# XML encoding techniques for storing XML data on memory limited (mobile) devices.

David A. Lee

Epocrates, Inc
727 Poplar Lane
Jasper, IN 47546
dlee@epocrates.com

## 1 Biography

David Lee has over 20 years experience in the software industry responsible for many major projects in small and large companies including Sun Microsystems, IBM, Centura Software (formerly Gupta.), Premenos, Epiphany (formerly RightPoint), WebGain. As senior member of the technical staff of Epocrates, Inc., Mr. Lee is responsible for managing data integration, storage, retrieval, and processing of clinical knowledge databases for the leading clinical information provider. Key career contributions include Real-time AIX OS extensions for optimizing transmission of real-time streaming video (IBM), secure encrypted EDI over internet email (Premenos), porting Centura Team Developer, a complex 4GL development system, from Win32 to Solaris (Gupta, Centura), optimizations of large Enterprise CRM systems (Epiphany), implementation of ecommerce systems for on-demand digital printing and CD replication (Nexstra).

## 2 Abstract

Epocrates is an industry leader in providing clinical references on mobile devices. Key constraints for storing content on handheld devices are small memory footprint and limited CPU power. Epocrates clinical content is 'rich data' requiring structural and presentation markup to be displayed, in addition some data sets are quite large pushing the limits of handheld devices. Some handheld operating systems (such as Palm/OS) have challenging performance characteristics for memory and storage formats, as well as limited display capabilities which require creative solutions for effective and efficient presentation of content. Commonly available tools for parsing and managing XML or HTML often perform miserably on handheld devices. Epocrates has selected "XText" encoding and compression to distribute clinical data on handheld devices.

This presentation is a case study of Epocrates' attempts to solve these problems. Initially we implemented a simplified "Rich Text" markup language as an attempt to bypass the size and performance problems with parsing XML on handheld devices. This early implementation lead to multiple problems and "dead ends" as the requirements evolved. Eventually a new XML based representation was developed which has the benefits of the concise representation and fast decoding of the simpler format, as well as the further

benefits of full structural representation, standards based markup and adaptability of XML.

This new encoding format (we call "XText") uses a hybrid of SAX Encoding and compression to produce a very efficient encoding of XML which is extremely fast and simple to decode. Presented are the real life lessons learned in the evolution of the format as well as techniques, examples, and sample code for producing an efficient encoding scheme for XML tailored to the constraints of handheld devices. Implementation is simple and can be coded in any language that can perform binary IO.

# 3 Introduction

## *3.1 Who Is Epocrates ?*

Epocrates is the industry leader in providing clinical references on handheld devices. We have 475,000 active subscribers. Our core products are a subscription based clinical publishing reference.

## *3.2 Common Terminology*

There are a few terms which are difficult to avoid, so they are defined here.

- **PDA** - "Personal Digital Assistant". In this paper refers to handheld devices, running a general purpose operating system, specifically the Palm/OS or Pocket PC OS (Win/CE).
- **Monograph** - From clinical terminology, refers to the information describing a single drug or preparation. In this paper refers more generally to the information describing a single drug, disease, lab test, preparation or other clinical entity. Roughly analogous to a section, or single reference work about a specific topic. A single data source usually provides a collection of Monographs of a particular type.
- **Palm** – A PDA device running the Palm/OS operating system
- **PPC** - A PDA device running Microsoft's Pocket PC operating system.
- **PDB** - From the Palm/OS terminology, a "PDB" is a "Palm Database". A very simple variable length record format with a single 16 bit key index.
- **Syncing** - The process of synchronizing a server's database with a PDA's database. New records on the server are inserted into the PDA database, modified records updated and records which no longer exist on the server are deleted from the PDA.
- **Sync Server** - The online servers in the Epocrates data center responsible for managing syncing to the PDA's.

# 4 Core Application

The Core Application for Epocrates is the Essentials product. *Epocrates* Essentials a multiplatform clinical reference tool which runs on handheld devices.


# 5 Background "The Problem Space"

The "problem" attempting to be solved is that of storing and parsing XML documents on handheld devices. Handheld devices have some special characteristics that can make storing and parsing full XML documents cumbersome, and sometimes impractical. CPU Speed, RAM availability and storage space are the main limitations. In addition there are issues of getting the data to the device in an efficient manor. It was determined experimentally that storing and parsing XML in serialized text format was impractical so an alternative needed to be found. There are many implementations of XML compression and encoding standards available which were tested including notably XMILL and XMLZIP. In addition, simply applying text compression (both zlib and

Huffman encoding) was prototyped. The problem with these technologies is that they appear to be targeted to a large computing environment. While they achieve good compression (which solves the space resource problem), they are complex and use significant RAM and CPU resources which make them non-ideal for handheld devices. Furthermore these technologies do not compress well for small data sets. For example, XMILL is documented as not compressing significantly for XML files under 20kbytes. Our XML data is usually in the 1-10 kbyte range per device document.

