A Portable C++ Framework for the Serialization of Floating-Point Numbers

by

Michael Crawford

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science

in partial fulfillment of the requirements for the degree of

Bachelor of Science

May 2014

APPROVED BY:

Cindy Norris, Ph.D., Thesis Project Director

E. Frank Barry, M.Sc., Second Reader

Dee Parks, Ph.D., Departmental Honors Director

James Wilkes, Ph.D., Chair, Computer Science

ABSTRACT

A Portable C++ Framework for the Serialization of Floating-Point Numbers.

(May 2014)

Michael Crawford, Appalachian State University

Appalachian State University

Thesis Chairperson: Cindy Norris, Ph.D.

The ISO C++ Standard standard does not specify the storage size or binary representation used for floating-point types. This ambiguity creates difficulties when a developer wishes to serialize floating-point values using a binary format that allows them to be successfully de-serialized on a variety of other systems. The most commonly used standard for floating-point values is the IEEE-754 Standard for Binary Floating-Point. However, this standard does not strictly define all aspects of its implementation. This means that even when the IEEE-754 standard is used, floating-point values can not be reliably serialized on one system that implements the standard and de-serialized on a different system implementing the same standard. Attempting to serialize values on an IEEE-754 system and de-serialize them on a non IEEE-754 (or the inverse) is a more significant challenge. This thesis presents a framework that may be used to serialize floating-point values to a standardized format and de-serialize those values on a variety of platforms. The framework is also placed through testing in order to show its viability.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The ISO C++ Standard defines three floating-point types to be used for the representation of real numbers: `float`, `double`, and `long double`. However, the standard does not specify the storage size or binary representation used for these types. This ambiguity creates difficulties when a developer wishes to serialize floating-point values using a binary format that allows them to be successfully de-serialized on a variety of other systems.

The most commonly used standard for floating-point values is the IEEE-754 Standard for Binary Floating-Point. However, this standard does not strictly define all aspects of its implementation. This means that even when the IEEE-754 standard is used, floating-point values can not be reliably serialized on one system that implements the standard and de-serialized on a different system implementing the same standard. Attempting to serialize values on an IEEE-754 system and de-serialize them on a non IEEE-754 (or the inverse) presents a more significant challenge.

This thesis presents a framework that may be used to serialize floating-point values to a standardized format and de-serialize those values on a variety of platforms. Developers using this framework must still implement appropriate code to convert the framework's floating-point format to that of the systems that they are using. Though perhaps less than ideal, this represents a substantial improvement over the current situation in which a developer is required to deal with floating-point data that is in an ambiguous or even unknown format.

Chapter 2 of the thesis provides background information related to the implementation of floating-point in C++, as well as details related to the IEEE-754 standard and the format that

it uses for the representation of floating-point values. Also discussed is the Boost Serialization Library, used by this thesis to provide the serialization backbone upon which the floating-point framework is implemented.

Chapter 3 provides additional detail regarding the challenges that must be addressed in order to serialize floating-point values successfully.

Chapter 4 describes the proposed framework, the utilities provided to assist in using the framework, and demonstrates the usage of the framework.

Chapter 5 discusses the testing that was done to verify the functionality of the framework utilities as well as the viability of the framework itself.

Chapter 6 summarizes this framework and its potential applications, as well as discussing possible future work to improve upon the framework.

Appendix A provides a listing of reports that detail the floating-point features of the systems used during testing.

Chapter 2

Background

Before describing the proposed framework, it is necessary to provide some details regarding the usage of floating-point within C++, the most common floating-point standard, and the Boost Serialization library.

2.1 Floating-Point in C++

The ISO C++11 Standard (3.9.1) [6] lists the following requirements for floating-point types:

> There are three *floating point* types: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much precision as `double`. The set of values of the type `float` is a subset of the set of values of the type `double`; the set of values of the type `double` is a subset of the set of values of the type `long double`. The value representation of floating-point types is implementation-defined.

The standard does not define how floating point numbers shall be represented, leaving compiler implementers free to use any floating-point format they desire while maintaining conformance with the ISO standard.

