

A record/playback method for simulating a device-under-test, with a custom file compression option

by

David W. Bozarth

A design project submitted to

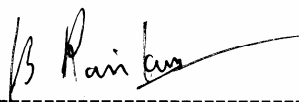
Sonoma State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

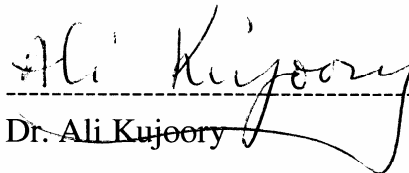
Computer and Engineering Science



Dr. Balasubramanian Ravikumar, Chair



Dr. Jagan Agrawal



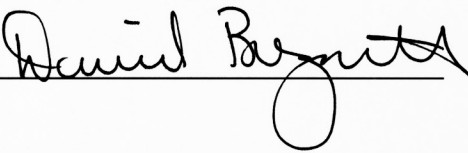
Dr. Ali Kujooory

9 April 2009

AUTHORIZATION FOR REPRODUCTION
OF MASTER'S THESIS/PROJECT

I grant permission for the reproduction of this project in its entirety, without further authorization from me, on the condition that the person or agency requesting reproduction absorb the cost and provide proper acknowledgment of authorship.

9 April 2009

Signature 

David Bozarth

A record/playback method for simulating a device under test, with a custom file compression option

Project by
David Bozarth

ABSTRACT

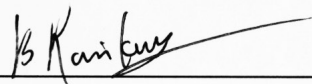
Contemporary test and measurement platforms acquire physical signal samples and process these digitally. The processing software of such an instrument must be carefully designed and tested. Sample error contributions propagate through digital processing stages, introducing uncertainty that can be costly to remedy or work around during performance testing of the instrument software.

The paper describes a method for mitigating this uncertainty. The paper outlines the design of a working software system that employs the method.

The method implies a potential secondary benefit: elimination of costly instrument hardware used in product software development - by moving some activities onto (much less expensive) general-purpose PC platforms that run the instrument processing software.

The author provides a file compression and decompression utility that could reduce the method's resource footprint. The paper presents results indicating a compression factor improved by 14% relative to that obtained with a popular open-source file compression utility.

Chair: Dr. Balasubramanian Ravikumar

Signature 

MSCES Program: Engineering Science Date 29 April 2009

TABLE OF CONTENTS

INTRODUCTION	1
Decoupling setup issues	4
Decoupling the platform	4
Basic design	5
Practical concerns	6
Summary	6
RECORD/PLAYBACK SIMULATOR REQUIREMENTS	7
Sweep	8
The fundamental use case	9
Staying in step with changes of Instrument State	9
Addressing multiple files	10
Excess number of Sweeps	10
Dissimilar States	11
File compression	11
Data tools	12
Summary	12
DESIGN OF THE RECORD/PLAYBACK SIMULATOR SOFTWARE	13
Organization of Sweep objects by SimData	14
File access operations	16
File compression	17
<i>What's available?</i>	17
<i>What's the problem?</i>	19
<i>Algorithmic entropy</i>	20
<i>Custom pre-compression</i>	22
<i>SimZip: compression and decompression utility for Record/Playback files</i>	23
<i>What kind of compression performance is likely?</i>	25
Data tools	26
Summary	27
IMPLEMENTATION NOTES	28
Software technologies	28
Sample stream decoding and encoding	28
Stream buffering	30
ALGORITHM ManageRecStream	30
ALGORITHM ManagePlyStream	32
File, data structure, and container issues	33
File compression	34
RESULTS	36
File compression	41
Summary	42

TABLE OF CONTENTS (continued)

DISCUSSION	43
Role of the Record/Playback method	43
Software testing	43
Model development	44
Applying custom data streams	44
Summary	45
GLOSSARY OF SPECIAL-USAGE TERMS	47
BIBLIOGRAPHY	48

A record/playback method for simulating a device-under-test, with a custom file compression option

David Bozarth (M.S. Candidate, SSU) and Susan Wood (Product Development Engineer)

9 April 2009

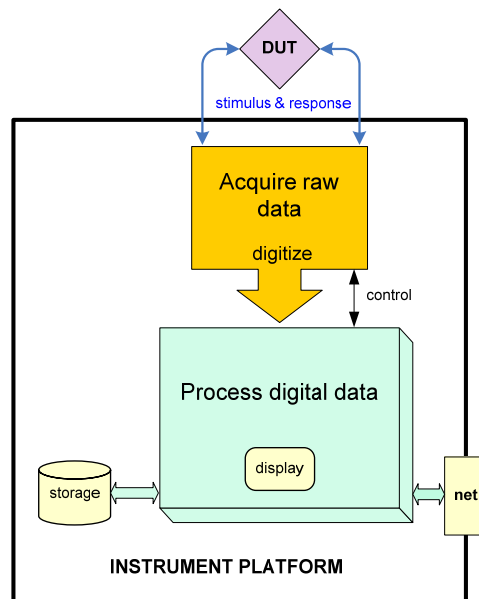
Design project advisory committee

Dr. Bala Ravikumar	Professor of Computer & Engineering Science, SSU
Dr. Jagan Agrawal	Director of Engineering Science, SSU
Dr. Ali Kujooory	Professor of Engineering Science, SSU
Dexter Yamaguchi	Product Development Manager

INTRODUCTION

The architecture of some contemporary test and measurement instrument platforms¹ invites recognition of two primary functional blocks (*Figure 1*):

- Acquire raw physical response/stimulus signal samples from a device or system under test (DUT), and convert these samples to a digital format (orange color in *Figure 1*)².
- Manipulate, output, store processed signal data or characterization information derived from it. These functions are typically encapsulated as an embedded computer (CPU) with graphical user interface (GUI), file storage, and network facilities (aqua color in *Figure 1*)³.



Basic test & measurement setup

Figure 1

¹ For a description and graphic of such instruments connected in a representative setup, see Rowe, M. Aug 1999.

² From the point of view of this paper, a DUT can be anything from a high-performance intelligent device, to a monstrous roll of copper cable – see (a) Bhattacharya, et. al, and (b) Rowe, M. Oct 2004.

³ Pervasive embedded computing in the test environment is illustrated in Rowe, M. Feb 2007. The graphic there shows a number of functional blocks, the majority of which contain embedded processors.

This design partition can play a role in managing cost and performance both for those who use such platforms in field applications⁴, and for those who develop and market these instruments⁵.

Software development for test and measurement products is complicated by the cost and bulk of the instrument platform (*Figure 1*). In order to test new or modified instrument software, each developer must have access to an appropriately fitted physical instrument platform together with the device or system being tested (DUT), cables and accessories. If activity increases over time, this can become expensive and logistically challenging for product development organizations.

Also, with some types of test and measurement applications, calibration or other exacting measurement setups can occupy a significant part of cost and activity budgets^{6,7}. Whether the task is to gather real-time device data in the field or to test software in the development lab, rigorous setup procedures or may be required to ensure correct measurement targeting, or to ensure accuracy or precision of measurements. Where specialized or custom setups involve many physical connections between instrument and DUT, cost or error likelihood associated with such procedures can grow exponentially with incremental expansion of setup functionality⁸.

Though metrology and instrumentation professionals develop powerful, novel approaches for reducing measurement uncertainty⁹ thereby improving instrument performance, little attention has been focused on the specific problem of **testing instrument processing software** in the presence of residual measurement noise and drift.

Susan Wood proposed an elegant method for partially decoupling setup issues - or even the instrument platform itself – from instrument software development and testing (*Figure 1a*). The method relies on partitioning the instrument platform as described above. The data path segment symbolized by the orange arrow (*Figure 2*) becomes a shunt point for extracting and re-injecting raw DUT data (signal sample values from the instrument detector(s)).

⁴ See (a) *Reed, G. 12 May 2008*, and (b) *Rowe, M. 1 Nov 2007*.

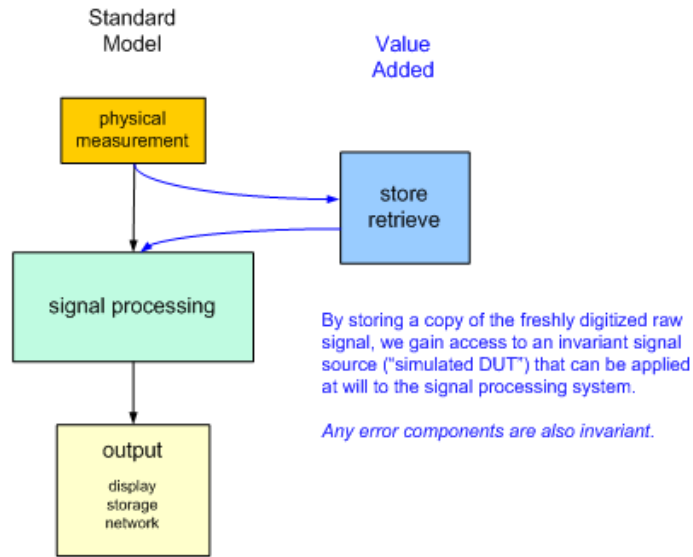
⁵ *Nelson, R. Apr 2001*

⁶ *Lyahou*, pp. 752-757.

⁷ *Delic-Ibukic*, pp. 1175-1179.

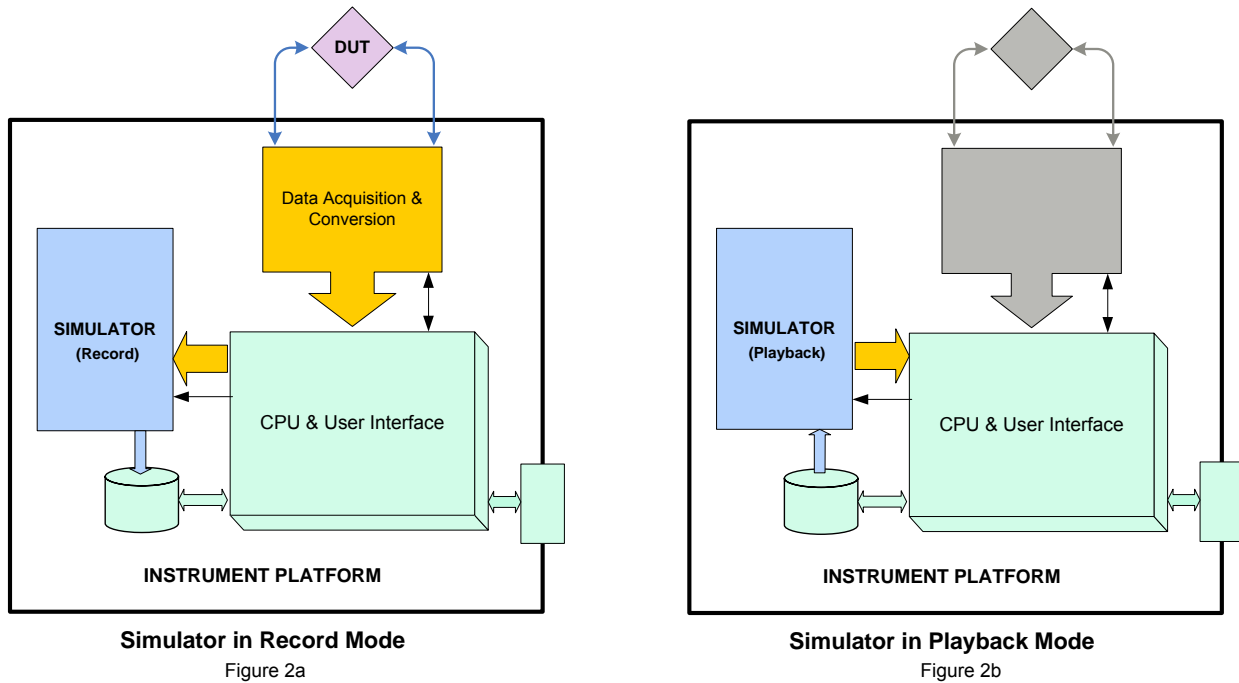
⁸ See (a) *Mayer, J. Nov 2002*, and (b) *Rowe, M. May 2000*.

⁹ See (a) *Maugard, et. al.*, (b) *Jenkins, K.*, and (c) *Farahmand, et. al.*



Method for standardizing error seen by the processing software
Figure 1a

This shunt point is accessed to “record” DUT physical data (stimulus and response signal sample values) by writing data to a disk file¹⁰ (Figure 2a). At a later time, the recorded file data is used to **simulate** (“play-back”) the DUT sample values (Figure 2b)¹¹. The instrument behaves as designed in the presence of these exactly reproduced sample values.



¹⁰ We sometimes use the term “disk” as a convenience to mean “file system”.

¹¹ We use the terms “simulate” and “simulator” for convenience – not to imply mathematical simulation or modeling.

In Figure 2, the thick orange arrows represent DUT signal data as catered by the front-end to the processing unit. The smaller blue arrows represent signal content formatted for file system storage and retrieval. The blue blocks marked “SIMULATOR” should be implemented only in software.

Once recorded, file data can be played back indefinitely and transferred among instruments and network storage as desired. Also, people should be able to edit or generate such recorded data offline, then play-back the data on target instruments as desired. This method could be applied to the generation of signal data sets for test and measurement applications, and to the generation of behavioral models for devices and systems.

Decoupling setup issues

The use of recorded sample data can mitigate effort and uncertainty involved in reproducing the exact setup characteristics from one measurement instance to another.

As one example, suppose a certain instrument software module has two mutually exclusive behaviors “original” and “modified”. Suppose one wants to determine the effect of the modification. Suppose also that the modification must be tested by observing a result that normally *depends on both the software performance and the hardware measurement setup*. It could be helpful to set up and obtain the measurement using “original” behavior, while saving the DUT samples using the Simulator’s Record Mode. Then the “modified” software could be tested using another repetition of *the same signal samples* using the Simulator’s Playback Mode. This tactic eliminates the possibility of hardware calibration or setup errors creeping in between the two test events, and focuses resolving power on the difference between the two software behaviors.

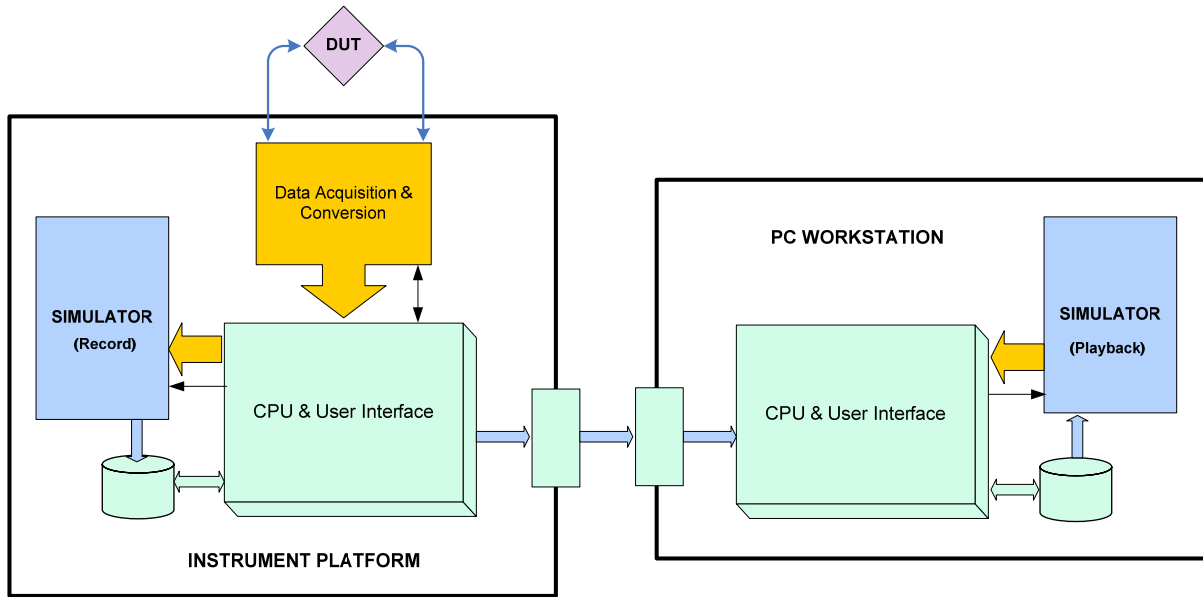
Decoupling the platform

The “greyed-out” areas in Figure 2b are those components that are non-functional during Playback Mode. With the grey components removed, Figure 2b potentially could represent a desktop or laptop PC running the instrument software application together with the Simulator software – a “virtual instrument”¹². Depending on an organization’s software development and testing activities, replacing some fraction of the physical instrument plant with ordinary PC workstations (*Figure 3*) could yield substantial cost savings.