The problem with handheld devices is a composite of resource constraints. RAM (total and heap), CPU, Storage and IO need to be optimized together without excessive use of any one. Furthermore, in our system we need multiple language support due to the server running Java and the handheld running C++. Code simplicity is also a significant issue for both debugging complexity and code size issues on handhelds.

Thus we set out for a solution that would solve the key problems in our application and also work within the constraints handheld devices.

## *5.1 Characteristics of the application*

The characteristics of the application with respect to XML usage help drive the design requirements for XML processing on the device. The application runs on handheld devices which have very limited memory and typically slow CPU. Palm devices have as little as 8 MB RAM of which an application may have as little as 200k heap space available, PPC devices as little as 16MB total ram and 2MB heap. CPU speed varies from 100mhz to 400mhz but this can be very misleading in the case of Palm OS5 devices. These devices run an "emulation layer". A Palm OS5 device running at 400 MHz can run C application code as slow 100 times slower then native code (a 200mhz processor can run application code equivalent to as slow as a 2mhz processor). Typical is 5-20x slower then the same code running in the native processor mode. Furthermore UI response time is very important. A single page should not take longer then about 500ms to display before the interface seems sluggish and annoys users. Memory or storage card speed another factor. Some devices use a file system implemented in dynamic RAM while others use nonvolatile memory (flash). Access speed to data can vary dramatically depending on the device further complicating design decisions.

Synchronization speed is also a critical factor. On palm devices, synchronization speed is primarily limited to the number of records modified and secondary determined by the amount of data transferred. For example, updating 100 10k records is faster then updating 1000 1k records. On PPC and windows Smartphone's, synchronization speed is more closely related to the amount of data transferred, not the number of records. The synchronization protocol does not enable updating of partial records, they need to be updated all or nothing.

Depending on the size of the dataset, optimizing for CPU or memory usage may be more important and often a compromise must be reached.

## *5.2 Characteristics of XML data*

Epocrates Essentials has multiple sources of data that originate as XML data. Each data source has different unique characteristics. Some examples include

### 5.2.1 Disease reference data

This originates as a large XML document from a third party source. The XML document contains about 1000 “monographs” of 4-30k each totaling approximately 10 MB. This data is a mixture of structural and markup type elements. The schema is fairly complex, but no namespaces or Unicode data is used.

### 5.2.2 Message text

Message text data is created as XML documents using a custom schema designed for displaying on handheld devices. About 100 messages average 1k each are used. Primarily markup type elements (bold, header, lists, links etc). Special symbols are frequently required (such as Copyright, Trademark).

### 5.2.3 Clinical References articles

Clinical reference articles originate as Microsoft Word documents written by outside authors are translated into XML documents. 20 to 100 XML documents per article averaging 4k each. Primarily markup (mixed) element content with some structural elements.

## *5.3 Why XML on the device ? A brief history*

### 5.3.1 Proprietary Markup

Initial implementation for representing markup on the device considered HTML and XML. Both were rejected as reasonable implementations.

Early performance tests with XML showed it to be very slow to parse on slow devices (Such as the Palm M500). It was also considered to be too large for small memory footprint devices (2x to 4x space increase due to text serialized form for XML). It was also considered to be too complicated for our needs. The main reason, however, was human not technical. There were simply no "XML Advocates" on the development team, it was considered a foreign "unknown" technology.

HTML, likewise, was rejected. Like XML, it was complicated and slow to parse. Also HTML tied us into the HTML markup schema which was not a perfect fit for the type of markup our applications needed. For example, we needed syntax for Popup windows

and for inter-application jumps (not using URL's) which were difficult to model using HTML.

Finally an in-house markup language was invented. At first it seemed a good idea. The language was roughly "RTF Like" in that it was full of slashes and single letter escape sequences. An example looked like this:

**/Bbold/btext/L/Anc32/aJump to app/l/Hheading/h**

This worked well, for a while. The parser was small and efficient; the markup was simple and concise. However in time, the predictable happened ... requirements grew beyond the ability of the design to accommodate.

### 5.3.2 XML Revisited

At this point the future of the in-house markup was reconsidered. It had grown to have many problems. It was too complicated to parse, no one could understand the code or complex documents. Furthermore, the simplistic design, which was good at the beginning, was too difficult to extend. For example, adding tables with cells was very tricky to merge into the existing structure in a way that anyone could comprehend.

