Energy-Efficient Transaction Serialization

Daniel Evans

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science

at

Seidenberg School of Computer Science and Information Systems
Pace University

March 2020

We hereby certify that this dissertation, submitted by Daniel Evans, satisfies the dissertation requirements for the degree of Doctor of Philosophy in Computer Science and has been approved.

Lixin Tao — 3/7/2020

Dr. Lixin Tao — Date
Chairperson of Dissertation Committee

[illegible] — 3/7/2020

Dr. Charles Tappert — Date
Dissertation Committee Member

Ronald Frank — 3/7/2020

Dr. Ronald Frank — Date
Dissertation Committee Member

Ivan G. Seidenberg School of Computer Science and Information Systems
Pace University 2020

Abstract

Energy-Efficient Transaction Serialization

Daniel Evans

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science

Seidenberg School of Computer Science and Information Systems
Pace University

March 2020

This dissertation presents two designs, the Transaction Serial Format (TSF) and the Transaction Array Model (TAM). Together, they provide full, efficient, transaction serialization facilities for devices with limited onboard energy, such as those in an Internet of Things (IoT) network. TSF provides a compact, non-parsed, format that requires minimal processing for transaction deserialization. TAM provides an internal data structure that needs minimal dynamic storage and directly uses the elements of TSF. The simple lexical units of TSF do not require parsing. The lexical units contain enough information to allocate the internal TAM data structure efficiently. TSF generality is equivalent to XML and JSON. TSF represents any XML document or JSON object without loss of information, including whitespace. The XML equivalence provides a foundation for the performance comparisons. A performance comparison of a C reference implementation of TSF and TAM to the popular Expat XML library, also written in C, shows that TSF reduces deserialization processor time by more than 80%.

Acknowledgements

This dissertation brings to completion a process started in 1995 when I was first admitted to the Computer Science Ph.D. program at the State University of New York at Stony Brook. My employer at the time, Periphonics, was supportive, providing tuition reimbursements and flexible hours, and I am grateful for that support. But, as Robert Burns wrote, "The best laid schemes o' mice an' men gang aft agley". Periphonics was bought by Nortel in 2000, and the downsizing began. Over the course of the next three years, over six hundred Long Island jobs were lost. Nortel ultimately went bankrupt, but that is a story for B-school study.

The hiatus in my Ph.D pursuit lasted until 2016 when I met Dr. Lixin Tao, and he admitted me to the Pace University DPS program. He was continually supportive, and allowed me to transfer to the Pace Computer Science Ph.D. program in 2018. The transfer would not have been possible without the support of Dr. Paul Benjamin, the director of the Ph.D. program, who reviewed my work at Stony Brook and allowed some transfers. Dr. Tao continued as my dissertation advisor after the transfer, provided valuable guidance, and assisting with publication. My work would not have been possible without his aid and encouragement.

In addition to Dr. Tao, the members of my committee, Dr. Charles Tappert and Dr. Ronald Frank, have always been willing to help on short notice. Dr. Tappert and Dr. Frank created the CS837 Quantum Computing course at Pace, introducing me to a topic with new research possibilities. Dr. Tappert mentored me on several special projects. Dr. Frank was willing to attend my defense by video, as he was on leave at the time. I appreciate all their support.

Throughout my quixotic pursuit, my wife Susan, and my two sons John and Mark, have always been encouraging, and never complained about the time taken from other activities.

As a Math major and senior at New Mexico State University, I was first exposed to computers when the brand new Computer Science department, and its chairman, Dr. J. Mack Adams, made an IBM 1130 available to all students after hours. Anyone could write Fortran or 1130 Assembler programs, punch them on cards, and read them into the computer late at night, as long as you got there before the EE's who would run power network simulations that seemed to take hours. New Mexico State

was an ideal university. It was large enough to have something for everyone. It had cattle pens one one side of the campus and a small aircraft runway on the other side. In addition to my Math requirements, I was able to take Electricity for Non-Majors, Machine Tool Theory and Use, Mathematical Physics, History of the English Language, Theater Arts, Philosophy, Sociology, Modern American Fiction (taught by Mark Medoff, the author of "Children of a Lesser God"), Archery, Classical Music, Money and Banking, and Astronomy (taught by Clyde Tombaugh, the discover of Pluto). Most of all, NMSU provided the Physical Science Laboratory for sponsored research. PSL in turn hired co-ops, students who worked for two semesters in the field, then returned for two semesters at the university, in a continuous work-study rotation. I was one of the co-ops, and the program allowed me to pay for and earn my BS degree. It also sent me to Anchorage, Shemya, Knob Noster, Kahului, Upernavik, and Godthaab. NMSU, PSL, and the professors I met there, laid the foundation for person I have become, and I am forever grateful.

Finally, I must mention the context within which this degree has been completed. I am reminded of the Marquez title "Love in the Time of Cholera". My defense was two weeks before the corona virus, Covid-19, began its relentless march on New York City. It has now forced Pace, and all New York universities, to shift completely to remote classes held through video connections. Pace has currently postponed graduation ceremonies. The future, particularly for older members of society, is uncertain. Still, I look back on my work since entering Pace with enjoyment, and I hope to have the opportunity use the results of my education.

Daniel Evans

March 2020

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Introduction

The rapid development of the Internet of Things (IoT) has renewed interest in transaction processing efficiency. The paper ”Energy Efficiency: A New Concern for Application Software Developers” [1] recently detailed the interaction between energy issues and software in IoT and mobile devices: ”. . . wasteful, poorly optimized software can deplete a device's battery much faster than necessary.” The initial energy supply of many IoT sensor devices limits their deployment lifetime. These devices sense information and transmit the results of sensing. They receive instruction from remote controllers. They continually serialize and deserialize transactions to and from a communications medium. Any reduction in processor time used by transaction serialization/deserialization contributes to an increase of the deployed lifetime of an IoT device. This work describes the Transaction Serial Format (TSF), whose primary goal is to use as little energy as possible to perform the serialization/deserialization tasks. The Transaction Array Model (TAM) supports the efficiency of TSF. Although a dynamic data structure, TAM directly integrates information from the TSF format.

1.2 Serialization Overview

The need for serialization and deserialization, converting data to and from serial media such as communications networks and long-term storage devices, has spawned many designs. Wikipedia lists 35 different formats in its ”Comparison of Data Serialization Formats” page. The differences are many and the reasons for development of each are not always clear. However, they have a common feature: the need to preserve data structure. The data structuring facilities of random access memory are not applicable when accessing data sequentially from beginning to end one byte at a time.

Serialization formats generally divide into two categories: formats specific to an application, and general formats intended for use by any application. Examples of application formats are Apache Avro [2] and Apache Parquet [3] both designed for the Hadoop parallel processing system. Binary JSON (BSON) [4] adds binary representations and extensibility to JSON-structured data in the MongoDB database system. Also in this category are language-specific serializations such as Java Object Serialization [5], Python Pickle [6], and Perl's DataDumper module [7], the first of many PERL serialization modules. Remote Procedure Invocation produced a number of formats designed to marshal parameters for a remote procedure call and to return the results. The Common Object Request Broker Architecture (CORBA) specifies the InterORB Protocol [8] for communication between clients and object request brokers. The Java library provides the Remote Method Invocation (Java RMI) [9] for Java programs to invoke remote methods available on any machine running an RMI server. D-Bus [10] is designed for both interprocess control on a local machine and remote invocation. Apache Thrift [11] is a design for cross-platform RPC. XML-based RPC serialization came from Microsoft in the form of XML-RPC [12], the ancestor to the World Wide Web Consortium (W3C) SOAP standard [13].

General serialization formats are for any application, and TSF is in this category.

In addition to the basic requirement of preserving data structure, many formats incorporate additional features that:

- reduce the size of serialized data to minimize data transmitted
- use an external schema to describe the serial format(s)
- work with multiple character encodings, such as UTF-8, UCS-2, and UCS-4
- use only characters recognized by a text editor so that the serialized data is human-readable and easily changed
- address some additional need, such as machine-independent data definitions that can be exchanged by machines of different architecture, or the restriction to 7-bit or 8-bit clean so that data can be transmitted through gateways and across networks with differing characteristics

The following examines serialization formats that incorporate one or more of these features, and note how TSF compares to them.

Data compaction was an early issue for the telecommunications industry when transmission speeds were much slower than they are today. The International Telecommunications Union (ITU), the standards body for international telecommunications, released in 1984, as part of CCITT-X.409, the Abstract Syntax Notation, version 1 (ASN.1), an interface description language (IDL). In 1988, ASN.1 became a separate standard, X.208 [14]. ASN.1 has gone through several revisions and is currently at Revision 5, but has not lost its original name, ASN.1.

ASN.1 is a declarative language (also called a schema language) for describing messages as general structures of arbitrary data types. A schema is a description of the serialization format, which both sender and receiver have. An ASN.1 schema declares data types first, and then declares messages as structured sequences of previously declared data types. ASN.1 provides an extensive set of built-in type constructors as a foundation for complex types. Every defined type has an unambiguous serial encoding, commonly known as type-length-value encoding. ASN.1's Basic Encoding Rules (BER define the rules for serializing data using bit- and byte- aligned fields. Initial implementations of ASN.1 compiled a schema of data types and message declarations into source code in a language such a C, providing both encoding and decoding functions. The generated encoding and decoding functions, specific to the described messages, could in turn be included as source code, or compiled and linked as a library, with application programs that sent and received the messages. ASN.1 found wide use within the telecommunications industry. A subset of ASN.1 became the Simple Network Management Protocol's language for describing SNMP's MIBs (Management Information Base)[15], which are abstract descriptions of network devices subject to management by the protocol. ASN.1 also found use in applications that use some form of the X.500 series of standards from the CCITT, such as the exchange of cryptographic metadata with X.509 certificates [16].

ASN.1 uses type codes and a schema to define and preserve structure. Another common structure-preservation technique is the use of delimiters that signify the beginning and end of data and provide for data nesting. XML [17] and JSON [18], two of the most popular formats in use today, both use this technique. XML uses named parentheses, called open and close tags, to delimit and nest data. JSON uses two types of structure delimiters, the characters "[" and "]" for arrays, and the characters "" and "" for collections, which are named sequences of data. Collections synonyms are associative arrays, maps, hashes, or dictionaries in other computer languages. The YAML format [19] uses indentation and the natural delimiting provided by line end characters to preserve data structuring. The lines in the YAML format have additional syntax, which may optionally include "flow" formats, close to the delimited design of JSON. YAML is also user editable with any general text editor program. Another delimited, editable format is "s-Expressions", originally designed by John McCarthy, the inventor of Lisp, and described in the Internet Memo [20] by Ronald Rivest. Like Lisp, it uses left and right parenthesis as delimiters and nests delimited data to provide structure. TSF does not use delimiters to define serialized structure, but takes an approach that is closer to ASN.1, although without a schema.

Although editing is not a design goal, TSF is editable with a normal text editor, if done carefully to update lengths after adding or deleting characters.

The concern for compact representations appears in many early serialization formats, coincident with communications speeds slower than today's. ASN.1's Basic Encoding Rules are the prototypical example. Later formats also dealt with compact representations. XML representation efficiency became an issue soon after XML use became widespread. The W3C chartered the XML Binary Characterization (XBC) Working Group and the Efficient XML Interchange (EXI) Working Groups in 2004. Both groups worked in the area of efficient representation of XML. The XBCWG produced the first draft of "XML Binary Characterization Properties" [21] in late 2004. The EXIWG produced its first draft in mid-2007 [22].

The work of these groups has always been informed by the desire to preserve as much of the primary XML specification as possible, and to be aware of XML schema definitions when they exist.

> "EXI is schema "informed", meaning that it can utilize available schema information to improve compactness and performance, but does not depend on accurate, complete or current schemas to work."[22]

The working group ultimately produced a very large code implementation of the specification and made it publicly available, but it does not appear to have been widely used[1].

In 2012, the W3C-chartered MicroXML Community Group produced the MicroXML specification [23]. The justification for MicroXML is at the beginning of the specification document.

[1]The entire OpenEXI package download is 355 Mb. Compare this to the total download size of 84Mb for the Expat XML parser.

> ”MicroXML is a subset of XML intended for use in contexts where full XML is, or is perceived to be, too large and complex. It has been designed to complement rather than replace XML, JSON and HTML. Like XML, it is a general format for making use of markup vocabularies rather than a specific markup vocabulary like HTML.”

MicroXML simplified XML by, among other things, eliminating DOCTYPE’s, namespaces, processing instructions, CDATA sections, and character set options. However, as MicroXML is a subset of XML, no alternate serialization format was proposed.

Improved line speeds resulted in less attention to compact representations, but the rise of mobile devices with their bandwidth limitations kept compact representations a concern, as evidenced by the recent Compact Binary Object Representation, described in 2013 in RFC 7049 as ”. . . a data format whose design goals include the possibility of extremely small code size, fairly small message size, . . . ” [24]. However, alternate compaction techniques have found wider commercial use. Web servers that download JSON have preprocessed the files to remove non-syntactic whitespace, and sometimes renaming all the variables to minimize their length, which has the effect of obscuring the source code. The files can also be processed by a standard compression algorithm, as all browsers have the ability to process several compression formats. In TSF, the elimination of redundancies yields some compaction, but compaction is not a primary motivation for the format. A TSF message is usually slightly smaller than its JSON equivalent. Any standard compression technique can further reduce the size.

Schema-based serialization formats tend to be more compact as they can eliminate some or all type information from the serialization by maintaining it in a separate schema, generally written in a custom interface description language (IDL). ASN.1 again is the prototypical example. A more recent example is Google’s Protocol Buffers [25]. Once written, a Protobuf schema compiler generates source code for any supported language. The generated code then is included in any program that uses the serialization described by the schema. Flatbuffers [26], similar to Protobuf, can use both its own IDL and that of Protobuf. TSF is not a schema-based serialization, but it does use a technique called "zero-copy deserialization", also used by Flatbuffers, to reduce the number of memory allocations required to construct the internal representation of a deserialized object.

Differences in machine architecture have motivated some serial formats. The External Data Representation first defined in RFC 1832 in 1995 and subsequently updated eleven years later in RFC 4502 [27] provides serial representations for standard binary data types such as signed and unsigned integers, 64-bit integers, called hyper integers, floating point values, enumerations, fixed length arrays, and more. However, the popularity of character-based serializations seems to have provided an alternate, simpler way to handle architecture differences, as conversions from character forms to internal data types are available on any machine and in every language. TSF delegates the data format definitions to the application.

Finally, there are two serialization formats that have some coincidental similarity to TSF. Bencoding, part of the BitTorrent specification[28], serializes string data with a count followed by a delimiter preceding a data field, similar to TSF. But Bencoding defines separate structures for lists (arrays) and dictionaries (hashes) and does not identify occurrences. It also imposes an order on dictionary strings. Binn[29], a more recent design, uses zero-copying and counts for structures, like TSF, but defines types using bit fields and varying length binary fields for data lengths and container lengths. Unlike TSF, it has a number of hard-coded data types, and three containers types, but it does make a provision for user types.

There are two serialization formats that have some coincidental similarity to TSF. Bencoding, part of the BitTorrent specification [28], serializes string data with a count followed by a delimiter preceding a data field, similar to TSF. However, Bencoding defines separate structures for lists (arrays) and dictionaries (hashes) and does not identify occurrences. It also imposes an order on dictionary strings. Binn [29], a more recent design, uses zero-copying and counts for structures, like TSF, but defines types using bit fields and varying length binary fields for data lengths and container lengths. Unlike TSF, it has a number of hard-coded data types, and three containers types, but it does make a provision for user types, similar to TSF.

