# A Normalized Data Library for Prototype Analysis

The goal was that each analysis and display tool to be included in the prototype analyzer should be designed and written only once. Therefore, the data library is designed to normalize the variety of basic logic analyzer data types and the variety of postacquisition data types generated by various analysis tools and to present this data to other analysis and display tools in a standard format.

**by Mark P. Schnaible**

Early in the design of the HP 16505A prototype analyzer we realized we needed a new library for storing and retrieving logic analyzer data if we were going to meet our project goals. We needed to duplicate the results of hundreds of software engineer-years of effort in the HP 16500A/B logic analysis system and measurement modules. We also had to lay the groundwork to meet the requirements of our prototype analyzer vision.

Understandably, we did not want to allocate enough people to meet the first challenge for the duration of our relatively short schedule. However, looking at where the original time was spent led to some insights. In the original HP 16500A/B system, effort had to be duplicated for each logic analyzer plug-in card introduced. That is, for each card, the lister, waveform, chart, and other software modules needed to be rewritten. Some code leveraging took place, but we did not come close to complete code reuse. The different ways in which the analyzer cards present data to the software (largely because of differing memory layouts) had much to do with this lack of reuse. Out of this observation came the design goal that each analysis and display tool to be included in the prototype analyzer should be designed and written only once. This meant that our library needed to handle the variety of basic logic analyzer data types as well as accommodate the variety of postacquisition data types that our envisioned analysis tools would generate, and present this data to other analysis and display tools in a normalized format. Fig. 1 shows the reduction in the dimensions of the coding effort we hoped to attain because of this library. It also shows that we would have automatic commonality of feature sets across acquisition modules. Previously, some features for a new logic analyzer card were not present at the initial release of that card.

Several other design goals emerged that would allow us to meet our challenges:

- Retrieval time for analysis and display tools to access the data should be minimized. In the HP 16500A/B environment, the acquired data is examined in typically one display per acquisition. In the prototype analyzer environment, the goal was to encourage data exploration and to view and analyze acquisitions with several tools. We wanted to permit users to examine data simultaneously with the same view in different locations and with different views in the same location. These use models led to multiple accesses of the same data, in contrast with the HP 16500A/B model of a single view of data in a single location. Fig. 2 shows the equivalent prototype analyzer graph of an HP 16500A/B use model while Fig. 3 shows a prototype analyzer graph with the multiple-view, multiple-location use model.
- Storage space for acquired data should be minimized. Given the potential size of some logic analyzer acquisitions (>40M bytes), undue expansion of the data was unacceptable. As always, however, there exists the trade-off between minimized retrieval time and minimized storage space.
- The storage mechanisms used by the library should be completely decoupled from application or client modules. That is, changing the way data is stored should not affect any lines of client code. Examples of client code include the tools that analyze and display the acquisition data such as the lister, waveform, chart, distribution, and pattern filter tools.
- The interface to the library should be a “natural” C++-style application programming interface. No new style of programming should be introduced that would be different from normal C++ syntax.
- The library should provide a layer of memory management over the acquired data. Again, the possibility of very large data sets makes it vitally important not to leak these huge memory chunks or provide stale pointers to nonexistent data. This goal is made more difficult by the potential complexity of graphs possible in the prototype analyzer. Fig. 3 shows that there may be many analysis and display tools examining and viewing the data simultaneously. All of these references to the data must be handled properly to prevent large memory leaks.
- Finally, time correlation between different sets of data or analyzer acquisitions must be possible. This requirement makes it possible for users to examine their systems on several different hierarchical levels

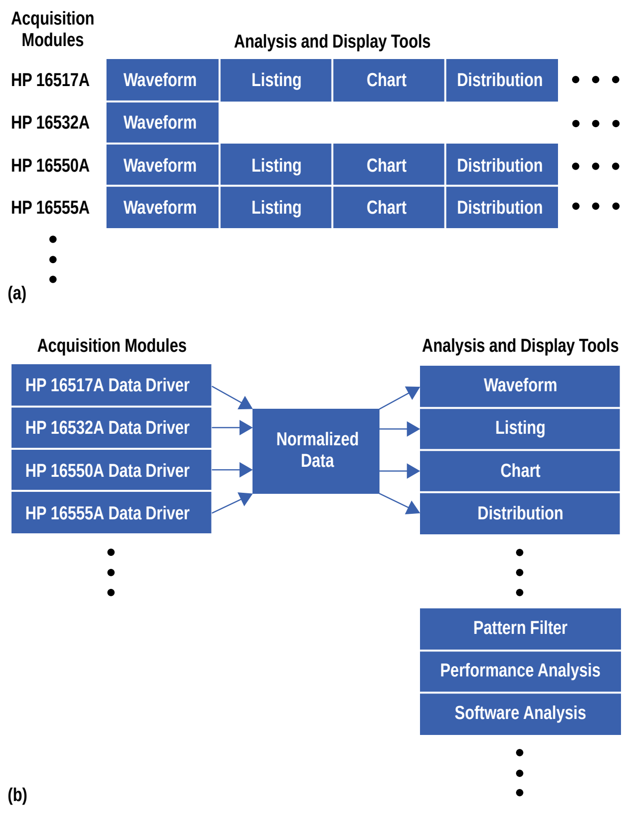


***Fig. 1.*** *(a) In the HP 16500A/B logic analysis system coding model, tool software had to be rewritten for each plug-in card. (b) In the HP 16505A prototype analyzer coding model, data is normalized immediately after acquisition and is available to all tools in a standard format. Thus, the tools had to be coded only once.*