2.2 The IEEE-754 Standard for Binary Floating-Point

IEEE-754 is the most common floating-point format, and it is the de facto standard for floating-point on home computers. Most modern CPUs include dedicated hardware for performing floating-point operations, such as the Intel 80x87 series of math co-processors. Typically this dedicated floating-point hardware implements the IEEE-754 standard [3], which specifies the binary format of floating-point numbers as well as the semantics of floating-point operations. Processors that lack dedicated hardware often implement floating-point using software libraries, which may also conform to the IEEE-754 standard. The most notable exceptions are VAX and Cray computers, whose processors typically use a proprietary floating-point format [8].

Floating-Point Types

IEEE-754 defines three floating point types [2] that are used in order to implement the types required by ISO C++.

Single Precision: A 32 bit type that is used as the C++ `float`.

Double Precision: A 64 bit type that is used as the C++ `double`.

Extended Double Precision: A type used as `long double` that is implementation-defined, but must meet certain minimum requirements laid out by the standard in order to provide a greater level of precision than the Double Precision format. `long double` is most commonly implemented as a type with a width of either 80 or 128 bits. If the platform does not implement an Extended Precision type, then `long double` is an alias for `double`.

Unless otherwise specified in this document, `float` will refer to the IEEE-754 Single Precision type, `double` will refer to the IEEE-754 Double Precision type, and `long double` will refer to the 80 bit Extended Double Precision type.

Floating-Point Format

All of the floating-point types share a common layout in memory. The number is divided into three fields, whose size may vary depending on the specific floating-point type being im-

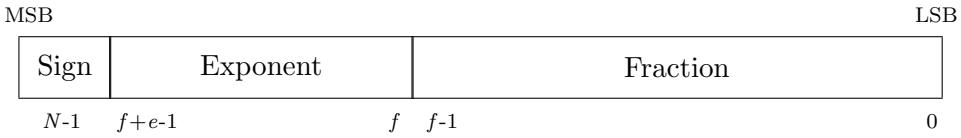

Figure 2.1: The memory layout of a `float`.

plemented. The three fields within the number are the sign, the exponent, and the fraction. Figure 2.1 demonstrates the memory layout.

The following terminology is used when referring to the fields of a floating-point number, and is common to all types of floating-point values:

Number of Bits (N): The total size of the type in bits.

Sign Bit (S): The most significant bit of each type is used to encode the sign of the value. $S = 0$ represents a positive value, while $S = 1$ represents a negative value.

Exponent (E): A signed integer exponent that modifies the magnitude of the number. For each type, valid exponent values are in the range $[E_{Min}, E_{Max}]$. $E_{Min} - 1$ and $E_{Max} + 1$ are both reserved to implement special values.

Exponent Bits (e): The number of bits that are used to encode the exponent.

Bias (b): A constant that is combined with the exponent to transform it into a non-negative integer. The bias for a type may be calculated as $b = 2^{e-1} - 1$.

Biased Exponent (E_b): The unsigned integer exponent value that is actually encoded in the exponent field of the floating-point number. The biased exponent is calculated as $E_b = E + b$.

Fraction Bits (f): The number of bits used to encode the fractional part of the value.

Fraction (F): A binary string consisting of f digits that determines the precision of the number being represented. If the value does not require f digits, the fraction is right-padded with zeros until it contains f digits. The fraction may also be referred to as the significand, or the mantissa.

Table 2.1: Common implementations of floating point types.

Type	N	e	b	E_{Min}	E_{Max}	f
`float`	32	8	127	-126	+127	23
`double`	64	11	1023	-1022	+1023	52
`long double`	80	15	16383	-16382	+16383	64
`long double`	128	15	16383	-16382	+16383	112

Value (V): The decimal value represented by the floating-point number. How the value is computed is dependent on the type being represented.

Many of the fields have values that are constant for each floating-point type; these values are listed in Table 2.1 [9].

Floating-Point Values

In order to provide maximum flexibility in representing a wide range of values as well as the ability to handle edge cases during computations, IEEE-754 defines four different types of floating-point values.

Normal Numbers

Normal numbers are the most commonly used floating-point values, representing most real numbers. Normal numbers modify the following field terminology:

Fraction (F): In order to provide additional precision, the fraction of a normal number is implied to have a leading one. When encoding a normal number, an appropriate exponent must be computed so that the fraction's leading digit is one. This digit is then dropped from the fraction string, and is not encoded into the fraction field of the number.

Value (V): The value of a normal number may be calculated as $V = (-1)^S \cdot (1.F)_2 \cdot 2^E$.

