Harold S. Henry
7511 329th Avenue SE
Fall City, WA 98024

January 1, 2011

harold@talerian.com

hshLib Data Overview

The hshLib distribution contains a portable static C-linkable library written by Harold Henry to provide consistent, lightweight, platform-independent support for a secure, scalable, distributed hierarchical store.

Contents

1. Introduction

1.1 Goals of the Current Release

1.1.1 Text Handling

- Lightweight platform-independent Unicode support (see hshLibUnicode.doc).
- Loading strings at runtime from a binary `.hkd` data file or the distributed store.
- Basic text parsing and formatting.

1.1.1.1 Utilities associated with text handling:

hshUcBuildBinary.exe

Builds the Unicode database text files published by Unicode.org and text files created for the hshLib library into a binary file named hshUnicode.bin that hshLib can load and use.

1.1.2 Date and Time Handling

Dates and times are not currently handled consistently across operating systems, and in some instances there are flaws in the implementations of date/time functions, particularly where time zones are concerned. As a result, hshLib provides platform-independent date and time functions that include support for:

- Leap seconds.
- Time-zones, including world-wide local times in past years.
- Dates in the distant past and future.

1.1.2.1 Utilities associated with date and time handling:

hshTzBuildBinary.exe

Builds a time-zone database for hshLib from the text files in the tz distribution (see http://www.twinsun.com/tz/tz-link.htm).

1.1.3 Memory Allocation

In a distributed world, marshaling data on and off of the wire can be a significant burden. To keep this overhead to a minimum, hshLib provides memory allocation and management optimized for data transfer.

1.1.4 Data Management

1.1.5 Document Creation and Navigation

The current release does not provide special support for document creation and navigation.

1.1.6 Decimal Arithmetic

Precise decimal arithmetic is essential for commercial computations. The library interfaces with Mike Cowlishaw's well-known decNumber library, currently at version 3.61, available under the ICU license, as well as direct from IBM under the alphaWorks license. The decNumber library provides full IEEE 754 support for decimal arithmetic in a package that is performant and widely used.

1.2 Features Planned for Later Releases

1.2.1 Text Handling

- Basic text display, including platform-independent font and string handling.

1.2.2 Date and Time Handling

- Event recurrence.

1.2.3 Data Management

The hshLib library will implement a scalable, fast, lightweight distributed store based on a hierarchical data model consistent with XML. The binary representation of data in this store is referred to here as `hshXml`. The library will provide:

- A message pump tuned to use TCP/IP and UDP to support large documents, data blocks and indices in the distributed store.
- Distributed indices that can return sorted results in a language-independent order and that are guaranteed to be stable over time.
- A distributed rules engine integrated into `hshXml`.
- A scripting subset of the C language (`hshCX`) that supports safe, portable comparison and transform functions for writing rules that operate on `hshXml` data.
- Fast data export and import to and from well-formed XML.
- Safe, synchronized access to data.
- Strong encryption and authentication.
- Permissions enforced at a per-node level.
- Efficient discovery of distributed data.

1.2.3.1 Utilities for data management:

hshCX.exe

Compiles code written in a subset of the C language (hshCX). This language can evaluate mathematical expressions and can call a subset of hshLib data functions.

It compiles into byte-code that is machine independent and can safely be incorporated into rules attached to `hshXml` data.

1.2.4 Document Creation and Navigation

The library will provide functions for organizing, browsing and searching data in an `hshXml` distributed store. It will also provide user-interface elements based on OpenGL for navigating, consuming and modifying hierarchical data and documents.

In addition to the library APIs, there will also be several tools included in the distribution for creating and navigating documents.

1.2.4.1 Utilities for document creation and navigation:

hshNav.exe

Provides a graphical interface for navigating hierarchical documents and data in the `hshXml` distributed store format that use the `hshDocSchema` generalized document schema.

hshXmlBuild.exe

Compiles one or more XML files that use the `hshDocSchema` generalized document schema into the `hshXml` distributed store format.

hshStringBuild.exe

Compiles a file containing Unicode text strings into a loadable string file in the `hshXml` distributed store format.

hshSdkBuild.exe

Builds a document written in XML using the hshLib documentation schema into one of a number of output formats, including a Word file, an Adobe Acrobat .pdf file, HTML files, an HTML-Help .chm file, an HTML-Help .hxs file, and an hshLib distributed store file. SDK documentation for the library is published in this form.

hshSdkCheck.exe

Compares SDK reference documentation written in XML using the hshLib documentation schema against header and source files and reports signature differences.

2. Text Handling

2.1 Rationale for Unicode Support in hshLib

The hshLib library provides Unicode support that is independent of the operating system where the code runs. This is necessary in order to implement consistent, fast, distributed data access that is portable and stable across version changes of Unicode.

The problem is that string indices in the library are built from sort keys, allowing the return of sorted results for browsing. The sort order provided by the Unicode Collation Algorithm (UCA) for this purpose is contained in the Unicode allkeys.txt data file, and is known as the "Default Unicode Collation Element Table" or DUCET.

As Philippe Verdy of Unicode.org articulated to me in an email of December 3, 2008 on the Unicode discussion list, this collation is not, and cannot be relied on to be, stable across versions of Unicode:

> "Note that the absolute values of collation keys in the DUCET are NOT stable. They are renumbered as needed when new characters are added. What is or should be (approximately) stable is the relative order of scripts (even if there's no policy for determining how they were initially positioned in the DUCET) and the relative order of characters within each script.
>
> "Also the absolute values are not meant to be universal: when tailoring is used, the gaps between numbers of the DUCET (whose default is to keep one position free between two successive number) may need to be changed.
>
> "…
>
> "The computed collation keys are then NOT stable and not portable. They are not meant to be transported from one system to another, even if they are based on the same version of the DUCET and even if they are not tailored at all. The UCA and DUCET is not meant to be handled like a character encoding standard. All pre-computed collation keys are for local use only. The only safe way to transport text from one system to another is to transport them encoded as sequences of Unicode code points with some character encoding, and then to compute the collation keys on the reception site, according to its local implementation of the UCA and DUCET."

Since hshLib is intended to support very large distributed indices that need to remain stable for years, it is necessary to be able to circumvent this inherent instability in Unicode collation so as to be able to create sort keys that do not change from system to system, or from Unicode version to Unicode version.

2.2 Design Goals

The library MUST implement the following functionality:

- Support all code points assigned in Planes 0, 1 and 2
- Generate custom sort keys that:
  - Are stable across all supported platforms and future versions of Unicode.
  - Produce a collation generally intelligible to users from any culture.
  - Can support case-insensitive and case-sensitive comparisons.
  - Can be generated efficiently, rapidly and incrementally.
  - Crudely approximate the collation produced by application of the "Default Unicode Collation Element Table" (DUCET) published in the allkeys.txt file.
- NFC Normalization (canonical decomposition followed by canonical composition); this MUST include rapid normalization checking.

- Case changing (to upper- and lower-case).
- Reliable numeric processing.
- Simple word-boundary and line-break detection.

The library SHOULD also provide:

- Rich character-category information.

The library does NOT support collation tailoring for individual languages or cultures.

2.3 hshLib Collation Approach

An hshLib sort key consists of a sequence of byte values, ordered such that a byte-by-byte comparison of two sort keys always determines the default sort order of the corresponding strings. The sort key has 5 levels, of which the first (level zero) is based on a custom semantic decomposition of the string in question. Levels 1 through 4 correspond to Unicode DUCET levels 1 through 4.

The actual values in the sort key are ordered according to the DUCET values, but are assigned within collation groups that roughly correspond to scripts, and are expressed using UTF-8 encoding. By assigning values within the context of a script or collation group, it is possible to maintain a stable value for sort keys across changes in version of the Unicode database.

The semantic decomposition used to generate level-zero values includes all the decompositions included in the Unicode database, as well as additional custom ones, full numeric decomposition, elimination of most "variable" collation values, and XML-style whitespace normalization.

While hshLib sort keys are designed to avoid unnecessary length, they are not intended to be stored with strings. Instead, it is assumed that individual sort-key bytes are implicitly embedded in index tables, and are then generated as needed for final string comparisons, since `hshUcKeyCmp` can do this relatively rapidly.

Before building a sort key, hshLib always normalizes the string in question to a Unicode `NFD` normalization form (*i.e.* full canonical decomposition), using `hshUcNormalize`. As a result, the library does not maintain collation data for code-points that can be canonically decomposed.

2.3.1 Level Separators and Sort-Key Termination

A null (zero-valued) byte is used to separate levels, and two null bytes indicate the end of the sort key:

`0x00`	Terminates a level.
`0x00 0x00`	Terminates the sort key

A null byte never appears in a sort key with any other significance.

2.3.2 Collation Groups and Group-Change Indicators

Character scripts that are either already in Unicode, or are on the Unicode roadmap as of now, or are currently being considered for inclusion, have hshLib IDs that are listed in Appendix A. Scripts in the Basic Multilingual Plane (BMP, or Plane 0) have script IDs in the range 10 (`0x0A`) through 78 (`0x4E`), while those on other planes have script IDs from 79 up.

Most collation groups correspond to these scripts, which are defined by the Unicode database in the Scripts.txt file, and in most cases a change in collation group within a string is signaled in level 0 and level 1 of the sort key by the byte value 0xFF followed by the script ID of the script to which the next character belongs.

Exceptions are the following collation group change values for common collation groups:

`0x01`	Change to the space and punctuation group.
`0x02`	Change to the modifier letter group.
`0x03`	Change to the first symbols group.
`0x04`	Change to the second symbols group.
`0x05`	Change to the third symbols group.
`0x06`	Change to the CJK (Han) symbols group.
`0x07`	Change to the numerics group.
`0x08`	Change to the currency signs group.
`0x09`	Change to the decimal digits group.
`0xF8`	Change to the Latin script.
`0xF9`	Change to the Cyrillic script.
`0xFA`	Change to the Arabic script.
`0xFB`	Change to the Hiragana/Katakana scripts.
`0xFC`	Change to the Hangul Jamo script.
`0xFD`	Change to the first CJK group.
`0xFE`	Change to the second CJK group.

2.4 System-Independent Text Display (Planned)

The library provides support for system-independent text display. This includes being able to:

- Determine what string, substring and character corresponds to a given cursor position (`x,y`).
- Allow text to be selected by dragging a cursor across it.
- Allow selected text to be copied and pasted or dragged and dropped, either retaining its formatting or taking on the formatting of its new context.
- Determine the `n`th character in a string.
- Allow new text to be typed in at the cursor position in a line.
- Wrap a paragraph as new text is typed in or existing text is deleted.
- Right-justify lines.
- Delete text within a line.
- Delete text across lines.