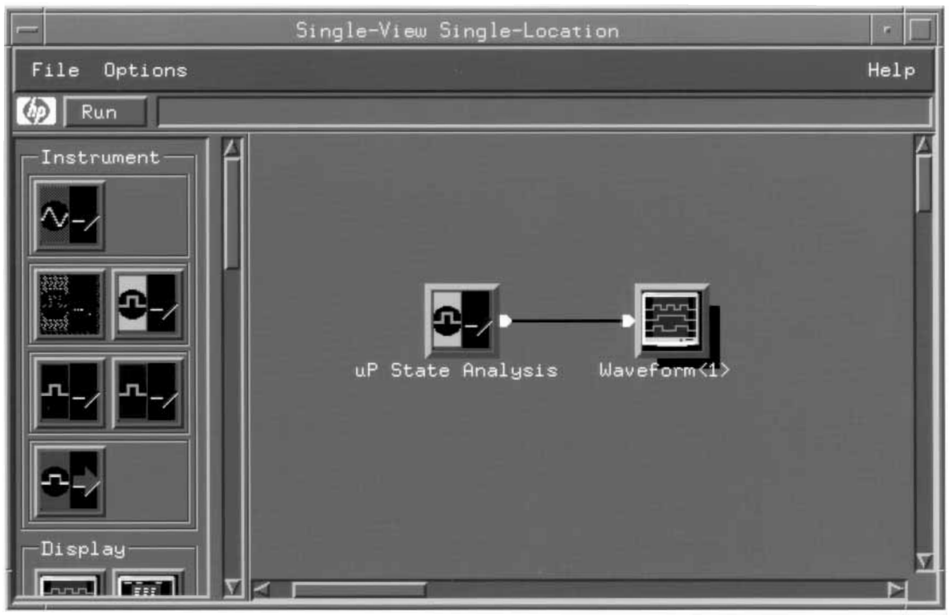


***Fig. 2.*** *Prototype analyzer graph of an HP 16500A/B logic analysis system single-view, single-location use model.*

concurrently, from looking at the analog nature of a ground pin bouncing to looking at the high-level source code executing at that moment in time. Time correlation of measurements also makes it feasible to plot one derived measurement against another derived measurement. Previously nonobvious relationships between variables may appear as a result of these charts. Simply having the ability to move markers in one view of an acquisition and watch that location automatically tracked in a different view of the same acquisition is another benefit of time correlation.

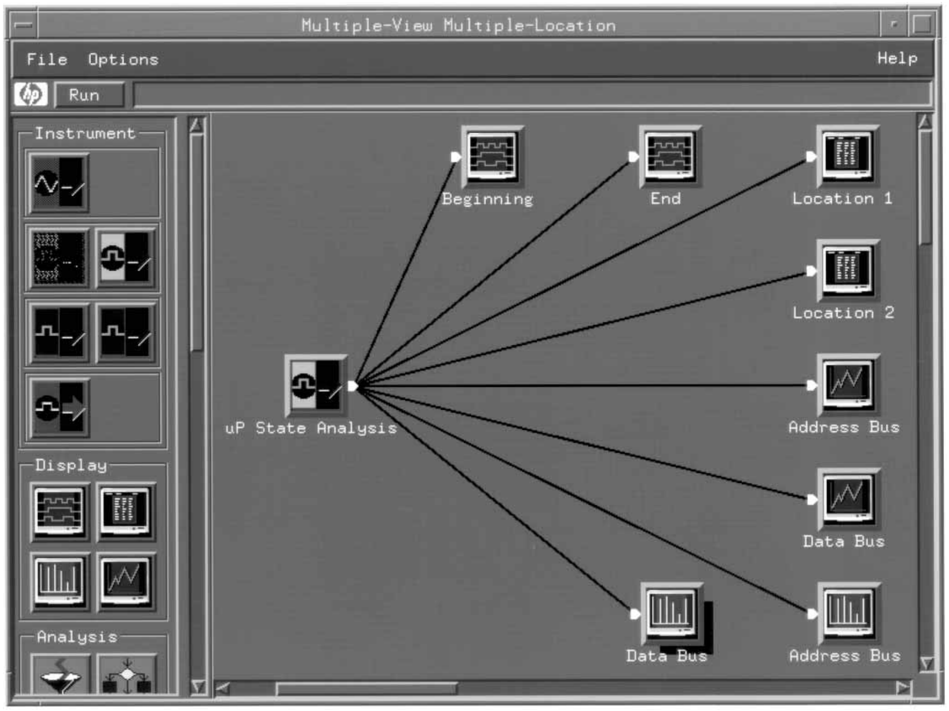


***Fig. 3.*** *Prototype analyzer graph of an HP 16505A prototype analyzer multiple-view, multiple location use model.*

Starting with this list of requirements, we searched for any existing libraries that would meet our needs. We examined three HP internal libraries and one public-domain library. None of these met all of our requirements. Among the reasons we rejected them were data set size limitations, non-C++ APIs, file-based memory storage, inefficient storage models, inability to handle the variety of data types, and mismatches between application models.

## Data Sets and Data Groups

The visual programming user interface of the HP 16505A prototype analyzer presents users with a left-to-right flow of data from sources to displays. Lines between the icons represent this flow. The first thing to decide was exactly what kind of objects move from icon to icon.