For example, the first six digits of π (3.14159) maybe encoded as a `float` in the form $(-1)^0 \cdot 1.10010010000111111101_2 \cdot 2^1$. The values used to form the `float` are $S = 0$, $E_b = 128$, and $F = 10010010000111111010000$, as shown in Figure 2.2.

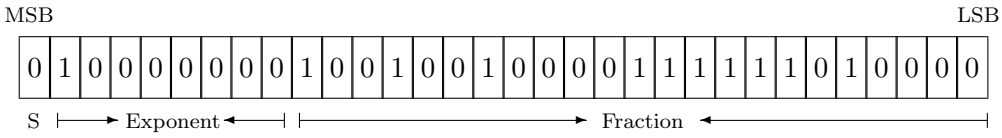

Figure 2.2: The binary representation of π as a `float`.

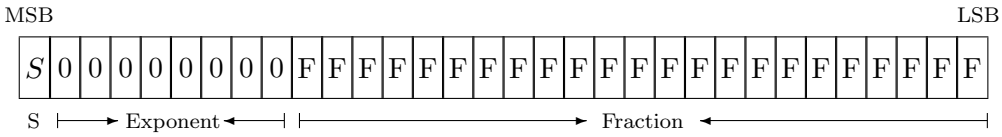

Figure 2.3: The binary representation of denormal `float` values.

Denormal Numbers

In some situations, a floating-point operation may yield a result that is not zero, but is smaller than the smallest possible normal number. Since it is undesirable to have an unexpected divide-by-zero or a situation where, for example, $a - b == 0$ even though $a \neq b$, IEEE-754 implements the concept of gradual underflow using denormal numbers to represent these very small values.

Additionally, normal numbers are not capable of representing zero due to the implied leading one of the fraction field, so zero must be represented as a denormal number. This means that both $+0$ and -0 are distinct floating-point numbers. However, the IEEE-754 standard does dictate that $+0 == -0$ always evaluates as true.

Denormal numbers modify the following field terminology:

Exponent (E): The exponent of a denormal number is fixed, and is always $E_{Min} - 1$. This corresponds to a biased exponent of zero.

Fraction (F): The fraction of a denormal number does not have an implied leading digit. Alternatively, it may be considered to have an implied leading zero.

Value (V): The value of a denormal number may be calculated as $V = (-1)^S \cdot (0.F)_2 \cdot 2^{E_{Min}-1}$.

Figure 2.3 illustrates the layout of a denormal number.

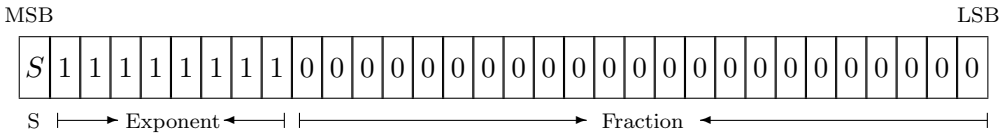

Figure 2.4: The binary representation of ∞ as a `float`.

Table 2.2: Operations involving ∞.

Operation	**Result**
$x/\pm\infty$	0
$\pm\infty \cdot \pm\infty$	$\pm\infty$
$x/0 : x \neq 0$	$\pm\infty$
$\pm 0/\pm 0$	NaN
$\infty + \infty$	∞
$\infty - \infty$	NaN
$\pm\infty/\pm\infty$	NaN
$\pm\infty \cdot 0$	NaN

Infinity

IEEE-754 requires that computations continue, even if the result of an operation has a magnitude greater than that which can be represented by the floating-point type as a normal value. Infinity is used to represent these overflow values. Infinity values modify the following field terminology:

Exponent (E): The exponent of an infinity value is fixed, and is always $E_{Max} + 1$. This corresponds to a biased exponent in which all bits of the exponent field are set to one.

Fraction (F): The fraction field of an infinity value is fixed, and consists of all zeros.

Value (V): No computation is necessary for an infinity value, as the only portion of the value which varies is the sign bit in order to distinguish $+\infty$ and $-\infty$.

Figure 2.4 shows the layout of an infinity value. The result of IEEE-754 operations that involve infinity sometimes produce results different than what one would expect based on standard mathematical operations. Table 2.2 lists the results of some common operations using infinity [8].