- Delete text in column mode.
- Change the formatting of selected text.

2.4.1 Text Forms

Text data intended for display in a user interface (UI) can be in one of the following forms:

Compressed	In a *serialized* form that has been compressed using the **`7z`** engine.
Serialized	UTF-8 or UTF-16 with HTML and custom XML markup
hshText	Loaded into an `hshText` structure for display and editing.

The `hshText` Structure

The `hshText` structure contains the following elements:

- The position, size and orientation of the text's bounding rectangle (in an **`hshSRect`** structure).
- An array of sub-string structures, each containing:
  - The offset of the sub-string's bounding rectangle from the start of its parent string's bounding rectangle.
  - The starting character index of the substring.
  - A reference to a string-style header that:
    - References a line structure that specifies:
      - padding around the line
      - border information
    - References a font structure that specifies:
      - font name
      - font family
      - font point size
      - font glyph size characteristics
      - font style options
    - References a text-style structure specifying:
      - background color
      - foreground color
      - font weight
      - font mode (normal, italic, etc.)
    - References a composition structure specifying:
      - text direction
      - character spacing
      - kerning guidelines
  - An array of character structures, each of which specifies:
    - The character code.

- Character type information.
- The offset of the start of the character's bounding rectangle from the start of the sub-string's bounding rectangle.
- Character accents and diacritical marks.

3. Date and Time Support

In hshLib, dates are generally stored as some positive number of days from the base date of 31 December 1969 or from the base date of February 29, -32400 in the proleptic Gregorian calendar, astronomical notation. The latter case, termed Arbitrary Gregorian Day or AGD computation, is intended to provide reliable date arithmetic across human history and much of human prehistory.

Time of day is generally stored either as a number of seconds after midnight (SAM) or as a number of ten-thousandths of a second after midnight (TAM).

Because date and time of day are stored separately, the most common ambiguities introduced by leap seconds and the gradual lengthening of the solar day are avoided. The library provides for identifying leap-second days and for calculating durations that include leap-seconds, but leap-seconds can usually be ignored, even at times very close to midnight.

Although only the Gregorian and Julian calendars are currently supported, support for other calendars may be added in the future.

A utility named hshTzBuildBinary.exe is included in the hshLib distribution to compile time-zone data from the tz database (currently at ftp://elsie.nci.nih.gov/pub/). This database provides excellent current and historical time-zone data that allow hshLib to convert UTC times accurately to and from local times over a wide range of dates and locations.

3.1 Timekeeping Background

3.1.1 The Gregorian and Julian Calendars

The calendar in general use today in the United States and most of Europe is the Gregorian calendar, introduced in 1582 by Pope Gregory XIII and gradually adopted throughout the world thereafter. In the Gregorian calendar, a year averages 365.2425 days in length, which is fairly close to a current average tropical year. The Gregorian calendar achieves this average year length through leap years, a leap year being one in which the month of February has 29 days instead of 28. In the Gregorian calendar, every fourth year is a leap year, except years that are evenly divisible by 100 and not evenly divisible by 400.

The European calendar in general use before the Gregorian calendar was the Julian calendar, established by Julius Caesar around 46 BCE. It is still in use in some places today. The average length of a year in the Julian calendar is 365.25 days, and every fourth year is a leap year without exception.

For dates before 46 BC, historians generally use an extension of either the Gregorian or the Julian calendar backwards in time; such a backwards extension is termed a proleptic calendar. The Gregorian proleptic calendar is preferable to the Julian one for more distant dates because it diverges more slowly from the actual orbit of the Earth around the Sun, and hence reflects the seasons more accurately. However, as we go thousands of years backwards or forwards, even the Gregorian calendar diverges significantly from Earth's orbit.

When dealing with dates concerning European or American historical events, it is often unclear which of these calendars applies. On request, hshLib selects a calendar for a given date at a given location as follows:

Before 46 BCE	Gregorian proleptic calendar, adjusted before 3275 BCE.
46 BC to 1582 CE	Julian calendar.
After 1582 CE	The calendar theoretically in local use at the time, if known, or otherwise the Gregorian Calendar. Dates at which the Gregorian calendar was adopted vary considerably — hshLib relies on the table in Appendix A to make the determination.

Julian Days (JD), which are often used in astronomy, are based at noon on Monday, January 1, 4713 BCE in the Julian proleptic calendar, which corresponds to Monday, November 24, 4714 BCE in the Gregorian proleptic calendar. Astronomers find it convenient to start a day at noon so that all observations made during a given night have the same date.

3.1.2 Negative Years BCE

The so-called astronomical calendar convention uses negative numbers to identify years BCE ("before the common era"), also known as years BC ("before Christ"). To understand how this works, keep in mind that there is no "year zero" in our normal reckoning— December 31st of the year 1 BCE is followed immediately by January 1st of the year 1 CE ("common era"), also known as AD (*anno domini*). Therefore, when years before the common era are expressed as negative numbers in the astronomical convention, they are offset by one:

```
1 CE    is year   1
1 BCE   is year   0
2 BCE   is year  -1
3 BCE   is year  -2
4 BCE   is year  -3
...and so on.
```

3.2 Time-Keeping Problems

3.2.1 The Issue of Three Clocks

A serious time-keeping problem is now gradually coming into focus, as Steve Allen of the University of California Observatories has thoroughly and thoughtfully documented (http://www.ucolick.org/~sla/leapsecs/dutc.html).

For the purposes of hshLib, this problem can be stated as follows: the three different types of clock that humans need in order to keep time don't stay synchronized and never have. These clocks are:

1. The atomic clock, currently based on the resonance of caesium atoms.
2. The year clock, based on the time it takes Earth to revolve around the sun.
3. The day clock, based on the time it takes Earth to rotate around its axis.

These three different clocks correspond to three broad purposes of time-keeping:

1. When measuring rates of change in science, including astronomy, it's essential to be able to keep time in a way that's precise, consistent and stable, so that two measured periods can reliably be compared. For these purposes, it doesn't matter at all what the unit of measurement is as long as it's clearly defined and unchanging. In general, stability is much more important for scientific time than matching seasons on Earth or the rising and setting of the sun.
2. As people have throughout the ages, we use the year clock primarily to keep track of Earth's seasons and the changes they bring.
3. We use the day clock primarily to keep track of our circadian cycles of sleep and wakefulness, and all the things we do during periods of wakefulness. Our biological clocks, like those of most animals and plants, are set by the pattern of light and darkness in our environment (i.e. day and night), which in turn is determined by the rotation of Earth around its axis.

Aside from scientists, the most demanding users of clocks throughout history have been navigators, who have needed (and continue to need) to be able to combine the year clock with the day clock to determine their position on the surface of the Earth relative to the stars. Although this is a specialized requirement, it's important enough to dictate a good deal about how we keep time.

3.2.2 Atomic Clocks and Synchronization

With respect to the three kinds of clock, the first one constitutes an amazing success story — atomic clocks are incredibly accurate. The caesium-based clocks on which the definition of the standard second is based are capable of accuracy to 1 part in 10^{15} or 10^{16}! Not only that, but advances in technology may soon allow us to keep time a thousand times more accurately still: Hidetoshi Katori and his colleagues at the University of Tokyo reported in Nature, vol 435 (2005), p 321 that they had succeeded in creating an

strontium-based "optical lattice" clock theoretically capable of an accuracy of 1 part in 10^{18}. An article in the February 7, 2009 issue of *New Scientist* (page 39) reviewed this and other advances in atomic clocks, and explored some of their implications for time-keeping.

Having such accuracy available has only served to bring into focus the lack of synchronization among the three clocks we need in our civil and technical activities.

Since the 1960s, the length of a second has been defined by the International System of Units (SI) as 9,192,631,770 cycles of the radiation which corresponds to the transition between two energy levels of the ground state of the caesium-133 atom. This value for a second is intended to create a day of 86,400 seconds that synchronizes both with the actual rotation of Earth around its axis and with the average length of a seasonal (tropical) year.

Synchronization is considered important because we live, travel and observe the sky from the surface of a spinning, wobbling, revolving orb, and we need to be able to calculate our position at any given point in time with respect to the solar system and the stars. In an underlying sense, that's always been the main reason we keep track of time.

And that's also where things get complicated — very complicated. In fact, as far as we can tell at present, it simply isn't possible to keep all three clocks stably synchronized — a problem that atomic-clock accuracy has only served to highlight and confirm.

3.2.3 Fluctuations in Earth's Orbit

The length of time it takes for earth's center of gravity to complete one entire revolution around the sun is quite stable; when measured with respect to distant stars, this type of year lasts about 31,558,149 standard seconds (SI) (365 days 6 hours 9 minutes 9 seconds, or 365.256363051 days) and is known as a sidereal year.

While the length of the sidereal year is stable, the planet's orbital path around the sun is not — it changes from year to year in accord with a variety of long-term periodic fluctuations known collectively as Milankovitch cycles. Not only do these fluctuations affect our position with respect to the sun and the stars, they also affect our measurement of the year and day lengths that we actually experience.

The seasons, after all, occur not at the planet's center of gravity but on its surface, which is rotating ever more slowly around a wobbling axis as it revolves around the sun. The so-called tropical year that corresponds to a year of seasons is measured at one place on the Earth's surface from one identifiable solar point to the same point in the following year (say, at Greenwich in England from one vernal equinox to the next). An average tropical year these days is not far from 31,556,925 seconds (SI), or 365 days 5 hours 48 minutes 45 seconds, or about 20 minutes shorter than a sidereal year.

Similarly, an "apparent" solar day (the day as we experience it on the surface of the planet) can vary in length by as much as 30 seconds over the course of a year just owing to the effects of the Milankovitch cycles. Once mechanical clocks became accurate

enough to detect this variation, the standard day was defined as an average apparent day (a "mean solar day" or MSD).