For example, suppose the instrument and Simulator software implementations may be run on a developer’s desktop workstation. Suppose also that development and testing of GUI-only features for the instrument occupies part of an organization’s activity budget. If these activities are separated from development and test activities that require the full instrument platform, then the organization could keep only a small number of physical instruments for essential work (including the recording of DUT sample sessions for later use) and do the remaining work on PC workstations¹³.

¹² This depends on computing platform compatibility with the instrument and Simulator software. Modern test instruments facilitate this potential by running commercial operating system software. See, for example, (a) *TMW:News Briefs Jul 2008*, and (b) *Nelson, R. Apr 2001*

¹³ A 2008 informal inventory in one such organization counted scores of instruments allocated to software development, each representing a cost in the high five figures.



Playback of recorded samples by "Virtual Instrument"

Figure 3

Basic design

The general design of a Record/Playback DUT Simulator should incorporate the following features.

- A representation of DUT signal samples acquired by the target instrument, *including all random and systematic error components associated with the instrument hardware*, can be stored in a suitably formatted disk file **f**.
- The target instrument can access file **f** and transform its contents into an exact reproduction of the instrument behavior that would be expected in response to the original DUT signal samples.
- The format of file **f** enables recognition and representation in some form by any computer processing platform with suitable software, including a "virtual instrument" such as a desktop PC or foreign instrument type.
- In principle, a person or software process could edit the content of file **f** (perhaps using suitable format translation), in order to specifically and determinately modify resulting instrument behavior.

Practical concerns

- In Playback Mode, response to recorded signal data may depend on the instrument's instantaneous state (or the PC software model's representation of instantaneous state), which may not match the instantaneous state of the instrument which originally recorded the data.
- The blue Simulator software block on the right side of Figure 3 may require additional facilities to simulate timing and control signals normally generated by the instrument platform hardware – since in this case the instrument hardware is “missing in action”.
- Neither Record nor Playback operation should place a significant performance load on the instrument relative to its normal processing cycle.
- Sample data contained a disk file and available for Playback could in principle be edited or even generated by external processes.

Each of the above issues will be addressed in some detail below.

Summary

Bozarth developed a Record/Playback Simulator software package that runs on the embedded computer of an instrument used widely for industrial and scientific test and measurement applications. This implementation targets a portable disk file containing information sufficient to drive the target instrument's signal sample processing software in the same manner that physical DUT stimulus and response signal samples normally drive the instrument's processing software. The source of the signal samples is transparent to the target instrument's processing software.

Bozarth also developed a separate package of Record/Playback data analysis tools, including a command-line compression/decompression utility that works with files that the Record/Playback Simulator typically writes and reads.

The following describes the requirements, design, implementation, and verification of the Record/Playback system and its associated software tools.

RECORD/PLAYBACK SIMULATOR REQUIREMENTS

As suggested above, the record/playback simulator software (hereafter denoted “Simulator”) is of modular design, and interacts with the pre-existing software that normally controls the target instrument (hereafter denoted “Instrument”).

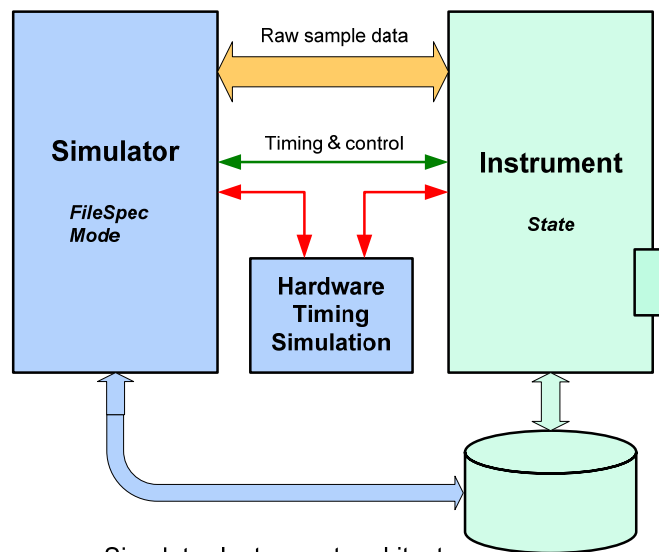
An Instrument may be either a physical instrument or a “virtual instrument” such as a desktop PC (Figure 3). In either case the Instrument’s computing processor runs its normal software integrated with the Simulator. If the Instrument is a virtual instrument, some “Hardware Timing Simulation” software may be needed to simulate the behavior of physical instrument hardware (Figure 4).

The Simulator has a state variable Mode with three states: Off, Record, and Playback.

The Simulator has a state variable FileSpec, from which the instrument can determine a unique read/write file specifier in its attached network file system domain.

The Operator (human or software system) is able to read and modify the states of Mode and FileSpec, and to operate the Instrument in the normally expected ways.

At any specific time the Instrument has a determinate state, a set of variables. Correct response of the Instrument to recorded Samples may depend on the values of a certain well-defined subset of these state variables. We use the term “State” to denote this critical subset of Instrument variables (Figure 4). The Instrument may change State at any time after the Instrument has finished processing a Sweep event, and before the Instrument generates a new Sweep event.



Simulator-Instrument architecture.
Timing & control path for virtual instrument is shown in red.

Figure 4

Each following instance of (i, j, k, n, m) represents a set of non-negative integers.

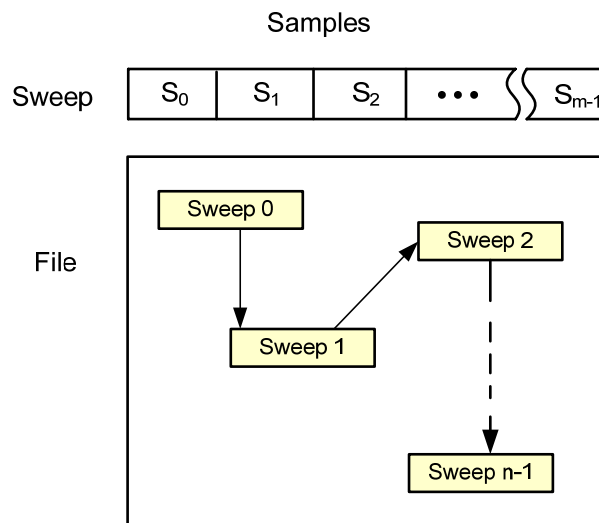
Sweep

Independently of the Simulator, the Instrument is designed to sweep values of an independent variable vector \mathbf{X} over a configured range resulting in a sequence of measurement vectors (hereafter denoted by “Sweep”¹⁴), and to detect values of a dependent variable vector \mathbf{Y} .

Also independently of the Simulator: For each Sweep element i , the Instrument applies to the DUT a set of stimulus signals having known characteristic \mathbf{X}_i , and measures a set of DUT response signals having unknown characteristic \mathbf{Y}_i . The instantaneous detected vectors \mathbf{X}_i and \mathbf{Y}_i are incorporated into a sample vector \mathbf{S}_i (hereafter denoted by “Sample”)¹⁵.

The Simulator detects each instance of the Instrument initiating a sweep event. Note that some Instrument may routinely generate multiple Sweeps during a single sweep event cycle.

When a Sweep occurs and the Simulator is in **Record Mode**, the Simulator collects all the Samples corresponding to the Sweep, and stores them together in a single disk file record (*Figure 5*).



**A Sweep is a sequence of Samples.
A File is a container of Sweeps with sequence information.**

Figure 5

When a Sweep occurs and the Simulator is in **Playback Mode**, the Simulator reads a Sweep record from its disk file (or memory object), and delivers the contained Samples to the instrument software in the same manner as the software normally expects physical DUT Samples. *The Instrument software ignores any physical DUT signals that may be present.*

While the Simulator is in Record Mode, it will continue writing successive Sweeps to the disk file (or file buffer) specified by FileSpec in such a way as to preserve the sequence of original Sweeps

¹⁴ We use the non-capitalized word “sweep” to denote hardware-related activity of the Instrument.

¹⁵ Without loss of generality, a target class of instruments may physically detect *both* the applied stimulus near the point of application to the DUT, and the DUT response near its point of emanation from the DUT.

(Figure 5). While the Simulator is in Playback Mode, it will continue reading successive Sweeps from its disk file (or file buffer) in such a way as to preserve the sequence of the recorded Sweeps.

The fundamental use case

- A physical instrument has State A. Operator addresses this Instrument.
- Operator sets Simulator FileSpec = Z.
- Operator sets Simulator Mode = Record.
- Instrument performs a sequence of n sweep events, processing Samples from DUT.
- Operator sets Simulator Mode = Off.
- ...
- Some Instrument has State A and can access file Z. Operator addresses this Instrument.
- Operator sets Simulator FileSpec = Z.
- Operator sets Simulator Mode = Playback.
- Instrument performs a sequence of n sweep events, processing Samples from file Z.
- Operator sets Simulator Mode = Off.

This is the simplest use case, serving as a basic functional test for the Simulator. No change of Instrument State or Simulator FileSpec occurs, and the number of recorded Sweeps n is the same as the number of playback Sweeps.

Staying in step with changes of Instrument State

- A physical instrument has State A_0 . Operator addresses this Instrument.
- Operator sets Simulator FileSpec = Z.
- Operator sets Simulator Mode = Record.
- FOR j = 0 to (n-1) DO
 - Instrument acquires State A_j
 - Instrument performs a sequence of k_j sweep events, processing Samples from DUT.
- Operator sets Simulator Mode = Off.
- ...
- Some Instrument has State A_0 and can access file Z. Operator addresses this Instrument.
- Operator sets Simulator FileSpec = Z.
- Operator sets Simulator Mode = Playback.
- FOR j = 0 to (n-1) DO
 - Instrument acquires State A_j
 - Instrument performs a sequence of k_j sweep events, processing Samples from file Z.
- Operator sets Simulator Mode = Off.

As with the fundamental case, the playback activity should faithfully reproduce the original behavior of the Instrument while recording Sweeps, since the Instrument State always matches. An example of this use case would be to record and playback a fixed setup or calibration routine, using just one Simulator disk file.

Note that this is merely an extension ($n > 1$) of the fundamental use case ($n = 1$).

Addressing multiple files

- A physical instrument has State A. Operator addresses this Instrument.
- Operator sets Simulator Mode = Record.
- FOR $j = 0$ to $(n-1)$ DO
 - Operator sets Simulator FileSpec = Z_j
 - Instrument performs a sequence of k_j sweep events, processing Samples from DUT.
- Operator sets Simulator Mode = Off.
- ...
- Some Instrument has State A and can access all files Z. Operator addresses this Instrument.
- Operator sets Simulator Mode = Playback.
- FOR $j = 0$ to $(n-1)$ DO
 - Operator sets Simulator FileSpec = Z_j
 - Instrument performs a sequence of k_j sweep events, processing Samples from file Z_j
- Operator sets Simulator Mode = Off.

Again we have extended the fundamental use case, keeping multiple files to store and playback data corresponding to various subroutines.

Note that State changes and FileSpec changes could be used together either in hierarchical or interleaved fashion. We have not tried to list all the possible use case forms.

Similarly, the Operator could address a sequence of multiple Instruments for recording and playback. In order to do so using the model presented in this paper (*Figure 4*), the Operator would need to address each individual Instrument with its own instance of the Simulator software. (A different architecture might provide a single standalone Simulator instance that could connect with various Instruments over a network.)

Whether addressing a single Instrument or multiple Instruments, the Operator should ensure that the Instrument State at the time of playback for each Sweep corresponds with the State that existed at the time of recording that Sweep.

Consider also that for use cases more complex than the fundamental, Simulator activity could be more conveniently managed by a script or software application Operator than by a human Operator.

Excess number of Sweeps

Recall that a single recorded Simulator file contains zero or more Sweeps that can be played back in the same sequence they were recorded. Suppose that while in Playback Mode, the Instrument continues generating sweep events until all the recorded Sweeps in the file have been delivered to the Instrument for processing. Suppose then the Instrument generates another Sweep event while the Simulator is in Playback Mode and its FileSpec remains unchanged – what should the Simulator do?

Two obvious answers are: (1) Do nothing since the end of the file has been reached, or (2) Start over at the beginning of the recorded file's sequence and continue delivering Sweeps to the Instrument.

We believe this should be an implementation decision, perhaps configured in software and modifiable by the Operator to match specific test or measurement applications. In the implementation described below, we opted to have the Simulator cycle through its recorded file until the Operator changes the state of either Mode or FileSpec.

The following use case diverges from those above.

Dissimilar States

If for any reason the Instrument State during Playback differs from the corresponding State during Record, the Simulator should be able to detect this condition and respond gracefully.

For the implementation described below, we reasoned that having the Instrument respond incorrectly or unexpectedly to a recorded Sweep would be undesirable. It appeared that the task of accommodating the Simulator's behavior to various instances of mismatched State, might be better left to future development efforts.

We set up our Simulator to do the following each time the Instrument generates a sweep event:

- Iterate forward from the most recently processed Sweep in the file, seeking the next recorded Sweep that matches the current Instrument State.
- If a matching Sweep is found, then deliver it to the Instrument for processing.
- If no matching Sweep is found, then obtain a default Sweep from a stored configuration, and deliver this to the Instrument for processing.

Note that matching a Sweep to its original state would require storing State information along with each Sweep in the disk file. Refinement of this nature is discussed in later sections on Design and Implementation.

File compression

There is no inherent requirement that Record/Playback files should be compressed; however, some advantages of compressing files in other contexts could apply here – storage efficiency and possibly time performance improvement.

As in other contexts, early attention to some details of design and implementation can help to manage the size of Record/Playback files.¹⁶

¹⁶ See, for example, the inclusion of flagged data fields in our discussion of “File, Data Structure, and Container Issues” under Implementation.

A compression/decompression utility for Record/Playback files should have, at minimum, the following characteristics:

- **lossless** encoding and decoding: the original file must be exactly reproducible from the encoded file. (This follows from the same constraint on the Record/Playback Simulator.)
- **compression performance** sufficient to justify enhanced development and maintenance costs

Also, one would prefer the utility to be **integrated** or closely-coupled with the Record/Playback software, so that no explicit, additional file conversion steps would be needed to store and retrieve Record/Playback data. This implies an additional requirement for negligible **time performance** penalty for both compression and decompression^{17, 18}.

Data tools

It would be helpful to have tools for inspecting and modifying Record/Playback file contents. As mentioned elsewhere in this paper, such tools could be integrated with the Record/Playback software. Another approach, perhaps more applicable to initial development scenarios, would be to maintain an offline package of data manipulation utilities.