1.3 TSF Design Objectives

Now that TSF has been differentiated by what it isn't, we following the example set by CBOR[24] in RFC 7049 and list the design objectives of the Transaction Serial Format (TSF), in order of importance.

- The format must be efficiently deserialized.
  - Deserialization should not require more than a few pages of C code.
  - The format must not require parsing for deserialization.
  - The format must not use data units smaller than a byte.
  - The format must avoid forcing data into specific bit representations.
  - Deserialization should not require a schema. Data types should be implied in the format.
- The design must be general enough to encode popular data formats such as XML and JSON, as well as other common Internet formats.
  - The format must support named and unnamed sequences, such as arrayed data.
  - The format must support named and unnamed collections, such as hashes.
  - Data structuring must support collections of sequences, and sequences of collections.
  - TSF is not a streaming format; it does not support unspecified data lengths or occurrences.

- The internal (in memory) representation of a deserialized transaction is integral to efficiency of the design and must be easily created.
  - Dynamic memory allocations should be minimized.
  - Data should not need to be copied into the internal representation.
- The format should support user data typing to allow it to be adapted to specific user applications.
- The internal API must include a standard serialization.
  - This is a common sense, ease of use objective.
  - Contrast this with XML, which has no serialization API.
- The serial representation should be reasonably compact.
  - Compactness is not a primary goal, but the equivalent JSON size is a compactness target.
  - Redundancies such as end tags and extra delimiters should be avoided.

The primary goal of efficiency is a result of the simplicity of the TSF design. Figure 1.1 shows the complete lexical structure of TSF.

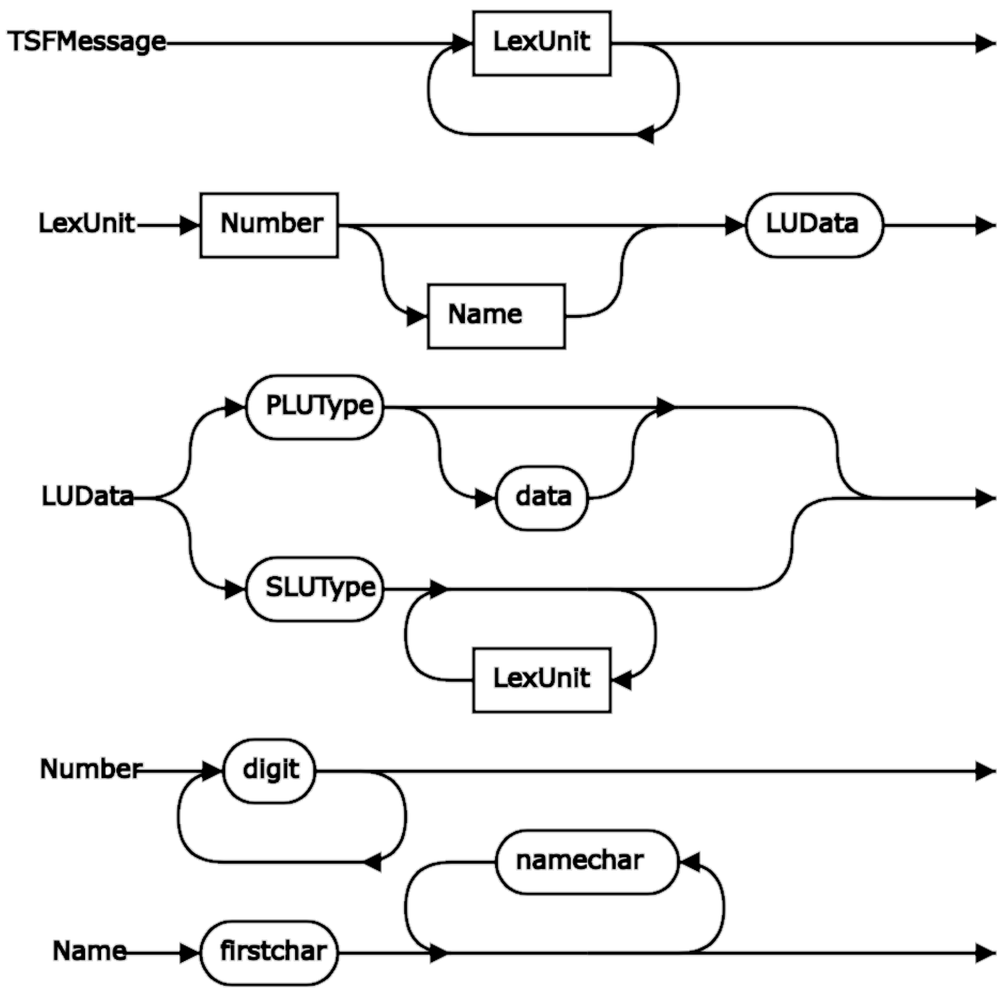

Figure 1.1: TSF Complete Lexical Structure

The following sections examine the design in detail and discuss the performance impact of TSF and its sister memory representation, TAM.

1.4 Summary

Chapter 1 (this chapter) introduces TSF and differentiates it from related works on serialization. It presents the goals that have guided the TSF design and summarizes the complete lexical structure in Figure 1.1

Chapter 2 presents the detailed design of TSF and illustrates TSF message deserialization without parsing. The lexical processing is limited to recognizing a sequence of digit characters as a number, and scanning to the end of names. The lexical grammar that underlies the TSF design illustrates the simple one-character lookahead requirement. An informal proof shows the grammar produces TSF lexical units.

Chapter 3 discusses the application of TSF to JSON. It generalizes the definition of a TSF name, and shows that it is possible to encode JSON data in equivalent TSF.

Chapter 4 presents TAM, and discusses the efficiency considerations which guide the design.

Chapter 5 presents the functions needed to implement deserialization, and presents formal proofs of correctness for these functions. The proofs are based on Program Logic.

Chapter 6 applies TSF to XML. applies TSF to XML. It shows that XML documents are a series of lexical units when represented in TSF, so that deserialization does not involve parsing.

Chapter 7 presents a comparison of the performance of deserialization of XML documents using the Expat C library implementations of the XML SAX parser, and the C/C++ implementation of TSF/TAM deserialization.

Chapter 8 summarizes the results and discusses the findings.

Appendix A discusses the implementation of TSF/TAM in C, and presents the API's for using the TSF/TAM implementation.

Chapter 2

TSF Design

2.1 Goals

TSF provides a general serial transaction format. The TSF library deserializes a TSF transaction to its Transaction Array Model internal format with minimal processing by compact code. The primary design goals are:

- Minimal processing for message serialization and deserialization
- Simple in-memory representation of a deserialized TSF transaction
- Standard serialization and deserialization API's
- Elimination of redundant information, such as found in XML and JSON formats

For IoT devices, the benefits are

- Reduced energy usage
- Smaller memory footprint
- Operations with slower, cheaper CPU's

2.2 Design

The elements of a TSF serialized message are lexical units (LU's), so called because recognition requires only simple lexical processing. There are two abstract lexical units. The first is a primitive lexical unit (PLU) that consists of a sequence of digit characters specifying the data length, a single type character that is not a digit, and length characters of data. TSF extracts a PLU from a message by lex'ing the length, recognizing the type, and extracting the corresponding data. The type character serves to distinguish various PLU types, as desired by the application using TSF. For example, type characters can distinguish between integers and floating point numbers.

A PLU can optionally have a name. In a named PLU, the name follows the initial length and ends with the type character. The name cannot begin with a digit and cannot contain any type character. In the case of XML-equivalent serialization, this is a simple restriction. XML tag name and attribute name come from a restricted character set. For complete generality, the restriction on name characters is removed with a convention described when we consider JSON, which places no restriction on the character composition of names.

The second type of TSF lexical unit is the structured lexical unit (SLU). An SLU consists of a sequence of digit characters specifying the contained unit *count*, a single type character which is not a digit, and *count* subsequent LU's. An SLU is a container in the sense that the *count* identifies the number of immediately contained units. Knowing the *count* at the beginning of the unit allows the pre-allocation of needed storage. An SLU can also optionally have name, with the name having the same restrictions as the PLU name. Each SLU-contained lexical unit may be any type, a named or unnamed PLU or SLU. In the case of XML equivalence, SLU's are used for XML elements and lists of XML attributes.

The second type of TSF lexical unit is the structured lexical unit (SLU). An SLU consists of a sequence of digit characters specifying the contained unit *count*, a single type character, which is not a digit, and *count* subsequent LU's. An SLU is a container in the sense that the *count* identifies the number of immediately contained units. Knowing the *count* at the beginning of the unit allows the pre-allocation of needed storage. An SLU can also optionally have a name, with the name having the same location and restrictions as a PLU name. Each SLU-contained lexical unit may be any type, a named or unnamed PLU or SLU. In the case of XML equivalence, XML elements and lists of XML attributes are SLUs. For JSON equivalence, arrays and objects are SLUs. In a TSF message, named and unnamed lexical units occur in any combination.

Table 2.1 shows the syntax of the TSF lexical units. Syntax descriptions use the Augmented Backus Naur Form described in the IETF's RFC5234[30].

Table 2.1: TSF Message Definitions

TSFMessage	=	1*LexicalUnits
LexicalUnits	=	(PLU / SLU)
PLU	=	Length 0*1Name Type data
SLU	=	Count 0*1Name Type LexicalUnits
Length	=	Number
Count	=	Number
Number	=	1*digit
Type	=	a character that is not a digit or a name character
Name	=	does not start with a digit, or contain any type character

The actual characters used to indicate types can be chosen to reflect the particular application. Type characters and name characters are disjoint. As shown in Table 2.1, type and name characters have the following restrictions[1].

[1]All but the first restriction can be removed. See Chapter 3

1. types cannot be digits
2. names cannot start with a digit
3. type characters cannot be used in names

2.3 Lexical Simplicity

This section discusses the syntax that describes TSF lexical units and shows that it requires only the simplest kind of lexical processing, one-character lookahead.

The syntax descriptions shown in Table 2.1 exhibit the lexical simplicity of TSF. The lexical units of TSF can be recognized using only one lookahead symbol. Each lexical unit begins with a sequence of numeric characters. The sequence is always terminated by a non-numeric character that is either the start of a name or a type character, which determines the type of processing needed to complete the lexical unit. If the type character implies a PLU, the numeric value is the number of characters that constitute the value. Thus, in the syntax description, this field is treated as a terminal symbol. If the type character implies an SLU, the numeric value is the number of contained lexical units. This number can be used to allocate the storage needed for the contained lexical units.

The syntax descriptions shown in Table 2.1 exhibit the lexical simplicity of TSF. The lexical units of TSF can be recognized using only one lookahead symbol. Each lexical unit begins with a sequence of numeric characters. The sequence is always terminated by a non-numeric character that is either the start of a name or a type character. If the type character implies a PLU, the numeric value is the number of characters that constitute the value. Thus, in the syntax description, this field, Data, is a terminal symbol. If the type character implies an SLU, the numeric value is the number of contained lexical units. This number also implies the storage needed for the associated TAM node.

Table 2.2 gives an alternative, right recursive, description of the TSF syntax. Instead of the ABNF zero occurrences syntax, an ϵ alternative explicitly indicates nonterminals that may be empty.

Table 2.2: TSF Right Recursive Syntax

TSFMessage	=	LexList
LexList	=	LexUnit LexList
LexUnit	=	Number Name LUData
LUData	=	PLUType data[1]
LUData	=	SLUType LexList[1]
Number	=	digit Digits
Digits	=	ϵ / digit Digits
Name	=	ϵ / firstchar Nchars
Nchars	=	ϵ / namechar Nchars

Figure 2.1 represent graphically the syntax of Table Table 2.2. In the diagram, a rectangle is a nonterminal symbol. A terminal symbol is a circle or elongated oval. The figure shows that the TSF syntax obeys the following two rules.

Figure 2.1 shows that the TSF syntax obeys two rules.

Rule 1: For any nonterminal, the set of first symbols for each of its alternatives is unique.

Rule 2: For any nullable nonterminal, such as *LexUnit* and *Name* which have the empty string as an alternative, the set of its follow symbols is disjoint from the set of its first symbols.

Together, these two rules guarantee that any sequence of TSF lexical units can be unambiguously recognized by looking at the next terminal in the sequence (one symbol lookahead). Table 2.3 shows the first and follow sets.

[1]zero length is indicated by the leading LexUnit Number, not the syntax

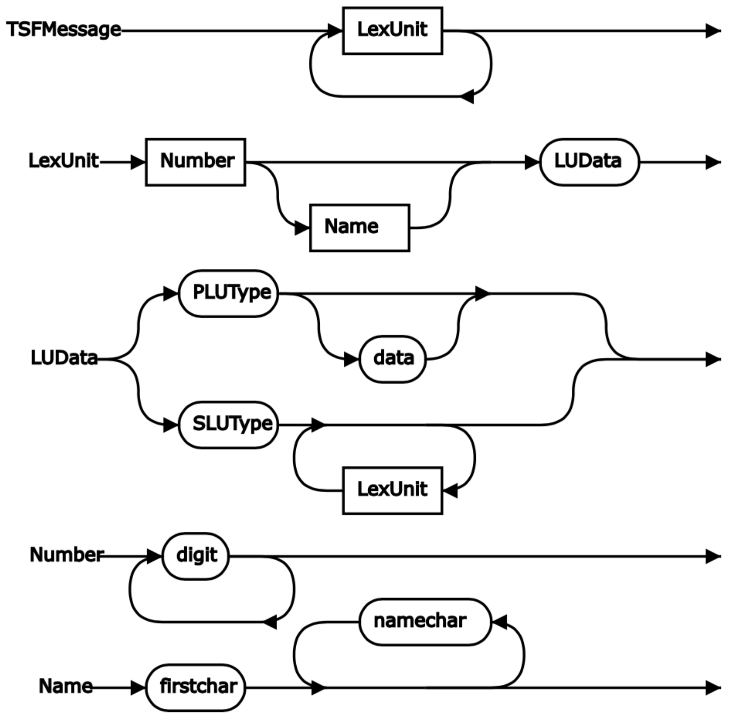

Figure 2.1: TSF Syntax Diagrams

Table 2.3: First and Follow Symbols for the TSF Syntax

Nonterminal	**Nullable**	**First Symbols**	**Follow Symbols**
TSFMessage	no	digit	
LexList	no	digit	digit
LexUnit	no	digit	digit
LUData	no	PLUType, SLUType	digit
Number	no	digit	firstchar, PLUType, SLUType
Digits	yes	digit	firstchar, PLUType, SLUType
Name	yes	firstchar	PLUType, SLUType
Nchars	yes	namechar	PLUType, SLUType

Chapter 6 shows how TSF can encode XML. In this application of TSF, the type characters are chosen to suggest XML meanings. The characters '[', ']', '!', '+', and

'?' are all *PLUType* characters. The characters '<' and '=' are *SLUType* characters.

TSF does not define specific type characters for *PLUType* and *SLUType.* These are user defined. The characters available for type characters are implied by the simple syntax. A type character cannot be a digit, and cannot be a character that may appear in a name. This is because lexically, the type character signifies the end of a number or a name. When names are defined as XML tag names or Javascript variable names, then all the remaining ASCII characters between 32 (0x20) and 127 (0x7f) are available for types. Specifically, these are the characters 32 to 47 (0x20-0x2f), 58 to 64 (0x3a-040), 91 to 96 (05b-0x60), and 123 to 126 (0x7b-0x7e). Although there is no lexical reason why 0 to 31 (0x00-0x1f) and 127 (0x7f) cannot be used, we avoid them to keep the TSF serializations text-editable. The characters in use by an application are initially set through API initialization.