One of the first things users do in the HP 16500A/B environment is to define what the analyzer probes are connected to. This seemed like a natural place to start our definition of the normalized data library. A *label entry* represents one or more probes defining a logical value. Typical examples are the logic analyzer probes connected to an address, data, or status bus of a microprocessor or an oscilloscope probe connected to $V_{CC}$, $V_{SS}$, or a clock signal. The label entry has a name, polymorphic ordinate data, attributes, and perhaps a database of value-to-symbol mappings associated with it. Fig. 4 shows the class diagram of a label entry using the Booch notation.[1] A logic analyzer or oscilloscope commonly probes several of these label entries.

If we collect all label entries that share *exactly* the same sampling information, we have a *data set*. All of the label entries from a single logic analyzer acquisition share the same sampling information, so they can be collected into a data set. Likewise, all of the label entries from a single oscilloscope acquisition share the same sampling information. The data set

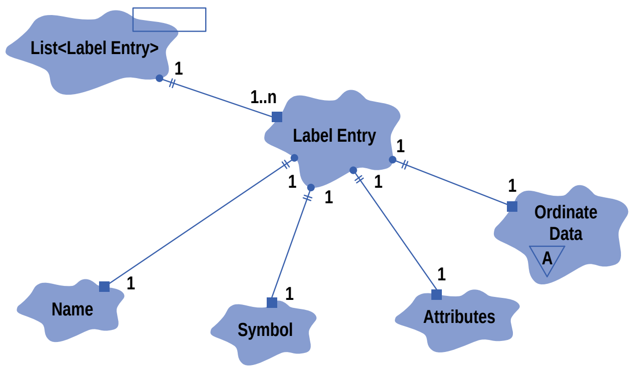


***Fig. 4.*** *Class diagram of the label entry class using Booch notation.*[1]

contains a pointer to some polymorphic data representing this sampling or x-axis information. We call this the *abscissa data*. The actual abscissa data object varies with the type of acquisition. The data set also contains time correlation information indicating which other data sets a particular data set can correlate with and indicating their relative trigger times. Another form of information contained in a data set is called *tags*. Tags represent the availability of each sample row in a data set. Tools such as the pattern filter or the sequencing filter can remove lines from a data set based on some pattern of data or a sequence of data. The memory for these lines is not deleted; the lines are simply marked as not available by these tools.

When we collect one or more data sets for input to a tool, we have a *data group*. Data groups are the objects that "go along the wires" of the visual programming graph. A data group is simply a list of data sets and some information indicating the nature of correlation among the data sets. A data group may contain data sets that have no correlation, time correlation, state correlation, or both time and state correlation. Each tool can indicate to the system what kind of correlation it requires incoming data sets to have. Inputs that do not match this criterion cause a warning message to be shown on the display. Fig. 5 shows the class diagram for data sets and data groups.

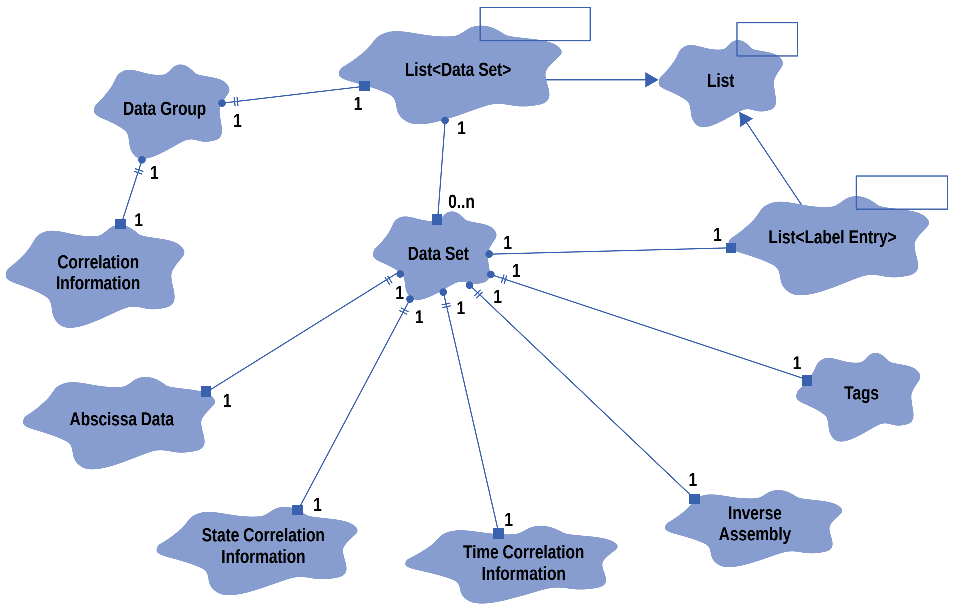


***Fig. 5.*** *Class diagram for data sets and data groups.*

## Abscissa Data

As mentioned above, the x-axis information (the abscissa data object contained within each data set) is a polymorphic type dependent on the type of acquisition or measurement of the data. Fig. 6 shows the class hierarchy of this information. The abscissa data base class represents a simple numbered sequence of states with one of those states being the trigger state. This concrete base class is the type used for generic logic analyzer state clocked acquisitions with no time information, that is, no time tags. Objects of type *periodic* are used for logic analyzer timing acquisitions as well as oscilloscope acquisitions. This class stores information about the sample period and exact time at trigger. The *time tags* class is used for logic analyzer state acquisitions with time tagging turned on. These tags indicate the exact time for each sample, which may or may not be periodic. This class is also used for calculated data sets derived from logic analyzer acquisitions. For instance, a series of setup and hold calculations derived from a timing acquisition will have a time tag for each pair of setup and hold values. We can save considerable space with this technique because the setup and hold times are sparse compared to the sample density of the original acquisition.