Therefore we reconsidered XML. If the performance and space issues could be resolved, the fundamental architecture that XML provides would solve many of our problems. Primarily it is well defined and extensible for future (as yet unknown) requirements

# 6 Architecture

The architecture chosen for our use case involved a combination of deign and implementation choices with an attempt to balance the constraints of the device. Much of this architecture is achieved by simplification of the XML sent to the device, and by offloading many of the 'traditional' XML processing tasks to the server.

## *6.1 Focus on Parsing, not creating*

In our use cases, XML is only delivered to the device, never constructed on the device. Thus all code for creating XML can be eliminated from the device. XML creation time is therefore not a factor in the design.

The final representation chosen is very simple, and could be created on the device efficiently if the XML follows the same design simplifications used for parsing.

## *6.2 Simplify Schemas*

Some feature of a fully compliant XML parser are simply never used in a vast majority of real world XML documents. These features can add extra code size, memory and CPU usage on the device if fully implemented on the device. A first step is to attempt to identify how to simplify your XML data to remove potentially expensive or complicated features. This could be done by simply recognizing that certain features are never used, or by transforming the source document into a simpler schema that guarantees the design choices. Things to look for to simplify and speed up your implementation include

### 6.2.1 Unicode or UTF8 support

Although the algorithm is simple, performance tests show that converting UTF8 to Unicode is a very expensive part of parsing XML data. Furthermore, not all devices support Unicode, for example Palm OS natively supports only 8 bit characters and a subset of the ISO8859-1 character set. Implement Unicode support not only requires parsing the UTF8 sequences, but also implementing multiple fonts and storing either Unicode data (as 16 bit words) or a sequence of 8 bit chars with font changing codes. If your can data can be simplified to not require Unicode text, or to only allow it in limited places, you will speed up your parser significantly.

If Unicode data is required, but infrequent, one technique can be to transform Unicode data on the server into special elements such as <SYMBOL value="1234"/>. This 'trick' moves encoding of Unicode to a slightly higher level then either using UTF8 (or UTF16) or XML entities. By doing this, the code that parses text need not be Unicode aware. Note that this only works within element text, not within attribute text.

In our implementation we choose a different strategy, which is to expand all entities in the server side parsing and use UTF8 as the encoded format. In addition, we allow a special SYMBOL element for common symbols supported on both palm and PPC (using a special font for palm). However we choose to use UTF8 only for element text, not attribute text. This choice was made because in the schemas we are using, attribute text is never used for any text values that require Unicode so it is a slight performance improvement to use ISO8859-1 encoding instead of UTF8 for attribute text.

### 6.2.2 Simplify your schema

The more elements and attributes the more complicated your device parser needs to be. Consider reducing the number of elements and attributes by using a schema on the device that is a simplification of the original document. Some XML features like namespaces may be able to be completely eliminated. Often XML data designed for exchange across organizations is very complex because it needs to encode a lot of information that cannot be assumed, and hence must be described explicitly. When deploying data to a handheld, the case is often vastly different. The server and device have a "trusted" relationship. Many assumptions can be made that can simplify processing. Consider that just because you start out with a complicated document, doesn't mean you have to

send that same document to the device. Rather consider what does the device really need at a bare minimum. Stripping out syntactic features (like namespaces) as well as data can greatly reduce the size and complexity of the XML.

## *6.3 Split parser into pieces*

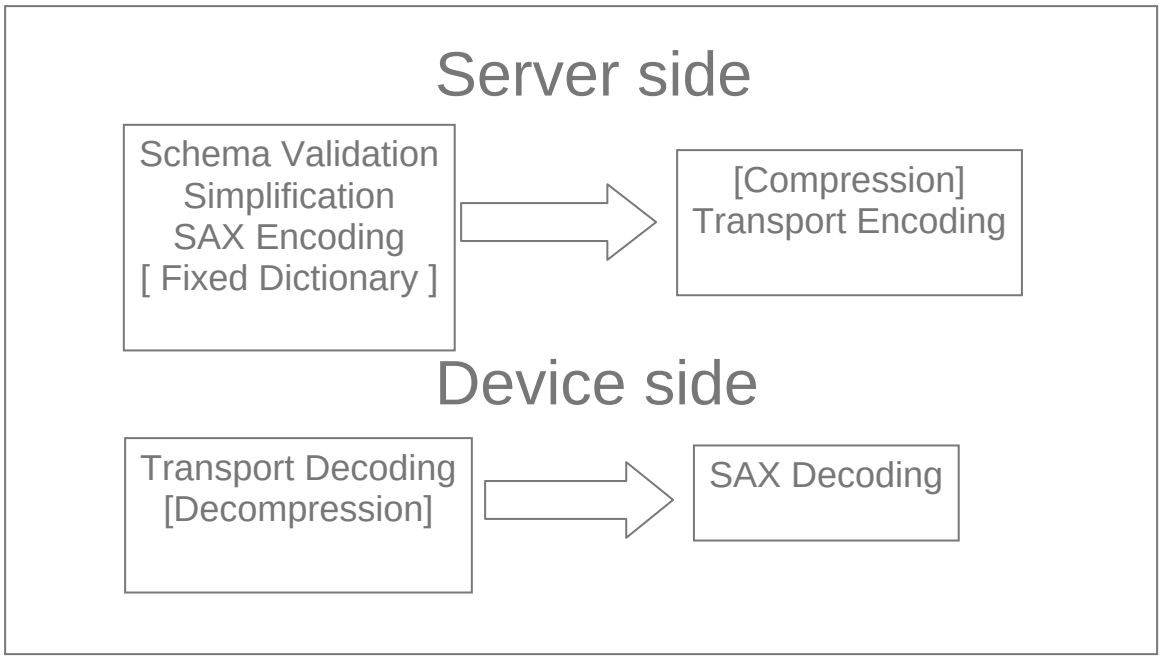
The biggest advantage of server side controlling the deployment of XML to a device is that from a data processing perspective, the pair (server plus device) can be viewed as a single logical entity. This means that much of the XML processing can be done on the server instead of the client. Validation, for example, both schema validation and application specific data validation can completely be done on the server. Interestingly, so can the actual lexical parsing of the XML text stream. There is no need for the device to actually handle the lexical parsing, which turns out to be the largest expense in SAX parsing model.