3.2.4 LOD Instability

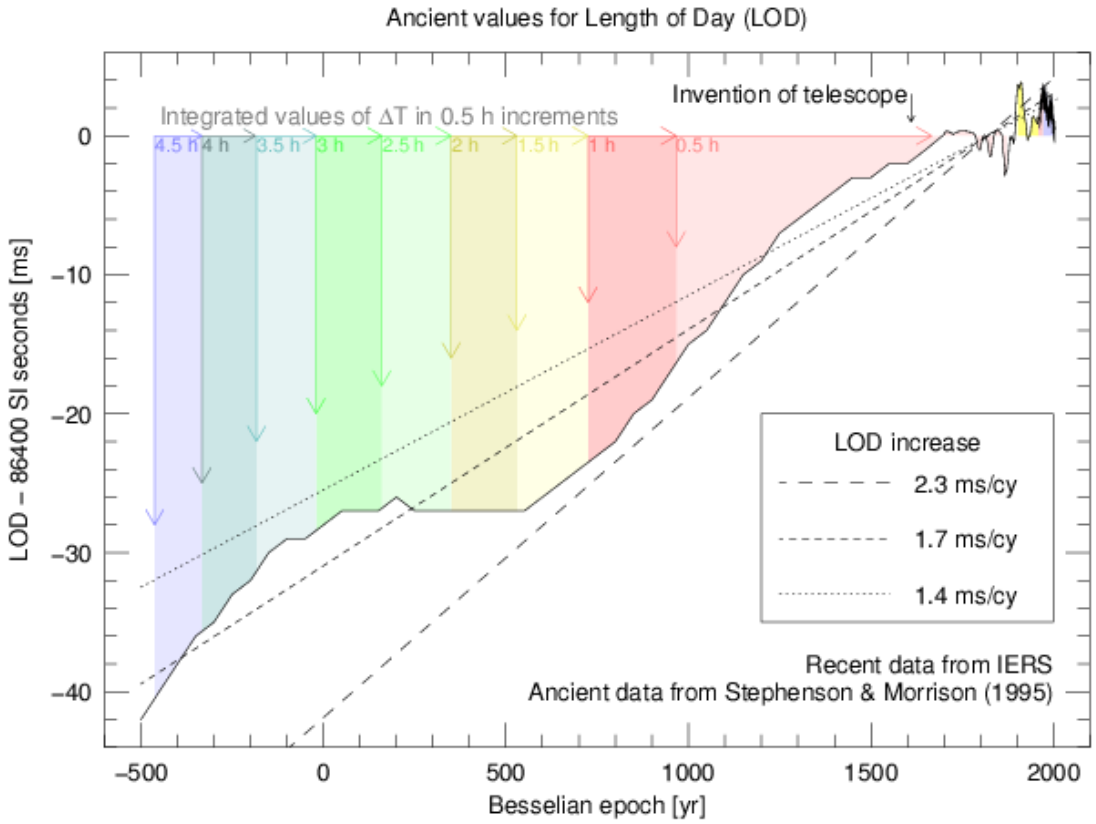
The trouble with the concept of a "mean solar day," however, is that there are factors significantly affecting the length of a day (LOD) that are not cyclic at all. Some of them are short-term and very irregular.

What determines how long a day lasts is the angular velocity of the earth, and that can be affected by all kinds of things, including changes in the distribution of mass resulting from climate change, magnetic interactions between the earth's core and the mantle, and even by changes created by short-term surface weather patterns.

The largest and most predictable of the factors affecting the planet's rotation speed is "tidal friction" resulting from gravitational interaction between the moon and the oceans. The effect of tidal friction can be estimated fairly accurately now to be causing the day to get longer by about 2.3 milliseconds per century. That figure may fluctuate with sea level changes, however, and it is by no means the only factor affecting changes in the LOD.

Careful analysis by F. Richard Stephenson and others of historical data has shown fairly convincingly that the actual lengthening of the day over the past 2,500 years is more on the order of 1.7 milliseconds per century. The discrepancy may be caused in part by crust and mantle rebound after ice-packs melted following the last ice age, and in part by fluid motions in the core of the earth that interact with and disturb the rotation of the mantle. During the relatively brief period during which we've had atomic clocks with which to take very precise measurements, moreover, it has become clear that the length of the day is constantly fluctuating somewhat chaotically.

Steve Allen provides on his site (http://www.ucolick.org/~sla/leapsecs/dutc.html) a graph that summarizes what we know at this point about the difference over the last 2,500 years between the length of an apparent day (LOD) and an atomic standard day of 86,400 seconds:



Finally, recent evidence gathered by analyzing tidal sediments in South Australia suggests that an apparent day 620 million years ago was around 21.9 hours long, which translates into an average change of only 1.2 milliseconds per century since that time (see George E. Williams, "Geological constraints on the Precambrian history of Earth's rotation and the Moon's orbit" in Reviews of Geophysics, vol. 38.1 (2000), p. 37-60; abstract available at http://adsabs.harvard.edu/abs/2000RvGeo..38...37W).

On the one hand, the changes we're discussing may not seem very significant — 2,500 years ago, the day was only a little more than 40 milliseconds shorter than it is today.

On the other hand, that small divergence is cumulative — over 2,500 years, it has added up to a difference of over 4.5 hours between day-clock time and atomic clock time. That means that in 500 BCE, atomic clock time would have noon happening at the equivalent of around 8:30 in the morning. Also, the further you go back into the past or forward into the future, the faster the cumulative divergence starts adding up.

3.2.5 UTC — Coordinated Universal Time

In most parts of the world, local civil time is defined as some number of hours earlier or later than Coordinated Universal Time (UTC). Computer systems generally use UTC when specifying a point in time, so that different times can be compared reliably. That way, for example, you can tell whether a file version that originated in Rome was last saved earlier or later than one originating in Hawaii or Bombay.

UTC is essentially equivalent to Greenwich Mean Time (GMT), also known as Zulu time — all specify a standard time at longitude zero. The "Coordinated" in UTC refers to coordinated radio broadcasts of time, which have served to make accurate synchronization of time possible around the world. Since the 1970s, UTC has been based on atomic clocks.

That's the simple version of the story of UTC, but there are also many other related time standards and variants that are referred to when precise synchronization for scientific or astronomical purposes is at issue. The best survey of these standards I know is Steve Allen's at the University of California Observatories (currently at http://www.ucolick.org/~sla/leapsecs/timescales.html).

The regulatory agency with the most historical influence over many of these various standards is the International Telecommunication Union (ITU) Radio-communications Sector (ITU-R), previously known as the *Comité consultatif international pour la radio* (CCIR), or Consultative Committee on International Radio. Its Web site is at http://www.itu.int/ITU-R/.

One of the most important stakeholders in time standard discussions is the community of astronomers, which is clearly the source of the best information regarding two of the clocks we're trying to keep synchronized with, namely earth's orbit and rotation. This community is represented in large part by the International Astronomical Union (IAU or UAI); its Web site is at http://www.iau.org/.

3.2.6 The IERS and Leap-Seconds

When you take into account that the earth's rotational period, and hence the length of a day, is constantly, unpredictably changing, and that the length of one complete year of seasons is partially dependent on the speed of earth's rotation, then you can see that any system tying together year, day and atomic clocks will require frequent adjustment.

Making the necessary frequent adjustments over the past 3 decades has been the responsibility of an organization called the International Earth Rotational Services (IERS — see www.iers.org/).

Since 1972, the IERS has handled the problem of synchronizing civil atomic time with the actual rotational period of the earth by adding a "leap-second" to atomic time whenever necessary to prevent the cumulative difference between atomic time and the current equivalent of a mean solar day from exceeding one second. It has sometimes been necessary to add two leap seconds in one year, and sometimes several years have elapsed before the next one needs to be added.

The time standard to which this applies is the one now used almost everywhere in the world, known as Coordinated Universal Time, and abbreviated as UTC. It's more or less equivalent to the older Greenwich Mean Time (GMT) except that it's regulated by atomic clocks.

For most civil purposes, leap seconds work fairly well. The main problem they introduce is that computer systems which use seconds-based time keeping don't always include them, which means that a time very close to midnight can easily be computed to fall in the wrong day if a mistaken assumption is made about whether leap seconds have been included or not. This is one of the main reasons that hshLib routinely stores date information separately from time-of-day information.

It may be that in the near future we switch to some other synchronization methodology, abandoning leap seconds. No matter what happens, however, it will still be important to keep our three clocks synchronized in some fashion, so that we can keep track of the things that they measure.

3.3 Times and Dates in hshLib

3.3.1 Arbitrary Gregorian Days (AGD)

The hshLib library introduces a new date convention called Arbitrary Gregorian Days that is designed to handle all of human history and most archeological dates. An AGD value is basically a count of the days elapsed since the AGD base, which is defined to be:

```
Gregorian proleptic:  00:00 (midnight)  Tuesday  February 29, 32401 BCE
Julian proleptic:  00:00 (midnight)  Tuesday   October 31, 32401 BCE
```

The earliest valid AGD date is January 1, 32001 BCE in the Gregorian proleptic calendar, corresponding to an AGD value of 145,997.

The latest valid AGD date is December 31, 12000 CE in the Gregorian calendar, corresponding to an AGD value of 16,217,076.

As you go back into prehistory or forward into the far future, the AGD number that corresponds to a given year, month and day may not be exactly what you expect, because the AGD system adjusts the conversion to and from year, month and day to take into account the effect of a changing length of a day (LOD). The purpose of this adjustment is to ensure that dates expressed in prehistory fall where you'd expect them to — for instance, that March 21 would be on or very near the equinox.

The value that the AGD system assumes for change in LOD over the 44,000 years it covers is 1.7 milliseconds per century rather than the higher value computed from estimates of tidal friction. Given the current state of our knowledge/ignorance, there are several reasons for choosing 1.7 ms/cy:

- It's the average rate deduced from actual observations over the past 2,500 years.
- If mantle rebound is a factor in counteracting the influence of tidal friction, that has likely been going on for the last 10,000 years anyway.
- The average change over the past 620 million years, estimated from the South Australian tidal sediment data, works out to only 1.2 milliseconds per year.

The assignment of a month or month and day to an event in prehistory is bound to be somewhat vague and arbitrary. By trying to provide at least some correction for length-

of-day change, AGD tries to ensure that dates in the past fall where we would expect them to in the year. And, for what it's worth, the day of the week is always calculated before correction, so it will always be right even if the month and day turn out to be a few days off.

For days in the distant future, on the other hand, we can be guess that civil time will continue to correct far more precisely for fluctuations in the LOD than can be predicted at present. Fortunately, at a resolution as coarse as one day, only 3 corrections are needed before 12,000 CE if we assume the LOD change rate of 1.7 ms/cy. Again, the point here is not to be precise with respect to how or when corrections will occur, but simply to ensure that a date chosen in the future falls roughly where we would expect it to in the year, and at roughly the time interval from the present that we intend.

Also, it's worth emphasizing that by storing the date value as an AGD and the time of day separately, the time of day will always correspond quite well to what you expect and intend because it has been isolated from the effects of cumulative divergences.

If the LOD has changed at a rate of 1.7 ms/cy, the average apparent day in 32,401 BCE was a little more than half a second shorter than it is today, resulting in the need for 42 prehistoric day adjustments, the most recent of which would occur in 3277 BCE. Going forward into the future, the first day adjustment would not occur until 7275 CE, and only 3 in all would be needed before 12,000 CE. The specific (somewhat arbitrary) days on which AGD day corrections occur are listed in Appendix B.