Summary

A generic Record/Playback Simulator captures raw Sample data from the Instrument and stores these as sequential Sweep units in a disk file. These Sweeps can later be read from the file and their Samples injected into the processing stream of a physical or virtual Instrument.

Scripts or other software routines can be built to exploit or enhance the capabilities of the Simulator for various test and measurement applications.

¹⁷ See discussions on file access, file compression, and performance in the Implementation section below.

¹⁸ This requirement holds for a closely-coupled (probably stream-based rather than file-based) encoder and decoder. For a more loosely-coupled (possibly batch-oriented) scenario, there might be opportunities to work around any reasonable time performance penalty.

DESIGN OF THE RECORD/PLAYBACK SIMULATOR SOFTWARE

A small set of software object classes partition the following Simulator functionality (*Figure 6*):

- internal state sequencing and interface with the Instrument: **SimControl**
- disk file management: **SimData**
- hardware timing simulation: **SimHW**

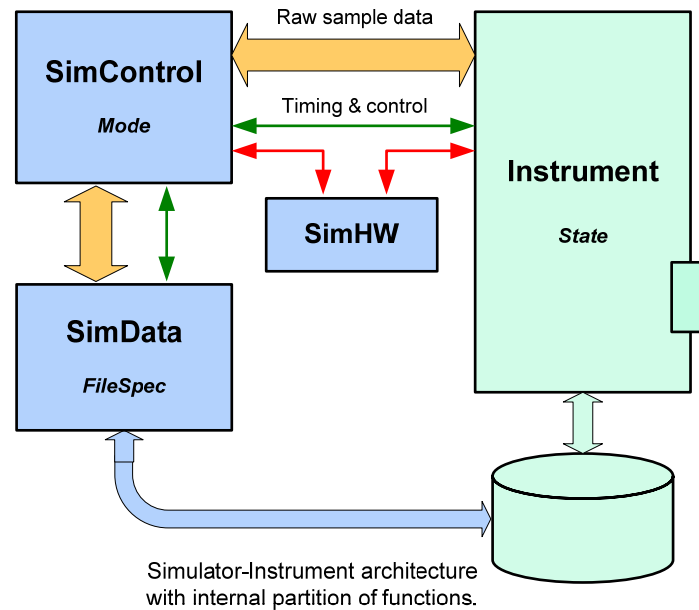


Figure 6

The SimHW software component was already in use supporting a previous project. After some modification, it now performs an expanded role supporting Simulator services for any “virtual Instrument” such as a desktop PC running appropriate software.

The remaining two components (SimControl, SimData) were designed and implemented from the ground up and integrated with the host organization’s Instrument software platform (which includes the SimHW component).

SimControl monitors all DUT sample data acquired by Instrument. When SimControl receives notification from the Instrument that a Sweep event has been triggered, then SimControl issues appropriate notification to SimData, depending on the state of Mode (Record, Playback, or Off).

If in Record Mode, then SimControl forwards sample data from the Instrument to SimData, with notification to write this data to disk. If in Playback Mode, then SimControl signals Instrument to drop any physical DUT sample data it has buffered, and signals SimData to read sample data from the disk. SimControl then forwards the disk data back to Instrument.

For simplicity, the basic design provides for asynchronous timing of these events: Instrument must wait for SimControl to complete its additional duties before proceeding with the normal Instrument

processing cycle. A revised or specialized implementation might attempt a synchronous approach, placing some different constraint on the internal performance of the Simulator while accommodating different performance requirements for the overall system.

SimControl also is tasked with managing stream buffering, multiplexing, and any other data formatting or flow control requirements. (With the present implementation, both stream buffering and multiplexing are in play.)

This implementation's target Instrument employs chunked or bursty rendering of Samples in order to accommodate hardware timing constraints. During the record process, SimControl must buffer these stream partitions and deliver them to SimData in an orderly fashion. Conversely during playback, SimControl must receive a full complement of stream data as supplied from the disk file, and partition it in the manner expected by Instrument. This expectation should be clearly specified by control signals available from Instrument.

An implementation might employ some multiplexing schemes that require decoding of the Instrument's supplied Sample stream in order to isolate separate Sweeps during the record process, and encoding to rebuild the expected Sample stream for return to the Instrument during the Playback process.

In principle, there might be no need for decoding and encoding. The stream of Samples might simply be recorded to the disk file and played back without reformatting. This would make for a simpler design, but doing so would make Sweep data less accessible for external editing or generation by algorithm. With decoding applied before the Record process, each Sweep can be stored in its conceptual form: a distinct sequence of Samples. In this context one can easily envision an array of Sample objects subject to manipulation.

A description of this implementation's stream buffering and encoding mechanisms is provided below in the Implementation section.

Organization of Sweep objects by SimData

The simplest data structure for storing a container of Sweep objects (*Figure 5*) would be an array of Sweeps, equivalent to a two-dimensional array of Samples. In practical terms, it can be useful to maintain some metadata related to each Sweep object. As mentioned above, for example, some information related to the Instrument State should be stored along with each Sweep. In our design, then, at minimum we envision a structure composed of "signal" (Sweep) plus "info" (Sweep metadata). We denote this object "SimSweep". This "signal plus info" organizational paradigm should characterize all Simulator data objects, present and future.

An array is the simplest organization for a SimSweep object container. For our implementation, there was no requirement for anything other than a simple sequence of SimSweep objects.

Suppose, though, that an application could benefit from a different type of container data structure. It is straightforward to imagine a testing scenario involving SimSweep objects, wherein path choices could be made based on prior testing results. Such a scenario efficiently might be implemented with a tree or graph structure of contained SimSweep objects.

With this consideration in mind, a decision was made to formulate the SimData component as an abstract base class from which various SimSweep container classes can be derived, each representing a data structure type suited to a specific situation (*Figure 7*). In fact, the SimSweep class itself is derived from the abstract SimData base class (representing the simplest container of one Sweep). The next-simpler sequence container of SimSweep objects is realized as another SimData-derived class SimSweepSet. Any newly-conceived SimData class is potentially a container for any other SimData class. One might imagine, for example, a “SimQueue” container of SimSweep objects, or a “SimSpecialGraph” container of SimSweepSet objects.

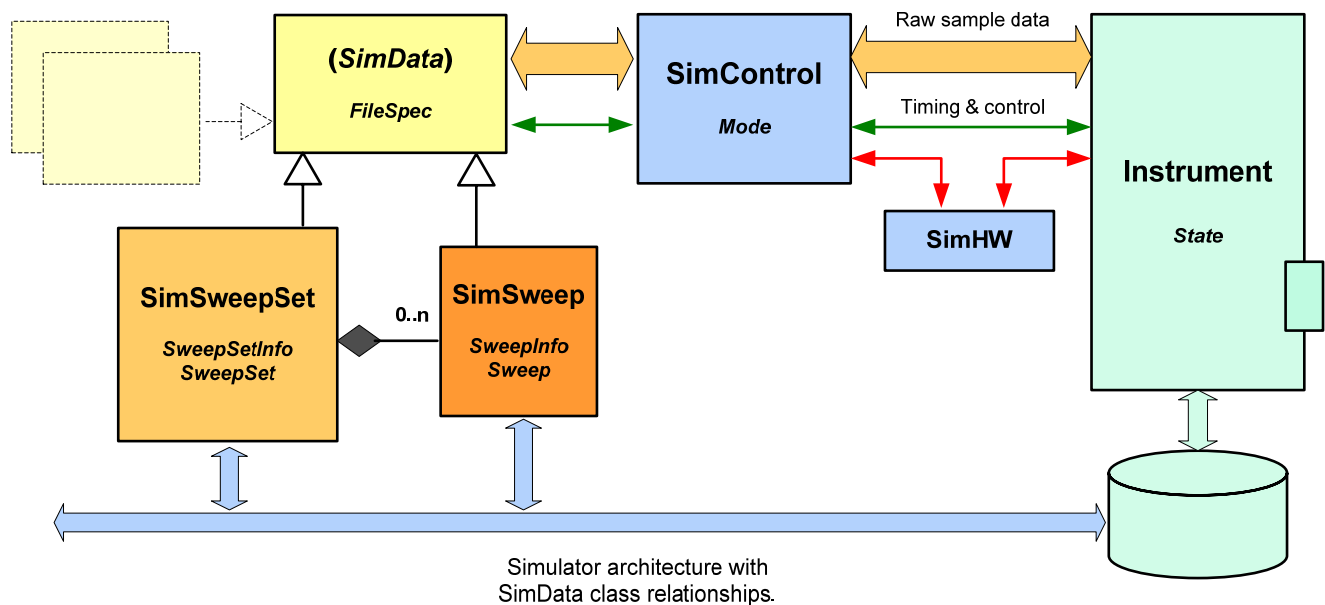


Figure 7

The following structural nomenclature can be a bit confusing, but the “info plus signal” paradigm is meant to apply to each SimData-derived class, including classes SimSweep and SimSweepSet. To clarify, note that each SimSweepSet contains some metadata (SweepSetInfo) and some signal (SweepSet, an array of SimSweep objects). From the point of view of SimSweepSet, each contained SimSweep object is an element of pure signal. From the point of view of SimSweep, each SimSweep object contains both metadata (SweepInfo) and pure signal (Sweep).

This organizational strategy solidifies the relationship between any single Sweep and its own specific metadata. The “info” component of any higher-level SimData-derived object pertains only to its own level of organization. SweepSetInfo, for example, contains no information related to any single Sweep object, but only information related to the SimSweepSet container object.

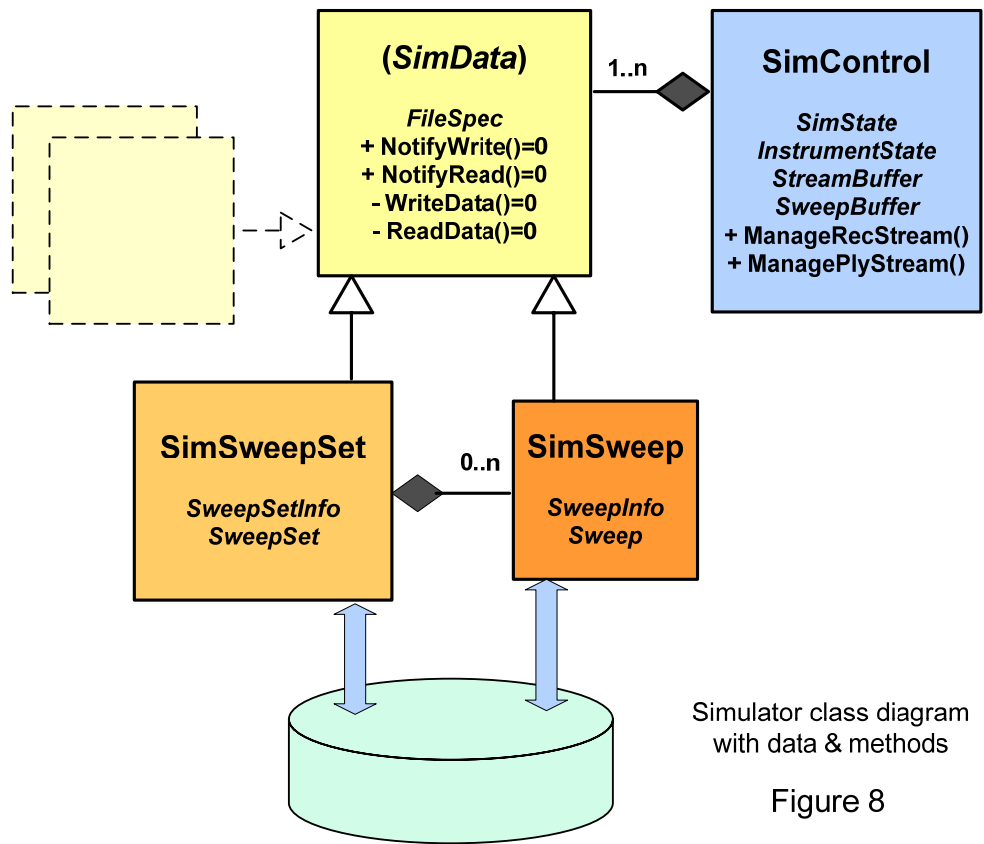
Any SimData container must have a mechanism for iterating among its contained SimData objects, whether for reading, writing, or any other purpose. In a purely object-oriented design, this would be accomplished through use of a separate or contained iterator class specialized for each SimData-derived type. For our implementation, there was no immediate need to develop a separate iterator class for SimSweepSet, but there was a priority on achieving deliverables within a limited time

frame. Sequencing logic for the simple array structure (including special conditioning for seeking matching State and for looping endlessly, as described above under Requirements) was implemented in a straightforward manner.

File access operations

Each SimData-derived class must define methods (Figure 8) enabling each SimData object to write its own content to the specified disk file as a unit, and for reading its own content from the disk file as a unit. Each SimSweep object, for example, is fully capable of writing and reading its own Sweep and SweepInfo content together as a single disk file record representing the SimSweep object.

Similarly, SimSweepSet is fully capable of writing its own content to the disk file specified by FileSpec, and of reading its own content from the same disk file. Since SimSweepSet is designed as a container of SimSweep objects, it may accomplish these tasks by invoking methods of SimSweep successively on each contained object. For example, upon receiving a NotifyWrite() signal from SimControl, a SimSweepSet object could perform housekeeping and invoke its own WriteData() method, which would in turn invoke the NotifyWrite() method of each contained SimSweep object (Figure 8). This scheme leverages code reuse in an object-oriented framework.



The basic design places few constraints on the file access and buffering strategy used by any specific implementation. At minimum, the file I/O scheme must support the following requirements (as portrayed above in the Introduction and Requirements sections):

- ready switching among file targets
- ready switching among Instrument states
- negligible performance penalty on Instrument operation

Included below (in the section on Implementation) is more detail on how our implementation moves SimData content to and from the file system.

Our implementation uses a binary file format for SimData files in order to optimize performance and use of storage. SimData classes might also include methods and data members to enable writing and reading non-binary files, for example comma-separated or XML files that might be used for editing or signal generation.

File compression

A stream-oriented, closely-coupled arrangement with excellent compression performance would be ideal.

A file-oriented implementation could be less useful. If a performance penalty accrues as a result of loose coupling (or if very good compression performance is not achieved), then a real-time service might not be practicable. Employment of a file-oriented service in a live test-and-measurement scenario might require extra work to set up batches of files offline to be compressed after, or decompressed prior to, those files being addressed by the primary Record/Playback software.¹⁹ Also it's messy to have various temporary files being written, read and (presumably) deleted.

For this scenario a file-oriented, loosely-coupled design was fairly simple to develop and (given modest compression performance) introduced little additional performance constraint.

Aesthetic and practical concerns suggest that the design of our file compressor should reflect aspects of the Record/Playback design. Given the centrality of the Sweep object, for example, one might think in terms of a corresponding "SweepCompressor" object capable of transforming the set of bits represented by a Sweep object into an encoded (hopefully smaller) bit set. We'll see why being rigorous about that might pose problems, and we'll consider a working compromise.

What's available?

Given the ready availability of open-source compression utilities with proven performance and reliability, the thought of building a full-blown compression scheme from scratch was of limited entertainment value. Two of these utilities stood out from the pack: one (LZMA) uses a

¹⁹ With a fast, highly-effective, tightly-coupled, streaming scenario, the service should be transparent to the client.

combination of dictionary-based and statistical encoding²⁰; the other (bzip2) prefixes a statistical encoder with a block-mode pre-compression stage²¹.