Consider the Server and Device to be single processing entity and choose to assign processing tasks to which part of this pair makes the most sense for your application.

For our system we use the following split of functionality across server and device

- Server
  Schema validation
  XML Simplification
  SAX parsing producing SAX events as a binary stream and an implied "Fixed Dictionary"
  Optional ZLib compression
  Packaging for transport to device

- Device
  Decoding the transport record format
  Optional decompression
  SAX Decoding of binary SAX event stream producing SAX callbacks




### *6.4 Efficient "SAX Encoding"*

From our performance tests we identified XML parsing (using SAX) to be a very expensive part of the total XML processing so we moved the SAX parsing stage into the server. This required some way of representing the output of SAX parsing as data. SAX parsing traditionally produces a set of language callbacks to represent SAX events.

We chose a common technique called "SAX Encoding" which produces a simple byte stream from SAX callbacks. We extended this technique to support an optional "Fixed Dictionary", which means that the mapping of element and attribute names to their encoded values is not explicit in the data stream.

### *6.5 Optional Compression*

If Space is more important then speed then compression of the encoded SAX stream is an option. We used the open source "ZLib" to compress the encoded byte stream because it is open source and a reasonable compromise of speed and compression ability. Being an open source version is available in pure "C" code it ports well to both Palm and PPC devices, as well as commonly available implementations in Java for server side. We also tested several other compression methods including 'home built' compression technologies.

In our uses, the usefulness of compression is mixed. Due to header and dictionary information contained in the output, compression ratios are very small and sometimes negative for small input sizes. As a rough 'rule of thumb' we found compression only useful when the input size is over 1kbyte. Furthermore, decompression performance is very expensive on slow devices. We found compression only useful for cases where

single documents are fairly large, there are many of them, and application use cases are where only one (or very few) need to be decompressed per UI "event" (screen display).

### *6.6 Pack for transport*

The raw data stream from the output of the SAX Encoding process needs to be packaged for transport to the device. Device databases usually have hard limits of record sizes which cant be exceeded. Some application constraints may need to be applied, (smaller record sizes, encoding of binary data, incorporation within a containing data structure of stream). Because the validation and parsing are done on the server side, some form of error checking, checksum, start and end markers etc may need to be incorporated to protect against transmission or application errors.

If compression of the SAX encoded data is not performed, some type of transport layer compression may be desired.

Finally the data needs to be deployed to the device, usually as part of a "Synchronization" but possibly as part of the application install, or via other methods such as SD Cards , RPC or web services.

# 7 Implementation

This section describes the actual implementation used for the prototype described in this paper, and used for the tests performance measurements presented. The actual implementation used in our production servers and device code is based on this implementation, but have been extended somewhat to accommodate application specific needs and release quality coding standards. The same general results are achieved in the production system, so it is a useful simplification to describe the prototype system as one which can be used to grow into a production quality system.

### *7.1 Server side implementation*

#### 7.1.1 Fixed ('offline') Dictionary

The "Dictionary" is a simple association between a text "name" and an integer encoded "value". At minimum these are used to encode element and attribute names, but can also be used to encode a known fixed set of enumerated values. An 'inline' dictionary is a dictionary which is embedded in the encoded data stream so that the parser can reconstruct the text representation. An 'offline' (or "fixed") dictionary is an implied dictionary. By 'implied' this means the encoder and decoder (parser) have a built in dictionary that does not need to be transmitted along with the data. This allows a significant compression of data for even very small XML files because the element and attribute text values are not included in the data.