AGD relates to other common dates and date-convention bases as follows:

```
JD (Julian Day) base = AGD base + 10,112,738.5 days
JRD (Julian Rata Die) base = AGD base + 11,834,161 days
GRD (Gregorian Rata Die) base = AGD base + 11,834,163 days
Microsoft FILETIME base = AGD base + 12,418,552 days
MJD (Modified Julian Day) base = AGD base + 12,512,739 days
POSIX base (1/1/1970) = AGD base + 12,553,326 days
January 1, 2000 = AGD base + 12,564,283 days
```

Julian Days (JD), which are often used in astronomy, are based at noon on Monday, January 1, 4713 BCE in the Julian proleptic calendar, which corresponds to November 24, 4714 BCE in the Gregorian proleptic calendar. They begin at noon because it's convenient for astronomers to assign the same date to observations made on a given night, rather than having the late evening portion assigned to one date and the early-morning portion to the following date.

3.3.2 Counting Weekdays and Weeks

The hshLib library counts weeks of the year as specified in the ISO 8601 standard. On page 2, the standard defines a week of the year as follows:

> A seven-day period within a calendar year, starting on a Monday and identified by its ordinal number within the year; the first calendar week of the year is the one that includes the first Thursday of that year. In the Gregorian calendar, this is equivalent to the week which includes 4 January.

The standard goes on to clarify numbering of weeks and weekdays (page 5):

> Calendar week is represented by two numeric digits. The first calendar week of a year shall be identified as [01], and subsequent weeks shall be numbered in ascending sequence.
>
> Day of the week is represented by one decimal digit. Monday shall be identified as day [1] of any calendar week, and subsequent days of the same week shall be numbered in ascending sequence to Sunday (day 7).

3.3.3 XML Time Representation

ISO 8601 and the XML-Schema specification call for time of day to be prefixed by an upper-case 'T', and to be either UTC, or to be local time followed by the local time offset from UTC.

In addition to supporting this standard representation, hshLib also supports a local wall-clock time representation: tHH:MM:SS (italics indicating optional elements). A lower-case t is followed by one or more digits representing hours in a 24-hour clock. The hours value is optionally followed by a colon and two minutes digits, optionally followed by a colon and two seconds digits.

Time represented in this format indicates what a local clock would read at the specified time, which is often useful for scheduling purposes. Once scheduled, a local time can then be placed in the context of one particular date and locale and converted to UTC for co-ordination purposes.

3.4 Recurring Events (Planned)

Events scheduled by humans often follow simple patterns of recurrence, such as "every Tuesday," or "every second Friday," or "every year on the third Sunday in March," or "on the 8th of every month," or "on the first and last Wednesday of every third month."

Described below is a concise text notation for representing such recurrence patterns, designed to be reasonably human-readable.

Introduction to Text-Format Recurrence Codes
A number of elements in text-format recurrence codes use the same general format: they start with a character that identifies them, followed optionally by a number. Lists and ranges are enclosed in parentheses.

Although you can enter whitespace characters (space, tab, or carriage return) within text-format recurrence codes to improve readability, whitespace is ignored and stripped out when the code is parsed.

The order of elements in a recurrence code is as follows (green indicates optional elements):

{ startDate ! stopDate r count cycle@anchor : pattern(s) ~exclusion(s) }

The table below briefly describes each of these elements in turn, followed by more complete definitions:

Element	*Required?*	*Purpose*
{	*required*	A recurrence code is enclosed in curly braces.
startDate	*optional*	Specifies a date on or after which the first instance of the event takes place.
stopDate	*optional*	Specifies a date on or before which the last instance of the event takes place.
r	*required*	The body of a recurrence codes begins with a lowercase letter "r".
count	*optional*	The "r" can be followed by the total number of times the event recurs.
cycle	*required*	Specifies the type and length of cycle on which the event recurs.
@anchor	*optional*	If necessary, the cycle is followed by an "@" sign and an indication of when the cycle begins.
:pattern(s)	*optional*	If necessary, specifies one or more patterns of day(s) of the week and/or month on which the event occurs, preceded by a colon (":"), and optionally including times at which the event starts and ends.
~exclusion(s)	*optional*	Specifies a day, list or pattern of days on which the event does not happen.
}	*required*	A recurrence code is enclosed in curly braces.

The *startDate* Element

[*optional*] The startDate element specifies a date on or after which the first instance of the event takes place, in standard XML-Schema date form: YYYY-MM-DD (for example, 2002-03-01). If a day of the month is not specified, the first of the month is assumed, and if a month is not specified, January is assumed. Thus, "2002" by itself is interpreted to mean "starting January 1, 2002".

The *stopDate* Element

[*optional*] The stopDate element, prefixed by an exclamation point ("!"), immediately follows the startDate element and specifies a date on or before which the last instance of the event takes place. It too has a standard XML-Schema date form: YYYY-MM-DD (for example, 2002-03-01). If a day of the month is not specified, the last day of the month is assumed, and if a month is not specified, December is assumed. Thus, "2002!2002" is interpreted to mean "starting January 1, 2002 and ending December 31, 2002."

The *count* Element

[*optional*] If present, the count element specifies how many times the event recurs. It consists of one or more decimal digits.

When neither a count nor an endDate element is present, the event is assumed to keep recurring until such time as the universe terminates.

The *cycle* Element

[*required*] The cycle element defines the interval between instances of the event. It begins with one of the following upper-case letter that identifies the unit of time used to measure the recurrence cycle:

```
D: - The cycle is counted in days.
W: - The cycle is counted in weeks.
M: - The cycle is counted in months.
Y: - The cycle is counted in years.
N: - The event does not occur on a regular cycle.
```

If the cycle spans more than one of the designated time units, the designator letter is followed by the number of units per cycle: W2, for example, means "every two weeks."

The *anchor* Element

Cycles that span more than one time-unit generally require an anchor point in order to be calculated properly — for example, you can't determine the dates of an event that occurs once every 2 weeks unless you know what week to start counting from.

An anchor is specified by following the cycle number by an "@" sign and date information identifying the point from which to start cycle calculations. Date information is expressed using one or more of the following sub-elements:

```
y# - Year designator  (where '#' is the four-digit number of the year)
m# - Month designator (where '#' is a number in the range 1-12)
d# - Day designator   (where '#' is a number in the range 1-31)
```

An anchor point does not necessarily have to fall within the period when the event actually occurs, and need not always be a complete date. Monthly cycles that divide evenly into a year (*i.e.* 2-, 3-, 4- and 6-month cycles) don't need more than an anchor month: `M4@m2:d2` means three times a year on February 2, June 2 and October 2. On the other hand, a cycle not evenly divisible into a year would need to specify a starting year as an anchor: for instance, `M5@y1999m2:d15` means every 5 months on the fifteenth of the month, starting in February 1999.

In the case of multi-week cycles, the anchor date must specify a particular day, month and year so as to identify a week from which the cycle count can be started. This date is used only to determine what week to start counting from, regardless of what day of the week it happens to fall on. For example, `{2003rw2@y2002m1d1:w[3]}` means every other Wednesday starting in 2003, but counting from the first week in 2002 — and the fact that January 1, 2002 was a Tuesday is not important.

If a startDate element is specified and can serve as the anchor date, then the anchor date may be omitted. For example, `{2002-10-14r5W2:w[3]}` means every other Wednesday for 5 Wednesdays, starting in the week of Monday, October 14, 2002.

A colon (":") must follow the end of the cycle element and anchor date (if any).

Several more examples of cycle elements are:

```
M3@m2:          - every third month counting from Febrary
D4@y1994m4d3:   - every fourth day counting from April 3, 1994
W2@y1998m7d12: - every other week counting from the week of July 12, 1998
```

The pattern Element
[*optional*] Depending on the time-unit and pattern of recurrence, days on which an event recurs may be expressed in several ways:

Calendar Days. To specify a day counted from the beginning of a month, use one or more of the following sub-elements:

```
y# - Year designator  (where '#' is the four-digit number of the year)
m# - Month designator (where '#' is a number in the range 1-12)
d# - Day designator   (where '#' is a number in the range 1-31)
```

Examples are:

```
M:d5          - the fifth of every month
Y:m3d15       - every year on the 15th of March
N:y2001m12d5 - on December 5, 2001
```

In a list, each sub-element specifier remains applicable until another of the same type occurs.

Days of the Week. Days of the week are specified using a lowercase "w" followed by square brackets enclosing a digit that identifies the day of the week, as follows:

```
w[0] - Sunday
w[1] - Monday
w[2] - Tuesday
w[3] - Wednesday
w[4] - Thursday
w[5] - Friday
w[6] - Saturday
w[7] - Sunday
```

Note that Sunday can be represented either by `w[0]` or by `w[7]`.

If an event happens on a numbered weekday within the month, put the number of the weekday in the month between the w and the opening square bracket. For example, the third Friday of every month would be `{rM:w3[5]}`. You can indicate the last instance of a weekday in a month with the number 5, even if the month does not contain 5 instances of that weekday. For instance, the last Sunday of every third month starting in March, 2002 would be: `{rM3@2002-03:w5[7]}`.

Lists and Ranges. Lists and ranges are always enclosed in parentheses except in the case of days of the week that are already enclosed in square brackets. When you need to specify two or more days, weekdays, months or years, separate the elements or their identifying numbers with commas (','). You can express a range of values by using a hyphen ('-') between the first and last values in the range.

The examples below illustrate some of the ways lists and ranges can be used:

`{rW:w[1,3,5]}`	Three times a week on Monday, Wednesday and Friday.
`{rW2@y2002m4d9:w[2,5]}`	Every other Tuesday and Friday, starting April 9, 2002.
`{rM5@y2001m4:d(1,15)}`	The 1st and 15th of every 5th month starting in April 2001.
`{rM:(w2[1],w5[5])}`	The second Monday and last Friday of every month.
`{rM:w(1,3)[0]}`	The first and third Sundays of every month.
`{rY:m(6,8,11)w3[2]}`	The third Tuesday of June, August and November every year.
`{rM:w5[1-5]}`	Monday through Friday of the last week of the month.

`{rY:m(1-5,9-12)w5[6]}`	The last Saturday of Jan.– May and Sept.– Dec.
`{rM:w5[6]~m(6-8)}`	also last Saturday Jan.– May and Sept.– Dec.
`{rN:y2001(m3d7,m8d15)}`	March 5 and August 15 of 2001.