In Chapter 3, since JSON already has a string representation for each of its primitive types, all primitive types are typed by a single 'string' type using the single quote character ('). The character '[' is the JSON array type, and '' is the JSON object type. The TSF serialization of JSON therefore requires three type characters. A different application could use more type characters to signify individual encodings of JSON primitive types.

Chapter 6 shows how TSF can serialize XML. In this TSF application, the type characters suggest XML meanings. The characters '[', ']', '!', '+', and '?' are all *PLUType* characters. The characters '<' and '=' are *SLUType* characters.

TSF Message Generation

In order to show that the TSF syntax of Table 2.2 is an accurate description of the TSF format, we show that the syntax generates TSF strings, with the following informal argument.

In order to analyze generated strings, one starts with the goal symbol and expands it by replacing it with each of its alternative definitions. These are the strings that it generates. The resulting strings are each expanded by replacing nonterminal symbols with their definitions. This process continues until a string contains only terminal symbols. A string of terminals derived this way is said to be generated by the syntax.

Beginning with a set consisting only of the goal symbol, *TSFMessage*, a new set is created containing all the strings generated by expanding *TSFMessage*. At each subsequent step, a new set of strings is generated from the previous set by replacement of the leftmost nonterminal of each string by each definition of the non-terminal. The intent of the process is to show when a generated string contains only terminals, it is a correct TSF message.

- *TSFMessage* generates one or more *LexUnit*'s.
- *LexUnit* generates a *Number* followed by a *Name*, followed by *LUData*. *Name* is optional since it may generate the empty string.
- *LUData* is either a primitive LU, if it starts with a *PLUType* (a terminal) character, or a structured LU if it starts with a *SLUType* (a terminal) character. A primitive LU is followed by *Data*, which may be *data* (a terminal), or empty. A structured LU is followed by zero or more *LexUnit*'s.
- A *Name* is a sequence of terminals.
- A *Number* is a sequence of terminals.

The informal analysis shows that a *TSFMessage* is a sequence of *LexUnit*'s, which in turn generate the terminal sequences of either primitive or structured lexical units. Therefore, the syntax generates only correct *TSFMessages*'s.

Chapter 3

TSF JSON Equivalence

TSF's design is easily applied to JSON, the Javascript Object Notation. JSON is a character-based serial encoding of general computing data structures that requires parsing for deserialization. The web site "Introducing JSON" [18] describes JSON as

> "... built on two structures:
>
> **A collection of name/value pairs.** In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
>
> **An ordered list of values.** In most languages, this is realized as an array, vector, list, or sequence."

A JSON collection is equivalent to a TSF SLU containing a sequence of named PLU's and SLU's. A JSON array is a TSF SLU containing a sequence of unnamed PLU's and SLU's. TSF's internal data model, TAM, handles the JSON data structures. Several export convention allow TSF messages to be exported as either XML strings or JSON strings. We should note that the primary purpose of TSF is to transmit and receive messages with efficient serialization and deserialization. This description of the application of TSF to JSON underscores the generality of the TSF design.

3.1 The Application of TSF to JSON

In order to apply TSF to JSON, we need:

1. a PLU type for each of the JSON primitive types
2. a structured lexical unit type to identify arrays
3. a convention for handling JSON names which contain type characters, or that begin with digits.

In all cases, we will use TSF type characters that suggest the JSON delimiters.

3.1.1 Primitive Types

Javascript's primitive types include integers, floating points, strings, booleans, and a null type. However, JSON does not encode these types differently, but instead uses their string representations. With PLU types, it would be possible to encode each JSON primitive type differently in a TSF transaction. However, for the pedagogical purposes of this section, we keep the string encoding and indicate all primitive types with the single quote type character (**'**).

3.1.2 Collections

A collection is the first of two JSON data structures, and is a sequence of named primitive and structured elements. This is exactly what a TSF SLU is, and we will use the left brace ({) as the SLU type character for a JSON collection.

3.1.3 Arrays

Arrays are second of the two data structures encoded by JSON. A separate SLU type character is needed for arrays, and we will use the left bracket ([) for arrays. The SLU count will be the number of elements in the array. As with all SLU's, an array can have a name, or it can be unnamed.

3.1.4 JSON Names Containing TSF Type Characters

TSF expects lexical unit names, if they are present, to exclude the characters that are used as type characters, (', {, [), so that the detection of a name can be accomplished by a short scan forward until a type character is encountered. In JSON, collection property names may contain any character. JSON handles this by delimiting all names using double-quotes. Double-quotes in a name are handled by escaping (preceding) a double-quote with a backslash (\) character. Since the possibility of names containing the three LU type characters is remote, it is not necessary to incur the overhead of processing every name as if it possibly contained a type character. The double-quote character (") used by JSON is a good introduction character to signify that special processing should be done with a name. In a TSF serialization, a name that starts with a double-quote (") will be processed as if it contained backslash (\) escape sequences. Any character following a \ is accepted as a name character without further examination. An unescaped type character will terminate the scan normally. As an example, the four character name {\[' will appear in a TSF serialization as "\{\\\[\'. If '"' appears as a name character, it should be escaped.

There is one other subtlety regarding this convention. A non-digit character always terminates TSF length and count fields. In JSON, it is possible for a name to begin with a digit character, so the '"' name convention must also be used in this situation. Digits in a name are not escaped, but a name such as `64th` must begin with the double-quote, as `"64th`. This convention allows a name to be created from any sequence of characters, while keeping the name processing simple in most cases.

3.2 String Export

TSF strings can be exported as either XML or JSON strings, by adopting certain conventions. This is more of an academic exercise than a practical one, but may be useful when a TSF message is exported from the realm of IoT to a different computing environment.

3.2.1 Exporting TSF as JSON Strings

Serializing a JSON string in TSF is in a sense a lossy transformation. Unlike XML, JSON ignores whitespace on deserialization, so parsing JSON loses whitespace formatting. This is the only possible difference between an input JSON string converted to a TSF representation and the output string converted from that representation back to JSON. If a JSON string has no ignorable whitespace, the conversion to TSF and back to JSON is lossless. However, TSF includes the possibility of preserving whitespace using unnamed text lexical units if the JSON-to-TSF converter recognizes and preserves whitespace.

3.2.2 Exporting TSF Serializations of JSON as XML Strings

Without JSON extensions, TSF representations can always be rendered as XML strings. With several conventions, it is possible to maintain this feature when TSF has been extended to JSON.

Unnamed Structured Lexical Units An XML representation requires a name, so in those cases, other than arrays, where the JSON object is unnamed, a name can be generated from the position, nesting level and sequence, of the element.

Arrays Array elements under the same parent have the same name, generated from the position of the parent. Optionally, the names can be made unique by including the sequence number of the array element.

Arbitrary Property Names Any name that does not obey the name rules for XML element names will have any invalid character converted to a five character sequence equivalent to the six character JSON UCS escape convention (\`uhhhh`), but with the leading \ dropped. With this convention, a name will be guaranteed to start with a letter, and contain only letters and digits.

Chapter 4

Transaction Array Model

The Transaction Array Model is the in-memory structure of a deserialized TSF transaction. TAM is a simple, conceptually straightforward, representation of a TSF transaction. It features

1. only one memory allocation needed to create the structure to store all the elements of an SLU
2. an array structure to minimize the data fields devoted to structure overhead
3. the direct use of the in-memory TSF transaction for value storage, also called zero-copying

TAM combines the lexical units that are the immediate content of an SLU into a single dynamically allocated node. This node has the structure of a small table. See Figure 4.1.

Row[1]	Type[2]	Name[3]	Value[4]
1	if SLU	data reference or null	TAM node reference or null
2	if PLU	data reference or null	data reference
…	PLU or SLU	…	…
n	…	…	…
[1] row numbers are not part of the structure			
[2] an 8-bit type character			
[3] null if no name, otherwise a direct reference to the TSF transaction memory			
[4] SLU: null if empty, or a TAM node reference			
[4] PLU: a direct reference to the TSF transaction memory			
Additional Fields in the Node			
a reference to the TSF transaction in memory			
a parent reference			
a count of the number of rows in this node			
a count of the number of rows used in this node			

Figure 4.1: A Transaction Array Model Node (TAMNode)

4.1 TAM Creation

TAM attempts to reduce the dynamic allocations needed to create the structure by taking advantage of the SLU occurrence counts embedded in a TSF string. Each SLU has an occurrence count for the number of directly contained LU's. TSF deserialization extracts these occurrence counts from the TSF string at the start of SLU processing. Each SLU count is the number of table rows needed for the SLU's TAMNode. The full size of a TAMNode is thus computable as soon as the number of rows is known. The implications of this are different for each implementing language. For example, in C, all of the storage needed for a TAMNode is allocated with only a single dynamic memory request. In Java, where arrays must be allocated separately, more allocations are needed, but still, when compared to the number of allocations needed for an XML DOM representation, there is a significant reduction. This reduction in dynamic allocations in TAM translates to reduced memory and processing overhead.

The pseudo-structure of a TAMNode in C is shown in Listing 4.1 in which each of the attributes of the table is given a separate array.

Listing 4.1 shows the pseudo-structure of a TAMNode in C, with each of the attributes of the table declared in a separate array.

```
struct TAMNode
{
    char *tsfXact;
    struct TAMNode *parent;
    unsigned elemUsed;
    unsigned elemCount;
    char  elemType[elemCount]
    char  *elemName[elemCount];
    void  *elemValue[elemCount];
};
```

Listing 4.1: A TAMNode

It is a pseudo-structure because in C, arrays, such as `elemName`, cannot be declared with a computable size. However, since the total number of lexical units is known before the structure is allocated, the actual amount of storage needed can be computed. The declaration in Listing 4.1 is informative. The actual structure uses double pointers to locate each of the variable length sections in the TAMNode so that they can be referenced within C code as simple arrays, even though they cannot be declared exactly as shown in the listing.

4.2 Value Storage

TAM also uses a compact approach to store names and data. A TSF transaction is read as a single string, contiguous in memory, and passed to a deserialize() method. Values and names are identified by their offset from the start of the string and their length. Note that a TAMNode references the start of the TSF string to anchor offsets. In this way, no extra storage or allocations are required. This is a language-neutral approach, and works for all languages such as Java, that do not use string terminators. It also supports binary data fields, because it needs no embedded terminators. However, terminators are also possible. For a language such as C, if the transaction consists only of character data, the TSF metadata fields can be overwritten, which allows each name and value to be 0-terminated. This is done on the fly as the TSF message is being processed. Names and values are then directly referenced as 0-terminated C strings. In either case, whether offsets and lengths or 0-terminated strings, names, and data values are all located in the original serialized TSF transaction and no additional allocations are required to store them. This is also true for generalized names (described in Chapter 3.1.4) after escape sequence removal.

4.3 The TAM Root Node

The individual nodes of a TAM structure are linked through SLU references and parent references. The TAM structure begins with a root node. The simplest TAM structure is a sequence of PLU's in a single node. If the PLU's are at the root level, they are effectively "contained" by the pointer to the root node.

A TSF transaction always begins with an SLU to provide the count for the root node. The transaction shown in Figure 4.2 begins with an SLU named `root`. It has three contained units, two PLU's (`+, !`) and an SLU (<) named `doc` with no value. The embedded CR is shown using the C escape convention '\n', and should be counted as a single character. Figure 4.3 shows the `TAMNode` for this transaction.

```
3root=9+ comment 31!doc [<!ELEMENT doc (#PCDATA)>\n]0doc<
```

Figure 4.2: TSF Transaction With Three Top Level PLU's

Row	Type	Name	Value
1	+	null	→ " comment "
2	=	null	→ "doc [<!ELEMENT doc (#PCDATA)>\n]"
3	<	→ "doc"	null
→ tsfxact			
null (a parent reference)			
3 (number of rows in this node)			
3 (number of rows used in this node)			

Figure 4.3: The Root Node With Three LU's

A close inspection of this node reveals that the name of the `root` SLU is missing. The structure shown in Figure 4.4 is a TAMNode with a single named SLU field and a value field that references the TAMNode containing the content of the SLU. It could be considered the root node for the Figure 4.2 transaction.

In order to avoid the need for an extra node just to store the name of the root SLU, TAM adopts a convention for the root node. The root node always represents the contained LU's of the top level SLU. If the top level SLU is named, the name is located using the `tfsxact` field that is in every TAMNode. This field points to the start of the transaction, and the top level SLU name, if it exists, is a few bytes beyond this pointer. This convention avoids the need for an additional memory allocation and overhead just to store the name of the root SLU. The root node is effectively indicated by a null parent pointer, so it is clear when the API code needs to use alternate logic to find an SLU name.

Row	Type	Name	Value
1	=	$\rightarrow$ "root"	$\rightarrow$ TAMNode
$\rightarrow$ tsfxact			
null (a parent reference)			
1 (number of rows in this node)			
1 (number of rows used in this node)			

Figure 4.4: A TAMNode With a Single Named SLU

Chapter 5

Deserialization - Proof of Program Correctness

This section uses the techniques of Program Logic, also called Hoare Logic, to assert the correctness of the deserialization program. First, the inference rules of Program Logic used by the proofs are defined. Then, proofs for each of the methods are presented culminating in a proof for the `deserialize()` method. The techniques used here owe a debt to the presentation of program proofs in Bernstein and Lewis [31].

5.1 Inference Rules

To prove a code fragment, Program Logic makes use of logical assertions and inference rules about individual executable statements. The general form is to establish the truth of a *precondition* P prior to statement execution. Then, using an inference rule specific to the statement to be executed, a *postcondition* Q can be inferred. Q, or an implication derived from Q, then becomes the precondition to the next executable statement, and the process is repeated. This method allows one to assert a precondition at the beginning of a code sequence, and infer a postcondition at the end of the sequence. If this is done with respect to the intended effect of the code, the postcondition proves the code does what was intended. If the conditions are not constructed regarding the effect of the program, they may still constitute a valid proof, but will not reflect insights about the execution of the program.

Program proofs use invariants. A *program invariant* is a condition that is true at the beginning of the code sequence and remains true throughout its execution. A *loop invariant* is a condition that is true at the beginning and end of each iteration of a loop, although not necessarily during execution of the loop body.

5.1.1 Assignment

The rule for assignment states that if the condition P, with any occurrence of the variable x in the condition substituted by the expression e, is true before the assignment of e to x, then the condition P is true after the assignment. There are no hypotheses required above the line.

$$\frac{}{\{Px/e\}\ x = e\ \{P\}}$$

The simple increment statement `x = x + 1;` illustrates the use of this rule. If the statement precondition is $5 = x$, by implication and the rules of arithmetic, we can infer that $6 = x + 1$ as $x + 1$ may be replaced by 6 wherever it occurs. The postcondition then is $6 = x$. As a line in a program proof,

$$\{(5 = x) \Rightarrow (6 = x + 1)\}\ \mathtt{x} = \mathtt{x} + \mathtt{1};\ \{6 = x\}.$$

5.1.2 If Statement

The rule for an **if** statement states that if the condition P and the condition C are true before the **if** body S, and the condition Q is true after S, then P is true before the **if** statement and Q is true following it.

$$\frac{\{P \wedge C\}\ S\ \{Q\}}{\{P\}\ \text{if}\ (C)\ S\ \{Q\}}$$

This may be generalized to the if-else statement as follows.

$$\frac{\{P \wedge C\}\ S_1\ \{Q\}, \{P \wedge \neg C\}\ S_2\ \{Q\}}{\{P\}\ \text{if}\ (C)\ S_1\ \text{else}\ S_2\ \{Q\}}$$

A multi-branch **if** statement, which is basically the same as a **switch** statement, can be handled as follows.

$$\frac{\{P \wedge C_1\}\ S_1\ \{Q\}, \{P \wedge C_2\}\ S_2\ \{Q\}, \ldots, \{P \wedge \neg(C_1 \vee C_2 \vee \cdots \vee C_n)\}\ S_{n+1}\ \{Q\}}{\{P\}\ \text{if}\ (C_1)\ S_1\ \text{else if}\ (C_2)\ S_2\ \text{else if}\ \ldots\ \text{else}\ S_{n+1}\ \{Q\}}$$

5.1.3 While Loop

The rule for a while loop states that if a loop invariant I and the loop condition C are true before each iteration of the loop body S and I is true after each execution of S, then the loop invariant may be asserted before the beginning of the loop and at the end of the loop, along with falsity of the loop condition.

$$\frac{\{I \wedge C\}\ S\ \{I\}}{\{I\}\ \text{while}\ (C)\ S\ \{I \wedge \neg C\}}$$

This rule says nothing about termination other than when the loop terminates, the condition C will no longer be true. Loop termination requires a separate argument. The while loop condition must be altered within the loop body so that it will ultimately be false. Most commonly, an integer within C receives monotonically increasing or decreasing values which must eventually cause C to change.