Dictionary encoders (e.g., LZMA) typically process data in a sequential stream, classifying structural elements already encountered and exploiting these to compress newly-encountered data. Imagine using a spoken language dictionary that initially contains only a few words, but accumulates a new entry for each word you learn. At the outset, this dictionary's usefulness will be limited, but as more material is encountered, it becomes more useful. Similarly, a dictionary encoder will in principle achieve better compression performance with increasing size of its input.²² In practice, the dictionary generally can not be allowed to grow without bound, so compression gain tends to be limited for larger files.

The Burrows-Wheeler Transform (the “b” in “bzip2”) is a sorting algorithm that (when followed in sequence by run-length and move-to-front encoding) is effective in preparing blocks of data for improved compression performance by a statistical encoder. It does so by permuting the content of a given block in a way that, on average, enhances local redundancy.²³ The effectiveness of this process is a function of block length²⁴, which requires a minimum input size and implies better performance with increasing input size.

For both utilities, then, we should expect a strategy that encodes many small chunks of data individually, to provide inferior compression performance. We would prefer a strategy that encodes all the chunks together in one lump. From this perspective, the concept of a “SweepCompressor” object loses attractiveness. Perhaps we should think rather of a “FileCompressor” object.

Table 1 shows a comparison of results obtained with LZMA and bzip2, using a selection of different file types and sizes. The compression factor²⁵ is (original size) / (compressed size); larger numbers are better. We include a text file, and a Microsoft Word file containing mostly JPEG (already compressed) content, as context for our target Record/Playback files.

file type	size, bytes	compression factor	
		lzma	bzip2
ANSI text	59,703	3.469	3.703
Word file with images	668,160	1.120	1.096
Record/Playback	19,508	1.246	1.162
Record/Playback	97,008	1.257	1.198
Record/Playback	1,250,130	1.253	1.215

Table 1. Comparing two popular open-source compressors

²⁰ *Wikipedia:LZMA*

²¹ *Wikipedia:Bzip2*

²² *Salomon*, p. 48

²³ This is an expression of Kolmogorov/algorithmic information content, as distinct from Shannon/entropic information content. *Salomon*, pp. 48-49

²⁴ *Salomon*, pp. 756-760

²⁵ We avoid use of its inverse: compression ratio, which invites confusion when expressed as a percentage.

Two points merit notice: LZMA does better with our Record/Playback files (at least those smaller than 2 MB or so), and neither utility does very well. In fact, neither does much better than the worst that should be expected – i.e., attempting to compress a file that was already compressed.

What’s the problem?

From Shannon information theory we have the quantity **entropy**, an index of the “surprise” information content of a data set. Entropy H is a measure of the “absence of” redundancy R contained in the set:

$$H = H_{\max} - R$$

where H_{\max} is the maximum entropy attainable by the symbol set, and R (a positive number representing a negative amount of redundancy) equates to a positive quantity of entropy. For a data set comprised of n distinct symbols with probability mass function p , H_{\max} is the entropy of a uniformly-distributed set of the same n symbols²⁶:

$$H = \sum_{k=1 \rightarrow n} p_k \log(1 / p_k) = n p \log(1 / p) - R$$

$$H = n (1/n) \log n - R$$

$$H = \log n - R$$

Using base-2 logarithms, H is the expected value of the number of bits required to encode a single symbol. With a uniform distribution, every symbol would require the same number of bits to encode – that’s maximum entropy: a condition we recognize intuitively as perfect “randomness” or “disorder”. The greater R , the more variation exists in the number of bits needed to encode various symbols - and the more compressible is the data set, in principle.²⁷

For the largest Record/Playback file in Table 1, the distribution of byte symbols and the calculated entropy are shown below, in Figure 8a. For this data set,

$$H = 7.37 \text{ bits per symbol (bps)}$$

$$H_{\max} = 8 \text{ bits (one byte) per symbol}$$

This file’s byte redundancy, or “lack of surprise” is manifested in the several byte values that are represented with larger frequency.

Assuming that both encoders process one-byte symbols, what can be learned by comparing this file’s H with its H_{\max} ? If a given file containing redundancy could undergo a process that removed *all* redundancy, this would be equivalent to achieving “perfect” compression. Since H_{\max} characterizes a file of bytes with no redundancy, and H characterizes our file with its redundancy, it seems reasonable that, for this file, the ratio (H_{\max} / H) should have a magnitude comparable to the

²⁶ Haykin, pp. 568-573, and Salomon, pp. 46-47

²⁷ Another way to view this is to consider that R relates to H_{\max} , similarly as the standard deviation of a distribution relates to its mean value.

All three sequences contain the same distribution of integers, so their H and H_{\max} values are all equal. If we estimate compressibility based only on statistical distribution, our estimate would be the same for each sequence.

Sequence (b), though, might be encoded as the following sequence of characters:

```
# 1 2 3 4 5 # 3
```

Where “#” is meant to enclose a string, and the integer following gives the number of repetitions of that string in sequence. The resulting compression factor is 1.875.

Sequence (c) might be encoded as:

```
1 3 , 4 , 5 , 2 , 3
```

Where the first trailing integer 3 represents number of repetitions, and “,” (in the absence of a new trailing integer) means “apply the same number of repetitions”. The resulting compression factor is 1.5.

Thus for two of our sequences, it is possible to construct a rule or algorithm for encoding the sequence that results in substantial compression, independent of statistical distribution. The length of the shortest such algorithm is the **algorithmic entropy** of the sequence²⁸.

So, while statistical compression reduces information redundancy within a sequence (or decorrelates the sequence elements) by exploiting its Shannon entropy²⁹, another type of compression might attempt to decorrelate the same sequence by applying some efficient algorithm for finding a shorter representation for the sequence. Dictionary-based compression (LZMA) does this; so does Burrows-Wheeler transformation followed by move-to-front encoding (bzip2)³⁰.

Now we see why, although calculating the Shannon entropy of our sample file gave a “ballpark” estimate of compression performance, the utilities actually performed somewhat better.

Preferring LZMA as the basis for a custom compression utility, I (Bozarth) sought to characterize a Record/Playback file with respect to its “dictionary-friendliness”. Let us define a dictionary entry as a unique sequence of bytes, and imagine a simplified version of a dictionary-based encoding scheme, which dutifully stores every entry and checks every newly-encountered byte sequence against the dictionary. In this scenario, relatively good compression performance could be obtained from a dictionary stocked with a high proportion of very long entries. This is true because in scanning the input stream, the encoder would very often encounter a long sequence that had already been stored in the dictionary - thus the encoded stream would increase in length by only the information required to reference the stored entry (rather than having to repeat the entire sequence in the encoded stream).

²⁸ *Devine*, p. 85 and *Salomon*, pp. 48-49.

²⁹ Specifically, statistical compression assigns variable-length codes to symbols based on a distribution. Shorter codes are assigned to those symbols with greater occurrence in the distribution, thereby reducing the average length of each encoded symbol.

³⁰ Both compression utilities combine statistical and algorithmic approaches in attempt to achieve a balance of compression performance with time performance, memory utilization, and other factors.

Also, were the input stream to contain many repeated occurrences of already-stored entries, this would enhance compression performance³¹. I collected the native sequence of bytes from the 1.25 MB test file into a simplified dictionary, as described above. I plotted the number of dictionary entries against two variables: length of entry, and number of times the entry was referenced from the input stream. The result is shown in Figure 8b.

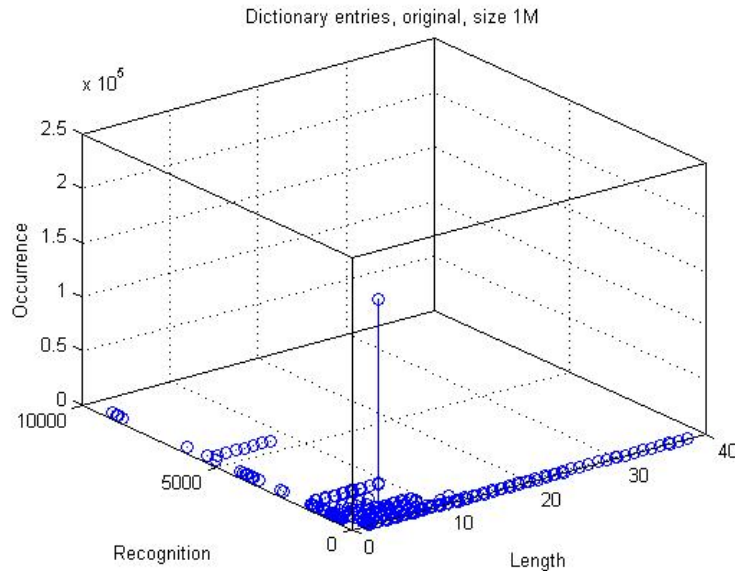


Fig. 8b: Record/Playback file is not especially “dictionary-friendly”.

For optimal compressibility with our prototypical dictionary method, we would like to see a large number of tall data points clustered in the opposite (far) corner – rather than being clustered near the origin and along both axes. As it turns out, though, longer entries are referenced very few times, and those entries that are referenced frequently are very short.

This observation is consistent with Table 1, which indicates that algorithmic compression has helped some, but not much, beyond what is theoretically attainable through entropic compression alone (compression factor = 1.085).

Custom pre-compression

Contents of the test files from Table 1 represent the prominent data features of files expected from the rollout version of the Record/Playback Simulator. I examined the content of the largest test file, looking for redundancy that might be exploitable by a pre-compression front-end stage that would feed the LZMA utility. Because LZMA is quite good at recognizing **local** redundancy³² (as distinct from the simplified, aggregate dictionary scheme described above), grouping redundant byte values closer together in a systematic manner could improve the compression performance of the LZMA

³¹ 40 entries of length 40, referenced 40 times each; would yield better compression than the same entries referenced only 20 times each – because the additional input would have to go into forming new entries, a less efficient process.

³² *Salomon*, pp. 169-171, 224-225.

encoder. Any such pre-compression process must also be reversible (as dictated by the **lossless** requirement).

Using the set of data tools described below, I identified five data components of the file that have distinct character with respect to redundant byte values:

- sweep headers (text and integer metadata specific to each Sweep object)
- some integer values contained in each Sample
- one byte from each single precision floating-point number contained in each Sample
- the other three floating-point bytes
- the file header (a few dozen bytes long)

I segregated and arranged each data component as a two-dimensional array of bytes. For each of the first three components listed above, I identified a short sequence of row or column transformations that would enhance local redundancy in the output stream.

For the fourth component, I did not find a way to enhance local redundancy – no pattern was evident. In fact, testing revealed that LZMA yielded “negative compression” (encoded file larger than the original) for this component. (Accordingly, the LZMA encoding and decoding steps below are not applied to this component.) For the relatively short file header, there is no need for pre-compression, but it is sent through the LZMA encoder.

SimZip: compression and decompression utility for Record/Playback files

Working with these five data components and three reversible custom array transformations, I built an encoder with the following behavior:

- Extract each data component from the source file.
- Apply a corresponding transformation to each component, where applicable.
- Compress each (transformed) component individually, using LZMA.
- Prepend a byte count to each compressed segment.
- Concatenate the segments, and prepend an uncompressed header for the compressed file.

The decoder works in the inverse manner one would expect:

- Read the uncompressed header and note any relevant information about the segments.
- Decatenate the segments (using the byte counts), and decompress them with LZMA.
- Apply each corresponding inverse transformation, where applicable.
- Load the destination data file with the recovered data components.

As mentioned earlier, the design of the encoder/decoder should reflect that of the Record/Playback system. The primary data objects of the Record/Playback system are Sample, Sweep, and SweepSet. In the rollout version, each Record/Playback file object is simply a SweepSet with some additional metadata (a file header). The SimSweepSet object is a linear array of Sweeps. Each Sweep is a linear array of Samples, plus metadata (a sweep header). Each Sample is a data structure composed of floating-point numbers and integers.

Thus the entire file object can be viewed as a file header, a linear array of sweep headers, and a two-dimensional array of Samples. In Figure 8c, the respective pre-compression data components (file header, sweep headers, integers, floating-point byte 3, and floating-point bytes 0-2) are color-coded.

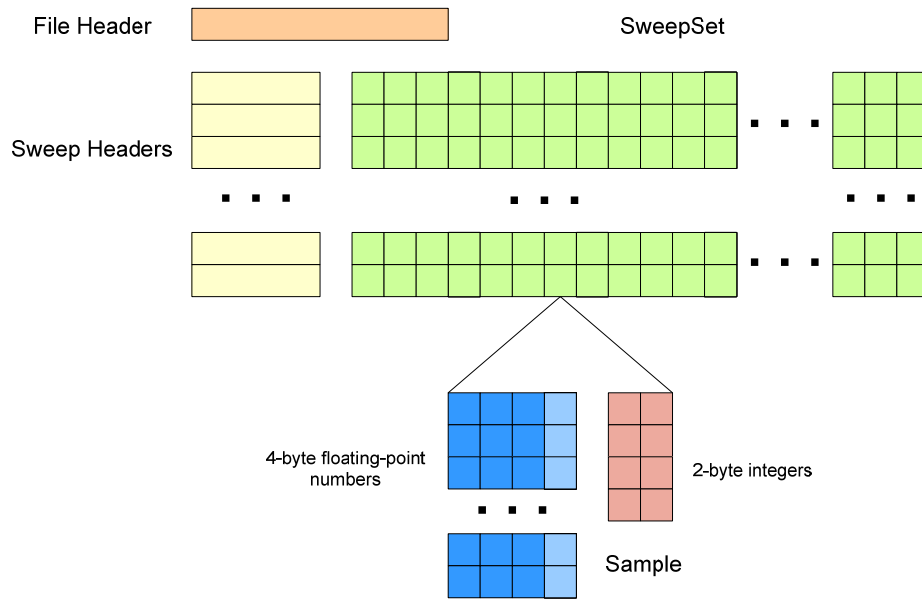


Fig. 8c Data components of Record/Playback file

All this nice symmetry can be compromised by the presence of odd-sized data in any component. A sweep, for example, may contain an arbitrary number of Samples. Sweep headers, moreover, may contain variable-length text strings. Perhaps a more realistic scenario would be represented in Figure 8d:

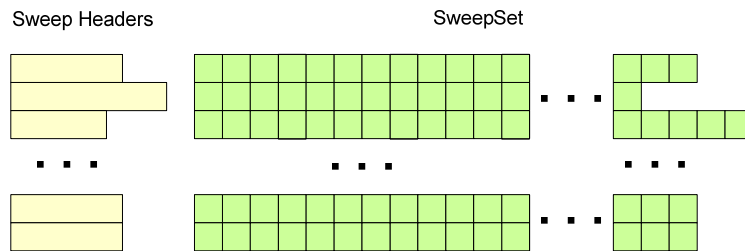


Fig. 8d Jagged data components

Correct handling of such non-rectangular (“jagged”) array components is built into the SimZip encoder/decoder, and is accomplished by means of additional information kept in the pre-compression header, together with alternative versions of the various transform modules. More information is provided in the Implementation section.

The SimZip class is derived from a base class SimFile, which is similar (pun intended) to a SimData-derived class in the Record/Playback software. SimFile and SimZip, though, are C# .NET classes built separately from the Record/Playback Simulator (a project written in unmanaged C++ code), during a secondary development cycle. SimFile has a limited role: it can read a specified SimSweepSet file from disk, and it can write data from memory to a specified file in SimSweepSet format.

In this context, it is worth revisiting our previous discussion about **not mirroring** the exact SimData object relationships among Sweeps. The careful attention to abstract derivation of the SimData class family (also discussed elsewhere in this paper) is not reiterated in SimFile. The benefits of run-time polymorphism and generation of arbitrary new classes that were designed into SimData, are not available with SimFile. A tradeoff was less effort invested in class design.