Event Times. To specify an event's start-time, use one of the following elements immediately following a day specification:

```
T - Standard XML-Schema time format
t - Local-time format
```

A start time of either format may optionally be followed by an ending time preceded by an exclamation point ('!'), or by a duration preceded by a pound sign ('#') as described below. An event time of this sort applies to the day specification it follows and also to all subsequent day specifications up to the next one that has a different time specification of its own.

'T' Start Time: A start time that begins with an uppercase 'T' takes the standard XML-Schema time form, and thus is assumed to refer to Universal Coordinated Time (UTC) unless suffixed by a time-zone-offset value.

't' Start Time A start time that begins with a lower-case 't', however, has a more relaxed format that specifies local time in the time-zone where the event takes place. The 't' is followed by one or more hour digits that assume a 24-hour clock, then optionally a colon (':') followed by two minute digits. For example, 1:00 A.M. would be `t1`, 1:15 A.M. would be `t1:15`, and 7:30 P.M. would be `t19:30`.

'!' End-time: An end-time is prefixed by an exclamation point, and always takes the same format as the start-time that it follows.

'#' Duration: A duration, preceded by a '#' sign, specifies the length of the event. By default, the duration is assumed to be expressed in hours, but other units, including years, months, weeks, days, minutes and seconds, may also be specified. In default mode, the '#' sign is followed by one or more hour digits, which can be followed optionally by a colon and two minute digits. In cases where a longer duration needs to be indicated, the designators, 'y', 'm', 'w', and 'd' may be used as well.

For example, a 2-hour event would be `#2`, a three-hour-and-five-minute event would be `#3:05`, a three-day event would be `#d3`, and a six-week event would be `#w6`.

The *exclude* Element
[*optional*] The exclude element allows you to list exceptions to a recurrence pattern; if present, it appears after the pattern, preceded by the lowercase letter 'x'.

Exclusions usually take the form of a list. For example, the following exclude element would indicate that Labor Day and Christmas are not part of a given pattern: `x(m9(w1[1]),m12d25)`.

An exclusion can also make reference to a data point in the XHX namespace where dates to be excluded are listed. Such a reference begins with an "@" sign, followed by a quotation mark, then the XPath of the data point, then a closing quotation mark.

Examples of Text-Format Recurrence Codes
The following examples further illustrate how patterns of repetition can be expressed using text-format recurrence codes:

`{rM:w3[5]}`
: Event occurs on the 3rd Friday of every month.

`{rM6@m6:d31]`
: Event occurs every June 30 and December 31.

`{rW:w[2,4]t19:30#1:30}`
: Event occurs every Tuesday and Thursday, starting at 7:30 p.m. local time and running for an hour and a half.

`{2002-06-28r8W2:w[5]}`
: Event happens eight times, on every other Friday, beginning in the week of June 28, 2002.

`{2002-03-17r6W:w[0]t13!16}`
: Event occurs every Sunday from 1:00 p.m. to 4:00 p.m. local time, for six weeks beginning on March 17, 2002.

`{2002-01-02rN:T19:30:00-8:00#2}`
: This is a one-time, two-hour event that starts at 7:30 p.m. Pacific Standard Time (PST) on January 2, 2002.

`{rM:w(1,3)[7]}`
: Event occurs on the first and third Sundays of every month

`{1904-03-31rM3:d31}`
: Event occurs once a quarter on the last day of the month, beginning on March 31, 1904.

`{2002!2002rW:w[1,3,5]~(m9(w1[1]),m12d25)}`
: Event occurs Monday, Wednesday and Friday of every week during 2002, except on Labor Day and Christmas.

4. Memory Allocation

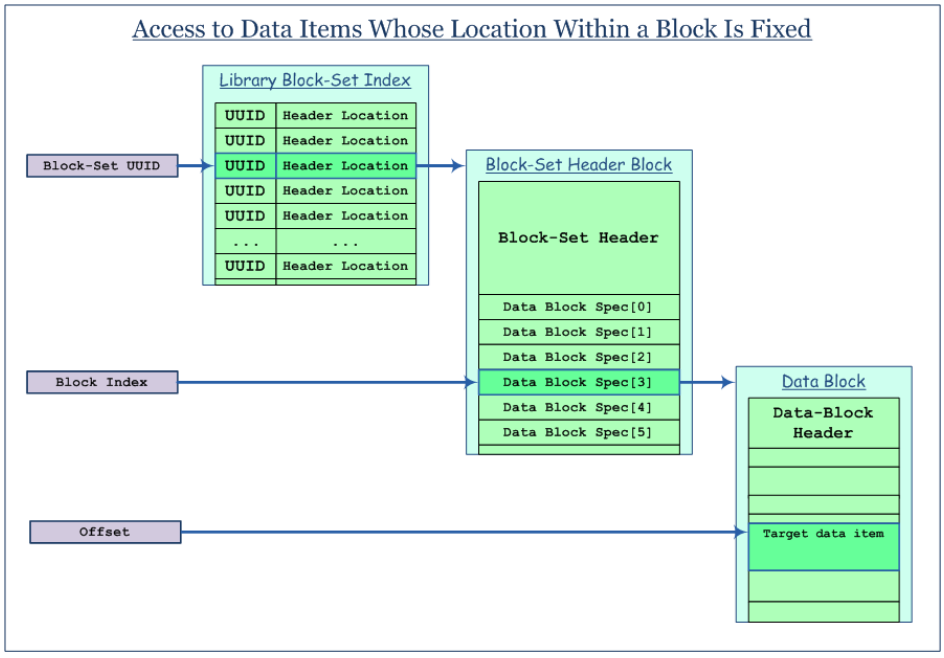
In a distributed system, data of all sorts must be as easily movable as possible. As a result, the hshLib library uses its own general memory allocation system using blocks that can be sent across the wire and reloaded with minimal overhead.

In this design, any data item can be identified using three values:

- A 16-byte universally unique identifier (UUID) used to locate a set of data blocks in which the data item is stored.
- A value that identifies a header containing information about the type and location of the particular data block in which the item resides.
- A value identifying the position of the item within that data block. This locator value does not change when the data block is moved.

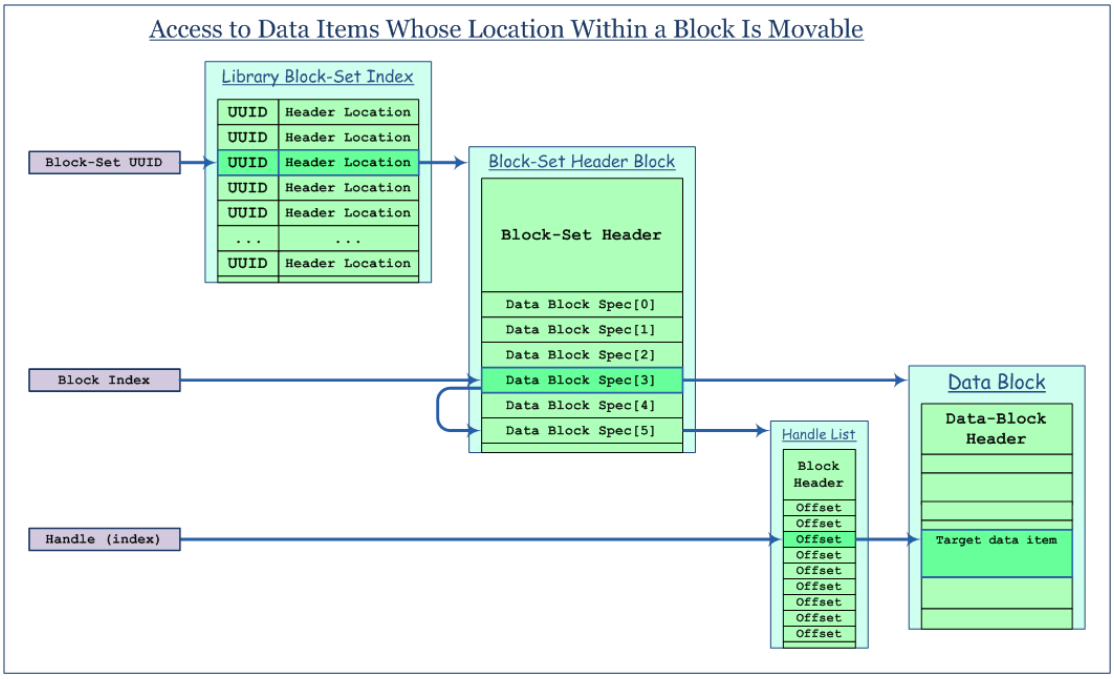
If it isn't necessary to move data items from their original position within the block that contains them, the start of each item can be located using an offset value indicating the number of bytes from the beginning of the block. Such offset values clearly do not change when the block is moved.

The diagram below illustrates how such data items are addressed in memory:



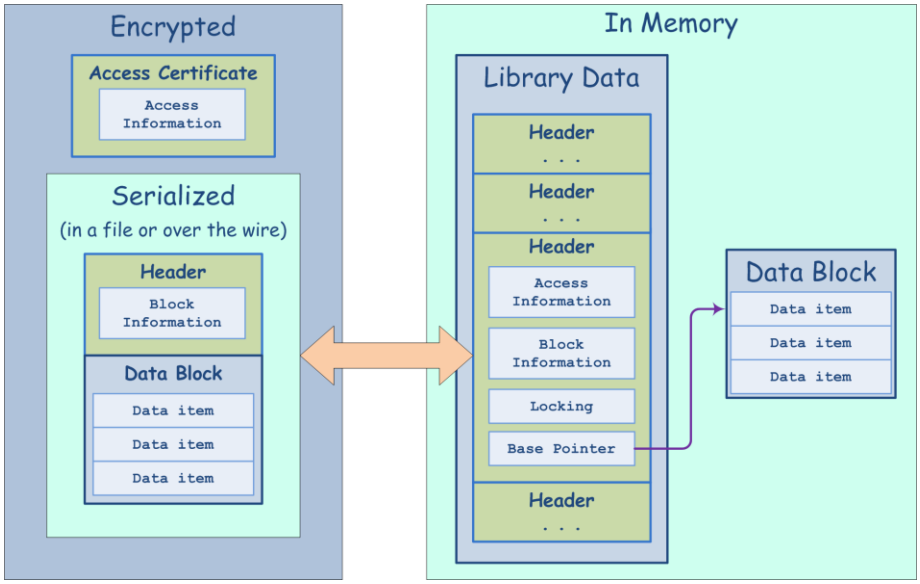
On the other hand, if items must be movable within the block that contains them, they can be accessed using a "handle," which is an index into an array of offset values. Allowing items to move lets them be resized and lets the block itself be compacted.