5.1.4 Restricted Procedure (Method) call

A procedure or method call is somewhat like an assignment statement. The result is a function of the input parameters. The parameters are restricted to value parameters, and the procedure returns a single value. This leads to an inference rule that looks much like assignment.

$$\frac{}{\{P\ x/f(p,q,\dots)\}\ x = f(p,q,\dots)\ \{P\}}$$

An equivalent view of a procedure call is to consider that it has i input parameters and j output parameters in its parameter list. Instead of a return statement, an assignment to an output parameter returns a value to the caller. The output parameters are like the left-hand expressions of one or more assignment statements, and are functions of the statements S of the procedure. Logically describing the input and output parameters as vectors, the procedure call f becomes $f(\overline{in}, \overline{out})$. An inference rule for the procedure call is then

$$\frac{}{\{P\ \overline{out}/f(\overline{in},\overline{out})\}\ f(\overline{in},\overline{out})\ \{P\}}$$

5.1.5 Return

A **return** statement must be inside a method f, and is effectively an assignment of a value to an output parameter which is the result of the method f. The postcondition of the return becomes a postcondition of the invocation of f, with f substituted for occurrences of x in Q. Q may not reference local variables other than the one mentioned in the **return** statement, and parameters that are not **final**.

$$\frac{}{\{Q\}\ \text{return}\ x\ \{Q\ x/f(\dots)\}}$$

5.2 Proof Example

As an example of the proof technique to be used with selected TSF functions, a proof tableau is given for the C function shown in Listing 5.1 which returns 2^n when passed n. The parameter n must be greater than 0, so $n > 0$ is a method invariant. The tableau is shown in Table 5.2. Each line in the tableau has three columns.

1. Proof Line Number - used to identify individual lines in the proof
2. Formula - an assertion about a program statement, which takes the form of {precondition} statement {postcondition}; the precondition is true before the statement is executed, and the rules of inference for the particular statement imply the postcondition
3. Justification - additional notes about the assertions of the proof line

```c
int iexp(int n) {
  int j, x;
  j = 0;
  x = 1;
  while (j < n) {
    j = j + 1;
    x = x * 2;
  }
  return x;
}
```

Listing 5.1: Method iexp() to Compute 2 to the nth Power

	Precondition	Statement	Postcondition	Justification
1	$\{n > 0\}$			program invariant
2	$\{0 = 0\}$	`j = 0;`	$\{j = 0\}$	assign, pgm 3
3	$\{1 = 1 \land j = 0\}$	`x = 1;`		
			$\{x = 1 \land j = 0 \Rightarrow x = 2^j\}$	assign, pgm 4
4	$\{(x = 2^j) \Rightarrow (x = 2^{-1}2^{j+1}), j < n \Rightarrow j + 1 \leq n\}$			line 1, 3
	$\{x = 2^j\}$			loop invariant
5	$\{x = 2^{-1}2^{j+1}\}$	`j = j + 1;`		
			$\{j \leq n \land (x = 2^{j-1} \Rightarrow 2x = 2^j)\}$	assign, pgm 6
6	$\{2x = 2^j\}$	`x = x * 2;`	$\{j \leq n \land x = 2^j\}$	assign, pgm 7
7	$\{x = 2^j \land j < n\}$			
		`while (j < n) S`		
			$\{x = 2^j \land \neg(j < n)\}$	while, pgm 5-8
8		termination: j monotonically increases		
9	$\{x = 2^j \land j \geq n\}$	`return x;`	$iexp = 2^n$	return, pgm 9
10		$\{iexp(n) = 2^n\}$		result

Table 5.1: Proof of method iexp()

5.3 Proof of Deserialization Helper Methods

In the following, actual program listings are modified to accommodate proof inference rules, which are designed for simple statements. For example, the loop condition in Listing 5.2 is rendered in a simplified form in Listing 5.3 with compound statements and conditions reduced to simpler forms for proof.

```
while (idx < ln && isdigit(ch = str[idx])) { . . .
}
```

Listing 5.2: Complex while Loop

```
while (idx < ln) {
    ch = str[idx];
    if (!isDigit(ch))
        break;
    . . .
}
```

Listing 5.3: A Complex while Loop Rendered for Proof

5.3.1 Proof of numericValue()

The **numericValue()** method, shown in Listing 5.4 converts a numeric character to its corresponding integer value. A non-numeric character receives the value 0. It is effectively a table lookup performed with an if/else statement. The proof tableau is shown in Table 5.2.

```
int numericValue(char ch) {
  int x;
  if (ch == '1')
    x = 1;
  else if (ch == '2')
    x = 2;
  else if (ch == '3')
    x = 3;
  else if (ch == '4')
    x = 4;
  else if (ch == '5')
    x = 5;
  else if (ch == '6')
    x = 6;
  else if (ch == '7')
    x = 7;
  else if (ch == '8')
    x = 8;
  else if (ch == '9')
    x = 9;
  else
    x = 0;
23    return x;
24  }
```

Listing 5.4: Method numericValue()

	Precondition	Statement	Postcondition	Justification
1	$P: \ (1=1 \ \wedge \ 2=2 \ \wedge \ \cdots \wedge \ 9=9 \ \wedge \ 0=0)$			definition
2	$Q: \ (x=1 \ \vee \ x=2 \ \vee \ \cdots \vee \ x=9 \ \vee \ x=0)$			definition
3	$\{1=1\}$	`x = 1;`	$\{x=1\}$	pgm 4
4	$\{2=2\}$	`x = 2;`	$\{x=2\}$	pgm 6
5	$\{P\}$	`x = i;`		
			$\{Q\}, \ 0 \le i \le 9$	pgm 4-22
6	$\{P\}$	`if (ch=='1') x = 1;...`	$\{Q\}$	line 4, if/else
7			$Q: (0 \le x \le 9)$	simplifying
8	$\{Q\}$	`return x;`		return inference
			$\{0 \le numericValue(ch) \le 9\}$	return inference

Table 5.2: Proof of numericValue()

	Precondition	Satetement	Postcondition	Justification
1	$C: \ \{'0','1',\ldots,'8','9'\}$			a set of digit characters
2	$Q: \ (x=1 \ \vee \ x=2 \ \vee \ \cdots \vee \ x=9 \ \vee \ x=0)$			definition
3	$\{i=i \wedge ch=C_i\}$			
		`x = i;`		i'th if, pgm 3-20
			$\{x=i, \ (0 \le i \le 9)\}$	
4	$\{0=0 \wedge \neg(ch=C_i)\}$			
		`x = 0;`		else, pgm 21-22
			$\{x=0, \ (0 \le i \le 9)\}$	
5	$\{Q\}$	`return x;`	$\{NV(ch)\}$	NV() defined above

Table 5.3: Proof(2) of method numericValue()

Although the proof for numericValue() indicates that a postcondition for the call is an integer return value from 0 to 9, we wish a more specific functional result that can be used in subsequent proofs.

$$NV(c) = \begin{cases} 0 & if\ c = '0' \\ 1 & if\ c = '1' \\ \ldots & \\ 9 & if\ c = '9' \\ 0 & otherwise \end{cases}$$

A revised proof tableau using the function $NV(c)$ is shown in Table 5.3.

5.3.2 Proof of fieldLength()

The `fieldLength()` method shown in Listing 5.5 scans the input string for a sequence of digit characters and returns the integer value corresponding to the digit character sequence. It uses the `numericLength()` method. The following code is not strictly Java code in that it uses two output parameters. The corresponding Java code uses a two element array to return two values. However, the inference rules for method calls are designed for multiple individual output parameters.

```
void fieldLength(String str, int len, int i, int[2] rtn) {
  int a, d;
  Char ch;
  //String T = "";
  a = 0;
  while (i < len) {
    ch = str[i];
    if (!iswdigit(ch))
      break;
    d = numericValue(ch);
    a = a * 10 + d;
    // T = T + ch;
    i = i + 1;
  }
  rtn[0] = a;
  rtn[1] = i;
}
```

Listing 5.5: Method fieldLength()

The proof of the method fieldLength() uses a non-program variable T which represents the sequence of digit characters selected from the string parameter, and reflects the implicit multiplication by 10 and addition that appending an additional digit to the right of a number representation accomplishes. It is referenced in comments in the program since there is no need for the program to actually compute its value. Using T, we define a function $IV(T)$, the integer value of a string of digit characters (d_i), as

$$IV(T) = \begin{cases} 0 & if\ T = \text{""} \\ d_1 d_2 \ldots d_n & if\ T = \text{"}d_1 d_2 \ldots d_n\text{"} \end{cases}$$

We also define the predicate $NC(i)$ to be true if i is the index of the next character in the string, and false otherwise. $NC(i)$ is true on entry to **fieldLength()**. Once the character i is accepted from the string, then NC(i) is false, but NC(i+1) is true. The function $NV()$ is the value of the **numericValue()** method defined above. With these definitions, the proof of the method **fieldLength()** can be shown in Table 5.4.

	Precondition	Statement	Postcondition	Justification
1	$C: \ \{'0','1',\ldots,'8','9'\}$			set of digit chars
2	$0 \leq i < \lvert str \rvert \wedge NC(i) \wedge len = \lvert str \rvert$			entry conditions
3	$\{"" = ""\}$	`T = "";`	$\{T = ""\}$	pgm 4
4	$\{0 = 0\}$	`a = 0;`	$\{a = 0\}$	pgm 5
5	$I: \ a = IV(T) \wedge NC(i)$			loop invariant, 2-4
6	$\{NC(i)\}$	`ch = str[i];`	$\{ch = str_i \wedge NC(i+1)\}$	pgm 7
7	$\{ch = str_i \wedge NC(i+1)\}$	`if (!iswdigit(ch) break;`	$\{ch = d_i \in C \wedge NC(i+1)\}$	line 1, pgm 8,9
8	$\{ch = d_i \in C\}$	`d = numericValue(ch);`	$\{d = NV(d_i)\}$	pgm 10
9	$\{10a + d = IV(Td_i)\}$	`a = a * 10 + d;`	$\{a = IV(Td_i)\}$	pgm 11
10	$\{a = IV(Td_i)\}$	`T = T.append(ch);`	$\{a = IV(T)\}$	append, pgm 12
11	$\{NC(i+1)\}$	`i = i + 1;`	$\{NC(i)\}$	pgm 13
12	$I: \ a = IV(T) \wedge NC(i)$			invariant, 10,11
13		termination	$\{i \geq \lvert str \rvert \vee ch \notin C\}$	
14	$\{a = IV(T)\}$	`rtn[0] = a;`	$\{rtn_0 = IV(T)\}$	pgm 15, line 10
15	$\{NC(i)\}$	`rtn[1] = i;`	$\{rtn_1 = i \wedge NC(i)\}$	pgm 16, line 11

Table 5.4: Proof of method fieldLength()

Notes on the proof of **fieldLength()**.

1. C is a set of digit characters.
2. On entry, the index i is greater than or equal to 0 and less than the length of the parameter str; the variable len equals the number of characters in str.
3. The logical variable T begins as an empty string, program line 4.
4. An initialization assignment sets the variable a to 0, the integer value of T, program line 5.
5. The loop invariant I asserts the variable a is equal to the integer value of the digit string T and the index i is the index of the next character to be processed in the parameter str.
6. Since $NC(i)$ is true, the assignment assigns to ch the i'th character of str.
7. Additional loop termination condition; if $ch \notin C$, jump to the end of the loop, program lines 8 and 9.
8. d is the numeric value of the character ch, program line 10.
9. The value of a is updated by appending the digit d, program line 11.
10. ch is appended to the logical variable T.
11. The loop index i is incremented by 1.
12. The loop invariant I can now be reasserted.
13. The loop must terminate because $i \geq |str| \vee ch \notin C$.
14. The first return variable rtn_0 is set to a, the numeric value of the length field.
15. The second return variable rtn_1 is set to the loop index i, the index of the next character to be processed.

5.4 Deserialization

The method **deserialize()**, shown in Listing 5.6, is a wrapper around the method **processElems()** that deserializes what are effectively top level document elements. The wrapper is needed because, unlike XML, the TSF root level element may be an unnamed container with multiple SLU's and PLU's. **deserialize()** lexes the count only, and calls **processElems()** to perform the root element deserialization.

In TSF format, if there are multiple top levels elements, the first numeric field is a count of the other elements followed by an '=' delimiter. If the transaction consists only of a single named SLU, the unnamed SLU is not present. See Section 4.3 for further details. The function $CT()$ represents the count of elements to be processed. We continue to use the predicate $NC(i)$ which is true when i is the index of the next character in the input to be processed.

5.4.1 Proof of deserialize()

The proof of **deserialize()** is given in Table 5.5 following its listing, Listing 5.6.

```
NpxUCElement *
deserializeUC(Char *wbuf, int siz)
{
    NpxUCElement *docElement;
    int ndx, idx, rtn[2], len;
    ndx = fieldLengthUC(wbuf, 0, rtn);
    if (L'=' != wbuf[ndx])
    { /* not unnamed SLU - count of 1 is implied */
        rtn[0] = 1;
        idx = 0;
    }
    else
    {
        idx = ndx + 1;
    }
    docElement = newNpxUCElement(wbuf, rtn[0], 0);
    idx = processElemsUC(docElement, wbuf, rtn[0], idx,
                         (char)wbuf[ndx]);
    return docElement;
}
```

Listing 5.6: Method deserialize()

	Precondition	Statement	Postcondition	Justification
1	$\{NC(0)\}$	`{fieldLength(str, 0, rtn));`	$\{NC(rtn_0) \wedge rtn_1 = CT()\}$	pgm 6
2	$\{CT(rtn_1)\}$	`n = rtn[1];`	$\{n = CT()\}$	line 1
3	$\{1 = 1\}$	`rtn[0] = 1;`	$\{(rtn_0 = 1) = CT()\}$	assign
4	$\{true\}$	`i = 0;`	$\{NC(i)\}$	assign
5	$\{rtn_0 + 1 = CT()\}$	`r0 = rtn[0] + 1;`	$\{(r0 = CT()\}$	pgm 7, assign
6	$\{NC(n+1)\}$	`i = 0;`	$\{NC(i)\}$	assign
7	$\{true\}$	`if (...)...`	$\{NC(i) \wedge r0 = CT()\}$	pgm 7-16, if
8	$\{r0 = CT()\}$	`doc = new Npx(r0, 0);`	$\{\}$	alloc root
9	$\{r0 = CT() \wedge NC(i)\}$	`processElems(doc,str,r0,i,r);`	$\{NC(r)\}$	processElems()
10	$\{\}$	`return doc;`	$\{\}$	return doc root

Table 5.5: Proof of method deserialize()

The method **processUCElems()**, shown in Listing 5.7 returns a fully constructed root element. **deserialize()** constructs a TAM document element which is returned to the caller.