Our implementation supports both an 'inline' and 'offline' dictionary, however for this case study we focus only on 'offline' dictionaries. This is because the core of the design focuses on a tightly coupled encoder/decoder system. It is entirely realistic that the

schema is known by both the server and handheld device, and in fact in our applications we currently use only one schema ("XText"). Furthermore, for our application, the actual text mappings of the element and attribute names are never actually needed on the device, only their meaning. So for the device side, the "Dictionary" is actually just a set of #define values used to identify the element and attribute encodings, and on the server (encoded) side the Dictionary is a static java structure. We also extend the concept of a dictionary to encompass enumerated attribute values. For example <TABLE align="right"> .. the attribute value for **align** may be one of only a few enumerated strings. By encoding enumerated values as integers the same way as element and attribute names enhanced compression and parsing speed enhancements can be implemented with minimal effort.

These encoding techniques, however, can be easily extended to supply an inline dictionary as part of the data stream, at the expense of additional data size and a small performance hit.

## *7.2 Server – SAX Parser*

The server component is the encoder. That is, it takes the text serialization format of XML and produces the encoded binary representation. This is implemented with the following components

- Schema Validation
  The XML is fully validated either via DTD or schema. This is typically done as part of the SAX parsing component as part of the Java XML SDK, but can also be done seperately.

- Simplification
  Complex XML constructs can be simplified by the server to optimize device side processing. For example we translate <SYMBOL> elements into Unicode representations to avoid lookups on the devices.

- SAX Encoding
  SAX Events (callbacks) are encoded into a simple byte stream which is very efficient to parse on the device.

## *7.3 SAX Encoding Simplified ... (server)*

The SAX Encoding is done on the server in Java. This makes use of a fixed (offline) dictionary and some very simple structural markers. Encoding is done using a SAX parser, then encoding each SAX event as a sequence of bytes. The following simplified example demonstrates the bytes produced for each SAX callback. A few optimizations are made to handle special cases of elements with no attributes which are encoded as a single byte, and attributes whose contents correspond to enumerated data, which are encoded as 2 bytes. These optimizations are very simple to parse and provide some

encoding compression at minimal cost or complication, but could be easily eliminated for simplicity. In our application, Unicode is required so all character data (but not attribute values) are encoded in UTF8. However this turns out to be the largest single performance factor in decoding so if possible should be considered for removal.

```
static  int kEXT_START_DOC    = 0xFA ;
static  int kEXT_END_DOC      = 0xFB ;
static  int kSTART_ELEM   = 0xFC ;
static  int kEND_ELEM     = 0xFD ;
static  int kCHARACTERS   = 0xFE ;

startDocument()
          [kSTART_DOC]
startElement("name", null,null)
          [ELEM_ID]
startElement( "name" , attrs , nattr )
     [kSTART_ELEM][ELEM_ID][NATTR]
     [ATTR_ID]”value”\0[ATTR_ID|0x80][ENUM_ID] ...
characters( data , count )
          [kCHARACTERS]”string”\0
```

## *7.4 SAX Decoding Simplified ... (client)*

On the device, decoding is very simple. Given a pointer to the encoded data, it is easily parsed. Enhancements could include better error checking, incremental parsing, and reading from a file or IO source instead of directly from memory. No look ahead or look-behind is required for parsing. No copying of data is required to produce the callback arguments which are based on SAX callbacks but modified to fit the parser data structures without changing the data. Since the element and attribute names are encoded as integers, they can be passed to the SAX callbacks directly as integers and never need to be translated into text (unless the client code needs text representations). In our application UTF8 is used for all character data. Not shown are details for decoding enumerated attribute values, UTF8 decoding and access to the offline dictionary.

```
while( p < end  ){
   int c = *p++;
   switch( c ) {
   case   kSTART_DOC :
       startDocument(); break;
   case   kCHARACTERS :
       characters( p ) ; break ;
   case   kSTART_ELEM :
```

```
            // start element
            ...
    default :
         startElement( c ) ;
}
```

# 8 Test Results

The following test results were obtained by taking an example XML file of 12473 bytes in text format. This XML file is largely text and markup data with some element structure and is typical of our application data. The same application is run on 3 devices and using 4 test cases. The code is written in C and is identical on all 3 devices (2 palm and one PPC devices), except for UI component which is only used to start the tests and display the results and do not effect the measurements. Test times are averaged over several runs and use the highest precision timers available on the device. The XML data is embedded within the application as a initialized C array to eliminate any file system IO as a contributor. The applications are simple native applications on each device with no special optimization beyond the normal compiler options for a "release" build. In the "XML" test cases the EXPAT parser, written in C, is used. EXPAT is a SAX based parser C parser and is considered a good example of industry standard efficient SAX XML parser. For compression, a static library port of the C source public domain "ZLIB" compression is used. The "XText" test cases are encoded using the techniques described in this paper, and optionally compressed with the same ZLIB compression code.