The diagram below illustrates how movable data items of this sort are addressed in memory:



If data items (or blocks) may be transient — in other words, if they may be discarded and replaced by different ones — then an ID value stored in the item or block in question should be checked before each access to ensure that the expected entity is still present.

More specifically, the indirection involved is illustrated as follows:



5. Data Management

5.1 Indexing in hshLib

The hshLib library implements its own distributed indexing algorithm referred to here as "partitioned binary search" (`pbs`).

Background on Indexing Algorithms
Although most efficient lookup algorithms (including hashes) can be understood as variants of a binary search, they're usually grouped into three categories:

- Hashes, in which keys are interpreted programmatically as bucket identifiers.
- Tries, in which the order of key comparisons follows the sequential layout of the keys.
- Trees, in which the order of key comparisons is organized into a tree structure.

On the surface, these look and perform quite differently, but they have a great deal in common as you look more closely — boundaries between the categories are somewhat arbitrary.

Hashing is generally regarded as the fastest simple approach to large-scale searching. B-trees, AVL trees and Red-Black trees are also popular in particular contexts. Donald R. Morrison's "Patricia" algorithm from 1968 remains the most elegant trie, while the "Judy array" system developed at Hewlett Packard is a more recent, less successful example.

5.2 The hshLib Partitioned Binary Search (`pbs`) Algorithm

The hshLib "partitioned binary search" (`pbs`) algorithm probably belongs in the *trie* category, inspired by the *Patricia* algorithm. Its purpose is to achieve a balance between the conflicting requirements of the following objectives:

- An index should be easily distributable — in other words, it should be easily separable into parts that can be moved around and kept in different places without a great loss in efficiency.
- Index size should be kept to a minimum.
- Several billion records with long keys should be indexable with reasonable performance.
- The index should be able to list data items in a reasonable default sort order.
- Code size and complexity should be minimal. Judy arrays, for example, though not as complicated as their documentation makes them appear, are too hard to maintain and extend.

In `pbs`, a key being searched for is examined at successive bytes that are not necessarily contiguous, and the key's value at those bytes is used to branch either to a subsequent byte, or to a data item that is the only potential match.

A `pbs` index can have up to 2^{32} pages, each of which contains between 2 and 256 nodes. This places an upper limit on the number of items that can be indexed of between 2^{32} and 2^{40} (between 4 billion and a trillion). Also, the length of a search key is limited to 65,535 bytes. For most purposes, these limits should suffice.

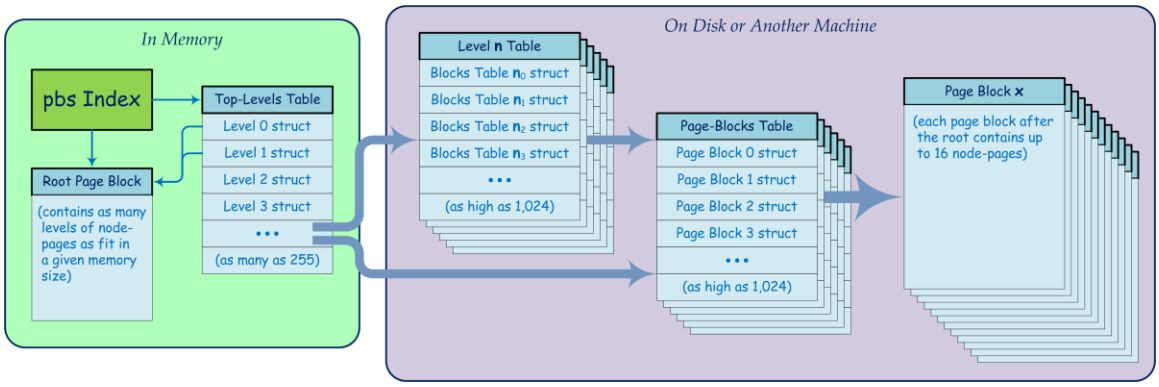
A leaf node maps to a data item or data locator that is contained in the page.

Each `pbs` page is associated with a 32-bit page identifier laid out as follows:

`0xFF000000`	The high byte indicates the page's "level" (the index of the byte in the key that the page branches on) if less than 255; otherwise, the high byte = 255.
`0x00FFFFF0`	The next 20 bits identify a page block associated with the high byte value.
`0x0000000F`	The low 4 bits identify a page within the page block.

Location of a given page begins in a level table that is kept in memory, and whose entries correspond to the values in the high byte of a page identifier. If the level in question has 1,024 or fewer page blocks associated with it, the level table identifies a page-blocks table in which the page-block for the page can be looked up. If the level has more than 1,024 page-blocks associated with it, the level table identifies an intermediary table in which the correct page-blocks table can be looked up. Once the correct page block has been located, the page can be located on it using the low 4 bits of the page identifier.

The following diagram illustrates the general layout of a `pbs` index:



This may seem like a lot of indirection, especially when locality of reference is the dominant factor governing performance, but this organization lets the index be distributed efficiently. Furthermore, the *trie* approach allows concise index-request messages to be sent to other machines, asking in effect, "On node page **n** of index **X**, is there a match for byte **b**, and if so, what is the result?"

Given that a key can be up to 64 kilobytes long, you can imagine a worst case in which a degenerate combination of keys like the following would required 65,536 comparisons over 65,536 pages:

```
a
x
aa
```

```
ax
aaa
aax
aaaa
aaax
 • • •
aaaaaaaaaaaaaaaa…a      (64KB long)
aaaaaaaaaaaaaaaa…x      (64KB long)
```

However, key sets like this are rare, and can be avoided, if need be, by using a key transform such as run-length encoding.

More to the point is how to handle page loading efficiently in the general case. The shallowest tree results from indexing sequential values of an unsigned 32-bit little-endian integer, starting at zero. This produces a *trie* having 4 levels, with the following number of pages at each level:

Level 0:	1	(2^0 pages)
Level 1:	256	(2^8 pages)
Level 2:	65,536	(2^{16} pages)
Level 3:	16,777,216	(2^{24} pages)

In this case, it makes sense only to keep levels 0 and 1 in memory.

On the other hand, when ASCII sentences are being indexed, the top level page will contain at most 26 nodes (one for each upper-case ASCII character), and pages at other levels are not likely to be larger. As a result, the chain of comparisons will be correspondingly longer for a comparable number of data items, and it will probably be desirable to keep more than the top two levels in memory.

To facilitate memory tuning of this sort, pages for levels 0–254 are segregated by level. From level 255–65,535, all pages are grouped together.

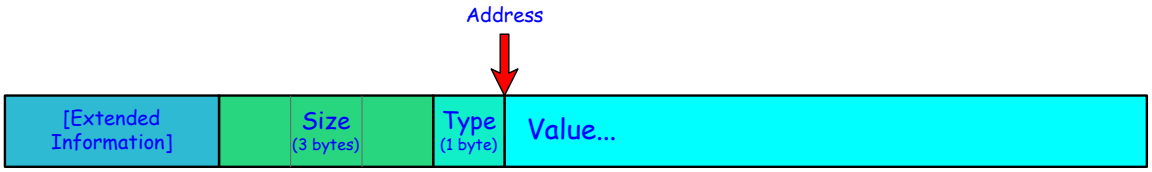
5.3 The `hshXml` Distributed Binary Hierarchical Data Store

The library provides for creation and management of a distributed hierarchical data store having the following characteristics:

- Can represent XML data in a compact binary form that can be loaded and navigated without text parsing.
- Scalable to trillions of nodes.
- Fast, and lightweight enough to run on cell phones.
- Incorporates secure distributed rules processing that can be used not only for data filtering, consolidation and transforms, but also to build simple applications.

5.3.1 Data Representation

Data values are stored in memory blocks, using the following protocol (illustrated in the diagram below):



- **Address:** The published address of a data value is the offset in the memory block at which the value itself starts (the red arrow in the diagram). It always falls on a four-byte boundary unless the value is a single byte, in which case it is word-aligned, or unless the value is a null-terminated string, in which case it can fall on any byte.
- **Base Type:** Immediately preceding the start of the value, there is always a one-byte base-type specifier. A zero-valued type specifier indicates a null-terminated string, allowing string blocks to be packed tight (the ending null of one string serving as the type specifier of the next).
- **Size:** Except in the case of null-terminated strings or values smaller than 4 bytes, the three bytes preceding the type specifier contain the size in bytes of the value itself (thus limiting most values to a maximum size of 16 megabytes).
- **Extended Information:** A type-specific "extended information" block may precede the *Size* bytes, of a layout and size determined by the type.

5.3.2 Data-Relationship Representation

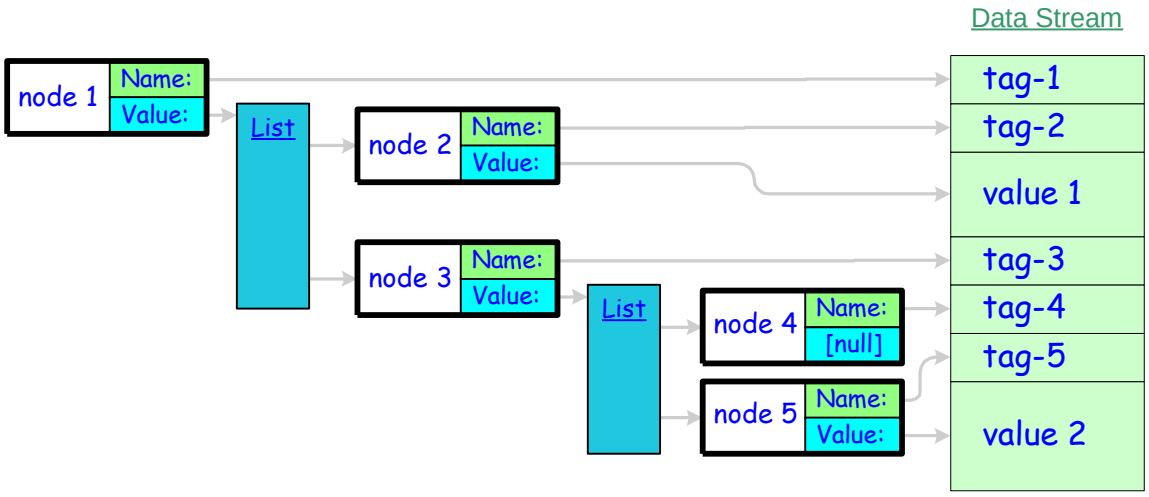
Two programming constructs are used to represent a graph of relationships between data values:

- A node containing a name offset and a value offset.
- A list that provides access to multiple child nodes.