5.4.2 Proof of processUCElems()

processUCElements() is called from deserialize() and performs the full deserialization of the input string the construction of all TAM nodes but the root node allocated in **deserialize()**.

```
static int
processElemsUC(NpxUCElement *npxElement, Char *str,
        const int ct, int ndx, char typ)
{
  int idx, ix;
  int rtn[2];
  Char ch;
  NpxUCElement *elm;
  idx = ndx;
  for (ix = 0; ix < ct; ++ix)
  {
    ndx = fieldLengthUC(str, idx, rtn);
    str[idx] = L'\0';
    ch = str[ndx];
    if (ch >= arraySize(ctype) || 0 == (ctype[ch] & ST))
    {
      if (*ENAME == (char)ch)
      {
        ndx = extendedNameUC(str, ndx + 1);
        ch = str[ndx];
      }
      else
      {
        while ((ch = str[ndx]) >= arraySize(ctype) ||
                0 == (ctype[ch] & ST))
        {
          ++ndx;
        }
      }
    }
    if (ch < arraySize(ctype) && 0 != (ctype[ch] & PL))
    {
      str[ndx] = L'\0';
      idx = rtn[0] + ndx + 1;
      self(npxElement).add(npxElement, str, rtn[1],
              ndx, idx, (char)ch);
      continue;
    }
    if (0 < rtn[0])
      elm = newNpxUCElement(str, rtn[0], npxElement);
    else
      elm = (NpxUCElement *)0;
    idx = ndx;
44      idx = processElemsUC(elm, str, rtn[0], idx + 1, (char)ch);
45      self(npxElement).addElem(npxElement, &str[rtn[1]], elm,
46              (char)ch);
47      str[ndx] = L'\0';
48    }
49    return idx;
50  }
```

Listing 5.7: Method processElems()

Based on the foregoing formal proofs, it should be clear that an equivalent proof can be constructed for **processUCElems()**. The principles reflected in the code have been used in the previous programs, so in the interest of brevity, they will be described informally in Table 5.6.

Lines	Explanation
1-3	`processUCElems()` is passed the TAM node being built, the transaction string in TSF format, a count of the lexical units to be found at this level, a starting index for the next lexical unit within the transaction string, and the type character for the current lexical unit
6	the method `fieldLengthUC()` and `processAttrList()` when called will return multiple values in array of reference parameters; this is the return array
10	loop for the number of lexical units at this level, the parameter `ct`
12	`fieldLengthUC()` processes the length/count field of the next lexical unit and returns the field's value, in `rtn[0]`, and the index of the character following the field, in `rtn[1]` and as the return vale
14	get the next character from the TSF transaction string
15	if the next character is not an SLU or PLU type character, execute lines 17-29
17-29	if the next character is an extended name signifier, process the extended name by calling `extendedNameUC()`, and set `ch` to the following type character, we have a normal name, which is lex'ed to the type character, and `ch` is again set

Lines	Explanation
31	if `ch` is a PLU type character, execute lines 33-37
33-37	add the new PLU to the current TAMNode being built, and go to the next iteration to process the next lexical unit
39-42	processing falls to here if the next lexical unit is an SLU; allocate a new TAM node for the SLU if the count is greater than 0; if the count is 0, the reference to the TAM node is null
44	call `processElems()` recursively to process all the lexical units at the next level, passing the number of units and the starting index of the first unit; the return value is the index of the next unit to be processed at this level
45	add a reference to SLU node just built to the current TAM node
49	the index of the next lexical unit following the units just processed is returned

Table 5.6: Informal Proof of Method processElems()

Chapter 6

XML Equivalence: An Application of TSF

This chapter demonstrates the ability of TSF to losslessly represent XML documents, in order to show that TSF is a least as general as XML.

Applying TSF to XML begins with relevant definitions from the XML 1.0 Recommendation [32], followed by a discussion of the lexical units needed by XML. Several short sections on various XML issues such as DOCTYPE's and name spaces follow. When syntax descriptions are needed, they are written using the Augmented Backus Naur Form described in the IETF's RFC5234[30]..

6.1 Relevant XML Definitions

The XML 1.0 recommendation[32] provides the definitions shown in Figure 6.1 and Figure 6.2. The numbers in square brackets are the identifiers of the definitions in the XML recommendation.

6.1.1 XML Names

"A *Name* is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters."[32]

[4]	NameChar	=	Letter / Digit / "." / "-" / "_" / ":" / CombiningChar / Extender
[5]	Name	=	(Letter / "_" / ":") *NameChar

Figure 6.1: The Syntax of an XML name

CombiningChar's and *Extender*'s are classes of Unicode characters that are not relevant to TSF. What is important for its design is that a *Name* (called an *XMLname* below) cannot contain the characters "<", ">", or "=".

The type characters used in TSF to represent XML lexical units must be disjoint from XML name characters.

6.1.2 Tag Names and Attribute Names

[40]	STag	=	"<" Name *(S Attribute) *S ">"
[41]	Attribute	=	Name "=" AttValue

Figure 6.2: The Syntax of an XML Open Tag

XML open tags are XML names, and the tag can contain a list of attributes. The XML definition of *AttValue* uses character exclusion which makes the ABNF definition awkward, so we describe attribute values in English. An *AttValue* can be any sequence of characters delimited by leading and trailing single quotes, or leading and trailing double quotes. Characters with XML syntactic meaning cannot be coded literally in an *AttValue*, but must be encoded using XML entity references. The relevant entity references for attribute values are `&lt;`, `&gt;`, `&amp;`, `&apos;`, and `&quot;`, for the characters "<", ">", "&", single quote (`%x27`), and double quote (`%x22`) , respectively. The *S* in the definition represents XML white space. TSF does not need entity references.

6.2 TSF Types for XML

TSF type characters identify the following XML types. The definitions all begin with lower case characters because they are terminal elements in the TSF syntax description of XML (Figure 6.4). As terminals, they do not need any auxiliary processing, such as scanning.

xML-doctype-content - a DOCTYPE entity

xML-processing-instruction - a Processing Instruction

xML-comment - character data in an XML comment following the opening four character `<!--` sequence and ending at (not including) the three character `-->` terminating sequence

xML-pcdata - parsed character data (XML PCDATA)

xML-cdata - character data (XML CDATA) (unexamined character data)

xML-attribute-content - the sequence of characters that make up the value of an XML attribute; for TSF purposes, attribute content contains no XML entity references.

xMLname - a sequence of characters conforming to the XML *Name* definition, shown in Figure 6.1, used as either an element tag name or an attribute name.

6.2.1 The Syntax of XML Lexical Units

The TSF design is based on the idea of a lexical unit, named that because of the minimal amount of lexical processing needed to recognize each unit in a TSF string. A *TSFXMLDoc* string is a sequence of lexical units, shown in Figure 6.3.

TSFXMLDoc	=	1*LexicalUnit
LexicalUnit	=	SLU / PLU
SLU	=	Element / Attrs
PLU	=	Doctype / ProcInst / Comment / Text / Cdata / Attr

Figure 6.3: The Syntax of an XML Document as a TSF String

Unstructured XML constructs are represented by Primitive Lexical Units, shown in Figure 6.4 with their single character types. The length indicates the number of value characters that follow the type character. The only named PLU is the *Attr*. The *xMLName* conforms to the XML definition.

Note that *Text* and *Attr* use the same type characters, but there is no ambiguity because they occur in different containers.

Number	=	1*digit
Length	=	Number
Doctype	=	Length ”`!`” xML-doctype-content
ProcInst	=	Length ”`?`” xML-processing-instruction
Comment	=	Length ”`+`” xML-comment
Text	=	Length ”`[`” xML-pcdata
Cdata	=	Length ”`]`” xML-cdata
Attr	=	Length xMLname ”`[`” xML-attribute-content

Figure 6.4: TSFString Length Unit Syntax

XML elements (*Element*) and attribute lists (*Attrs*) are SLU's, shown in Figure 6.5. With SLU's, the leading number is the count of contained lexical units. In the *Element* SLU, contained units include processing instructions, comments, parsed character data (PCDATA), character data (CDATA), and subsidiary (child) elements. The count is greater than or equal to zero. If the *Element* is empty, the count is zero. The occurrence of *Element* in the definition of *Content* provides the recursive definition for a nested XML data structure. With the *Attrs* SLU, the count is the number of attributes in the attribute list. If there are no attributes, there is no *Attrs* SLU, which is distinguished by its type code.

Count	=	Number
Element	=	Count xMLname "<" 0*1Attrs Content
Attrs	=	Count "=" 1*Attr
Content	=	*(ProcInst / Comment / Text / Cdata / Element)

Figure 6.5: TSFString Count Unit Syntax

6.2.2 DOCTYPEs, Processing Instructions, Comments

A complete serialized format for XML transactions must handle DOCTYPE's, comments, and processing instructions that are outside the root element, as well as a single document root element. The complete TSFXMLMsg, Figure 6.6, has an optional DOCTYPE followed by zero or more processing instructions and/or comments, one XML (root) element (TSFXMLDoc), and zero or more processing instructions and/or comments. The TSFXMLMsg is an unnamed SLU whose type character is '='. This overloading of the '=' character is not ambiguous since it is not contained in an *Element*. Most often, a *TSFXMLMsg* will be just the XML root element, composed of the lexical units of the *TSFXMLDoc* definition. In this case, the leading "1=" sequence can be heuristically implied and omitted. For additional information about the root node, see Chapter 4.3.

```
TSFXMLMsg   =   *1( Doctype ) *( Procinst / Comment ) TFSXMLDoc
                *( ProcInst / Comment )
```

Figure 6.6: TSFString With Outside XML Elements

6.3 XML Issues

6.3.1 Namespaces

XML namespaces are not given any special treatment in the serialized format. An XML tag name or attribute name may have a namespace prefix. The fully qualified name, when it occurs, is embedded as an *XML-name* in the format. Namespace URL's are represented in the normal way as attributes.

6.3.2 Character and Entity References

XML markup gives certain characters special meaning. Among these are the angle brackets (”less than” and ”greater than” characters), the ampersand, and the single and double quotes. When these characters appear as normal data characters, they must be treated specially so that they are not given their markup meaning. XML provides for this need with character references and entity references, which are special character sequences that encode these characters when they are used as data. These have been mentioned above.

Because TSF is not parsed, it has no need for these sequences and TSF does not use them. All data characters appear in a TSF string in their normal encoding. There is no additional escaping of special characters needed.

6.3.3 Encoding

An XML document is normally introduced with the

```
<?xml version="1.0" encoding="..."?>
```

processing instruction specifying the XML version and the encoding of the following document. Until very recently, there was only one XML version, 1.0. The new XML 1.1 version handles unusual situations that do not impact the core of XML usage.

While the ability to flexibly exchange documents encoded with different encodings is useful, the UTF-8 encoding is a superset of US-ASCII and UCS4. The TSF/TAM design addresses limited capability devices, often found in sensor networks, and makes the assumption that encoding is not an issue. As such, the default behavior is equivalent to

```
<?xml version="1.0" encoding="UTF-8"?>
```

so that this processing instruction need not be present in a TSF string.

The TSF design does not preclude the use of this processing instruction should an application have need for it. It can be included as a lexical unit at the document level. But, the code accompanying this dissertation conditionally compiles to either `USASCII`, using 8-bit characters internally, or `UTF-8` encoding, using 32-bit (UCS-4) characters internally. The fixed length 8-bit `USASCII` provides better performance and should be used unless there is a requirement for `UTF-8`. Eight-bit encoding is also important in that it can support binary data in a TSF transaction.

6.3.4 XML Document Reconstruction

The design of TSF is intended to encompass all XML documents, and to support a complete reconstruction of any TSF-encoded XML document. However, there are certain limitations brought about by parsers and XML equivalences.

- When converting XML to TSF, the converter will serialize the attributes of an XML element in the order they are provided by the parser in the XML library. If this is not the order in which they appear in the document (maybe the parser hashed them), when the document is reconstructed, the attributes will possibly not be in their original order within the element start tag.
- An empty XML element will be serialized using the single tag abbreviation `<element/>`. If the document originally contained `<element></element>`, the reconstructed document will differ. This can be optionally controlled.
- An XML parser will not preserve whitespace between components such as processing instructions and comments that appear outside of the root element. If these were originally on separate lines, thus being separated by whitespace, the reconstructed document will not have this whitespace.
- An XML parser will not preserve whitespace separating attributes within a start tag. The reconstructed document will have only a single space character separating attributes, with no spaces surrounding the "=" between the attribute name and the attribute value, and no spaces at the markup characters.
- Line end sequences can be optionally represented by a single newline character, or a carriage return/line feed sequence.

6.4 The Overhead of TSF

Although the motivation for TSF is not compactness, a side effect of the format is a smaller transaction when compared to its XML equivalent.

The Primitive Lexical Units each have one character indicating the type and a length field whose width is dependent upon the number of data characters in the unit. If l is the number of data characters, the width of the length field is $\lceil \log l \rceil$, so the overhead is $1 + \lceil \log l \rceil$.

Structured Lexical Units's have the same kind of overhead. If c is the count of contained lexical units, the width of the count field is $\lceil \log c \rceil$, so, with the SLU type character, the overhead is $1 + \lceil \log c \rceil$. An XML element with only PCDATA is equivalent to an SLU/ PLU combination. Finally, if a document consists of content in addition to the root element, a count of the number of document elements other than the root element is also provided, with one additional delimiter character.

Table 6.1 summarizes the overhead of various lengths of an SLU/PLU combination vs an XML element.

		Data Characters		
Lexical Unit	**Character Overhead**	**1-9**	**10-99**	**100-999**
Unstructured	$1 + \lceil \log dataLength \rceil$	2	3	4
Attribute list	$2 + \lceil \log attrCount \rceil$	3	4	5
Element	$1 + \lceil \log contentCount \rceil$	2	3	4
Document	$1 + \lceil \log docCount \rceil$	2		

Table 6.1: TSF Overhead

For a simple XML element whose tag name is t characters, the overhead is $5 + t$. This reflects the 5 delimiter characters, 2 <, 2 >, and 1 /, plus an extra occurrence of the tag name. TSF uses 1 delimiter for the tag name, but it also adds 1 delimiter and a length field for the character content. TSF also has a count field for each tag. The primary size difference between a TSF string and its equivalent XML is the missing redundant end tag. The savings improves with the number of elements in a transaction. Chapter 7, Table 7.2 shows test file size comparisons between XML and TSF.

6.5 Examples

A few examples of XML serialization in TSF are shown in Figure 6.7 to illustrate the foregoing descriptions.

Form	Serialization*
XML	`<project/>`
TSF	`0project>`
XML	`<ns:personnel xmlns:ns="urn:foo"><ns:person id="Boss"/><ns:person id="worker"/></ns:personnel>`
TSF	`3ns:personnel<1=7xmlns:ns[urn:foo1ns:person<1=4id[Boss1ns:person<1=6id[worker>`
XML	`<?peri rset?><!--Introduction--><project>content</project><!--Epilog--><?peri sset?>`
TSF	`5=9?peri rset12+Introduction1project<7[content6+Epilog9?peri sset`
	* line wrapping is not part of the serialization

Figure 6.7: TSF Overhead

6.6 Embedding TSF in XML

A TSF string can be embedded in an XML Processing instruction in order to send a TSF string within an existing XML infrastructure. A program using the XML SAX API could then handle the TSF PI as a special case. Consider

<?`tfx` *TSFString*?>

As in all situations where control characters may be recognized as data, the ending processing instruction two-character sequence '?>' cannot appear in the TSFString. If it does, the processing instruction would be prematurely terminated. How likely is this?