SimFile contains a member class SimSweep. Like SimData-derived classes, it also contains a set of SimSweep objects.

Unlike SimData, SimFile and its member SimSweep rely fundamentally on stream processing facilities, but (like SimData) not on high-level serialization facilities. Any method that addresses a file first opens a stream connected to that file, then invokes a SimFile stream processing method. Reading and writing of data fields is done through primitive stream methods.

Being derived from SimFile, SimZip can read and write. SimZip methods also conduct all the business of compressing and decompressing.

Both SimFile and SimZip rely on an auxiliary namespace Utility, which contains helper classes Pair, Io, StreamProcessor, FileProcessor, ArrayQuery, ByteArrayManipulator, and ByteArrayConverter³³. Classes in the Utility namespace are restricted to providing generic, low-level functionality not tied to any specific application, and are therefore likely to be re-used in other applications at some later time.

What kind of compression performance is likely?

As mentioned above, the triple-byte signal component (dark blue in Fig. 8c) was found to be not compressible using LZMA. This component occupies 62.2 % of the byte footprint of our 1.25 MB test file. If all other file components could be compressed to zero bytes each, we would have a theoretical maximum compression factor of

$$1 / 0.622 = 1.61$$

This is a common-sense calculation, but is also an expression of Amdahl's Law which bounds the performance improvement to a system with a non-improving component:

$$s \leq 1 / f$$

³³ This class represents wasted effort resulting from my use of the project as an opportunity to develop my first substantial C# application. The .NET Framework has a BitConverter class that fulfills the purpose.

where s is the overall performance multiplier, and f is the proportion of the system's performance determined by the non-performing component³⁴.

Clearly the theoretical maximum is unattainable for this system. By consulting Table 1 and assuming that pre-compression will yield some improvement, we might **estimate 1.3 to 1.5** as the range of enhanced compression factor for this file.

Data tools

These are not so much a designed subsystem as a related set of utility classes. I developed them in a rather *ad hoc* manner, mostly to support my investigations into potential file compression schemes. Here we glance at their organization and note the activities for which they have been useful.

Earlier I mentioned the Utility namespace. These are low-level classes used by the other tool classes. Not really a unifying theme for these, except that they are collections of static methods that are usable by a variety of applications and have no dependencies on custom entities outside the Utility namespace.

The Utility.StreamProcessor class contains a delegate (function object) named Transform. This is the prototype for all the “transform”-styled static methods that provide pre-compression (and inverse pre-compression) for file data components. Some of these methods remain in StreamProcessor, having no dependency on a SimFile object. Others are static methods of the SimZip class.

There is a class named DataTools (also derived from SimFile), but it's really just a collection of methods that I used for testing ideas. These all began life as instance methods of SimFile, prior to that class being pared down to essentials. Included are methods that support some very specific SimFile retrieval, storage, and format conversion processes. The instance method GetSignal, for example, returns a byte array containing a precisely specified subset of the object's signal data.

Static methods ReadSignal and WriteSignal enable format conversions while moving data between stream and byte array representations. One special data format that was very useful in visualizing binary data during my investigative work, is called CsvByte: a comma-separated sequence of zero-padded, 3-digit integers, each representing the value of one byte.

The Model class supports some very basic statistical analysis on general data files. I used it to characterize Record/Playback data files.

There is also a collection of Matlab functions. These were quite helpful in the earlier stages of analyzing test files. Several of these were eventually translated directly into C# code used in the SimZip solution. One example is PickPeriodicValues, a static method of the ByteArrayManipulator class in the Utility namespace. This method (which more accurately might be named “selectively segregate values with periodic indices”) returns a specified subset of bytes from a two-dimensional byte array. Another example is a set of pre-compression transform functions used by SimZip.

³⁴ Parhami, pp. 65-66 and *Wikipedia:Amdahl*.

Summary

The basic design of the Record/Playback Simulator partitions control, timing, and data management among a system of object classes. The SimControl component interfaces with the Instrument component, extracting or injecting streaming Sample data. The SimData component interfaces with the file system. The structure of SimData blends inheritance with composition, supporting reliable storage and retrieval of potentially complex Sweep structures. Ideally, SimControl would agnostically, polymorphically reference an appropriate SimData class object.

The Simulator design seeks to incorporate elements of adaptability and scalability. Coupled with an incremental implementation, the development process took a pragmatic turn toward rapid fulfillment of the host organization's specific requirements.

The SimZip file encoder/decoder utility reflects some design features of the SimData class family, while addressing requirements specific to file compression. Other tools for data analysis were developed as needed, and organized in C# class files. All the source code for these is available for download:

http://www.dbozarth.com/Project_MS/compress/SimFile/index.htm

IMPLEMENTATION NOTES

Previous sections outlined the basic requirements and design features that are generally applicable to Record/Playback Simulator implementations across various families of instrument types and among various test and measurement application areas. This section provides some detail of one such implementation, and describes some development and verification activities that translated design principles into a functioning engineering application. This work was conducted in an industrial laboratory facility during late June through mid-September of 2008. We will also look at a file compression implementation that resulted from work conducted off-site in early 2009.

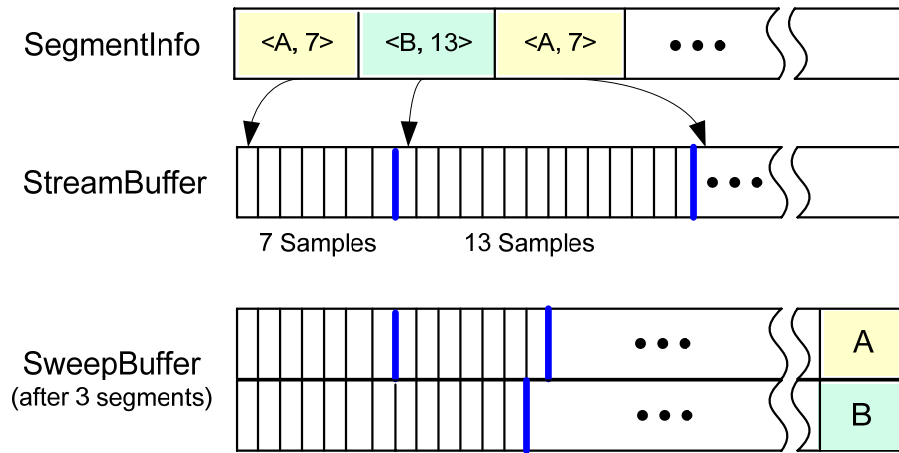
Software technologies

By design the Record/Playback Simulator software assembly is loosely but directly coupled to the Instrument software application - the operational software environment local to one specific Instrument. For our implementation, the programming technologies used were identically those used in other development work for the target Instrument software application.

We built the Record/Playback Simulator software in a Microsoft Windows environment using unmanaged C++ code using Standard Template Library (STL). We used Component Object Model (COM) technology for controlling the Instrument via a programming interface (rather than front-panel controls). The development platform was Microsoft Visual Studio 2005 enhanced by Whole Tomato's Visual Assist X productivity tool, and supported by an IBM Rational ClearCase source code management system. We also took advantage of the group's TWiki collaboration tool.

Sample stream decoding and encoding

The SimControl object (*Figure 8*) contains its own set of state variables (SimState) which includes Mode as discussed above. SimControl also monitors Instrument State and the stream of Samples (StreamBuffer) from the DUT. Let us assume the StreamBuffer contains a sequence of undifferentiated Samples (*Figure 9*), and assume that the Simulator Mode is set to "Record".



Decoding a Sample stream

Figure 9

There are two components of our Instrument State that pertain to data encoding and decoding. For this report we will call one of these components “SweepTag”. Its purpose is to identify Samples from a stream that belong together in the same Sweep. Another Instrument State component we need is an integer value representing the length of a sequence of Samples having the same SweepTag. We call this integer “SegmentSize”. We use the term “SegmentInfo” to denote a single pair $\langle \text{SweepTag}, \text{SegmentSize} \rangle$ (Figure 9).

Recall that a single Instrument sweep event may generate multiple Sweep objects. At the time the Instrument initiates any new sweep event, SimControl reads an array of SegmentInfo records from the Instrument. These will inform SimControl as it processes Sample stream data from the StreamBuffer.

The ManageRecStream() method of the SimControl object (Figure 8, Figure 9) loads the StreamBuffer with Samples from the Instrument as they become available. When a “ready” signal is asserted by Instrument, then ManageRecStream() decodes the content of StreamBuffer (with guidance from the SegmentInfo array) and appends successive Samples to their correct Sweep positions in the receiving SweepBuffer. Subsequently the content of SweepBuffer will be forwarded to SimData with a NotifyWrite() signal.

The software tooling is consistent for both Record and Playback modes, so the converse behavior occurs during Playback. When a “ready” signal is asserted, the ManagePlyStream() method of SimControl generally invokes the NotifyRead() method of SimData, which loads Sweep data from disk into the SweepBuffer³⁵ (Figure 8, Figure 9). Then ManagePlyStream() addresses the SweepBuffer and multiplexes Samples into the StreamBuffer (with guidance from the SegmentInfo array). The StreamBuffer content is then forwarded to Instrument for processing.

³⁵ If for some reason a suitable set of Sweeps is not obtained from disk, a “canned” Sweep will be generated from a preconfiguration.

Stream buffering

Our Instrument employs potentially bursty Sample streaming - in Record Mode, the StreamBuffer receives Samples from Instrument in variable-length chunks which collectively partition the Samples belonging to a single hardware sweep event. SimControl must decode the Samples in its StreamBuffer as they become available and collect these in the SweepBuffer.

In Playback Mode, Instrument (being agnostic to the Simulator) expects to manage Sample bursts as hardware timing conditions of the moment dictate. SimControl must the supply exactly the expected-size burst of Samples to Instrument on each occasion, while keeping track of SweepBuffer indexing. Again, these bursts partition the overall sweep event in progress.

The instantaneous size information needed to process the bursts is contained in Instrument State. The size information is supplied (as indices) to SimControl with the invocation of method ManageRecStream() (when in Record Mode) or method ManagePlyStream() (when in Playback Mode).

During the Record process, ManageRecStream() *implements stream buffering and demultiplexing simultaneously*, signaling NotifyWrite() when a full complement of Samples has been collected in the SweepBuffer. This process is sketched in algorithmic form below.

```
ALGORITHM ManageRecStream (startPos, finitPos, StreamBuffer[])
-----
Method of class SimControl.
-----
startPos, finitPos:
    indices of sliding position of StreamBuffer[] within the overall sweep event.
StreamBuffer[]: a multiplexed sequence of Samples supplied by the Instrument.
StreamBuffer[]: will have size (finitPos - startPos + 1).
-----
Each position of SegInfoArray[] describes a single Sweep segment. It contains:
    - sweepTag    Identifies a single row of SweepBuffer[][] (a distinct Sweep).
    - segSize     Number of buckets required to fill the segment.
-----
eventSize        Number of Samples comprising the overall sweep event.
segCount[]      Number of "filled" Samples per segment.
-----
Using SegInfoArray[]:
    Demultiplex SampleArray[], append Samples to SweepBuffer[][].
When the overall sweep event is complete, send Notifywrite() signal to SimData.
-----
BEGIN
  IF startPos = 0 THEN Clear segCount[], SweepBuffer[][].
  infoNdx ← (index of the first segment with an unfilled segment count)

  FOR EACH Sample in StreamBuffer[] DO
    mySweepTag ← SegInfoArray[infoNdx].sweepTag
    Locate target row[] in SweepBuffer[][], by matching mySweepTag.

    IF (target not found) THEN
      Add a new empty row[] to SweepBuffer[][].
      Make this the target row[].
    END

    Append Sample to the the target row[] in SweepBuffer[][].
    segCount[infoNdx] += 1.

    IF (segCount[infoNdx] = segSize) THEN infoNdx += 1.
  NEXT

  IF (eventSize ≤ finitPos) THEN
    Signal Notifywrite() to SimData object.
    Clear SweepBuffer[][].
  END
END
DONE
```

One detail glossed over here is the modeling of SweepBuffer as an array of Sweep objects. In practical terms, each position of SweepBuffer would be occupied by a Sweep (sequence of Samples) plus some metadata, a structure similar but not identical to the SimSweep object. For this report, it is not necessary to flesh out this distinction, but note that a SweepTag field would be the minimum required metadata for each Sweep in the SweepBuffer.

During the Playback process, ManagePlyStream() ensures the SweepBuffer gets loaded with Samples⁷ at the beginning of an Instrument sweep event, then *implements stream buffering and multiplexing simultaneously*. This process is sketched in algorithmic form below.

The querying of SimState shown below exhibits the reliance of this implementation on state sequencing managed by SimControl (rather than by a separate iterator class associated with SimData). The details are not shown, but SimState would include some flags and/or indices to accomplish this sequencing.

ALGORITHM ManagePlyStream (startPos, finitPos, StreamBuffer[])

Method of class SimControl.

startPos, finitPos:
indices of sliding position of StreamBuffer[] within the overall sweep event.
StreamBuffer[]: a multiplexed sequence of Samples built by this routine.
StreamBuffer[]: will have size (finitPos - startPos + 1).

Each position of SegInfoArray[] describes a single Sweep segment. It contains:
- sweepTag Identifies a single row of SweepBuffer[][] (a distinct Sweep).
- segSize Number of buckets required to fill the segment.

infoSize Number of positions in SegInfoArray[]
eventSize Number of Samples comprising the overall sweep event.
sweepNdx Current row (Sweep) index of SweepBuffer[][]
SampleNdx[] Current column (Sample) index for each SweepBuffer[][] row[]
TmpBuffer[] Temporary buffer for Samples.

If startPos is 0, then load SweepBuffer[][] with Sweep record(s).
Locate buffered Sweep record(s) according to startPos, finitPos, and State.
Using SegInfoArray[], multiplex Samples onto StreamBuffer[].

```
BEGIN
  IF (startPos = 0) THEN
    Clear TmpBuffer[].

    IF (SimState shows need to initialize buffers) THEN
      Clear SweepBuffer[][], sweepNdx, SampleNdx[].

      readFlag ← (Does SimState indicate a need to read from disk?)

      IF (readFlag) THEN
        Send NotifyRead() signal to SimData.
      END

      IF ( NOT(readFlag) OR (NotifyRead() did not succeed) ) THEN
        Load SweepBuffer[][] with "canned" Samples by invoking special configuration.
      END

    END

    Zero all SampleNdx[].

    FOR EACH infoNdx up to infoSize DO
      mySweepTag ← SegInfoArray[infoNdx].sweepTag

      Start with sweepNdx, find j:
      j ← ( index of next unused SweepBuffer[][] row[] having (sweepTag = mySweepTag) )

      IF (matching sweepTag was found) THEN (mySweep[] ← SweepBuffer[j])
        SampleNdx[j] ← 0
        Clear TmpBuffer.

      IF NOT(matching sweepTag was found) THEN
        Load SweepBuffer[][] with "canned" Samples by invoking special configuration.
      END

      limit ← SegInfoArray[infoNdx].segSize

      FOR EACH count up to limit DO
        mySample ← mySweep[SampleNdx[j]]
        Append mySample to TmpBuffer[]
        SampleNdx[j] += 1
      NEXT

    NEXT
  END

  FOR i = startPos to finitPos DO
    IF TmpBuffer.size < i THEN
      Append new empty Sample to StreamBuffer[]
    ELSE
      Append TmpBuffer[i] to StreamBuffer[]
    END
  NEXT

  IF eventSize <= finitPos THEN
    sweepNdx ← j
    Clear SampleNdx[]
    Clear SegInfoArray[]
    Clear TmpBuffer[]
  END
END
DONE
```

File, data structure, and container issues

The binary file format for SimData-built files conserves both storage space and file system access time, but the division of labor between SimSweep and SimSweepSet entails some inefficiency. (This also could apply to higher-order SimData container classes.) Since each individual SimData-derived object comes with its own “info” component, there could be repetitive data stored or retrieved during each file system access operation.