## *8.1 Test Cases*

| | |
|---|---|
| XML | Text XML<br>parsed with EXPAT |
| XML<br>Compressed | Text XML compressed with GZIP<br>Uncompressed then parsed with EXPAT |
| XText | XML SAX Encoded fixed dictionary<br>Parsed with C++ SAX Decoder |
| XText<br>Compressed | XML SAX Encoded and compressed with GZIP<br>Uncompressed then parsed with C++ SAX Decoder |

## *8.2 Test Devices*

The test devices used are standard "off the shelf" handheld devices, two Palm devices and one Pocket PC device.

| | |
|---|---|
| TE | Palm - Tungsten E<br>126 MHz |
| M500 | Palm - M500<br>33 MHz |
| PPC | Pocket PC - HP IPAQ 4150<br>400 mhz |

## 8.3 Sample Doc (partial) – 12kbytes

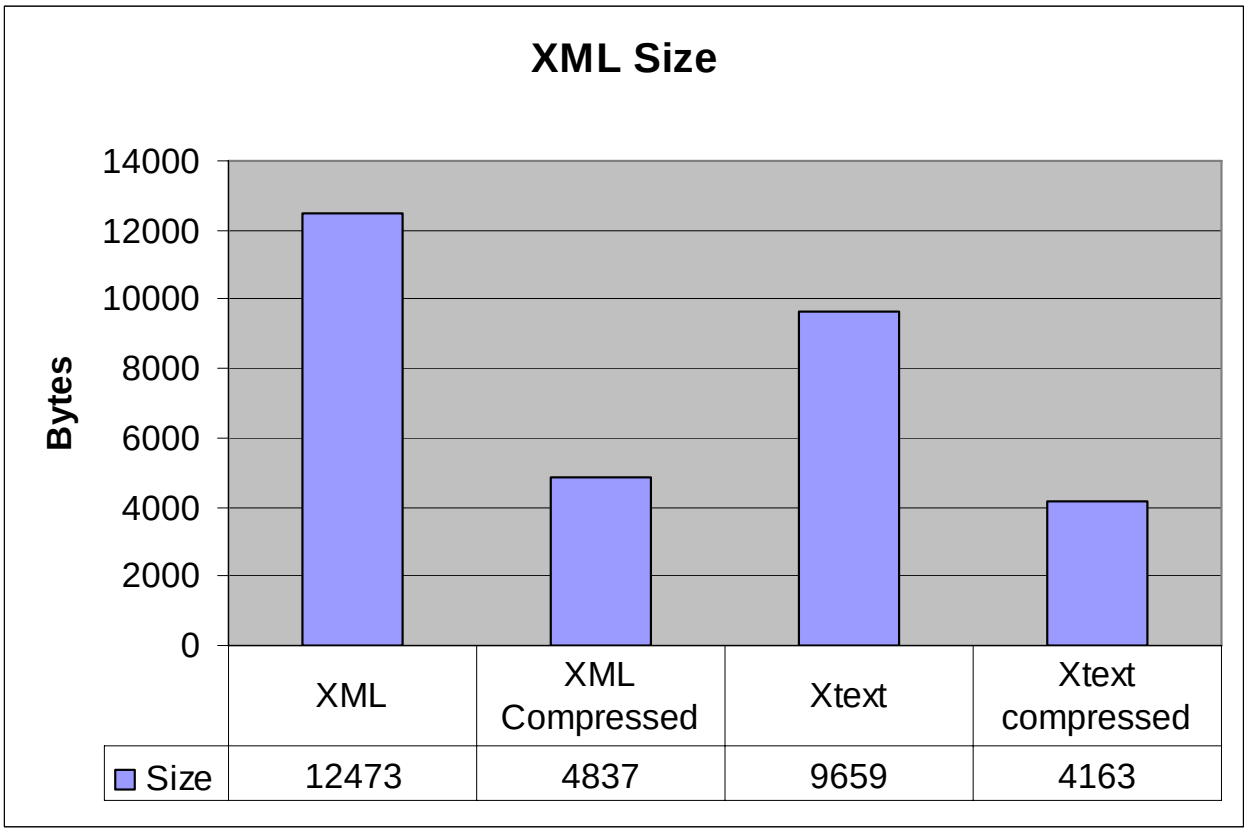
The sample document is a single topic from the "5 minute clinical consult" and looks like the following.