TSF uses the '?' to indicate a processing instruction, but XML precludes a '>' from being the first character of a processing instruction target, so this conflict will never arise.

TSF uses the '>' to delimit a tag name from a list of the tag's attributes. A tag name will never end in '?', so this conflict will never arise.

Therefore, the only potential conflicts would be in application data, where the '?>' sequence might appear in data in PCDATA, CDATA, or an attribute value. It would be necessary to handle this within the application. A simple approach would be to encode occurrences of '?' in the data as a sequence such as '?-'. This would keep the character adjacency '?>' from ever occurring in data, allowing the TSFString to be embedded within an XML processing instruction.

Chapter 7

TSF/TAM Performance

This section compares the performance of standard XML deserialization processing against TSF deserialization. The focus of the performance testing is on deserialization, as opposed to serialization, because, of the two operations, deserialization has a formal API. XML parsing has two standard deserialization API's, SAX and DOM. XML serialization depends upon the internal data model. Some libraries will provide serialization from DOM, but if one is using a SAX parser for performance, then one is also building a custom data structure from the parse, which implies that a serializer must also be custom. The TAM library provides a serializer, which could be compared to an XML DOM serializer, should one desire, but the following compares only deserialization performance.

7.1 Test Files

The input test files are shown in Table 7.1, where they are given IDs for reference in other tables. The ID's are assigned in file size order. The size in bytes and a brief description are included.

Table 7.1: Test File Descriptions

File ID	File Name	File Size	Description
F1	future001.xml	70358	Scenario file from the Mana Game Series
F2	bpmnxpdl_40a.xsd.xml	160946	XSD file for XPDL 2.0
F3	eric.map.osm.xml	218015	OpenStreetMap export from northern WVa
F4	cshl.map.osm.xml	298233	OSM export of a research laboratory
F5	sccc.map.osm.xml	404977	OSM export of a community college
F6	British-Royals.xhtml	482666	British Royalty Lineage from Alfred the Great
F7	csh_lirr_osm.xml	712661	OSM export of a train station
F8	exoplanet-catalog.xml	2147926	NASA Kepler Exoplanet Catalog
F9	LARGEbasicXML.xml	3420388	Military Strategy Game Unit Order of Battle

Table 7.2 compares the sizes of the XML test files and their TSF equivalents.

Table 7.2: Test File Size Comparisons

ID	XML Size	TSF Size	Reduction
F1	70358	54049	23.18%
F2	160946	142280	11.60%
F3	218015	206800	5.14%
F4	298233	284065	4.75%
F5	404977	386928	4.46%
F6	482666	477051	1.16%
F7	712661	677853	4.88%
F8	2147926	1456993	32.17%
F9	3420388	2797092	18.22%

Table 7.3 shows the XML characteristics of each file are shown. Files F3, F4, F5, and F7 are similar and serve as a consistency check. Although differing slightly in size, the table shows that they have the same internal structure. The other files were selected because of their size and differing internal structures. Detailed explanations of the columns follow Table 7.3.

Table 7.3: Test File Characteristics

					Lex	Avg	Depth		Children	
ID	Elems	Attrs	DATA	Cmt	Units	Bytes	Avg	Max	Avg	Max
F1	1936	6	2596	0	4538	11.9	3.5	7	2.2	251
F2	2565	3317	4011	29	9922	14.5	3.7	11	2.4	379
F3	2515	11021	2815	0	16351	12.8	1.6	3	1.3	2017
F4	3544	15360	3616	0	22520	12.8	1.6	3	1.3	2709
F5*	5135	20566	5294	0	30995	12.6	1.7	3	1.3	3523
F6	4589	391	7948	5	12934	36.9	10.2	15	2.0	3556
F7*	8630	36855	8915	0	54400	12.6	1.6	3	1.3	6385
F8	168728	420	66247	0	235395	6.2	6.0	7	1.4	4215
F9	73156	15989	146227	1	235373	11.9	4.6	6	3.0	1129
* contains multibyte characters										

Table 7.3 column explanations.

Elems the number of individual XML elements in the document

Attrs the total number of attributes on all the elements in the document

DATA the total number of CDATA and PCDATA occurrences in the document

Cmt the number of comments in the document

Lex Units the total number of lexical units in the document, which should equal the sum of the previous four columns

Avg Bytes (per lexical unit) the number of bytes in the document divided by the number of lexical units

Avg Depth the average depth of the subtree below an XML element

Max Depth the maximum depth of the document; the maximum n umber of elements encountered in the path from the root to the lowest leaf element

Avg Children the average number of child elements for any given element (Max Children omitted from this calculation to avoid skewing the value)

Max Children the maximum number of children parented by any element; in these documents, this is almost always the number of children of the root element

7.2 Performance of the C/C++ Implementation Compared to Libexpat

Libexpat [33] is a library, written in C, for parsing XML documents. It is a popular parser used in many industry-wide programs, including the open source Mozilla project, Perl's **XML::Parser** package, and Python's **xml.parsers.expat** module. It has undergone extensive development, testing, and release-to-release improvements. The release used for the following work is libexpat-2.2.6. The C compiler used to build TSF/TAM, libexpat, and the deserialization performance drivers on the MacBook Pro is:

```
Apple LLVM version 10.0.0 (clang-1000.10.44.4)
Target: x86\_64-apple-darwin18.2.0
Thread model: posix
Processor: 2 GHz Intel Core i7
```

The performance tests were run on the same machine.

The following points should be kept in mind when reading the information on comparative performance statistics between libexpat and TSF/TAM.

- libexpat is a SAX parser. When parsing an XML file using libexpat, minimal callback functions are used.

- The libexpat callbacks build a stripped-down DOM in order to avoid bias. In order to minimize memory allocation overhead, single allocations are used for multiple strings. For example, to build an attribute element from name and a value, a single memory request is made and the null-terminated name and value are both copied into the allocation.
- Namespace processing in libexpat was disabled to correspond with the TSF design. In this situation, libexpat treats a namespace prefix-qualified tag name or attribute name as a single sequence of characters. `xmlns`-prefixed attribute names are not given special treatment.
- CPU time is collected using the `getrusage()` C library function.
- All libexpat and TSF processing is done using in-memory input with no threading. Timing comparisons are not started until input is completely read.

Table 7.4 shows Libexpat CPU times to deserialize each of the nine test files. There are five separate runs for each file, and the mean and standard deviations for the runs are shown in the last two columns. Table 7.5 shows equivalent statistics for TSF deserialization operating on each of the test files over five runs. Note that this is an apples-to-apples comparison in that the TSF program is working with UTF-8 TSF files and supports a UCS-4 character set internally.

Table 7.6 shows the performance of an 8-bit character implementation of the TSF algorithm, using the seven input files that do not have multi-byte character input. As expected, the performance is better.

Table 7.4: Five Run Deserialization Performance Using a Libexpat C library (microseconds CPU time)

File	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stdev
F1	3051	2856	3213	3360	3126	3121.20	167.77
F2	7501	7046	7704	7786	7184	7444.20	287.69
F3	11413	11025	11053	11114	11287	11178.40	148.48
F4	15635	15704	15531	15117	16352	15667.80	398.15
F5	23402	22393	25996	21137	22454	23076.40	1627.62
F6	10001	10922	11212	11100	10938	10834.60	430.37
F7	42218	40058	44230	41468	39866	41568.00	1593.46
F8	114265	121418	125337	134613	128113	124749.20	6781.90
F9	158042	195949	180174	181541	185936	180328.40	12438.86

Table 7.5: Five Run Deserialization Performance Using a Wide Character C Implementation of the TSF Algorithm (microseconds CPU time)

File	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stdev
F1	662	585	579	551	550	585.40	40.85
F2	1678	1442	1398	1397	1402	1463.40	108.60
F3	2616	2053	2469	1887	1894	2183.80	302.43
F4	3134	2549	2836	2681	2592	2758.40	211.96
F5	4208	4166	3842	3663	4295	4034.80	240.97
F6	3367	2413	2670	3332	2482	2852.80	414.33
F7	7355	7097	6759	6549	6450	6842.00	339.00
F8	20513	20151	19287	21379	21208	20507.60	757.23
F9	35575	35133	29692	29965	37841	33641.20	3246.97

Table 7.6: Five Run Deserialization Performance Using an 8-bit C Implementation of the TSF Algorithm (microseconds CPU time)

File	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stdev
F1	409	402	402	411	401	405.00	4.15
F2	1285	1050	1050	1050	1050	1097.00	94.00
F3	1372	1468	1339	1302	1335	1363.20	56.90
F4	2081	1809	1814	2042	1813	1911.80	122.86
F6	1203	1102	1084	1083	1133	1121.00	44.82
F8	19242	23759	22204	19090	16728	20204.60	2485.44
F9	21972	21580	21138	20849	21530	21413.80	386.72

Table 7.7: Deserialization Performance Improvement Factor, TSF vs Libexpat

File ID	F1	F2	F3	F4	F5	F6	F7	F8	F9	Mean
TSF Improvement (UTF-8)	5.3	5.1	5.1	5.7	5.7	3.8	6.1	6.1	5.4	5.4
TSF Improvement (8-bit)	7.7	6.8	8.2	8.2	-	9.6	-	6.1	8.4	7.9

Table 7.7 summarizes the deserialization performance improvement provided by TSF (UTF-8). The improvement factor is the mean Expat CPU time divided by the mean TSF CPU time for each file. The overall mean improvement factor is 5.4, a reduction of the CPU time of more than 80%.

As an additional indication of the consistency of the results, the CPU times for both libexpat XML deserialization and TSF deserialization are highly correlated with the number of lexical units in each file, given in Table 7.3. For TSF, the correlation is 0.956. For libexpat, the correlation is 0.975.

The reduction in deserialization time for TSF by a factor of 5.4 in comparison to XML shows that TSF can be a significant energy reduction component of an IoT device that sends and receives structured data.

As an added bonus, the headers and source code for TSF/TAM comprise less than 600 lines. The libexpat XML parse source code is approximately 15,400 lines.

Chapter 8

Conclusion

As the Internet of Things expands with limited capability devices, efficient transactions formats can help move data faster, and with less energy. Less energy means a longer field life for an IoT device that does not have an external power source. The Transaction Serialization Format provides such a format. It is general enough to support full XML document and JSON object serialization and deserialization for a small fraction of the memory and CPU cost, as demonstrated by performance analyses comparing a traditional XML library. The Transaction Array Model provides a simple internal memory structure for handling the lexical units of a TSF message. The TAM structures can be created and destroyed with fewer requests for dynamic memory than needed for the well-known XML Document Object Model, and at the same time are memory conservative. The code supporting this work is available from https://github/dde/TSF. The full TSF/TAM library is presented in only five pages of code in Appendix A.

Chapter 9

Future Work

Although the raison d'etre for TSF is serialization efficiency, TSF is also a serial format. Information may be stored on any serial medium using TSF to preserve its structure. This immediately raises the problem of updating information stored in TSF, and related issues.

1. How is information located in a TSF data record? Since data is generally stored in the clear in TSF, a design analogous to XPath might be possible.
2. How is information updated in a TSF record? Can the changes needed for update, insert, and delete be localized without extensive modification to the record?
3. Since TSF has a standard serialization/deserialization API, is the best way to update a record simply to add to the API and modify the in-storage TAM?
4. Since TSF data is generally in character form, is it possible to create a simple, TSF-aware editor to simplify direct update?

It is also possible to question some of the TSF design decisions. For example, TSF uses counts and lengths are encoded using digit characters. Would there be a significant storage reduction if varying length bit fields were used to encode these values, and would this improve processing efficiency?

There is code is the TSF repository for converting XML and JSON to TSF. This code was written not as an end in itself, but to aid in the processing comparisons. Would standard conversions make it easier for users to convert to TSF? This dissertation describes practical conventions for converting TSF to both XML and JSON. Would these conversions support the use of TSF as an intermediate form to provide conversion between XML and JSON. Does the use of TSF as an intermediate form simplify other conversions?

Bibliography

[1] G. Pinto and F. Castor, “Energy Efficiency: A New Concern for Application Software Developers,” Communications of the ACM, December 2017.

[2] Apache Group, “Apache Avro,” 2012. [Online]. Available: https://avro.apache.org/docs/current/spec.html

[3] ——, “Apache Parquet,” 2013. [Online]. Available: https://parquet.apache.org/documentation/latest/

[4] I. MongoDB, “Bson (binary json),” 2018. [Online]. Available: http://bsonspec.org

[5] Java Language, “Java Object Serialization,” 1993. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html

[6] G. van Rossum, “Python Pickle - PEP 3154,” 2011. [Online]. Available: https://www.python.org/dev/peps/pep-3154/

[7] L. Wall, “PERL Modules DataDumper, FreezwThaw, Storable,” 1991. [Online]. Available: https://perldoc.perl.org/Storable.html

[8] The Object Management Group, “CORBA,” 2008. [Online]. Available: https://www.omg.org/spec/CORBA/3.1/Interoperability/PDF

[9] The Java Language, “Java Remote Method Invocation,” 1993. [Online]. Available: https://docs.oracle.com/en/java/javase/13/docs/api/java.rmi/module-summary.html

[10] H. Pennington *et al.*, “The D-Bus Specification,” 2003. [Online]. Available: https://dbus.freedesktop.org/doc/dbus-specification.html

[11] Apache Group, “Apache Thrift,” 2007. [Online]. Available: http://thrift.apache.org/static/files/thrift-20070401.pdf

[12] D. Winer, “XML-RPC,” 1998. [Online]. Available: http://xmlrpc.scripting.com

[13] M. Gudgin *et al.*, Eds., “SOAP Version 1.2,” 2007. [Online]. Available: https://www.w3.org/TR/soap12/

[14] International Telecommunications Union, “Specification of Abstract Syntax Notation One (ASN.1),” ITU Standard (Blue Book), 1988. [Online]. Available: https://www.itu.int/rec/T-REC-X.208/en

[15] K. McCloghrie, D. Perkins, and J. Schoenwaelder, Eds., “RFC 2578 - Structure of Management Information Version 2 (SMIv2),” 1999. [Online]. Available: https://tools.ietf.org/html/rfc2578

[16] International Telecommunications Union, “X.509 - IT OSI - Public-key and attribute certificate frameworks x.509 - it osi - public-key and attribute certificate frameworks,” 2008. [Online]. Available: https://www.itu.int/rec/T-REC-X.509

[17] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, Eds., “Extensible Markup Language (XML) 1.0,” February 1998. [Online]. Available: https://www.w3.org/TR/1998/REC-xml-19980210

[18] D. Crockford, “Introducing json,” http://json.org/, 2005. [Online]. Available: http://json.org/

[19] O. Ben-Kiki, C. Evans, and I. döt Net, "YAML Ain't Markup Language - Version 1.2," 2001. [Online]. Available: https://yaml.org/spec/1.2/spec.html

[20] R. Rivest, "S-Expressions," 1997. [Online]. Available: http://people.csail.mit.edu/rivest/Sexp.txt

[21] M. Cokus and S. Pericas-Geertsen, Eds., "XML Binary Characterization Properties, W3C Working Draft 05 October 2004," 2004. [Online]. Available: https://www.w3.org/TR/2004/WD-xbc-properties-20041005/

[22] J. Schneider *et al.*, Eds., "Efficient XML Interchange (EXI) Format 1.0 - W3C Working Draft 16 July 2007," July 2007. [Online]. Available: https://www.w3.org/TR/2007/WD-exi-20070716/

[23] J. Clark and J. Cowan, Eds., "MicroXML," October 2012. [Online]. Available: https://dvcs.w3.org/hg/microxml/raw-file/tip/spec/microxml.html

[24] C. Bormann and P. Hoffman, "Compact Binary Object Format," 2013. [Online]. Available: https://tools.ietf.org/html/rfc7049

[25] Google, "Protocol Buffers," 2008. [Online]. Available: https://developers.google.com/protocol-buffers/docs/proto

[26] W. van Oortmerssen, "Flatbuffers," 2014. [Online]. Available: https://github.com/google/flatbuffers

[27] M. Eisler, Ed., "RFC 4506 - XDR - External Data Representation Standard," 2006, obsoletes RFC 1832. [Online]. Available: https://tools.ietf.org/html/rfc4506

[28] B. Cohen, "Bencoding - Part of BitTorrent," 2008. [Online]. Available: http://bittorrent.org/beps/bep_0003.html

[29] B. Ramos, "Binn - Binary Data Serilization," 2015. [Online]. Available: https://github.com/liteserver/binn/

[30] D. Crocker, Ed., "RFC 5234 - Augmented BNF for Syntax Specifications: ABNF," Internet Engineering Task Force Request for Comments, January 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5234.txt

[31] A. J. Bernstein and P. M. Lewis, *Concurrency in Programming and Database Systems.* Jones and Bartlett Publishers, Inc, 1993.