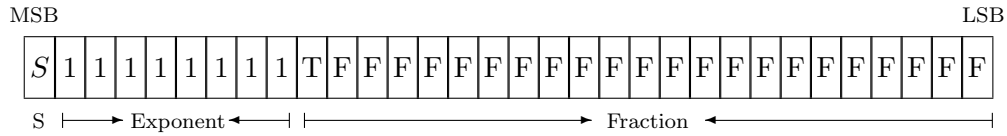

Figure 2.5: The binary representation of NaN as a `float`.

Not a Number

As with infinity values, Not a Number (NaN) is implemented by IEEE-754 in order to allow computations to continue in the case of exceptional conditions such as divide by zero or $\sqrt{x} : x < 0$ by defining an "invalid number" value that may be used as the result. Additionally, the standard defines two types of NaN: Quiet NaN (QNaN) and Signaling NaN (SNaN). When a SNaN value is used in an arithmetic operation an exception is signaled, while a QNaN used in an arithmetic operation allows computations to continue without interruption. NaN values modify the following field terminology:

Sign Bit (S): The sign bit of a NaN value is ignored; that is, there is no distinction between +NaN and -NaN.

Exponent (E): The exponent of a NaN value is fixed, and is always $E_{Max} + 1$. This corresponds to a biased exponent in which all bits of the exponent field are set to one.

Fraction (F): The fraction field of a NaN must be non-zero. The fraction field is also used to distinguish between quiet and signaling NaN values. The original IEEE-754-1985 standard specified that QNaN and SNaN values were implementation-defined, however the IEEE-754-2008 [4] revision of the standard requires that the most significant digit of the fraction field be used to determine QNaN (1) or SNaN (0). The remaining bits of the fraction may still be used for implementation-specific data, such as details about the type of the exception that caused the NaN to be generated.

Value (V): No computation is necessary for a NaN value.

Figure 2.5 shows the layout of NaN types.

2.3 The Boost Serialization Library

The Boost C++ Libraries [1] are a collection of portable, open-source, peer-reviewed libraries that extend the features and functionality of the C++ language. Many of the founders of Boost are members of the C++ Standards Committee, and numerous Boost libraries have been incorporated into ISO C++ either as the basis for new language features or as part of the C++ Standard Library. The Boost Libraries are free for both commercial and non-commercial use. The contributors and maintainers of Boost attempt to hold submitted libraries to high standards so that they are suitable for eventual standardization. As of this writing the current stable release of Boost, version 1.55.0, includes more than 100 libraries.

The Boost Serialization Library[1] is designed to allow a developer to easily deconstruct an object into a sequence of bytes, to store those bytes in some manner, and to later reconstruct an equivalent object using the stored bytes. The library intends to maintain portability while depending only on ISO C++ and while taking advantage of modern C++ features to improve simplicity and easy of use. Included in the library is support for the serialization of built-in C++ primitive types as well as data structures from the C++ Standard Library.

Boost.Serialization uses an `Archive` concept to determine how an object is rendered as a byte sequence. `Archive` classes must implement a standardized interface, allowing a class to have a single serialization method that is compatible with any compliant `Archive`. For example, a developer may wish to use a text-based `Archive` to produce a human readable output file that allows for easier testing and debugging, while changing to a binary output format when better performance is needed. Such a change only requires the developer to modify the instantiation of the output `Archive`; the actual serialization code within their classes need not be changed.

Listing 2.1 provides a simple demonstration of the library's usage. Two classes implement serialization, `Position` and `Person`. Both classes use the simplest option provided by the library, with a single template `serialize()` method to perform both saving and loading of objects. The `Archive` template parameter allows the method to work with any compatible `Archive` type. `Archive` classes overload the bitwise-and operator (`&`) to save objects to the

[1] `http://www.boost.org/doc/libs/1_55_0/libs/serialization/doc/index.html`

`Archive`, or extract them from an `Archive`. Any member objects that support serialization may seamlessly use the same operator, as shown when the `Person` class serializes its `Position` member.

By default the library includes `Archive` classes that allow for serialized data to be stored in files as plain text, XML, or binary data. Users may add their own `Archive` implementations if desired. Although the binary `Archive` that is included with Boost.Serialization does not produce portable files, the library does provide an example `Archive` for producing portable binary files; however, this `Archive` is considered incomplete. While it does handle some portability issues such as endianness, the `Archive` does not reliably serialize floating-point types in a portable manner, which is the most significant obstacle to completion of a truly portable serialization system[2].