## Ordinate Data

The probed data or calculated data contained in the label entries is also a polymorphic object. The base class of this hierarchy is an abstract base class called *ordinate data* which defines the interface for classes derived from ordinate data. Fig. 7 shows the class hierarchy rooted with ordinate data. Classes derived from ordinate data are *analog*, *states*, *glitch*, and *state count*. Other classes can be added to this hierarchy as needed. The analog class contains a polymorphic pointer to the quantized samples and an object specifying how to convert these quantized values to floating-point values. In the case of an oscilloscope acquisition, this header contains the formula to convert from quantized samples to voltage. The states class contains a pointer to some polymorphic data representing the binary values acquired at the logic analyzer probes. The glitch class contains the same information as the states class as well as a parallel data object indicating the presence of glitches for each sample. The HP 16550A logic analyzer card is capable of acquiring this type of data. The state count class holds the data representing the number of states that transpired between acquired or store-qualified states. The trigger systems of HP logic

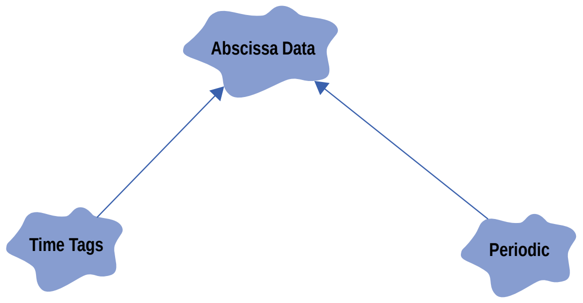


***Fig. 6.*** *Class hierarchy for abscissa data.*

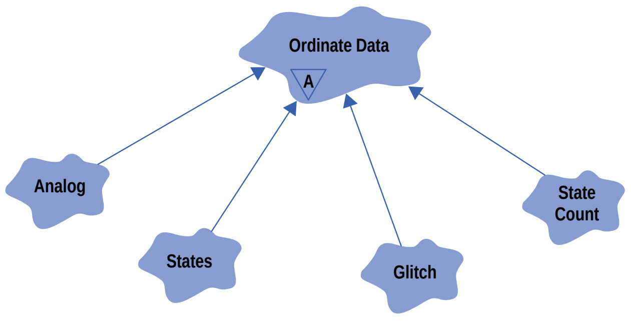


***Fig. 7.*** *Class hierarchy for ordinate data.*

analyzers allow users to specify which states to store out of all the states the logic analyzer sees based on some sequence of events. The state count object holds values that represent the number of states that were *not* stored.

## Analyzer Memory Layout

Our definition of a label entry shows that the data for each label entry is stored with that label and the data for no other label entry is stored with that label. This contrasts with the HP 16500A/B series of logic analyzer cards in which all of the acquisition data is stored in one block of RAM. Each time the data for a particular label needs to be retrieved, the bits must be extracted from this block and packed into a value representing a sample. As Figs. 8 to 10 show, the format of these blocks of memory differs from analyzer to analyzer.[2,3] The data formats in the acquisition cards are designed to be optimally space-efficient. There is a one-to-one mapping between bits of memory and probe tips so no memory waste occurs. There are two downsides to this design choice: (1) time must be spent every time a sample needs to be extracted from this block of memory and (2) the memory layouts are different for each analyzer. For the multiple-view, multiple-location use model of the prototype analyzer, access time for each sample of data is critical. For code reuse reasons, we need to hide the memory layouts from client code. Since these acquisition cards were already in production, we had little motivation to alter the way they presented data to the prototype analyzer. We wanted to minimize the software investment in these finished products.