[32] T. Bray *et al.*, Eds., "Extensible Markup Language (XML) 1.0 (Fourth Edition) - W3C Recommendation 16 August 2006," W3C Recommendation, August 2006. [Online]. Available: https://www.w3.org/TR/2006/REC-xml-20060816/

[33] The Expat Development Team, "LibExpat - Version 2.2.6," 2018. [Online]. Available: https://libexpat.github.io

[34] P. V. Biron *et al.*, Eds., "XML Schema Part 2: Datatypes Second Edition, W3C Working Draft 28 October 2004," 2004. [Online]. Available: https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[35] ECMA International, "ECMA-404 The JSON Data Interchange Standard," ECMA International, 2017. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[36] D. C. Fallside and P. Walmsley, Eds., "XML Schema Part 0: Primer Second Edition, W3C Working Draft 28 October 2004," 2004. [Online]. Available: https://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

[37] D. Gruhl, D. Meredith, and J. Pieper, "A case study on alternate representations of data structures in XML," Proceedings of the 2005 ACM symposium on Document engineering, November 2005. [Online]. Available: http://delivery.acm.org/10.1145/1100000/1096652/p217-gruhl.pdf

[38] International Telecommunications Union, "ITU-T Recommendation X.693 - Information technology – ASN.1 encoding rules: XML Encoding Rules (XER)," 2001. [Online]. Available: https://www.itu.int/ITU-T/studygroups/com17/languages/X.693-0112.pdf

[39] J. Kangasharju and S. Tarkoma, "Benefits of Alternate XML Serialization Formats in Scientific Computing ," SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches, June 2007. [Online]. Available: http://delivery.acm.org/10.1145/1280000/1272461/p23-kangasharju.pdf

[40] A. L. Hors *et al.*, Eds., "Document Object Model (DOM) Level 2 Core Specification, Version 1.0, W3C Recommendation 13 November, 2000," November 2000. [Online]. Available: https://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/

[41] D. P. Miranker and B. J. Lofaso, "The organization and performance of a treat-based production system compiler," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 1, pp. 3–10, 1991.

[42] OpenEXI Project, "A Quick Introduction to OpenEXI," web page, March 2012. [Online]. Available: https://www.dropbox.com/s/n2545xm0jjyui2d/IntroToOpenEXI.pdf?dl=0

[43] M. Oshry *et al.*, Eds., "Voice Extensible Markup Language (VoiceXML) 2.1 - W3C Recommendation 19 June 2007," 2007.

[44] D. Raggett, *Raggett on HTML 4.* Addison Wesley, 1998.

[45] A. Samaray and S. K. Makki, "A Comparison of Data Serialization Formats For Optimal Efficiency on a Mobile Platform," Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication Article No. 48, February 2012. [Online]. Available: http://delivery.acm.org/10.1145/2190000/2184810/a48-sumaray.pdf

[46] H. S. Thompson *et al.*, Eds., "XML Schema Part 1: Structures Second Edition, W3C Working Draft 28 October 2004," 2004. [Online]. Available: https://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[47] XML Working Group, "XML Conformance Test Suites," September 2013. [Online]. Available: https://www.w3.org/XML/Test/

Appendix A

Listings

The C code that follows is written using idiomatic C which follows object-oriented techniques, in effect object-oriented C. It is strict ANSI C and does not require a C++ compiler. The author feels that OO techniques are easily adapted to C code and provide the same advantages as do OO languages.

The UTF-8 (UCS-4 internal) versions of the code are presented. The primary difference is the use of the type `wchar_t`, `typedef`'ed as `Char`, and its corresponding functions, instead of the basic character type `char`. There are equivalent 8-bit versions for use with binary encoding.

The author uses three custom packages that contain low-level processing that is common to many C programs.

Memory.h These functions are an interface to the C memory allocation functions. They provide an out-of-memory error handling capability based on `setjmp.h`, a debugging capability that tracks memory allocations to avoid memory leaks, and a fence around allocation blocks that can detect some writes that exceed the boundary of an allocated block of memory.

String.h These functions manage C-string assignments, concatenations, and substrings, and the memory allocations that are needed by these operations. These functions depend upon `Memory.h`.

StringBufffer.h These functions provide string creation that does not require prior knowledge of the length of a string. String buffers are chunks of string data that are linked together internally. The functions provide an interface that hides the fact that the data is not stored contiguously. These functions depend upon `Memory.h`.

A.1 NpxUCElement.h

Declaration of the method table for the NpxUCElement object. This is the only "public" member of the NpxUCElement object.

```
#ifndef NpxUCElement_INCLUDE_GUARD
#define NpxUCElement_INCLUDE_GUARD
#ifdef NpxUCElement_EXPORT
#define NpxUCElement_API
#else
#define NpxUCElement_API extern
#endif
#define UCS_CHAR
#ifdef UCS_CHAR
typedef wchar_t Char;
#else
typedef char Char;
#endif
typedef char String;
typedef struct NpxUCElement NpxUCElement;
typedef struct NpxUCElementMtb
{
    void (*add)(NpxUCElement *, Char *str, int nfm, int ndx, int idx, char typ);
    void (*addElem)(NpxUCElement *, Char *nm, NpxUCElement *elm, char typ);
    void (*forEachContent)  (NpxUCElement *s, void *prm, void (*cb)(NpxUCElement *,
        int, Char *nm, void *obj, char typ, void *prm));
    int  (*isSLU)(NpxUCElement *, Char *tp);
    int  (*isPLU)(NpxUCElement *, Char *tp);
    Char *(*getObjName)(NpxUCElement *, void *obj);
    void *(*getObjByName)(NpxUCElement *, Char *nm);
    int  (*hasContent)(NpxUCElement *);
    int  (*getElementCount)(NpxUCElement *s);
    char (*getElementType)(NpxUCElement *s, int sub);
    Char *(*getElementName)(NpxUCElement *s, int sub);
    void *(*getElementValue)(NpxUCElement *s, int sub);
    NpxUCElement *(*getParent)(NpxUCElement *s);
    Char *(*getTransaction)(NpxUCElement *s);
} NpxUCElementMtb;
struct NpxUCElement
{
    NpxUCElementMtb *mtb;
};
NpxUCElement_API NpxUCElement *newNpxUCElement(Char *str, int esiz, NpxUCElement
    *par);
NpxUCElement_API void delNpxUCElement(NpxUCElement *);
#undef NpxUCElement_API
#endif
```

Listing A.1: NpxElement.h

A.2 NpxUCElement.c

The NpxElement.c file is the implementation of the NpxElement object, and contains all the methods referenced in the method table. It also declares the NpxElementI structure which is the ”private” implementation of the properties of the NpxElement object. The code uses an interface (Memory.h) to the C library dynamic memory management functions.

```
#include <wchar.h>

#include "Memory.h"
/**
 * Import the API declarations.
*/
#define API_EXPORT
#include "NpxUCElement.h"
#undef API_EXPORT
#include "NpxRoot.h"
#define arraySize(a) ((sizeof a)/(sizeof a[0]))

typedef char String;

static void add(NpxUCElement *self, Char *str, int nfm, int fm, int to, char typ);
static void addElem(NpxUCElement *self, Char *nm, NpxUCElement *cntnt, char typ);
static void forEachContent(NpxUCElement *s, void *, void (*callback)(NpxUCElement
    *, int, Char *name, void *obj, char typ, void *prm));
static int  isSLU(NpxUCElement *s, Char *tp);
static int  isPLU(NpxUCElement *s, Char *tp);
static Char *getObjName(NpxUCElement *s, void *obj);
static void *getObjByName(NpxUCElement *s, Char *nm);
static int  hasContent(NpxUCElement *s);
static int  getElementCount(NpxUCElement *s);
static char getElementType(NpxUCElement *s, int sub);
static Char *getElementName(NpxUCElement *s, int sub);
static void *getElementValue(NpxUCElement *s, int sub);
static NpxUCElement *getParent(NpxUCElement *s);
static Char *getTransaction(NpxUCElement *s);

/* method table */
static NpxUCElementMtb npxUCElementMtb = {add, addElem, forEachContent,
    isSLU, isPLU, getObjName, getObjByName,
    hasContent, getElementCount, getElementType, getElementName, getElementValue,
        getParent, getTransaction};

typedef struct NpxUCElementI
{
    NpxUCElementMtb *mtb;
    Char            *xact;
    NpxUCElement    *parent;
    Char            **elemNames;
    void            **elemValues;
    unsigned short  elemCount;
    unsigned short  elemSize;
    char            elemTypes[8];
} NpxUCElementI;
/**
 * Construct a new NpxUCElement object.  The elemType array is sized in units of 8
     chars.  If the passed esiz is
 * greater than 8, it requires additional storage allocated at the beginning of the
     variable section of the
 * object in units of 8 bytes. E.g. if esiz is 23, it requires 24 bytes or 2
     additional units the size of elemTypes.
 * ((23 - 1) / 8) * 8) = 16 additional bytes
*/
NpxUCElement *newNpxUCElement(Char *str, int esiz, NpxUCElement *par)
{
    NpxUCElementI *self;
    unsigned siz, szt;
    int ix;
    siz = sizeof(NpxUCElementI);
    siz += esiz * (sizeof(*self->elemNames) + sizeof(*self->elemValues));
    szt = sizeof(self->elemTypes);
    if (esiz > szt)
        szt = ((esiz - 1) / szt) * szt;
    else
        szt = 0;
    self = mAlloc(siz + szt, "newNpxUCElement");
    self->mtb = &npxUCElementMtb;
    self->parent = par;
    self->xact = str;
    self->elemSize = esiz;
    self->elemCount = 0;
    self->elemNames = (Char **)((char *)(self + 1) + szt);
    for (ix = 0; ix < (int)(sizeof self->elemTypes + szt); ++ix)
        self->elemTypes[ix] = '\0';
    self->elemValues = (void **)&self->elemNames[esiz];
    return (NpxUCElement *)self;
}
void delNpxUCElement(NpxUCElement *npxUCElement)
{
    int ix, ct;
    NpxUCElementI *self = (NpxUCElementI *)npxUCElement;
    ct = self->elemCount;
    for (ix = 0; ix < ct; ++ix)
    {
        if (0 != (ctype[self->elemTypes[ix]] & SL) && 0 != self->elemValues[ix])
            delNpxUCElement((NpxUCElement *)self->elemValues[ix]);
    }
    mFree((NpxUCElementI *)npxUCElement, "delNpxUCElementI");
}
static void forEachContent(NpxUCElement *s, void *prm, void
    (*callback)(NpxUCElement *, int, Char *name, void *obj, char typ, void *prm))
{
    int ix;
    NpxUCElementI *self = (NpxUCElementI *)s;
    for (ix = 0; ix < self->elemCount; ++ix)
    {
        callback(s, ix, self->elemNames[ix], self->elemValues[ix],
            self->elemTypes[ix], prm);
    }
}
static int isSLU(NpxUCElement *s, Char *tp)
{
    Char typ = *tp;
    if (typ < arraySize(ctype) && 0 != (typ & SL))
        return 1;
    return 0;
}
static int isPLU(NpxUCElement *s, Char *tp)
{
    unsigned char typ = *tp;
    if (typ < arraySize(ctype) && 0 != (typ & PL))
        return 1;
    return 0;
}
static Char *getObjName(NpxUCElement *s, void *obj)
{
    int ix;
    NpxUCElementI *self = (NpxUCElementI *)s;
    for (ix = 0; ix <self->elemCount; ++ix)
    {
        if (self->elemValues[ix] == obj)
            return self->elemNames[ix];
    }
    return 0;
}
static void *getObjByName(NpxUCElement *s, Char *nm)
{
    int ix;
    NpxUCElementI *self = (NpxUCElementI *)s;
    for (ix = 0; ix <self->elemCount; ++ix)
    {
        if (0 == wcscmp(self->elemNames[ix], nm))
            return self->elemValues[ix];
    }
    return 0;
}
static int getElementCount(NpxUCElement *s)
{
    NpxUCElementI *self = (NpxUCElementI *)s;
    return self->elemCount;
}
static char getElementType(NpxUCElement *s, int sub)
{
    NpxUCElementI *self = (NpxUCElementI *)s;
    return self->elemTypes[sub];
}
static Char *getElementName(NpxUCElement *s, int sub)
{
    NpxUCElementI *self = (NpxUCElementI *)s;
    return self->elemNames[sub];
}
static void *getElementValue(NpxUCElement *s, int sub)
{
    NpxUCElementI *self = (NpxUCElementI *)s;
    return self->elemValues[sub];
}
static NpxUCElement *getParent(NpxUCElement *s)
{
    NpxUCElementI *self = (NpxUCElementI *)s;
    return self->parent;
}
static Char *getTransaction(NpxUCElement *s)
{
    NpxUCElementI *self = (NpxUCElementI *)s;
    return self->xact;
}
static int hasContent(NpxUCElement *s)
{
    NpxUCElementI *self = (NpxUCElementI *)s;
    return 0 != self->elemCount;
}
static void add(NpxUCElement *s, Char *str, int nfm, int fm, int to, char typ)
{
    int sub;
    NpxUCElementI *self = (NpxUCElementI *)s;
    sub = self->elemCount++;
    self->elemNames[sub] = (nfm == fm) ? (Char *)0 : (Char *)&str[nfm];
    self->elemValues[sub] = (void *)&str[fm + 1];
    self->elemTypes[sub] = typ;
}
static void addElem(NpxUCElement *s, Char *nm, NpxUCElement *cntnt, char typ)
{
    int sub;
    NpxUCElementI *self = (NpxUCElementI *)s;
    sub = self->elemCount++;
    self->elemNames[sub] = nm;
    self->elemValues[sub] = (void *)cntnt;
    self->elemTypes[sub] = typ;
}
```