<book>
<long_topic>
<id>TP0002</id>
<name>Abruptio placentae</name>
<content>
<basics>
<description> Premature separation of otherwise normally implanted placenta. Sher's grades:
1: Minimal or no bleeding; detected as retroplacental clot after delivery of viable fetus
2: Viable fetus with bleeding and tender irritable uterus
3: Type A with dead fetus and no coagulopathy; type B with dead fetus and coagulopathy (about 30% of grade 3's)
<systems_affected>
<system>Cardiovascular</system>
<system>Reproductive</system>
</systems_affected>
<genetics> N/A</genetics>
**.....** ***12 k bytes***

## 8.4 Size comparison of XML Encoding

This graph shows the resulting sizes of the XML document before compression, using ZLib compression, using XText encoding, and XText encoding plus ZLIB

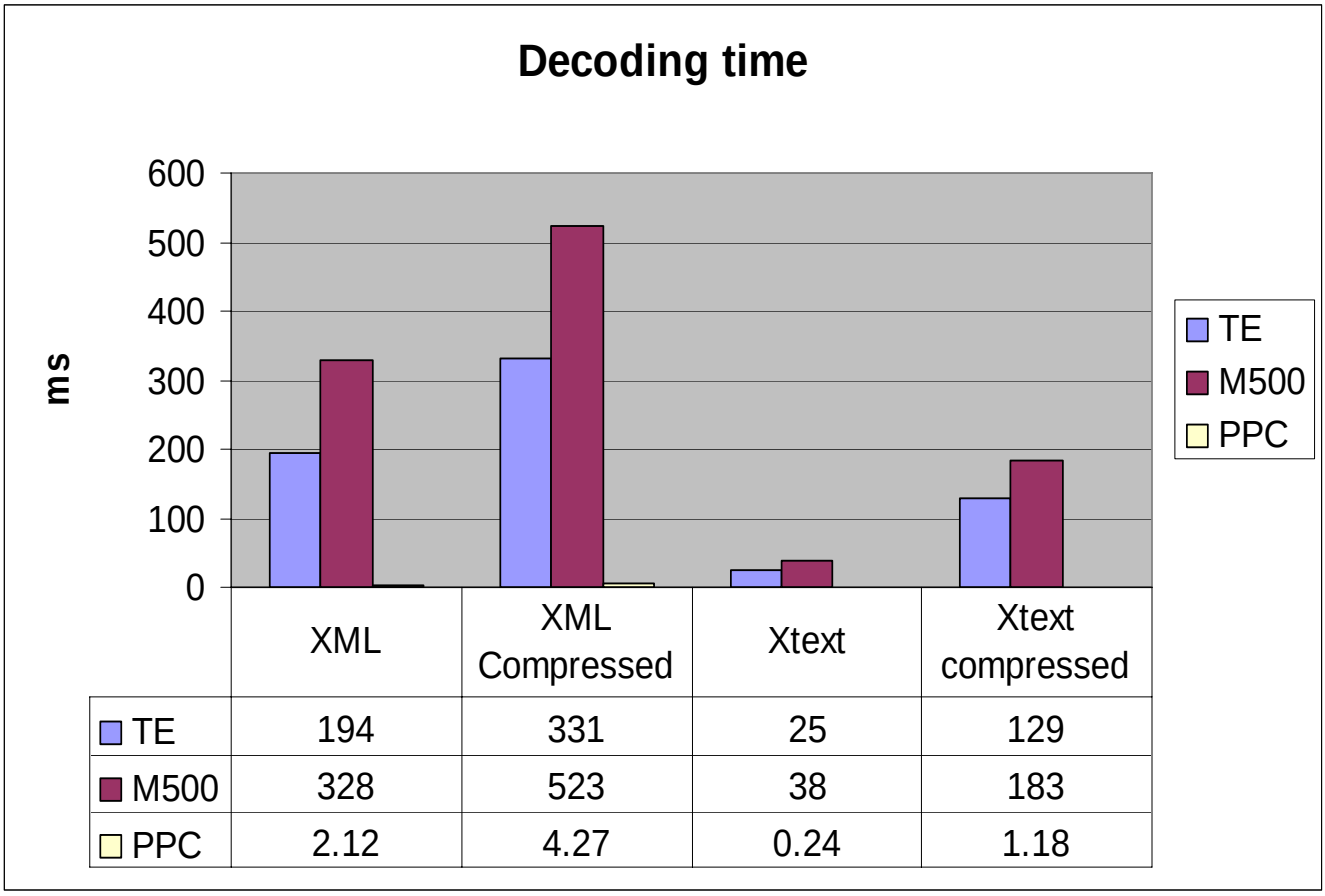
compression.