Consider a simple XML hierarchy such as the following:

```
<tag-1>
  <tag-2>value 1</tag-2>
  <tag-3>
    <tag-4/>
    <tag-5>value 2</tag-5>
  </tag-3>
</tag-1>
```

Using nodes and lists, the `hshXml` tree structure can be represented as follows:



Locating a given data value thus requires traversing a sequence of nodes and lists. Although this may be slower than traversing a single large list or hash table, partitioning the lookup allows sub-trees to be relocated without requiring corresponding tables to be rebuilt.

5.3.3 Node Concepts

Nodes are more uniform than XML — whereas XML elements have a structure that includes a tag name, attributes, and content consisting of text, other elements, or a mixture of both, `hshXml` nodes represent the XML tag name, attributes, text content and element content all more or less consistently as "children" of the node. This uniformity of representation simplifies navigation and editing considerably by reducing the number of primitive operations required.

At the same time, there are additional primitives not implied by XML that are required to implement security, a distributed store, rules, and other features of `hshXml`. Some of these extensions are internal and entirely transparent to a user, while others provide functionality that substantially alters the document itself.

5.3.4 Parent-Child Hierarchy

Every `hshXml` node has *one and only one* parent — in other words, it belongs to a hierarchy that forms a directed acyclic graph (DAG). This hierarchical structure has several important characteristics:

- Parent-child relationships are doubly linked so they can be traversed in both directions.
- Physical storage is managed through this hierarchy.
- Important “extensive” properties and policies, such as permissions, can be propagated only across parent-child links.

5.3.5 Storage Segments and Storage Nodes

The `hshXml` tree is divided into many *storage segments*, each of which consists of a "storage node" and some or all of its direct descendents. As the name implies, a storage node is one that manages memory allocation and persistent storage for that segment. Among the structures used to store hierarchy and content are the following:

- **sysName[ ]** The `hshLib` library provides an array of pointers into a block of up to 64K system tag names that is always present on every node. Any tag name that appears in this list may be referenced using its `sysName` index, rather than including the tag-name text in the document. Because the display text can be changed without affecting the tag indices, this allows easy localization of shared schema.
- **tagSpace[ ]** In addition, any `hshXml` document can make reference to one or more standard `hshXml` namespaces of up to 64K tag names each (a "`tagSpace`"), analogous to the `sysName` array. Again in this case, tag-name text is not included in the document, on the assumption that any recipient of the document either already has the indicated tag-name block, or will download it from the indicated URL on receipt of the document. Note, however, that no more than 4,095 tag namespaces can be in use on a node at one time. As a result, tagSpaces should be used sparingly, when many documents are expected to share the same tags.
- **node[ ]** Each storage node manages an array of node structures. Note that nodes in this array are referenced using offsets rather than indices, to save a few cycles while navigating.
- **dataBlock[ ]** Text and binary data contained in XML elements (numbers are converted to binary), along with tag names not provided by the system, child-list structures, indices, and heap-management information, are all stored in a private heap managed by the owning storage node.

5.3.6 Forms of Indirection: Proxy *vs.* Link

A child node in `hshXml` may contain as its value a redirect or reference to another node, the *target*, possibly in a different memory block or on a different machine. The relationship created by such a reference is termed *indirection* and such a child is said to act as a *pointer* to the target.

A pointer node can be either a *proxy* or a *link*. These two types may be treated identically when navigating, but they have different purposes and create different relationships.

- **Proxy indirection** A proxy child is *always* used to bridge a change in storage segment, whether local or remote. Conceptually, a proxy node is an intermediary that should not formally be considered a real child of its parent or parent of its target — if its target is relocated into the proxy's storage segment, the proxy simply disappears, since its only purpose is to serve as a stand-in and locator for the target. Although it has the same structure as any other node, it is an internal construct of the distributed `hshXml` system, not an integral part of the XML tree.

- **Link indirection** A link, by contrast, is itself a real, persistent child of its parent node, but points to a target that is the child of a different parent. Links are not hierarchical, and may be arbitrarily cyclic. They can be used to build indices and to store groupings, state information and dynamic associations that permit `hshXml` to respond efficiently to complex queries. During navigation, only "restrictive" properties and policies can propagate across links.

5.3.7 Intrinsic *vs.* Navigation-Dependent Children

In a normal XML document, all children are intrinsic in the sense that each forms an integral part of the document and belongs to its parent node under all circumstances.

In `hshXml`, however, transient children may be assigned during the process of navigating the tree, through a mechanism termed "co-navigation."

In its simplest form, co-navigation implements a form of inheritance, causing a specified node to be treated as a child of every node arrived at along the path being navigated. Co-navigation can also be used in more complex ways — when it is invoked, one or more secondary tree-segments are activated so that navigation in the primary tree-segment results in matching navigation in the secondary segment(s), with the result that nodes in the secondary segment(s) are treated as transient children of nodes in the primary segment.

Besides implementing navigation-based inheritance, co-navigation also supports "rule-sets" expressed in the secondary tree-segments, along with "state nodes" needed for efficient rule evaluation.

Rules can take different forms, including the following (although these may not all be supported in version 1.0):

- XML Schema elements
- Access-restriction elements
- XSLT templates
- `hshXml` event-handler elements

5.3.8 State Nodes

State nodes reside in a special kind of co-navigation tree-segment and are used to support complex conditions involving logical ANDs. Logical ORs, by contrast, simply generate multiple rules.

Suppose a rule contains a complex condition such as the following:

```
if( ( nodeA < 10 ) && ( nodeB > 5 ) && ( nodeC == 3 ) )
```

To support this condition, a state node would be created to express the join of the three sub-conditions. Separate rules under each of the three nodes would contribute to the value of this state node, whose own rule-child would determine when the entire condition is satisfied.

5.3.9 Propagation

Whereas co-navigation supports a navigation-dependent form of inheritance based on temporarily assigning children in a particular context, “propagation” supports context-invariant inheritance by assigning persistent children across specified portions of the tree, regardless of how they are navigated to.

Propagation allows an `hshXml` element to be designated as a child not only of its parent, but also of other descendents of that parent. This lets a property become intrinsic to an entire sub-tree of elements.

Propagation does not, however, implicitly cross links.

5.3.10 The Child List

At the heart of `hshXml`, the child-list implementation provides economic access to node children (including the unnamed text-node children in "mixed" content types). This is easy when there are only a few children, harder when there are a few hundred, and quite complicated when there are millions. In child lists provide memory management, serialization and file-system support to storage nodes and their direct descendents.

One of the enormous advantages of XML is its ability to represent a variety of different arbitrarily structured data. `hshXml` is designed to be flexible enough to support this kind of diversity without sacrificing size or performance.

An XML element, for example, may contain a deeply structured page of text where the order and hierarchy of elements is crucial to its meaning, or it may contain a long, flat list of name-value pairs whose order is not important, but over which the ability to perform fast lookups is essential. An element may also be empty, or it may contain a single number. In `hshXml`, it may even contain a large executable or `.zip` file. All of these scenarios must be robustly and economically supported.

As a result, the child-list implementation uses different lookup strategies and structures depending on the size and purpose of the list it has to manage. It relies on optional child elements, and a range of algorithms to support these various scenarios.

5.3.11 Child-List Characteristics

The following characteristics affect how a node's children are managed:

- What is the total number of children?
- Are the child-elements suitable for tabular storage? In other words, is the structure of the child-elements completely uniform, each having the same tag name and possessing the same number of subelements of the same names?
- How deep is the hierarchy beneath this node?
- Does order of child-elements matter?
- Is the child list expected to be highly dynamic (frequent additions and deletions)?

5.3.12 Ordering Considerations

If order is *primary* and relatively static, as in a structured page of text, then it is efficient to keep memory and document order the same, and handle insertions and deletions using memory moves.

If order is *secondary* and relatively dynamic, on the other hand, as in a routing table or other database-like table, then memory order should be maintained independent of document order, and standard techniques for allocating and freeing memory blocks should be used to handle insertions and deletions.

5.4 The hshCX Tool

The hshLib library needs a scripting language in order to support end users in writing custom functions that are portable and safely distributable, for:

- Comparing data items referenced in an index.
- Rule conditions.

These applications require:

- High performance.
- Small footprint.
- High security.
- Cross-platform portability.

In addition, this language must have:

- A source-code syntax that is easy for developers to use.
- A compilation methodology that allows for testing and debugging.

5.4.1 Design Overview of `hshCX`

The hshLib scripting language, called hshCX, joins a subset of ANSI C with a subset of the types and functions of the hshLib library.

As a result, functions intended to be used as hshCX functions can be compiled and tested using a standard C or C++ compiler linking to hshLib`.lib`.

Using the hshCX.exe tool, such functions can also be compiled to compact byte-code that can be sent to other computers. On any supported platform, hshLib can then rapidly compile such byte-code into fast local machine code as soon as it is received.

It is safe to send and receive functions compiled by hshCX because the language restricts memory access to a very constrained local context and supports I/O only through hshLib's secure data-access calls. A function to be compiled should immediately be preceded by a comment that contains the XML start element, `<hshCX>`. Similarly, the end of the function should be followed immediately by a comment that contains the XML end element, `</hshCX>`.

Because hshCX.exe only processes one file a time, and does not process `#include` directives, it is often necessary to reference data structures, such as structures or unions, that are defined elsewhere. Similarly, for purposes of conditional compilation, it may be necessary to redefine tokens that are defined in other files. The way to do this is illustrated below:

```
/*  <hshCX>
      <headerRef path="..\include</path" file="someFile.h">
        <def type="struct"
             name="addressListStruct"
             typedef="addrList, *pAddrList" />
        <def type="define"
             name="SOME_IMPORTANT_CONSTANT" />
        <global type="pu8"
                name="myHelloWorldString" />
        <global type="addrList"
                name="currentRecipients" />
      </headerRef>
*/
void myHelloWorld( )
{
  hshSendMsg( myHelloWorldString, currentRecipients.addrs );
}
/*  </hshCX>  */
```

Note that referenced declarations and definitions can only contain language elements that are supported by hshCX.

Relevant declarations and definitions need only be referenced once in a file; that is, they need not be repeated before every function being compiled.

5.4.2 Language Details:

Operators

The following are supported, with normal precedence:

Arithmetic:	+	-	*	/	%	++	--	**			
Shift:	>>	<<									
Bitwise:	&	~	^								
Assignment:	=	+=	-=	*=	/=	%=	&=	|=	^=	<<=	>>=
Logical:	||	&&	!								
Relational:	<	<=	==	!=	>=	>					

Note that a binary exponentiation operator (**) is supported, for both integers and floating-point values.

