap); /* Like vsprintf */
void utSetEnvironmentVariable(char *name, char *value); /* Set an environment variable */
char *utGetEnvironmentVariable(char *name); /* Get an environment variable */
```

20.8 File/Directory Information

The following commands can be handy when you just need some file information.

```
char *utGetcwd(void);
bool utFileExists(char *fileName);
bool utDirectoryExists(char *dirName);
bool utAccess(char *name, char *mode);
uint64 utFindFileSize(char *fileName);
char *utExecPath(char *name);
char *utFullPath(char *relativePath);
char *utFindInPath(char *name, char *path);
void utTruncateFile(char *fileName, uint64 length);
```

The utAccess command returns true if the mode is supported, which can be “r”, “w”, or “x”.

20.9 Miscellaneous

Some handy macros:

```
utAbs(a) – return a if a >= 0, otherwise -a
utMin(a, b) – return a if <= b, otherwise b
utMax(a, b) – return a if >= b, otherwise b
```

If you've gotten this far in this manual, you're probably a hard-core programmer. This being the case, I'll introduce you to a new looping construct that should have been built into C in the first place:

```
utDo {
    do-body
} utWhile(condition) {
    while-body
} utRepeat;
```

The utDo, utWhile, and utRepeat macros support this new looping construct. What happens is that the do-body is executed, and then the condition is evaluated. If true, the while-body is executed, and we then repeat, starting at the do-body. This handy looping construct eliminates the vast majority of cases where you would be tempted to place an assignment in a while condition. Consider this classic example:

```
while((c = getc(file)) && c != EOF) {
    /* Process character */
}
```

Many of us are uncomfortable allowing assignments in conditions because of the common bug of writing “=” when we mean “==”. This code is typically rewritten thus:

```
c = getc(file);
while(c != EOF) {
    /* Process character */
    c = getc(file);
}
```

While this only introduces one extra line of code, in more complex examples, the do-body can be very complex, and duplicating it twice is ugly. The utDo-utWhile-utRepeat loops fixes this. It was argued on the “D” language forum that this loop be included in D, but without success, even though this feature actually simplifies the language by combining both “do” and “while” loops into a single loop with optional parts. If you like it, feel free to use it. Most DataDraw programmer find it useful.

21 Appendix A – DataDraw's Database file

DataDraw's uses a DataDraw generated database itself. It makes a nice simple example for the DataDraw language. Here is DataDraw's database description file:

```
// These are DataDraw's data structures

module Database dv volatile

enum RelationshipType REL_
    LINKED_LIST
    DOUBLY_LINKED
    TAIL_LINKED
    POINTER
    ARRAY
    HASHED

enum PropertyType PROP_
    INT
    UINT
    FLOAT
    DOUBLE
    BIT
    BOOL
    CHAR
    ENUM
    TYPEDEF
    POINTER
    SYM
    UNBOUND

enum MemoryStyle MEM_
    CREATE_ONLY
    FREE_LIST

// This just gives us a way to keep track of top level data structures
class Root create_only

class Modpath create_only

class Module create_only
    sym prefixSym
    bit Persistent
    bit UndoRedo
    uint16 numFields // The total number of fields to be tracked
    uint32 numClasses // The number of classes in the module

// Helper class to break many-to-many relationship between modules and modules they import
class Link create_only
```

```
class Schema create_only

class Enum create_only
   sym prefixSym

class Entry create_only
   uint32 value

class Typedef create_only

class Class create_only
   MemoryStyle memoryStyle
   uint8 referenceSize
   bit generateArrayClass
   uint16 numFields

class Property create_only
   PropertyType type
   bit Array
   bit Cascade
   uint32 fieldNumber // Used in persistent databases
   Property firstElementProp // Used only for arrays
   Property numElementsProp// Used only for arrays
   bit hidden // Hides fields in the manager.  Used if we have to add a free-list property
   union type
         Enum enumProp: ENUM
         Typedef typedefProp: TYPEDEF
         Class classProp: POINTER
         sym typeSym: SYM // This is only used during parsing, to allow for forward references
         uint8 width: INT UINT
   uint32 line // This is set in parsing so we can report a line number in binding errors

class Relationship create_only
   RelationshipType type
   sym parentLabelSym
   sym childLabelSym
   bit Mandatory // Upper case avoids keyword collision
   bit Cascade
   bit accessChild
   bit accessParent
   bit sharedParent // Set for all but one relationship that share a common parent pointer

class Union
   sym propertySym
   Property typeProperty
   uint32 line // This is set in parsing so we can report a line number in binding errors
   uint16 number // This is used rather than a name, since unions are not named
```