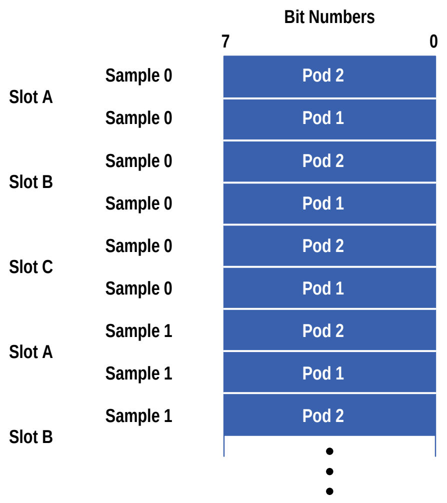


***Fig. 8.*** *HP 16517A acquired data format (three-card analyzer, full-channel mode only).*

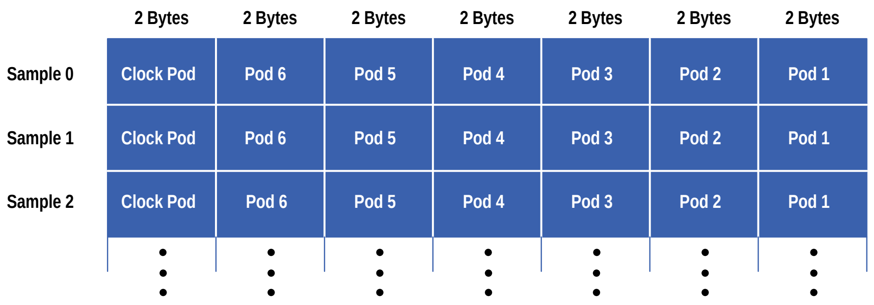


***Fig. 9.*** *HP 16550A acquired data format (one-card analyzer only).*

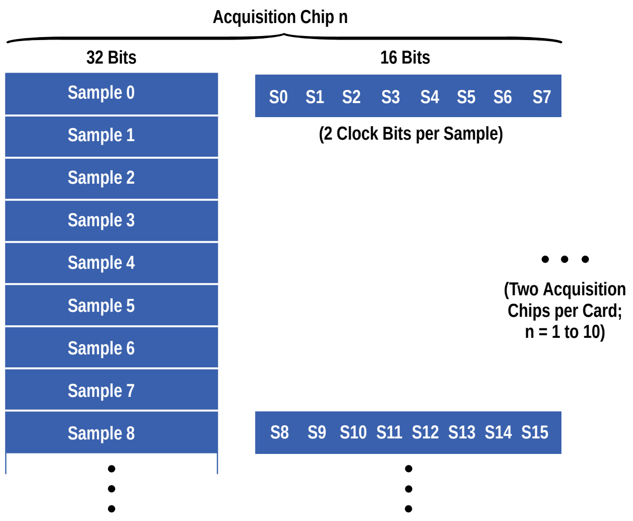
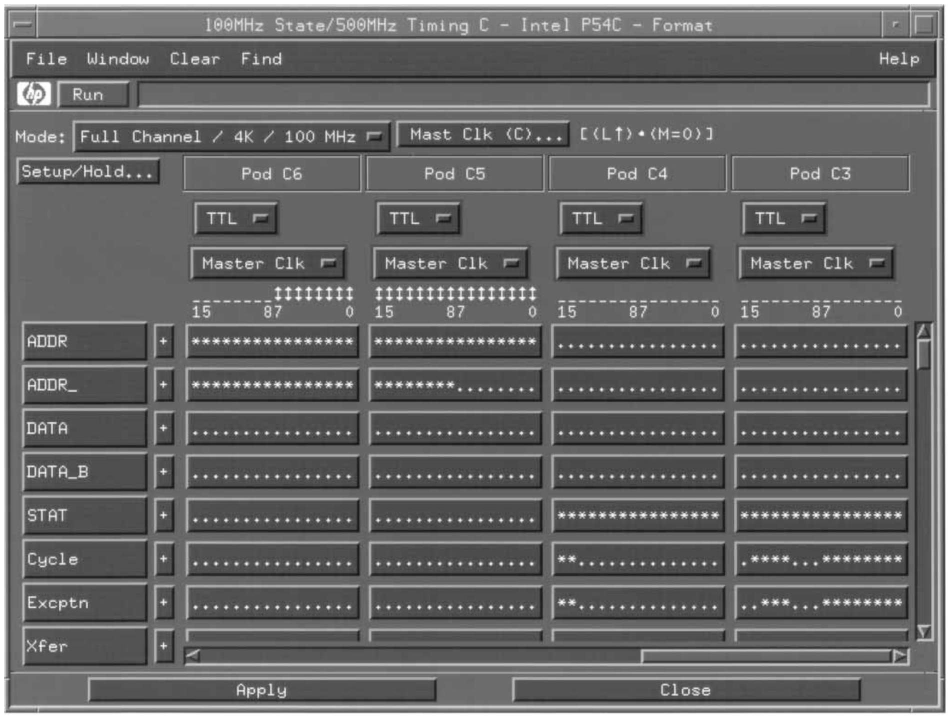


***Fig. 10.*** *HP 16555A acquired data format (full-channel mode only).*

## Format Specifications for Labels

Fig. 11 shows a partial format specification for the Intel P54C microprocessor. Customers use this dialog to specify which pins on various pods of logic analyzer probes are connected to different labels. Some labels, such as for an address bus or a data bus, tend to be probed with contiguous probe tips. However, other labels can be probed with discontinuous pins from different pods. The exception label (Excptn) in Fig. 11 is an example of this in the P54C specification. Depending on the complexity of the format specification, the extraction time for these labels can vary greatly. Even for contiguous labels with widths of 32 or 16 bits such as address or data, considerable time can be spent extracting these values from the monolithic block of memory.



Mode: Full Channel / 4K / 100 MHz    Mast Clk (C)...  [(L↑)•(M=0)]

| | Pod C6 | Pod C5 | Pod C4 | Pod C3 |
|---|---|---|---|---|
| | TTL | TTL | TTL | TTL |
| | Master Clk | Master Clk | Master Clk | Master Clk |
| | 15 87 0 | 15 87 0 | 15 87 0 | 15 87 0 |
| ADDR | **************** | **************** | ................ | ................ |
| ADDR_ | **************** | ********........ | ................ | ................ |
| DATA | ................ | ................ | ................ | ................ |
| DATA_B | ................ | ................ | ................ | ................ |
| STAT | ................ | ................ | **************** | **************** |
| Cycle | ................ | ................ | **.............. | .****...******** |
| Excptn | ................ | ................ | **.............. | ..***...******** |
| Xfer | | | | |

***Fig. 11.*** *Partial format specification for the Intel P54C microprocessor.*

For the normalized data library, we decided to extract the data for each label once for each acquisition and store it in its own chunk of memory. Label entries whose bit widths match the native machine data type widths (8, 16, or 32 bits) are extracted and stored as arrays of C++-type chars, shorts, or ints. Once these samples have been extracted or *normalized*, subsequent retrievals are simply array lookups. All labels that have non-native widths are combined into a block of bytes. Before the values are inserted into this block, however, any discontinuity in the ordering of the bits is removed to speed later retrievals. Some space inefficiency can result from this technique. The total possible bit wastage is:

$$\left[\left(\sum \text{non-native widths}\right) \bmod 8\right] \times \text{acquisition depth}.$$

We make this trade-off to provide faster access times to the data. Data drivers for each logic analyzer module supported are responsible for normalizing the data. These drivers can be arbitrary about choosing data types because they know the memory layouts of the cards and the specific widths of labels. Fig. 12 shows the class diagram of the ordinate data types with the various *integral data* types.