We mitigate this type of inefficiency by including some flags for each SimData class that tell the data engine whether or not to write certain “info” components to the disk file.

One such write-flagged “info” component was conceived as a vector of integers “LocationIndex”, which is declared as a protected member of the abstract base class SimData. (As such, it is by definition a member of every SimData-derived object.) This member is perhaps unique in our implementation by seeming to have no practical utility. It actually is dedicated for the benefit of any higher-order SimData container that may be developed in the future.

The purpose of LocationIndex is to uniquely identify each elemental data unit belonging to a SimData container. Each SimSweep object in a SimSweepSet container, for example, has its LocationIndex equal to the object’s corresponding index of SweepSet (the “signal” array component of SimSweepSet) (*Figure 8*). The SimSweepSet container itself has an empty LocationIndex.

The operative conjecture is that a vector of integers, suitably encoded, should be capable of uniquely identifying any specific elemental component of an *arbitrary* data structure. The LocationIndex would identify, for example, each node of a higher-order SimData container such as a graph, tree, or other. The highest-order SimData container in such a schema should have an empty LocationIndex – since it would not be itself a member of any SimData container.

Why maintain a LocationIndex for the elements of a SimData container? An iterator object would need to map the internal container structure. Similarly insert, modify, and delete operations would need to know where each element should fit within the container’s data structure. For example, there could be some application that builds new SimData elements and writes them in some specific manner to a SimData container. Or, some application might read SimData elements from a container, process these in some manner, and write them back a SimData container in a manner dependent upon their original container location. In this case, the location information for each data element would need to be read from disk.

Other “flagged” data members in our implementation are the primary and alternate file specifiers. The alternate specifier is intended for use in converting a SimData container file from its native binary format to an alternate format such as comma-delimited text, or XML. A Boolean flag member of each SimData object is used to indicate whether the primary or alternate format is currently being addressed. Other flags indicate whether the primary and alternate file specifiers should be written to disk. SimSweep objects in our implementation should normally have no need to write or read any file specifier, since the container knows the correct file name, and efficiently can provide its contained objects with any needed data. This is an example of using a Boolean flag to offset inefficiency by conditioning file system writes.

File compression

In the Design section we overviewed the behavior of Compress and Decompress instance methods of SimZip. Specifically, we mentioned that each file component is assigned a unique pre-compression data transformation that enhances local redundancy, and that a complementary unique transformation is applied during post-decompression to restore the original structure of the component.

Here we'll look at SimZip's way of managing jagged array data. This scheme involves two metadata headers that get prepended to the compressed file by Compress, and that get read from the compressed file by Decompress.³⁶

When the Compress method is invoked, the SimZip object inspects its own data to determine whether any non-rectangular (jagged) array content exists. There are two distinct ways that a SimZip object can have jagged content. One is in the sweep headers; the other is in the signal data. During pre-compression and post-decompression operations of SimZip, the sweep headers are represented in an homogeneous array. Similarly, the several components of signal data are represented in homogeneous arrays. If any such array contains a row that differs in length from another row in the same array, then a jagged array condition exists. Such conditions cause problems for pre-compression and post-decompression. One reason for this is that array transpose operations are used to compress and decompress some file components. Special processing is used when jagged array conditions are detected.

The sweep headers consist partially of variable-length strings. Some of these fields are optional for Record/Playback operation, and may be empty strings. Other fields may be non-empty but identical across all sweeps, for a given Record/Playback file. If corresponding fields have variable length and that variation is cancelled out in the aggregate by other length variations, then compression performance may suffer, but no jagged array detection and special processing will occur.

The structure and size of each Sample element is invariant, but any Sweep may contain any number of Samples. An array of Sweeps, then, will be non-rectangular if any contained Sweep has a different number of Samples than another contained Sweep.

If a jagged array of sweep headers is detected, this condition is flagged by Compress. Similarly, if a jagged signal data array is detected, this condition is flagged. Next, the flag value for each condition is written to the new disk file as part of its initial, uncompressed VersionHeader. Then, if the flag for jagged signal data is true, a special array of integers is written to a secondary, compressed DataSizeHeader. This array gives, for each Sweep, the difference in number-of-samples between that Sweep and the smallest number-of-samples over all Sweeps³⁷.

³⁶ Elsewhere in this paper we reference a "file header". This is part of the structure of the standard Record/Playback file. The headers discussed in this section relate to file compression. To avoid confusion, I will not refer to these as "file headers".

³⁷ In Figure 8d, the second row of the SweepSet contains the apparent minimum number of Samples. This number would be the common row size of a rectangular sub-component of this SweepSet. All other rows would contain excess Samples.

If the flag for jagged sweep headers is true, then the pre-compression transformation for sweep headers consists of a byte-for-byte copy (no transformation). Otherwise, a special transformation is performed that involves an array transposition.

If the flag for jagged signal data is true, then during pre-compression each signal data component gets separated into two sub-components: a rectangular sub-component, and a “leftover” sub-component. The rectangular subcomponent is formed using the minimum number of Samples across all Sweeps. The rectangular subcomponent gets transformed in the standard manner unique to that component. The “leftover” subcomponent, consisting of all samples in excess of the minimum number, forms a linear array. The DataSizeHeader partitions this linear array, keeping track of the number of samples that belong to each Sweep. The “leftover” subcomponent gets appended to the end of the byte stream produced from the rectangular transformation.

During post-decompression, the correct set of inverse transformations gets invoked depending on the state of the two flags in VersionHeader, and the content of DataSizeHeader.

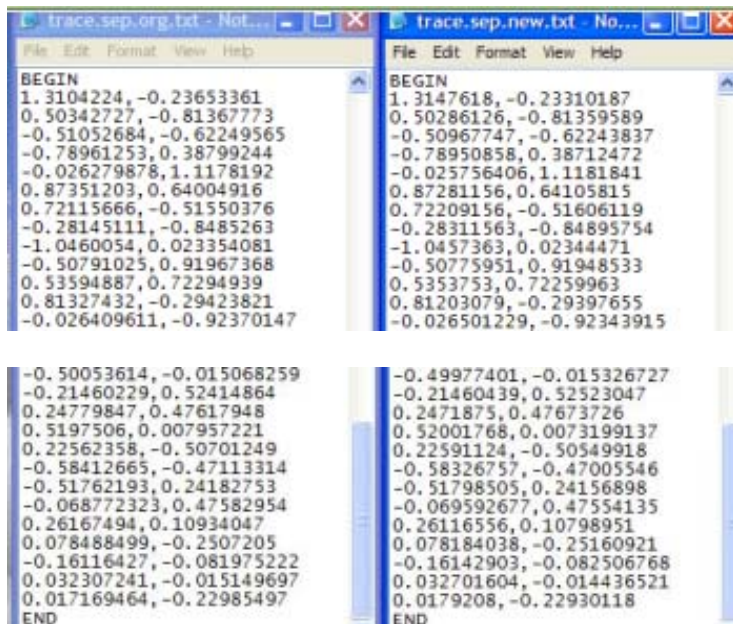
RESULTS

Functional verification of the Record/Playback Simulator is conceptually simple. One only needs to verify that the Sample stream obtained from disk in Playback Mode is identical with the corresponding Sample stream originally rendered by the Instrument in Record Mode. One should also verify that this identity remains consistent over a range of measurement scenarios and expected operational conditions.

We ran many unit tests during development to determine this identity between Playback and Record Sample values. With our Instrument, though, the detected Sample values are normally processed by the Instrument using a mathematical algorithm, and the resulting signal is made available to the User. (In other words, the quantity of real-world interest is the processed signal, not the Samples³⁸.)

An approach to systematic testing that was both simple and meaningful, then, was to place the Simulator in Record Mode and run some arbitrary scripted procedure to exercise the Instrument, while obtaining also a text or graphical representation of the output signal associated with the resulting set of Samples. Then, we place the Simulator in Playback Mode and run the same procedure, again obtaining a separate text or graphical signal representation. If some difference can be detected between the two similar representations, then an error has occurred in the process.

Below is an image of the *beginning* and of the *end* of a pair of lengthy text files that were obtained with one of our Instrument's standard output options. The window on the left displays signal information derived from one run of a certain test procedure with the Simulator in Record Mode. The window on the right shows the signal derived from a second run, moments later, of the same procedure with the Simulator also in Record Mode. Each pair of numbers was derived from a single Sample. Note that the two runs produce similar, but not identical results. For this particular Instrument, differences of this magnitude can be accounted for by random measurement noise.



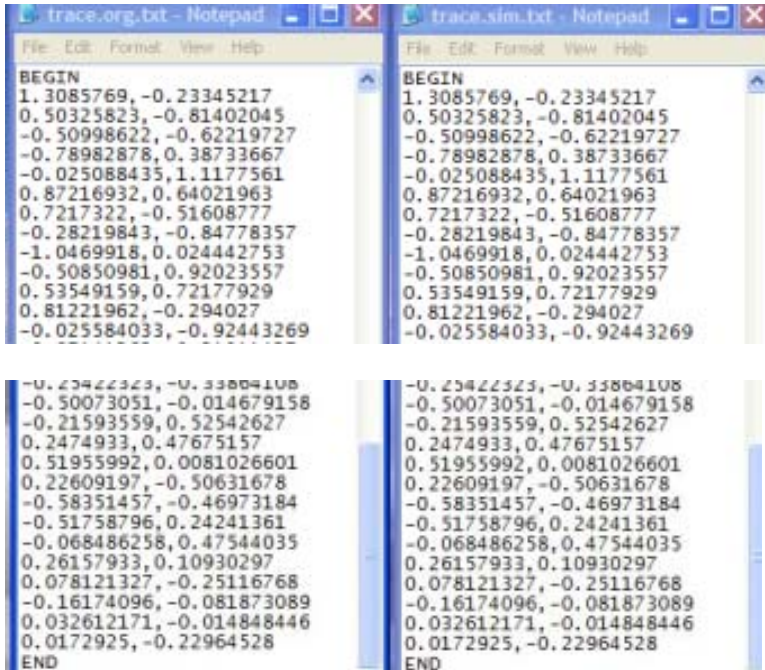
Effect of noise on derived signal.

Figure 10

← Missing file data was visually inspected.

³⁸ Yet the utility and power of the Record/Playback Simulator method relies on working directly with the fundamental Samples, rather than working with the processed signal.

Below is a similar comparison of text files. On the left is a signal obtained with the Simulator in Record Mode. On the right is the signal derived from Playback of the recorded Samples. Note that the two runs produce identical results. This indicates that the Instrument application software has processed exactly the same Samples during two runs of the same procedure. There is *no noise and no systematic error* in the second run of the procedure, relative to the first run.



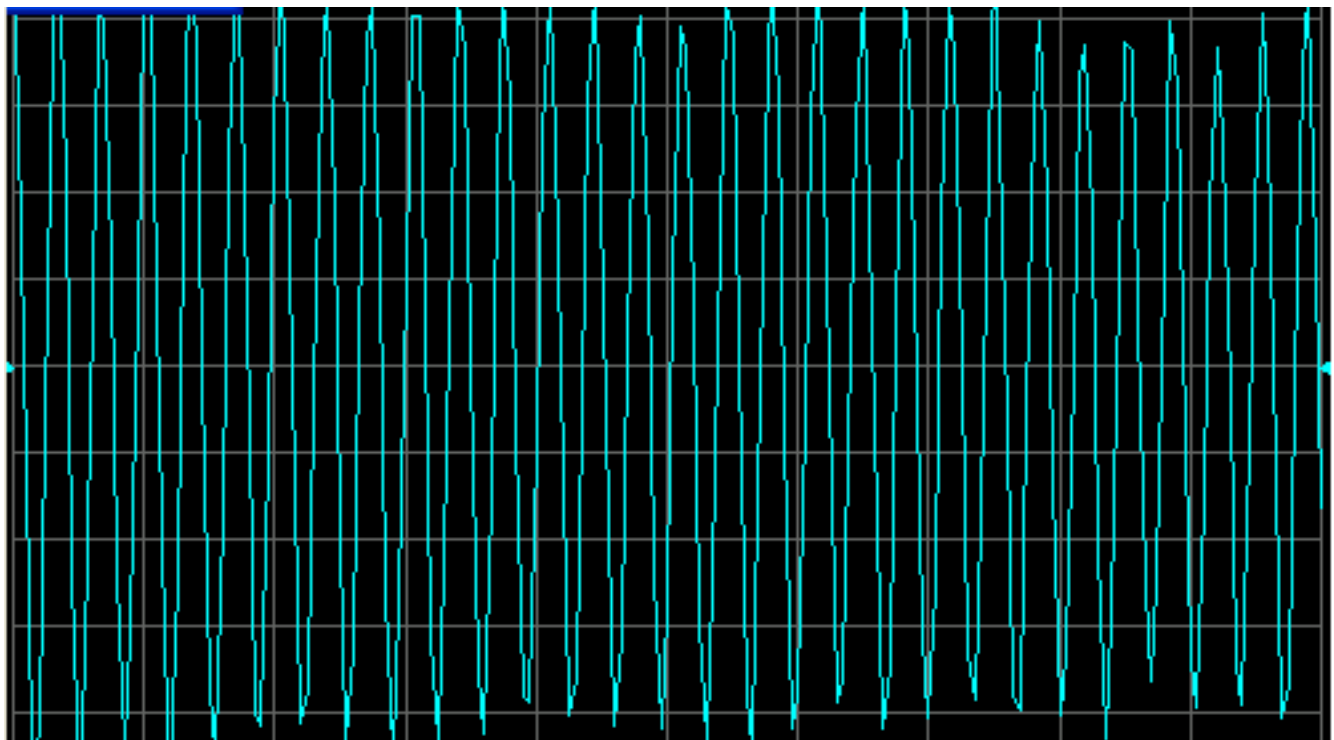
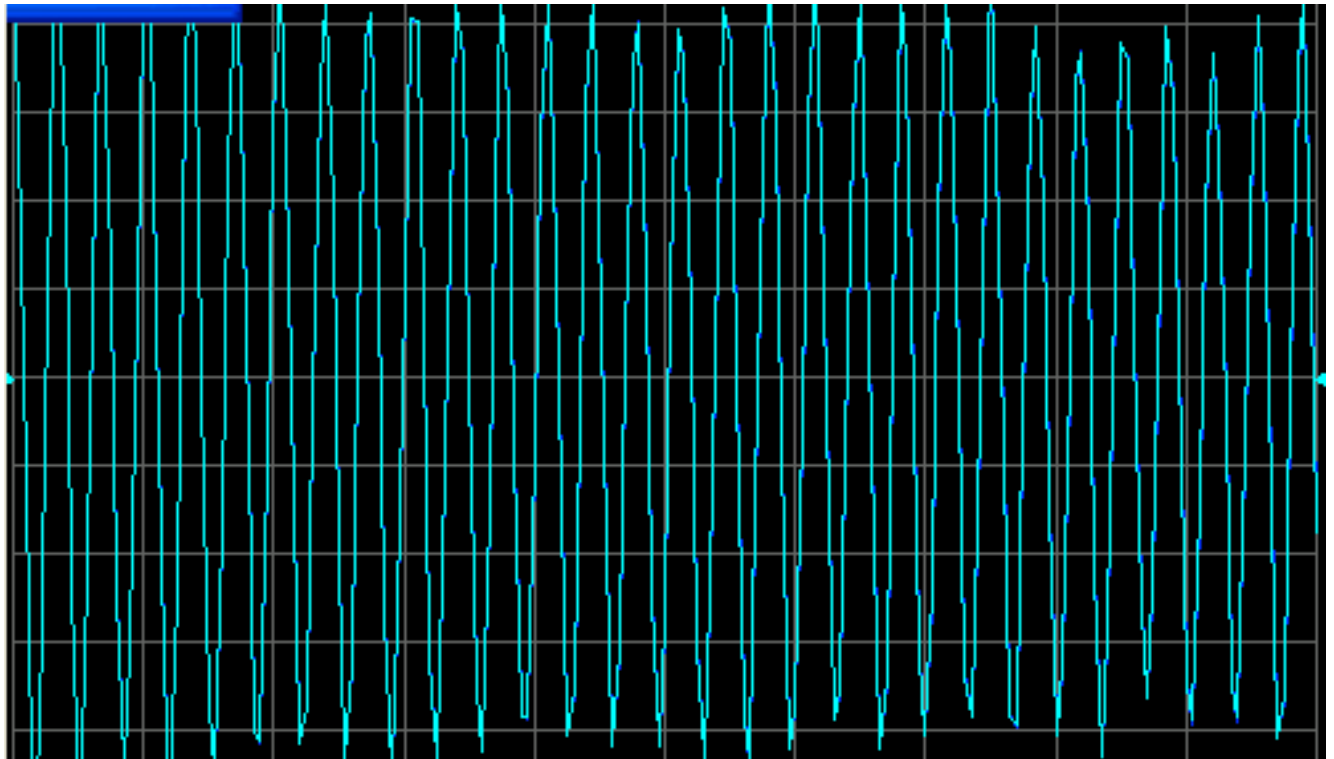
Signal derived from Record and from Playback.