## *8.5 Parsing Performance*

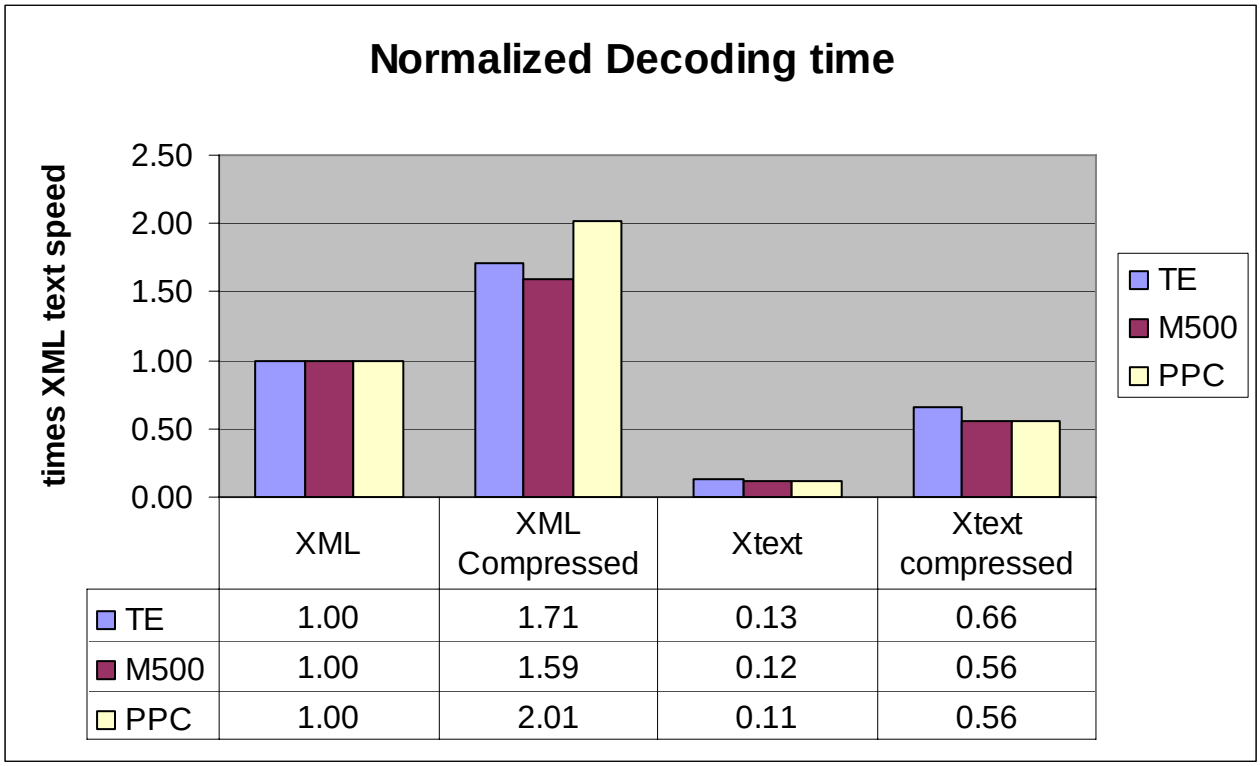
The first graph shows device parser performance as a measurement of decoding time in milliseconds. XText parsing without compression is about 10x faster then raw XML parsing. Note the absolute speed on the TE and M500 devices, this approches the limit for acceptable UI performance *Not Including* any time for processing (displaying) the data. Adding in the cost of formatting and displaying the data the time becomes unacceptable. Even with compressed XText the parsing time is non ideal, suggesting that compression only be used when absolutely necessary.

Of side interest is that the TE times are almost 100x that of the PPC device even though the PPC CPU is only 3x faster. (126MHZ vs 400MHZ). This is due to the emulation layer on palm OS5 devices.




| | XML | XML Compressed | Xtext | Xtext compressed |
|---|---|---|---|---|
| TE | 194 | 331 | 25 | 129 |
| M500 | 328 | 523 | 38 | 183 |
| PPC | 2.12 | 4.27 | 0.24 | 1.18 |

The following is a normalized view of the same data. Each device is normalized to 1.0 being the speed for processing the text XML with EXPAT on that device




| | XML | XML Compressed | Xtext | Xtext compressed |
|---|---|---|---|---|
| TE | 1.00 | 1.71 | 0.13 | 0.66 |
| M500 | 1.00 | 1.59 | 0.12 | 0.56 |
| PPC | 1.00 | 2.01 | 0.11 | 0.56 |

# 9 References

This project borrowed greatly from ideas and implementation of others including the following.


- ZLIB compression software
  http://www.zlib.net/

- XMILL xml compression
  http://sourceforge.net/projects/xmill/

- XMLZip xml compressor
  http://www.w3.org/2003/08/binary-interchange-workshop/31-oracle-BinaryXML_pos.htm