[2] `http://www.boost.org/doc/libs/1_55_0/libs/serialization/doc/todo.html#portablebinaryarchives`

Listing 2.1: An example usage of Boost.Serialization.

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <fstream>
#include <string>

class Position {
public:
    int x, y;
    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        // primitive types are handled automatically
        ar & x;
        ar & y;
    }
};

class Person {
public:
    std::string name;
    Position pos;
    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        // std::string is handled automatically
        ar & name;
        // Position::serialize() will be called by the archive
        ar & pos;
    }
};

int main() {
    {
        // create the person
        Person op = { "John Doe", { 0, 1 } };
        // output file
        std::ofstream ofs("person.dat");
        // text archive wraps the output file
        boost::archive::text_oarchive oar(ofs);
        // write the object to the file
        oar << op;
    }
    {
        // empty person that will be loaded
        Person ip;
        // input file
        std::ifstream ifs("person.dat");
        // text archive wraps the input file
        boost::archive::text_iarchive iar(ifs);
        // read the object from the file
        iar >> ip;
    }
}
```

Chapter 3

Challenges

In the course of developing this serialization framework, numerous challenges were discovered during the background research and testing process. This chapter discusses those challenges. Chapter 4 discusses the approach that was taken in the design of the serialization framework and the justification for that approach.

3.1 Supporting Non IEEE-754 Platforms

The original goal of this project was to develop a system that would have the ability to completely handle the serialization of floating-point numbers. When developing a portable framework, it is desirable that the end user be able to simply use the framework while minimizing the additional setup required as well as the necessary knowledge regarding the details of their platform. Unfortunately, it quickly became apparent that such a system was implausible.

C++ does not define any requirements for how floating-point numbers are implemented (see Section 2.1), meaning that it is impossible to predict what floating-point format will be used by a given platform. Most applications developed on desktop/server platforms using x86 or x86-64 processors may be reasonably expected to use IEEE-754. However, C++ compilers are available on a wide variety of system architectures, making it impossible to know in advance all of the different floating-point formats that may be encountered, whether they are implemented with hardware support or via software libraries.

In order to develop a portable framework, the developer must be required to know some information about the system configurations that their software will be used on, so that they may correctly interface with the framework. Serializing floating-point data in a standardized format (see Section 4.1) requires the end user to provide the necessary code to convert the standardized floating-point data into a format compatible with the floating-point types used by their platform. It is also expected that some systems may lack the ability to represent IEEE-754 special values. Infinity and NaN are, after all, a result of the IEEE-754 requirement that computations must always continue, and floating-point formats without that requirement may not have a comparable construct. The framework must therefore have a mechanism to signal when special values are loaded, so that the developer may translate them to an equivalent value or, if unsupported, discard them entirely.

3.2 Identifying IEEE-754 Types

It is often necessary to identify the specific type of a floating-point value. ISO C++11 requires the `cmath` header of the Standard Library to implement several functions related to floating-point classification [6]. `isnormal()`, `isnan()`, and `isinf()` may be used to determine if their respective parameters are normal numbers, NaN values, or infinity. The function `fpclassify()` may also be used; it returns an enumerated value indicating if the parameter is normal, subnormal (denormal), infinity, NaN, or an implementation-defined type that does not match one the IEEE-754 categories. However, it should be noted that the enumerated values returned by `fpclassify()` are implementation-defined, meaning that they require additional manipulation before they may be used in a portable manner.

The above functionality is not available when using compilers that do not support the C++11 standard. Prior to C++11, many compilers implemented extensions to their libraries which added functions such as `isnan()`, `isinf()`, etc. There is no standardized way to detect the presence of these extensions short of manually implementing checks for specific compiler versions. Section 4.2 describes a set of utility functions that are provided by the framework to assist in the identification of floating-point types.

3.3 Not a Number