Figure 11

← Missing file data was visually inspected.

Another perspective from our testing is shown below (*Figure 12*). We see in the top image the graphical representation of a DUT-derived signal processed through our Instrument. Superimposed on this is a second, separate signal obtained moments later, but under the same conditions. The first signal is rendered in a dark blue color, and the “prickly” aspect visible in the image results from bits of dark blue color “peeking out from behind” the second signal which is rendered in light blue. As in the previous scenario, the small differences represent random noise.

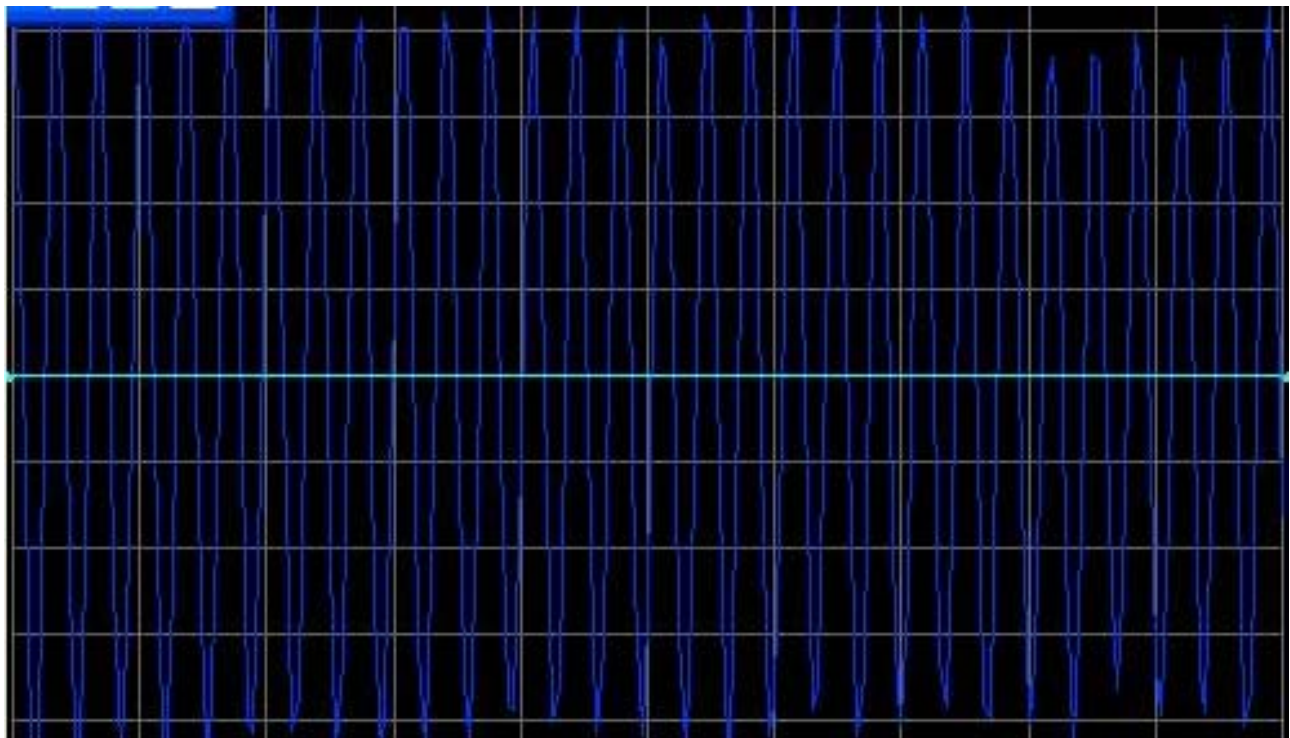
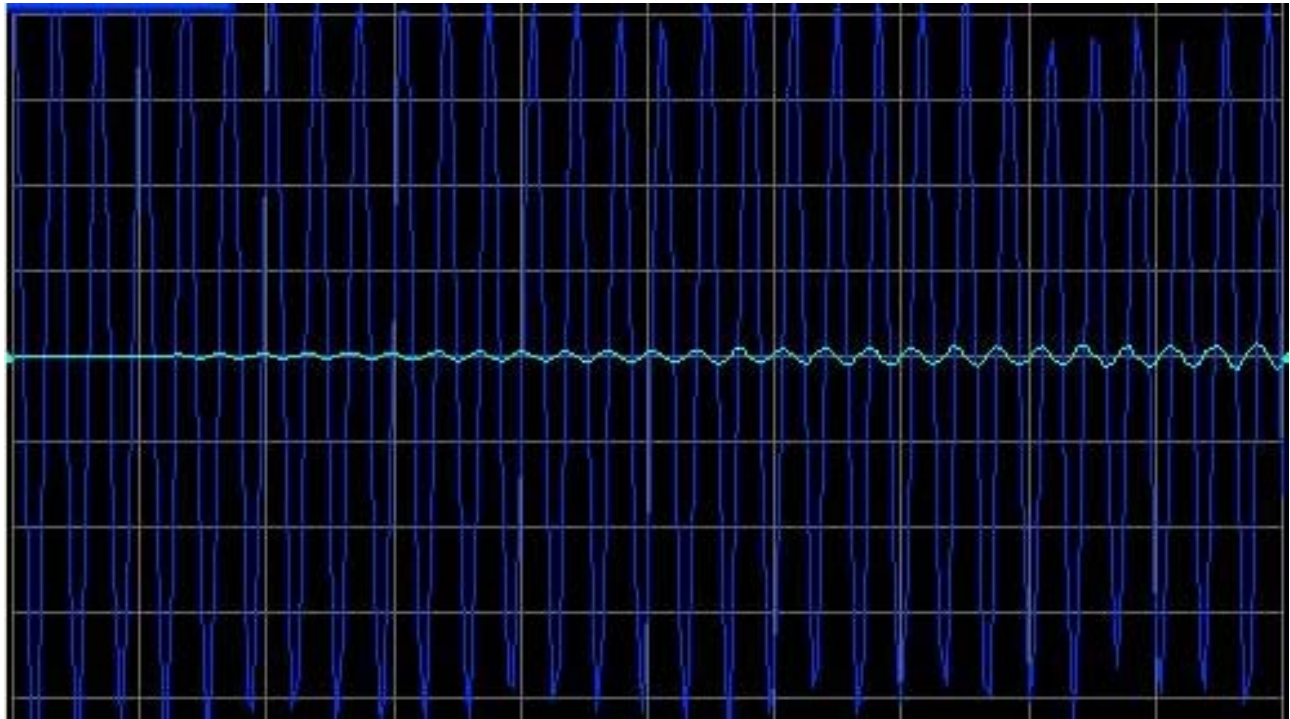
The bottom image is constructed similarly using dark blue for a DUT-derived signal that was processed through the Instrument and also recorded to a disk file by the Simulator. The corresponding Playback signal is superimposed using a light-blue trace. Note that in the bottom image, the dark blue trace is nowhere visible. The superimposition appears to be perfect, indicating that the two signals are essentially the same.



Two separate signals superimposed (top); Record & Playback signals superimposed (bottom)

Figure 12

One may need to closely examine and compare the previous two images (*Figure 12*) in order to recognize the difference between them. Another way to become convinced is to redefine the second, light-blue trace in each image as the arithmetic *difference* between the two signals.

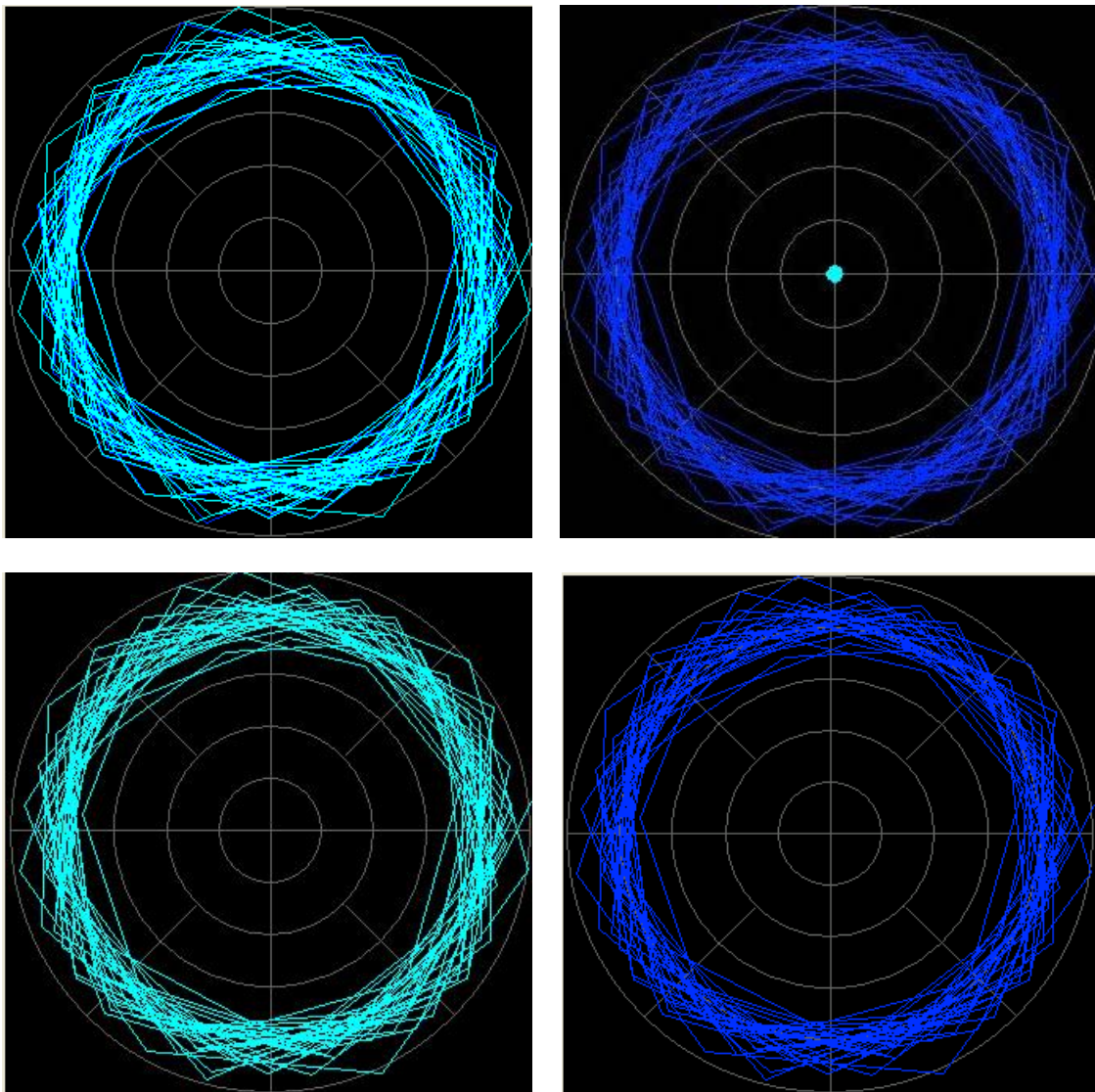


Signal and difference signal (top); difference of Playback relative to Record signal (bottom)

Figure 13

The above pair of images (*Figure 13*) display the same events as the previous pair (*Figure 12*). The only difference is that the light-blue trace in the second pair shows a difference signal. In the top image (*Figure 13*), we see that the two separately-obtained DUT signals diverge increasingly from left to right. In the bottom image (*Figure 13*), we see a flat line at the zero level, indicating no difference between the Playback signal and its corresponding Record signal.

Below is a similar comparison using a polar representation of a complex-valued signal. Each image in the top row superimposes two signals differing is only by noise. On the right the difference signal has a small finite magnitude. Each image in the bottom row superimposes a Playback signal with its corresponding Record signal. On the right, the difference is not visible – if it were visible, it would occupy only a single pixel at the center of the display.



**Two separate signals superimposed (top); Record & Playback signals superimposed (bottom).
In each case, on the right is shown a difference signal in light blue color.**

Figure 14

In testing the performance of our implementation of the Record/Playback Simulator, we used a variety of recorded file sizes and scripted procedures to emulate various combinations of the fundamental use cases outlined in the Requirements section above. Once the mechanism for recording and playing back Sweeps was determined to work correctly for these scenarios, the basic development work was deemed complete for this implementation.

This is the only implementation of the Record/Playback Simulator within the scope of this project. Due to project resource constraints, some desirable objectives were not pursued:

- No iterator class was defined for SimData or its derivatives. This is not expected to impact performance of the existing package.
- No SimData instance methods exist for converting among the native binary file and other potentially useful file formats such as comma-delimited text, XML, etc. The host organization currently has no defined use case for this capability. (See Data Tools subsections in this paper for API-level data extraction, manipulation, and format conversion utilities.)
- The Record/Playback Simulator package was not fully integrated into the host organization’s normal business activities. That task may be accomplished by others at some later time.

File compression

On the 1.25 MB Record/Playback test file, the SimZip utility improved the compression factor by about 14% relative to that of the basic LZMA utility.

Size, bytes	File name	Compress factor	Comment
871,919	original.cim	1.43	SimZip (pre-compression + LZMA)
997,397	original.lzma	1.25	standard LZMA compression
1,250,130	original.sim	1.00	uncompressed file

Figure 8d - **SimZip improved compression of a Record/Playback file**

The file was recorded during a session while the physical instrument was continuously triggering sweeps, but the Instrument State and signal inputs changed rarely. This yielded a large proportion of repeated sweeps, creating a file that is quite amenable to the pre-compressor’s bag of tricks. In the Design section of this paper, we estimated a compression factor in the range **1.3 to 1.5** for the SimZip utility. It seems likely that the performance reported here represents the higher end of what can be realized with this first incarnation of SimZip.

Another file characteristic that could degrade compression performance is the presence of odd-sized sweep headers or diverse sweep sizes. SimZip will encode and decode such files successfully, but with some variable cost.