5.4.3 Data Types

Instead of the basic data types native to ANSI C, hshCX only supports the following explicitly-sized base types defined in hshLib (associated pointer types are also supported):

s8	u8	f32	hB
s16	u16	f64	nil

```
s32   u32
```

Values can be cast between the data types above where appropriate causing conversions to occur in the usual way.

5.4.4 Supported Keywords

Only a subset of C/C++ keywords are supported in hshCX functions:

```
break
case
continue
default
do
else
enum
for
goto
if
return
struct
switch
typedef
void
while
```

The following standard C/C++ keywords are simply ignored:

```
asm           bool          catch          char
auto          const         const_cast     dynamic_cast
extern        inline        register       reinterpret_cast
static        static_cast   volatile
```

Annotations (enclosed in square brackets) are also ignored.

The following standard C/C++ keywords are recognized as such but cause `hshCX.exe` to generate a compile error if they appear in any function designated as an **`hshCX`** function:

```
asm           bool          catch          char
class         delete        double         explicit
false         float         friend         int
long          mutable       namespace      new
operator      private       protected      public
short         signed        sizeof         template
this          throw         true           try
typeid        typename      unsigned       using
virtual       wchar_t
```

Any other unrecognized expressions in an hshCX function also cause a compile error.

5.4.5 Precompiler Directives

Supported precompiler directives are:

```
#define
#ifdef
#ifndef
#else
#endif
```

All others are simply ignored.

5.4.6 Sandboxing And Side-Effects

An **hshCX** function can only return one of the following hshLib types:

```
void
hB
s8      u8
s16    u16
s32    u32
f32
```

hshCX can read from memory locations identified in parameters passed in to functions, but cannot write to them.

The only way **hshCX** can change anything but local variables is by returning a value, or by calling one of the supported hshLib data access functions.

6. Document Creation and Navigation (Planned)

To make life easier for developers who use the hshLib library, the distribution contains several tools aimed at making it easy to deliver SDK documentation and user help as `hshXml`.

6.1 The hshNav Tool

Provided with source code, the hshNav tool uses the library to navigate any `hshXml` tree that uses the `hshDocSchema` generalized document schema. The tool shows a collapsible hierarchical table of contents and/or index in a pane at the left, if the document provides them, and pages on the right, in much the same way as Microsoft's HTML help does.

6.2 The hshXmlBuild Tool

The hshXmlBuild tool, also provided with source code, builds XML files written using the `hshDocSchema` generalized document schema into an hshXml document.

6.3 The hshStringBuild Tool

The hshStringBuild.exe utility takes as input a text file of a specific format and compiles it into a binary `hshXml` string file with a .str extension that is loadable by the hshLib library. The library contains functions for loading a whole file, or groups of strings, or individual strings on demand.

Format of an hshStringBuild Source Text File

6.3.1 Encoding

A string text source file must be encoded either as UTF-8 or UTF-16. If it is encoded as UTF-16, it must begin with a standard byte-order mark character (BOM, = `0xFEFF`). If it is encoded as UTF-8, the initial byte-order mark character is optional, but is never treated as a zero-width space.

6.3.2 Header Line

The first line of the file must contain the following header information:

An hshLib identifier. The word "hshLib" (case-sensitive) MUST appear at very the beginning of the file, right after the byte-order mark, if there is one.

A content identifier to go in the output-file name. After one or more space or tab characters following "hshLib," a word must appear that will be included in the name the output file generated by `hshStringBuild.exe` to identify what kind of strings the file contains. This identifier must be made up only of alphanumeric characters (no spaces or punctuation).

A language identifier. After one or more space or tab characters following the content identifier, a 3- to 6-character language identifier MUST appear. The first three characters of this language identifier must be a standard ISO 639-2 language code, while the optional last three characters must begin with a hyphen ('-') and may then be consist of any two alphanumeric characters.

An optional loading hint. The letter 'm' (or 'M'), standing for "memory," may optionally appear after the language identifier on the same line, optionally separated by one or more space characters, and followed immediately by a digit between 0 and 2, with the following significance:

m0 An 'm followed by a zero causes the `HSH_OPEN_FILE_ON_DEMAND_HINT` value to be set in the `fileInfo` field of the string file header, suggesting that the string file should generally be opened only when one of its strings is needed.

m1 An 'm' followed by a one causes the `HSH_HOLD_FILE_OPEN_HINT` value to be set in the `fileInfo` field of the string file header, suggesting that the string file be kept open to provide access to any strings in it that are not loaded into memory (this is the default setting if no loading hint is provided on the first line of the file).

m2 An 'm' followed by a two causes the `HSH_LOAD_FILE_HINT` value to be set in the `fileInfo` field of the string file header, suggesting that the entire string file should be loaded into memory and kept there for rapid access, allowing the file itself to be closed.

6.3.3 Comments

A pound sign ('#') that is not within a string begins a comment that the hshStringBuild compiler ignores. Such a comment extends to the end of the line. Empty lines are ignored.

6.3.4 String Groups

Every string in a loadable string source file must be contained in a numbered string group. When a particular string is loaded from a string file, it is identified by two numbers: the index of the group it is in, and the index of the string within the group.

A string group may not contain more than 65,534 (*i.e.* $2^{16} - 1$) strings.

A string group must begin with an opening square bracket character ('`[`') at the start of a line, followed by its group number expressed either in decimal digits, or in hex digits preceded by "`0x`". The first group in a file must be number zero [`0`], and the number of every other group must be one larger than that of the preceding one. In other words, the groups must be sequentially numbered, starting at zero. The group number must be followed by a closing square bracket ('`]`').

The closing square bracket may optionally be followed immediately by a plus sign ('`+`'), which indicates that the group is intended to be loaded into memory as a string array rather than accessed from disk — this causes the `HSH_LOAD_STRING_ARRAY_HINT` flag to be set in its header, which can then serve as a guideline for applications that access the resulting string file.

Space characters and an identifying comment will usually then follow on the same line.

Following the line with the string group number must be a line containing nothing but an opening curly bracket ('`{`'). There then must follow a series of sequentially numbered string lines, followed by an ending line that contains nothing but a closing curly bracket ('`}`').

6.3.5 String Lines

A string line may begin with one or more spaces or tabs, and must then contain a number expressed either in decimal digits or in hex digits preceded by "`0x`". The number must be followed by one or more spaces or tabs and then the string itself contained within curly brackets ('`{ }`'). Curly brackets are used to bound strings rather than quote characters so as to make it easier to include quote characters within the strings.

For a similar reason, the tilde ('`~`') is used in place of the backslash as an escape character; this makes it easier to represent Windows paths.

The following simple escape sequences are supported:

`~}`	==	A literal closing curly bracket character, not the string end.
`~[`	==	The beginning of a string inclusion specification.
`~~`	==	A single literal tilde character.

~r == A carriage return character (ASCII 13).
~n == A newline character (ASCII 10)
~t == A tab character (ASCII 9).

In addition, a tilde followed by four hex digits in succession is interpreted as the Unicode code point that the hex number represents. For example, ~2028 is interpreted as a Unicode line-separator character.

A tilde preceding any other character is interpreted simply as a tilde.

Including a String Within Another String
It is often useful to be able to build a string that contains one or more other strings. To include one string within another, insert its string-group number, a period, and its index number, all enclosed in square brackets and preceded by a tilde. Some examples are: ~[2.12], ~[0.0xA3], and ~[15.617].

Including a string from another file is not supported. Inclusions can be nested, but care must be taken not to create a circular inclusion chain (*e.g.* where a includes b which includes a).

For the sake of localization, however, be cautious about assembling sentences from fragments — every string in your application should be localizable independently of the rest.

6.3.6 Example String Source File

```
hshLib SampleFile eng-us   m1
#=================================================================
#  This is a sample hshLib library loadable string source file.
#  The hshStringBuild.exe utility can compile it into a binary
#  loadable file with a '.str' extension that hshLib can load,
#  either at startup or on demand.  This file would be compiled
#  into a binary file named SampleFile_eng-us.str.
#=================================================================

[0]+   # User Interface Strings (the plus sign indicates that
       # this string group is intended to be loaded into memory)
{
   0   {Press the "Esc" key to cancel}
   1   {First Name:}
   2   {Middle Initial:}
   3   {Last Name:}
   4   { Enter your name below }
   5   {C:\Program Files\Wigglewort\Nurds\}
   6   {The default path is:~r~n   "~[0.5]".}
}

[1]    # Error Messages
{
   0   {You must enter a value in the "~[0.1]" field!}
   1   {Warning: Only enter one character in the "~[0.2]" field.}
   2   {Only enter one word in the "~[0.3]" field!}
}
```

6.4 The hshDocBuild Tool

Builds a document written in XML using hshDocSchema into one of a number of output formats, including a Word file, an Adobe Acrobat .pdf file, HTML files, an HTML-Help .chm file, an HTML-Help .hxs file, and an `hshXml` distributed store file. SDK documentation for the library is published in this form.

6.5 The hshSdkCheck Tool

Checks C/C++ source code and either generates skeleton SDK reference pages using `hshDocSchema`, or compares signatures against existing reference pages of this sort, and reports signature differences.

7. Decimal Arithmetic

hshLib links to the most recent version of the decNumber C Library, which provides full support for IEEE-754-2008 numeric processing.

Quoting from Mike Cowlishaw's manual for version 3.68 of the decNumber C library:

> The decNumber library implements the **General Decimal Arithmetic Specification**[1] in ANSI C. This specification defines a decimal arithmetic which meets the requirements of commercial, financial, and human-oriented applications. It also matches the decimal arithmetic in the IEEE 754 Standard for Floating Point Arithmetic.[2]
>
> The library fully implements the specification, and hence supports integer, fixed-point, and floating-point decimal numbers directly, including infinite, NaN (Not a Number), and subnormal values. Both arbitrary-precision and fixed-size representations are supported.
>
> The arbitrary-precision code is optimized and tunable for common values (tens of digits) but can be used without alteration for up to a billion digits of precision and 9-digit exponents. It also provides functions for conversions between concrete representations of decimal numbers, including Packed BCD (4-bit Binary Coded Decimal) and the three primary IEEE 754 fixed-size formats of decimal floating-point (`decimal32`, `decimal64`, and `decimal128`).
>
> The three fixed-size formats are also supported by three modules called *decFloats* …which have an extensive set of functions that work directly from the formats and provide arithmetical, logical, and shifting operations, together with conversions to binary integers, Packed BCD, and 8-bit BCD. Most of the functions defined in IEEE 754 are included, together with other functions outside the scope of that standard but essential for a decimal-only language implementation.