Listing A.2: NpxElement.c

A.3 Serialization/Deserialization Code

The serialization and deserialization conditionally compile either 8-bit or UCS-4 (`wchar_t`) functions.

```
//
//  main.c
//  Test driver for Npx
//
//  Created by Dan Evans on 8/31/17.
//  Copyright    2017 Dan Evans. All rights reserved.
//
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <wchar.h>
#include <setjmp.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/resource.h>
/* #define MEMORY_TRAK */
#include "Memory.h"
#include "Sstring.h"
#include "StringBuffer.h"
#include "UCStringBuffer.h"
#include "NpxElement.h"
#include "NpxUCElement.h"

#define self(s) (*s->mtb)

extern int npx2XML(NpxElement *elm, StringBuffer *sb, int nopts, ...);
extern int npx2XMLUC(NpxUCElement *elm, UCStringBuffer *sb, int nopts, ...);

/* extern NpxElement *deserialize(unsigned char *, int len); */
extern NpxUCElement *deserializeUC(unsigned char *, int len);
extern void *expatDeserialize(unsigned char *buf, int len);
extern char *serialize(NpxElement *);
extern void delDocumentTree(void *root);
extern clock_t getDOMTime();
extern clock_t getTAMTime();


typedef char String;
```

```
static char *version = "2.0";
static char *separator = "/";
static jmp_buf env;
static int opt_writexmlc = 0;
static int opt_reserialize = 0;
static int opt_repeat = 1;
static FILE *opt_dupfile = 0;
static int opt_expat = 0;
static char *opt_suffixes[10];

static UCStringBuffer *utf8toUCS(const unsigned char *str, int ln)
{
    int ix;
    unsigned char ch;
    UCStringBuffer *wbuf;
    wbuf = newUCStringBuffer((UCStringBuffer *) 0, 0);
    ix = 0;
    while (ix < ln)
    {
        ch = str[ix];
        if (ch < 0x80)
        {
            *ucstringBufferPostInc(wbuf) = (wchar_t) ch;
            ix += 1;
        }
        else if (ch >= 0xc0)
        {
            if (ch < 0xe0)
            {
                *ucstringBufferPostInc(wbuf) = (wchar_t) ((ch & 0x1fu) << 6 |
                    (str[ix + 1] & 0x3fu));
                ix += 2;
            }
            else if (ch < 0xf0)
            {
                *ucstringBufferPostInc(wbuf) = (wchar_t) (((ch & 0x0fu) << 12) |
                    ((str[ix + 1] & 0x3fu) << 6) |
                                                          (str[ix + 2] & 0x3f));
                ix += 3;
            }
            else if (ch < 0xf8)
            {
                *ucstringBufferPostInc(wbuf) = (wchar_t) (((ch & 0x07u) << 18) |
                    ((str[ix + 1] & 0x3fu) << 12) |
                                                          ((str[ix + 2] & 0x3fu) <<
                                                              6) | (str[ix + 3] &
                                                              0x3fu));
                ix += 4;
            }
            else if (ch < 0xfc)
            {
                *ucstringBufferPostInc(wbuf) = (wchar_t) (((ch & 0x03u) << 24) |
                    ((str[ix + 1] & 0x3fu) << 18) |
                                                          ((str[ix + 2] & 0x3fu) <<
                                                              12) | ((str[ix + 3] &
                                                              0x3fu) << 6) |
                                                          (str[ix + 4] & 0x3fu));
                ix += 5;
            }
            else if (ch < 0xfe)
            {
                *ucstringBufferPostInc(wbuf) = (wchar_t) (((ch & 0x01u) << 30) |
                    ((str[ix + 1] & 0x3fu) << 24) |
                                                    ((str[ix + 2] & 0x3fu) <<
                                                        18) | ((str[ix + 3] &
                                                        0x3fu) << 12) |
                                                    ((str[ix + 4] & 0x3fu) <<
                                                         6) |  (str[ix + 4] &
                                                        0x3fu));
                ix += 6;
            }
            else
            {
                /* encoding error */
                delUCStringBuffer(wbuf);
                return 0;
            }
        }
        else
        {
            /* encoding error */
            delUCStringBuffer(wbuf);
            return 0;
        }
    }
    return wbuf;
}
static int wchar2Utf8(wchar_t ch, StringBuffer *sb)
{
    int ix = 0;
    if (ch < 0x0080)
    {
        *stringBufferPostInc(sb) = (char)ch;
        return 1;
    }
    if (ch < 0x0800)
    {
        *stringBufferPostInc(sb) = (char)(((ch & 0x07c0u) >> 6) |  0xc0u);
        ix = 2;
        goto l0;
    }
    else if (ch < 0x10000)
    {
        *stringBufferPostInc(sb) = (char)(((ch & 0xf000) >> 12) |  0xe0);
        ix = 3;
        goto l6;
    }
    else if (ch < 0x200000)
    {
        *stringBufferPostInc(sb) = (char)(((ch & 0x1c0000) >> 18) | 0xf0);
        ix = 4;
        goto l12;
    }
    else if (ch < 0x4000000)
    {
        *stringBufferPostInc(sb) = (char)(((ch & 0x3000000) >> 24) | 0xf80);
        ix = 5;
        goto l18;
    }
    else
    {
        *stringBufferPostInc(sb) = (char)(((ch & 0x40000000) >> 30) | 0xfc0);
        ix = 6;
        goto l24;
    }
  l24:
    *stringBufferPostInc(sb) = (char)(((ch & 0x3f000000) >> 24) | 0x80);
  l18:
    *stringBufferPostInc(sb) = (char)(((ch & 0x00fc0000) >> 18) | 0x80);
  l12:
    *stringBufferPostInc(sb) = (char)(((ch & 0x0003f000) >> 12) | 0x80);
  l6:
    *stringBufferPostInc(sb) = (char)(((ch & 0x00000fc0) >>  6) | 0x80);
  l0:
    *stringBufferPostInc(sb) = (char)(( ch & 0x0000003f) | 0x80);
    return ix;
}
static StringBuffer *ucsToUtf8(wchar_t *wbuf, StringBuffer *sb)
{
    int ix;
    for (ix = 0; L'\0' != wbuf[ix]; ++ix)
    {
        wchar2Utf8(wbuf[ix], sb);
    }
    return sb;
}
static StringBuffer *ucsToUtf8L(wchar_t *wbuf, int len, StringBuffer *sb)
{
    int ix;
    for (ix = 0; ix < len; ++ix)
    {
        wchar2Utf8(wbuf[ix], sb);
    }
    return sb;
}
static int rprintf(const char *format, ...)
{
    int rtn;
    va_list args;
    va_start(args, format);
    rtn = vfprintf(stdout, format, args);
    va_end(args);
    if (0 != opt_dupfile)
    {
        va_start(args, format);
        vfprintf(opt_dupfile, format, args);
        va_end(args);
    }
    return rtn;
}
static int rwprintf(const wchar_t *format, ...)
{
    int rtn;
    va_list args;
    va_start(args, format);
    rtn = vfwprintf(stdout, format, args);
    va_end(args);
    if (0 != opt_dupfile)
    {
        va_start(args, format);
        vfwprintf(opt_dupfile, format, args);
        va_end(args);
    }
    return rtn;
}
static long timediff(struct timeval *tfm, struct timeval *tto)
{
    return (tto->tv_sec * 1000000 + tto->tv_usec) - (tfm->tv_sec * 1000000 +
        tfm->tv_usec);
}
static int exec(const char *fileName)
{
    FILE *file, *ofil;
    unsigned char *buf;
    char *cp, *outf;
    wchar_t *cbuf, *wdoc;
    long siz;
    struct stat statbuf;
    NpxUCElement *dsobj;
    void *exobj;
    UCStringBuffer *ucsb;
    clock_t sttime = 0, entime = 0;
    struct timeval sttv, entv;
    int rtn = 0, ix;
    unsigned len;
    struct rusage rsc;
    if (0 == (file = fopen(fileName, "rb")))
    {
        fprintf(stderr, "unable␣to␣open␣%s\n", fileName);
        rtn = -1;
        goto ex1;
    }
    dsobj = 0;
    fstat(fileno(file), &statbuf);
    siz = statbuf.st_size;
    buf = mAlloc((int) siz + 1, "execbuf");
    if (siz != fread(buf, sizeof(char), siz, file))
    {
        fprintf(stderr, "error␣reading␣file␣%s␣(size␣%ld)\n", fileName, siz);
        rtn = -1;
        goto ex2;
    }
    buf[siz] = '\0';
    rprintf("read␣%ld␣byte␣file␣%s\n", siz, fileName);

    for (ix = 1; ix <= opt_repeat; ++ix)
    {
        if (0 != opt_expat)
        {
            getrusage(0, &rsc);
            sttv = rsc.ru_utime;
            //sttime = clock();
            if (0 == (exobj = expatDeserialize(buf, (int)siz)))
            {
                fprintf(stderr, "expat␣deserialize␣failed\n");
                rtn = -1;
                goto ex2;
            }
            //entime = clock();
            //rprintf("DOM build time %ld\n", getDOMTime());
            getrusage(0, &rsc);
            entv = rsc.ru_utime;
            rprintf("rusage␣CPU␣%ld\n", timediff(&sttv, &entv));
            delDocumentTree(exobj);
        }
        else
        {
            getrusage(0, &rsc);
            sttv = rsc.ru_utime;
            //sttime = clock();
            if (0 == (dsobj = deserializeUC(buf, (int)siz)))
            {
                fprintf(stderr, "deserializeUC␣failed\n");
                rtn = -1;
                goto ex2;
            }
            //entime = clock();
            //rprintf("TAM build time %ld\n", getTAMTime());
            getrusage(0, &rsc);
            entv = rsc.ru_utime;
            rprintf("rusage␣CPU␣%ld\n", timediff(&sttv, &entv));
            if (ix != opt_repeat)
            {
                cbuf = self(dsobj).getTransaction(dsobj);
                mFree(cbuf, "wcxact");
                delNpxUCElement(dsobj);
                dsobj = 0;
            }
        }
        rprintf("deserializeUC␣time␣%ld␣(clocks␣per␣sec␣%d)\n", entime - sttime,
            CLOCKS_PER_SEC);
    }
    if (opt_writexmlc != 0)
    {
        ucsb = newUCStringBuffer((UCStringBuffer *)0, 0);
        rtn = npx2XMLUC(dsobj, ucsb, 1, (int)1);
        wdoc = ucstringBufferToString(ucsb, 0);
        if (opt_writexmlc != 0 && 0 != (cp = strstr(fileName, ".txt")) && 4 ==
            strlen(cp))
        {
            siz = ucstringBufferGetOffset(ucsb) - 1;
            cp = stringCut(fileName, cp + 1);
            outf = stringConcat(cp, "xml.outc");
            delString(cp);
            if (0 == (ofil = fopen(outf, "wb")))
            {
                fprintf(stderr, "unable␣to␣open␣%s\n", outf);
                rtn = -1;
                goto ex3;
            }
```

```
            rprintf("writing␣%s\n", outf);
            if (siz != fwrite(wdoc, sizeof(char), siz, ofil))
            {
                fprintf(stderr, "error␣writing␣file␣%s␣(size␣%ld)\n", outf, siz);
                rtn = -1;
                goto ex4;
            }
            ex4:
            fclose(ofil);
            ex3:
            delString(outf);
        }
        if (1 == opt_reserialize)
            fwprintf(stdout, L"%ls\n", wdoc);
        mFree(wdoc, "ucsBuf2string");
        delUCStringBuffer(ucsb);
    }
/*    if (1 == opt_reserialize && 0 == opt_writexmlc)
    {
        doc = serialize(dsobj);
        fprintf(stdout, "%s\n", doc);
        delString(doc);
    }*/
ex2:
    if (0 != dsobj)
    {
        cbuf = self(dsobj).getTransaction(dsobj);
        mFree(cbuf, "wcxact2");
        delNpxUCElement(dsobj);
    }
    mFree(buf, "execbufF");
    fclose(file);
ex1:
    return rtn;
}
static int execDir(const char *dirName)
{
    DIR           *d;
    struct dirent *dir;
    char *cp, *path;
    int ix, found;
    d = opendir(dirName);
    if (0 != d)
    {
        while ((dir = readdir(d)) != NULL)
        {
            found = 0;
            for (ix = 0; 0 != opt_suffixes[ix]; ++ix)
            {
                if (0 != (cp = strstr(dir->d_name, opt_suffixes[ix])) &&
                    strlen(opt_suffixes[ix]) == strlen(cp))
                {
                    found = 1;
                    break;
                }
            }
            if (0 == found)
                continue;
            rprintf("processing␣%s\n", dir->d_name);
            path = stringConcatV(dirName, separator, dir->d_name, (char *)0);
            exec(path);
            delString(path);
        }
        closedir(d);
    }
    else
    {
        fprintf(stderr, "cannot␣open␣directory␣%s\n", dirName);
        return -1;
    }
    return 0;
}
static void usage(const char *pgm)
{
    fprintf(stderr, "%s␣[-e]␣[-h]␣[-ofile]␣[-rn]␣[-v]␣[-w]␣[-x]␣path\n", pgm);
    fprintf(stderr, "␣␣-e␣␣deserialize␣input␣XML␣files␣using␣Expat\n");
    fprintf(stderr, "␣␣-h␣␣display␣this␣help␣information\n");
    fprintf(stderr, "␣␣-ofile␣␣duplicate␣stdout␣to␣this␣file\n");
    fprintf(stderr, "␣␣-rn␣␣repeat␣the␣deserialization␣n␣times\n");
    fprintf(stderr, "␣␣-ssuffix␣␣if␣path␣is␣a␣directory,␣input␣files␣have␣this␣
        suffix␣(default␣.txt)\n");
    fprintf(stderr, "␣␣␣␣␣␣␣␣␣␣␣␣␣(may␣be␣repeated␣for␣a␣maximum␣of␣9␣suffixes)\n");
    fprintf(stderr, "␣␣-v␣␣display␣the␣version␣and␣exit\n");
    fprintf(stderr, "␣␣-w␣␣write␣deserialized␣output␣to␣.xml.outc␣file\n");
    fprintf(stderr, "␣␣-x␣␣write␣reserialized␣XML␣version␣to␣.xml.outc␣file\n");
    exit(1);
}
int main(int argc, const char *argv[])
{
    struct stat statbuf;
    int n, ni, ss, sl;
    if (setjmp(env))
    {
        exit(254);
    }
    ss = 0;
    ni = 1;
    for (n = 1; n < argc; n += ni)
    {
        if (argv[n][0] == '-')
        {
            switch (argv[n][1])
            {
                case 'e':
                    opt_expat = 1;
                    break;
                case 'h':
                    usage(argv[0]);
                    break;
                case 'o':
                    opt_dupfile = fopen(&argv[n][2], "wb");
                    break;
                case 'r':
                    opt_repeat = (int) strtol(&argv[n][2], (char **) 0, 10);
                    break;
                case 's':
                    if (ss < 9)
                    {
                        sl = (int)strlen(&argv[n][2]);
                        opt_suffixes[ss] = mAlloc(sl + 1, "optSuf");
                        memcpy(opt_suffixes[ss], &argv[n][2], sl);
                        opt_suffixes[ss][sl] = '\0';
                        ++ss;
                    }
                    break;
                case 'v':
                    fprintf(stderr, "Version:␣%s\n", version);
                    exit(1);
                case 'w':
                    opt_writexmlc = 1;
                    break;
                case 'x':
                    opt_reserialize = 1;
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    if (0 == ss)
    {
        opt_suffixes[ss] = mAlloc(5, "optSuf");
        strcpy(opt_suffixes[ss], ".txt");
        ++ss;
    }
    opt_suffixes[ss] = 0;
    for (n = 1; n < argc; n += ni)
    {
        if (argv[n][0] != '-')
        {
            if (0 <= lstat(argv[n], &statbuf))
            {
                if (0 != ((unsigned)S_IFDIR & statbuf.st_mode))
                {
                    execDir(argv[n]);
                }
                else if (0 != ((unsigned)S_IFREG & statbuf.st_mode))
                {
                    exec(argv[n]);
                }
            }
            else
            {
                fprintf(stderr, "cannot␣stat␣%s\n", argv[n]);
            }
        }
    }
    for (n = 0; 0 != opt_suffixes[n]; n +=1)
    {
        mFree(opt_suffixes[n], "optSuf");
    }
#ifdef MEMORY_TRAK
    trakReport(__FILE__);
```

```
#endif
    return 0;
}
```

Listing A.3: NpxDeserial.c