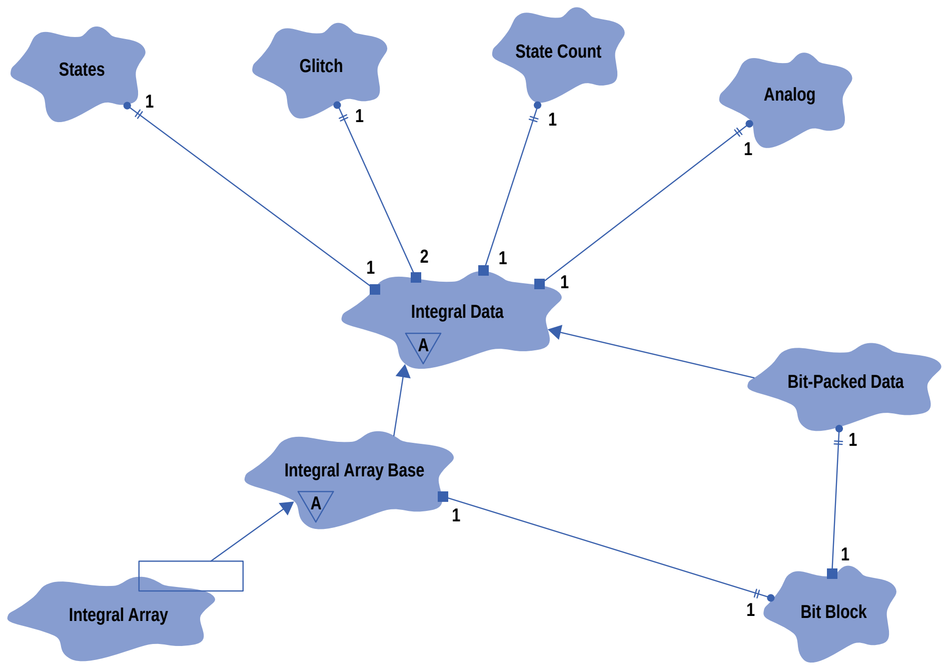


***Fig. 12.*** *Class diagram for the ordinate data types (top) with the various integral data types.*

## Accessing the Data

Once we have stored the data in a normalized format, we need methods to access the data. A study of existing code and an analysis of how future tools might need to access the data showed that retrieval tended to be very sequential in nature. If we picture an acquisition as a matrix of data, with rows representing samples or time and columns representing the various labels, two retrieval styles emerge: row major and column major. Tools that use the row major style want to examine all of the labels for a certain sample before they proceed to the next sample. The lister, pattern filter, and sequencing filter use this style. Tools that use the column major style look at all samples of a particular label or column before they look at the next label. Waveform drawing, charting, and histogramming tools access the data this way.

A programming idiom that supports this sequential nature of accessing data is called an *iterator*.[4] We defined classes for iterating over data contained in the abscissa data class hierarchy as well as data contained in the ordinate data class hierarchy. Fig. 13 shows the *abscissa-itor* class hierarchy and Fig. 14 shows the *ordinate-itor* class hierarchy.

Both diagrams show that there is a one-to-one correspondence between data types and iterators over those data types. Since we don't want client code to know how the data is stored, we need some way to hide this information from clients while still providing them with a way to get at the data. We use the letter/envelope programming idiom to accomplish this.[5] Clients construct an envelope object (*general abscissa-itor* and *general ordinate-itor* in the diagram), which can then ask the data to construct an iterator over itself. Clients never have to know what kind of data is being accessed. The envelope and letters are derived from a common base class which defines the interface for the iterators. The envelope serves as a forwarding object to the real iterator, the letter. The iterators have an orthogonal interface for data retrieval and iterator positioning. There are methods for looking at the next and previous data elements (which then alter the position of the iterator), methods for peeking at the next and previous data elements (do not alter the position), methods for querying and setting the position of the iterator, and methods for testing forward and backward iterator exhaustion. Multiple iterators can be constructed to look at the same data; this would be the case in a prototype analyzer graph that takes advantage of the multiple-view, multiple-location use model. Iterators are declared so that they can only access the data. We provided no iterator methods that would modify the data.

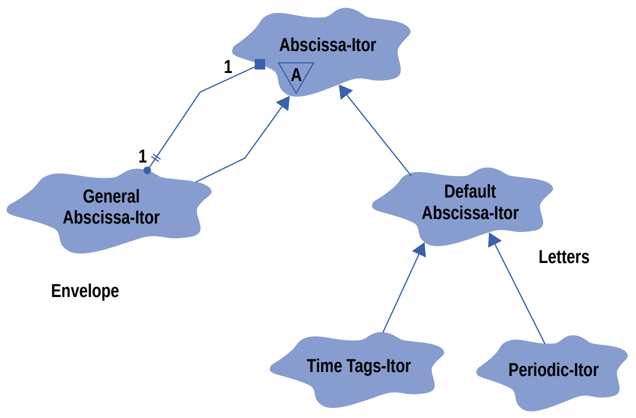


***Fig. 13.*** *Abscissa-itor class hierarchy.*

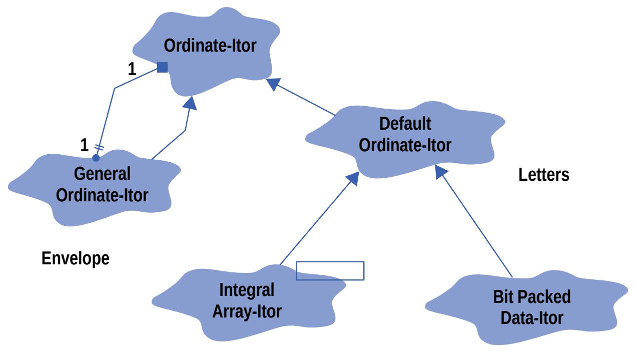


***Fig. 14.*** *Ordinate-itor class hierarchy.*

We also defined an iterator that is a combination of an abscissa-itor and an ordinate-itor. The *data-itor* retrieves pairs of values representing the value of a label and the sample or time at which it occurred. In addition to the orthogonal interface described above, this iterator also provides methods to report the value and location of the next or previous change in the ordinate data. These methods are useful for waveform drawing algorithms.