No effort was made to optimize the time performance of SimZip. Encoding is slow; decoding is slower. Using my vintage-2003 desktop PC with AMD Athlon processor and 768 MB RAM, encoding followed immediately by decoding of the 1.25 MB file takes about 35 seconds³⁹. It is likely that future optimization can cut this performance penalty by a factor of 2 to 10.

It should be noted also (while comparing SimZip to basic LZMA) that there is virtually no chance of picking a random file of any type and having SimZip compress it successfully. SimZip works only on the rollout version of Record/Playback Simulator files.

Simulator files have version labeling in the file header. The SimZip encoder does not check this label explicitly, but it checks the values of certain data fields against their expected format, and raises a “bad format” exception if anything is found to be amiss. It also places SimZip versioning information in the encoded file’s header. The SimZip decoder checks this version information and also verifies the format of encountered fields, raising a “bad format” exception as needed.

Summary

Our implementation of the Record/Playback Simulator faithfully reproduces recorded Samples. The method can be a boon for any software-driven Instrument scenario that relies on consistently precise Sample input.

SimZip, a custom file compression utility for Simulator files, has been carefully tested and is provided with the intent of being production-ready. The time performance limitation of SimZip 1.0 should be noted, though.

³⁹ With the same file on the same hardware, LZMA alone runs the same test in about 5 seconds.

DISCUSSION

With the 21st century well underway, our global civilization engages the opportunity of crisis on multiple fronts. Enmeshed challenges of climate change, energy production and conservation, economic upheaval and realignment, health and environmental stewardship place the significance of test and measurement technologies firmly on a nondecreasing trajectory with time.

Today's test and measurement systems, moreover, represent the convergence of engineering, physical, and information sciences. A common thread among these fields is uncompromising interdependence within the whole. The observable world is analog in nature; the observed world is digital.

In test and measurement, software technologies organize activity similarly as confidence networks in the financial markets, neuronal patterns in the animal brain, and molecular circuits in systems biology. The burgeoning adolescences of embedded computing, digital signal processing, and object-oriented design have multiplied the powers of hardware engineers, but developments in measurement technology in turn made these developments feasible. The continuing interplay of hardware, software, and fundamental science advances resolving power at the point of physical measurement while yielding ever more sophisticated analytical and knowledge-management structures.

Software development should not dominate the efforts of hardware engineers - but on the whole, information science tends inexorably into the core of test and measurement.

Role of the Record/Playback method

A review of the above pictorial comparisons (*Figures 10 - 14*) may suggest a range of potential applications of the Record/Playback method in software-intensive test and measurement technologies. **The leverage provided by this method is in consistently being able to have the Instrument software respond to an *exactly determined* sequence of Samples representing real or modeled system-under-test (DUT) characteristics.** Noise and systematic error differences between separate runs of the same procedure, can simply be eliminated.

As suggested earlier, known applications for the Record/Playback method include testing Instrument software offline or in the context of complex instrumentation setups. Given a well-matched set of conditions, moreover, the Record/Playback method might be used effectively within a larger scenario of developing a stimulus/response model of a given DUT, interfacing with other simulation tools, or providing standard currency for Sample processing in automated testing setups. Let us briefly visit each of these scenarios.

Software testing

Anyone who has dabbled in technology testing - even in a rudimentary scenario - understands the need for a disciplined approach. A rule of thumb is, "Change only one thing at a time." When one wishes to test a certain software modification, for example, a winning strategy might be first to use

a specific data set to test the unmodified software, then test again using the *same data set* together with the modified software. The only thing that should change in this scenario is the software itself.

If the software under modification performs tasks for which variation in observable results is comparable in magnitude to expected variation produced by sample noise or instrument drift, then to “change only one thing at a time” may become a thorny challenge.

Suppose, for example, that a certain Instrument software module takes a Sample stream as input, and yields a random process as observable output. Suppose one wants to determine the effect of an algorithm modification on the variance of the output process. If many Sample streams are introduced and the output process monitored, what component of the output variance is due to the modification, and what component is due to variance in the population of Sample input streams? If the latter variance can be held to a known constant value (zero, perhaps) then there is hope for determining the former variance. An implementation of the Record/Playback Simulator could be used to provide such a well-defined set of Sample streams.

In some test and measurement field applications, the reliability of custom setups may determine the success or failure of project activities. For example, there may be numerous physical connector port surfaces that represent precision measurement planes, all of which must be carefully characterized in a system calibration model. If software configuration or programming changes need to be tested under these conditions, it could be very useful to record a set of “golden” Sample streams derived from a “known good” calibration model, and use this set repeatedly in testing. With testing done incrementally, iteratively, or over an extended period of time, costly recalibration procedures can be circumvented.

Model development

In general, the Instrument and its Record/Playback Simulator may capture both stimulus and response sample values measured close to the DUT access port(s). Recorded streams of Sample vectors might be useful as part of an overall effort to model DUT performance. If the performance domain is linear and quality standards permit, the derived transfer function or impulse response could be used to simulate DUT performance over some regions of interest, reducing the need for costly exhaustive testing and processing of measurement data⁴⁰.

Applying custom data streams

Many above references to “the Instrument software” actually generalize to any process that operates on streams of vectors representing raw sample data. With appropriate support including file format conversion tools, custom Sample streams could be built algorithmically and filed in the same manner used to Record physical DUT data. We mentioned this above in the context of software testing associated with product development and custom field applications.

Such custom data sets also might be used in automated test scenarios that target the Instrument in production. Since the Simulator software is integrated with the Instrument software application,

⁴⁰ For an example of using derived signal data to simulate DUT characteristics, see *Trinh and Tran*.

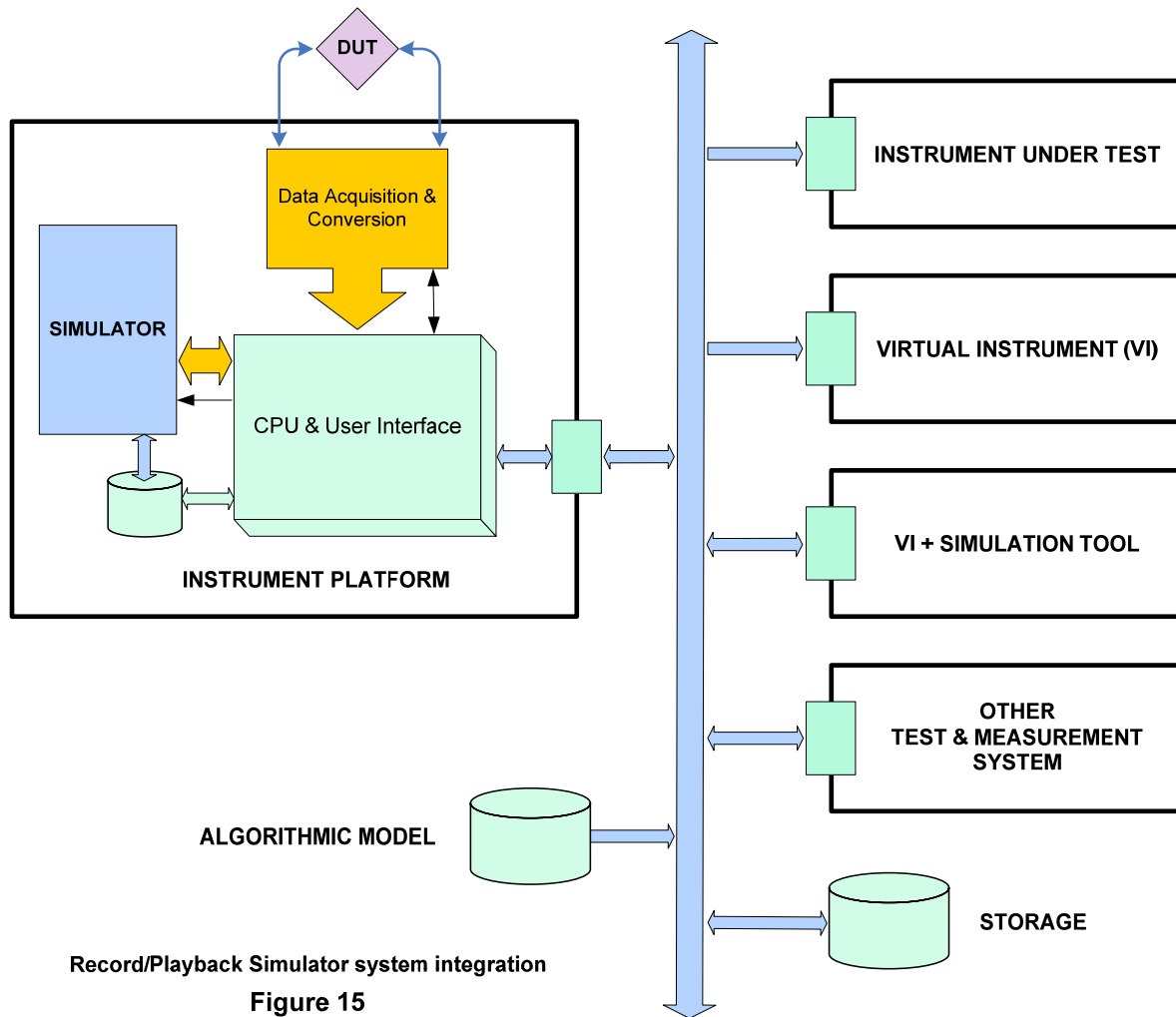
each Instrument is capable of translating correctly formatted files into Sample streams and processing these as if the Samples had been obtained from a DUT. Moreover, these Sample streams can be specified free of noise and drift. This method may open a door to new cost efficiencies in production testing.

Similarly, algorithmic or DUT-derived Sample streams might be used to develop signals that could serve as input to downstream simulation or modeling processes. For example, a noise-free wireless channel simulation could be mediated by a file containing a set of Sample streams. These data, perhaps with appropriate translation, might be catered to another simulation system that adds features to the channel.

Summary

We have described the Record/Playback Simulator and suggested some ways that the method can empower test and measurement instrumentation projects while supporting cost management. A simple design was carefully implemented and verified at an industrial facility.

An integrated test and measurement system including Record/Playback Simulator capabilities might include one or more Instrument Platforms connected with other functional blocks shown in the following diagram.



Sample streams could be physically captured from the DUT or algorithmically generated, and stored locally or remotely. Software development, production test, field operations, simulation, or feed-forward test and measurement applications could be supported by these recorded Sample streams⁴¹.

⁴¹ For a real-world example of an integrated system of modest complexity, see *Nelson, R. Sep 2003*.

GLOSSARY OF SPECIAL-USAGE TERMS

Sample	collection of signal sample values that represents a single measurement event (data point)
Sweep	ordered sequence of Samples associated with a sweep event performed by the instrument hardware
Simulator	a Record/Playback DUT Simulator software system
Instrument	a software system that interfaces with a Simulator, and processes Samples
State	subset of Instrument state variables that are critical for Simulator performance
Operator	human or program control of Instrument & Simulator

BIBLIOGRAPHY

Bhattacharya, S., Senquuttuvan, R., Chatterjee, A. "Production test enhancement techniques for MB-OFDM ultra-wide band (UWB) devices: EVM and CCDF", pp.10 pp.-245, *IEEE International Conference on Test*, 2005.

Source: <http://www2.computer.org/portal/web/csdl/doi/10.1109/TEST.2005.1583981>

Delic-Ibukic and Hummels. "Continuous digital calibration of pipeline A/D converters", *IEEE Transactions on Instrumentation and Measurement* 55:4, Aug 2006

Devine, S. "The insights of algorithmic entropy", *Entropy* 2009, 11(1), 85-110; doi:10.3390/e11010085, 4 Mar 2009. Source: <http://www.mdpi.com/1099-4300/11/1/85>

Fahrahmand, T., Tabatabaei, S., Ben-Zeev, F., Ivanov, A. "A DDJ calibration methodology for high-speed test and measurement equipments", *IEEE International Conference on Test*, 2005

Source: <http://www2.computer.org/portal/web/csdl/doi/10.1109/TEST.2005.1583978>

Haykin, S. *Communication Systems*, 4th Ed. (Hoboken, NJ: John Wiley & Sons, 2001)

Jenkins, K. "Detecting and preventing measurement errors", *IEEE Design and Test of Computers*, vol. 14, no. 4, pp. 78-86, Oct.-Dec. 1997, doi:10.1109/54.632884

Source: <http://www2.computer.org/portal/web/csdl/doi/10.1109/54.632884>

Lyahou, van der Horn, and Huijsing. "A noniterative polynomial 2-D calibration method implemented in a microcontroller", *IEEE Transactions on Instrumentation and Measurement* 46:4, Aug 1997

Maugard, G., Wegener, C., O'Dwyer, T., Kennedy, M. "Method of reducing contactor effect when testing high-precision ADC's", *International Test Conference 2003*

Source: <http://www2.computer.org/portal/web/csdl/doi/10.1109/TEST.2003.1270842>

Mayer, J. "3G delays force flexibility", *Test & Measurement World*, 1 Nov 2002.

Source: <http://www.tmworld.com/index.asp?layout=article&articleid=CA254575>

Nelson, R. "Vector network analyzers grade microwave components", *Test & Measurement World*, 1 April 2001. Source: <http://www.tmworld.com/article/CA187325.html>

Nelson, R. "Cut time-to-market for wireless designs", *Test & Measurement World*, 1 Sep 2003.

Source: <http://www.tmworld.com/index.asp?layout=article&articleid=CA318888>

Parhami, B. *Computer Architecture: from Microprocessors to Supercomputers* (New York: Oxford University Press, 2005)

Reed, G. "Check electronics success with RF/microwave test", *Test & Measurement World*, 12 May 2008. Source: <http://www.tmworld.com/article/CA6559882.html>

BIBLIOGRAPHY (continued)

Rowe, M. "Measure a disk drive's read channel signals", *Test & Measurement World*, 1 Aug 1999.
Source: <http://www.tmworld.com/article/CA187504.html>

Rowe, M. "Take good care of your RF calibration kits", *Test & Measurement World*, 1 May 2000.
Source: <http://www.tmworld.com/index.asp?layout=article&articleid=CA187395>

Rowe, M. "Rolling in the copper", *Test & Measurement World*, 1 Oct 2004.
Source: <http://www.tmworld.com/index.asp?layout=article&articleid=CA457494>

Rowe, M. "The analog in software radio: Testing SDRs", *Test & Measurement World*, 1 Feb 2007.
Source: <http://www.tmworld.com/article/CA6411374.html>

Rowe, M. "RF testing spins through transmission paths", *Test & Measurement World*, 1 Nov 2007.
Source: <http://www.tmworld.com/article/CA6495734.html>

Salomon, D. *Data Compression: The Complete Reference*, 3rd Ed. (New York: Springer-Verlag, 2004)

TMW:News Briefs. "Wide Screen, Quick Measurements", *Test & Measurement World*, 1 July 2008. Source: <http://www.tmworld.com/article/CA6573584.html>

Trinh, A. and Tran, M. "Signal Integrity Performance of the Teledyne Relays GRF300/GRF303 Relay Series", *Teledyne Corporation White Paper*.
Source: <http://www.teledyne-europe.com/pdf/Teledyne%20Signal%20Integrity%20white%20paper.pdf>

Wikipedia:Amdahl. "Amdahl's law". Source: http://en.wikipedia.org/wiki/Amdahl's_law

Wikipedia:Bzip2. "bzip2". Source: <http://en.wikipedia.org/wiki/Bzip2>

Wikipedia:LZMA. "Lempel-Ziv-Markov chain algorithm". Source: <http://en.wikipedia.org/wiki/LZMA>