```
    uint32 fieldNumber // Used in persistent databases

class Case
    sym entrySym // Used in binding to an entry

// Relationships are not owned by one class, but shared equally between "parent" and "child"
relationship Root Modpath hashed mandatory
relationship Root Module hashed child_only mandatory

relationship Module Class hashed mandatory
relationship Module Enum hashed mandatory
relationship Module Typedef hashed mandatory
relationship Module Schema hashed mandatory
relationship Module:Import Link:Import tail_linked mandatory
relationship Module:Export Link:Export tail_linked mandatory

relationship Schema Class tail_linked

relationship Class Property hashed mandatory
relationship Class Property:freeList child_only

relationship Enum Entry hashed mandatory

relationship Class:base Class:derived tail_linked mandatory
relationship Class:parent Relationship:child tail_linked mandatory
relationship Class:child Relationship:parent tail_linked mandatory
relationship Class Union tail_linked

relationship Union Property tail_linked
relationship Entry Case tail_linked mandatory
relationship Property Case tail_linked mandatory
```

22 Appendix B – DataDraw Syntax

This section defines the DataDraw syntax. It can be used with bison to create a parser.

```
goal: module

module: moduleHeader moduleParameters "\n" moduleStuff

moduleParameters: /* Empty */
| moduleParameters moduleParameter

moduleParameter: "volatile"
| "persistent"
| "undo_redo"

moduleHeader: "module" upperIdent optIdent
```

```
optIdent: /* Empty */
| IDENT

moduleStuff: /* Empty */
| moduleStuff moduleElement

moduleElement: import
| enum
| typedef
| schema
| class
| relationship

import: "import" upperIdent "\n"

enum: enumHeader "begin" entries "end"

enumHeader: "enum" upperIdent optIdent "\n"

entries: entry
| entries entry

entry: IDENT "\n"
| IDENT "=" INTEGER "\n"

typedef: "typedef" typedefIdents "\n"

typedefIdents: IDENT
| typedefIdents IDENT

schema: "schema" upperIdent "\n"

class: classHeader classOptions "\n" "begin" properties "end"
| classHeader classOptions "\n"

classHeader: "class" upperIdent optLabel

classOptions: /* Empty */
| classOptions classOption

classOption: "reference_size" INTEGER
| "free_list"
| "create_only"
| "array"

properties: property "\n"
| union
| properties property "\n"
```

```
| properties union

property: baseProperty
| baseProperty "cascade"
| "array" baseProperty
| "array" baseProperty "cascade"

baseProperty: IDENT upperIdent
| moduleSpec IDENT upperIdent
| propertyType upperIdent

moduleSpec: IDENT ":"

propertyType: INTTYPE
| UINTTYPE
| "float"
| "double"
| "bit"
| "bool"
| "char"
| "enum"
| "typedef"
| "sym"

union: unionHeader "begin" nonUnionProperties "end"

unionHeader: "union" upperIdent "\n"

nonUnionProperties: property ":" unionCases "\n"
| nonUnionProperties property ":" unionCases "\n"

unionCases: unionCase
| unionCases unionCase

unionCase: IDENT
relationship: relationshipHeader relationshipOptions "\n"

relationshipHeader: "relationship" upperIdent optLabel upperIdent optLabel relationshipType

optLabel: /* Empty */
| ":" upperIdent

relationshipOptions: /* Empty */
| relationshipOptions relationshipOption

relationshipType: /* Empty */
| "linked_list"
| "doubly_linked"
```

| "tail_linked"
| "array"
| "hashed"

relationshipOption: "cascade"
| "mandatory"
| "parent_only"
| "child_only"

upperIdent: IDENT