Row major iteration over multiple correlated data sets is aided by group abscissa iterators. These iterators examine a list of abscissa-itors (one for each data set in the data group) and return the next or previous x-axis value and a list of identifiers that select the data sets from which that value comes. These iterators come in handy, for example, in visual programming graphs that have multiple analyzers fanning in to a single analysis tool (Fig. 15). For a lister display that has multiple data sets merged (Fig. 16), we need to show the data from the various analyzers interleaved in time as they were actually sampled.

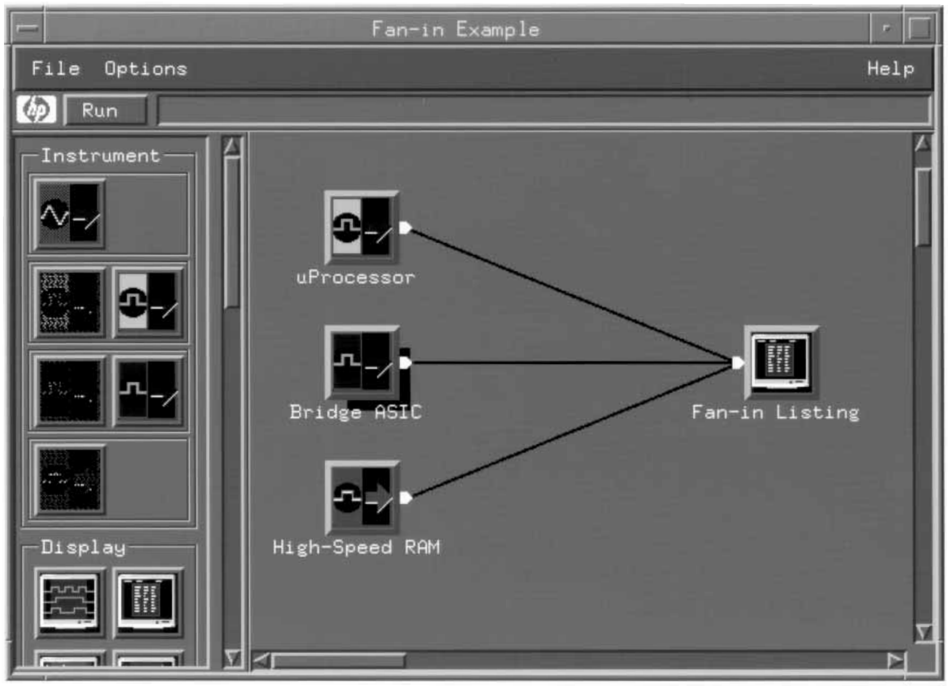
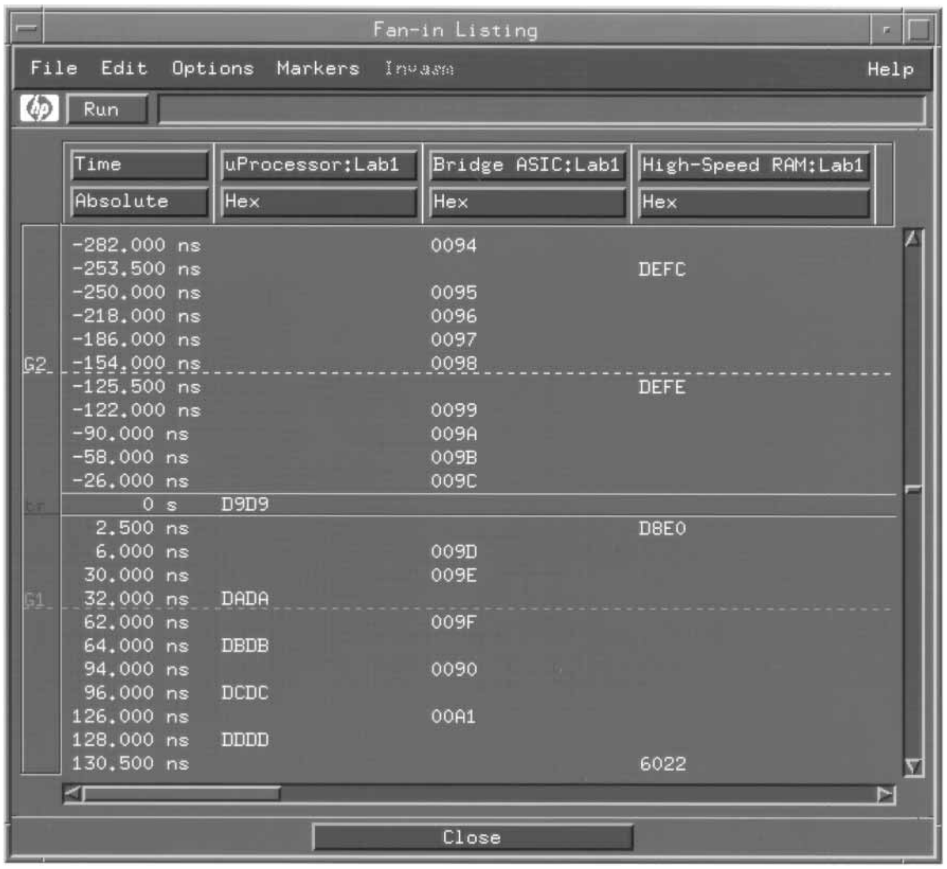


***Fig. 15.*** *A visual programming graph with multiple analyzers fanning in to a single analysis tool.*

## Memory Management

With the increasing acquisition depth and width of HP logic analyzers, data sets can have sizes greater than 40M bytes. In the prototype analyzer environment, users can be examining these data sets in several locations with several views. Having so many references to the data can easily lead to memory management problems, particularly nasty among them being memory leaks and pointers to stale memory. A memory leak occurs when a process no longer has any references to a chunk of memory that has been allocated from the heap of memory available to the process, and thus there is no way to return the memory back to the heap. As an application leaks more and more memory, less is available to that process and eventually



```
   Time             uProcessor:Lab1   Bridge ASIC:Lab1   High-Speed RAM:Lab1
   Absolute         Hex               Hex                Hex
   -282.000 ns                        0094
   -253.500 ns                                           DEFC
   -250.000 ns                        0095
   -218.000 ns                        0096
   -186.000 ns                        0097
G2 -154.000 ns                        0098
   -125.500 ns                                           DEFE
   -122.000 ns                        0099
   -90.000 ns                         009A
   -58.000 ns                         009B
   -26.000 ns                         009C
tr        0 s       D9D9
      2.500 ns                                           D8E0
      6.000 ns                        009D
     30.000 ns                        009E
G1   32.000 ns      DADA
     62.000 ns                        009F
     64.000 ns      DBDB
     94.000 ns                        0090
     96.000 ns      DCDC
    126.000 ns                        00A1
    128.000 ns      DDDD
    130.500 ns                                           6022
```

***Fig. 16.*** *A lister display with multiple data sets fanned in. The data sets from the various analyzers are interleaved in time as they were actually acquired.*

the application will halt because of an out-of-memory error. Leaking very large blocks of memory will cause this condition to occur sooner. Pointers to stale memory result when the application does return a chunk of memory to the heap but keeps other pointers to that same memory, which the application no longer owns.

The normalized data library uses the reference counting idiom[5] to overcome these potential problems. Passing a data group from one tool to the next creates a copy of the data group. These copies result in incrementing reference counts to the individual blocks of data contained in the abscissa data and ordinate data object hierarchies. Defining an iterator over this data also constructs a copy of the data, which causes a reference count increase. Destroying the data (by deleting a tool) or deleting iterators causes these copies to be deleted, which then causes the reference counts to be decremented. When the reference counts reach zero it is safe to release the memory back to the heap. At that time we are sure no other references to that memory exist. The normal execution of a tool includes these steps:

- Delete the old output data group of the tool.
- Delete the old input data group of the tool.
- Construct a new input data group for the tool by merging all inputs.
- Execute the tool (delete old iterators and construct new ones).
- Construct a new output data group for the tool.

We defined our iterators such that they do not modify the underlying reference counted data. In cases where tools need to modify the reference counted data (such as pattern filters which modify the tags object in a data set), we use the copy-on-write optimization: operations that would change the data cause a completely new copy of the data to be created and cause reference counts to be adjusted accordingly.

## Case Study

During the design of the normalized data library for the prototype analyzer we made a conscious decision to incur the overhead of normalizing the acquisition data immediately after transferring the data from the HP 16500B mainframe to the prototype analyzer. This happens once per acquisition and before any tools examine the data. We do this immediately to optimize the response time of postacquisition data exploration. For very large acquisitions, however, this normalization time can be considerable.

At the initial release of the prototype analyzer, one customer was making such an acquisition with 1M-byte-deep HP 16555A logic analyzer cards probing an Intel P6 microprocessor and a PCI bus. We found the normalization time to be too large in this case—the acquisition data exceeded 30M bytes. After studying where our code was spending time during the normalization process, we optimized certain sections of code as well as significantly changed the way in which the data driver for that module stored the data in the object hierarchies. After our optimizations, we reduced the normalization time by a factor of ten. More important, we did not need to change a single line of client code (lister, waveform, chart, etc.) to take advantage of this optimization.

## Conclusions

At the introduction of the HP 16505A prototype analyzer we met most of our project goals. We duplicated almost all of the HP 16500B functionality for the HP 16550/4/5/6A, HP 16517A, and HP 16532/3/4A acquisition cards and added several significant features such as pattern filtering and multiple-view, multiple-location data exploration. The ability to write a single dynamically loaded shared library for each analysis and display tool that would work for any of the data retrieved from the acquisition cards was prominent among the reasons we met our goals. We continue to create new analysis and display tools that will help our customers solve their most difficult design problems.

## References

1. G. Booch, *Object-Oriented Design with Applications*, Benjamin Cummings, 1991.
2. *HP 16517A/18A Programmer's Guide*, Hewlett-Packard Part Number 16517-97000.
3. *HP 16550A Programmer's Guide*, Hewlett-Packard Part Number 16550-90902.
4. E. Gamma, et al, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
5. J.O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

▶ Go To Next Article

▶ Go Table of Contents

▶ Go To HP Journal Home Page