The bit pattern that IEEE-754 defines to identify NaN values is not fixed (see Section 2.2). That is, any floating-point value that contains a biased exponent with all bits set and a non-zero fraction field will be recognized as NaN; the necessary bit pattern of the fraction field is not defined by the standard. Additionally, the fraction field of a NaN value may be used to encode implementation-specific data which may be interpreted differently on various platforms. For example, on one platform a specific bit pattern may represent "NaN caused by divide by zero." If this bit pattern is moved to another platform it will still be interpreted as NaN, however, it may take on a different meaning such as "NaN caused by $\infty - \infty$."

Also complicating this problem is the distinction between Quiet and Signaling NaN values. The 2008 update to the IEEE-754 standard addresses this deficiency by requiring that the most significant digit of the fraction string be used to indicate the type of NaN value [4]. Some platforms, however, have not yet adopted the updated standard. For example, Microsoft Visual C++ version 12 (Visual Studio 2013) operating on both Windows 7 and Windows 8 generates NaN values that do not conform to the IEEE-754-2008 standard (see Appendix A). Since conformance to the 2008 revision of the standard can not be assumed, it is therefore impossible to distinguish between Quiet and Signaling NaN values in a portable manner. As detailed in Section 4.1, the framework will consider all NaN values to be QNaN so that this issue may be avoided.

3.4 Size of `long double` Values

As detailed in Section 2.2, the Extended Precision type used to implement `long double` is not defined to be a fixed size. There are in fact three common sizes that may be used for `long double`: 8, 10, or 16 bytes. The width of the `long double` type may vary depending on the available hardware floating-point unit, the operating system, or even on compiler settings.

Conversions between the 10 and 16 byte formats are relatively easy. The two formats are identical except for the width of the fraction field (see Table 2.1), therefore a conversion from 10 to 16 bytes may be done by right-padding the fraction field with zeros to the appropriate width, while converting from 16 to 10 bytes requires dropping some bits from the fraction string.

Conversion from the 8 byte format to either 10 or 16 bytes requires modification of the exponent field, as the formats have exponents of differing widths. If the value is normal, the biased exponent must be modified using the formula $E_b = E_b \pm (b_{\texttt{long double}} - b_{\texttt{double}})$, performing addition when the conversion is from 8 bytes to 10/16 bytes, or subtraction when converting from 10/16 to 8 bytes. If the value is denormal, infinity, or NaN, 8 byte to 10/16 byte conversions require the exponent field must be right padded to the correct width by duplicating the least significant digit of the exponent field; conversion from 10/16 byte to 8 byte formats require the exponent field to be trimmed to the correct width. In conversions of all value types, the fraction field must also be modified to the correct width by right-padding or removing extra digits using a round to nearest mode.

An additional problem occurs in some implementations of the 10 byte type. Testing revealed that some implementations (see Appendix A) use a 10 byte `long double` type that is actually stored in 16 bytes of memory. In these situations `sizeof(long double)` will return 16, meaning that `sizeof(long double)` alone is insufficient to distinguish between the 10 and 16 byte types. As a solution to this issue, the algorithm shown in Listing 3.1 was used. This algorithm sets a negative `long double` value and detects if the most significant bit is set, as would be expected for a true 16 byte value. Unfortunately testing also showed that some implementations of the 10 byte format using 16 bytes of storage make use of the additional bytes to store implementation-specific data whose purpose is unknown. In the systems observed during testing, the most significant bit of the storage was not used for storing additional information. If, however, an implementation did choose to use the most significant bit for extra information, it would invalidate this solution.

Listing 3.1: An algorithm to detect 10 byte values that use 16 bytes of storage.

```
1  function detect_10_byte_format_in_16_byte_storage:
2      if long double size is not 16 bytes then
3          return false
4      end if
5
6      set a long double variable to -1
7      obtain the most significant bit of the variable
8      if the bit is 1 then
9          return false
10     else
11         return true
12     end if
```

Table 3.1: x86 Extended Precision explicit fraction bits.

Value Type	**Fraction msb**
Normal Numbers	1
Denormal Numbers	0
Infinity	1
Not a Number	1

3.5 x86 Extended Precision Format

Most sources indicate that a 10 byte `long double` may be expected to use the standard IEEE-754 format as described in Section 2.2. However, additional research and practical testing determined that many 10 byte types do not strictly conform to the standard. Instead, they use the x86 Extended Precision floating-point format implemented by Intel x87 hardware floating-point units. This format is based on IEEE-754, but differs from the standard by requiring the most significant digit of the fraction to be explicitly encoded in the value as listed in Table 3.1 [7]. The remainder of the fraction field conforms to the IEEE-754 standard.

The explicitly encoded leading fraction digit causes numerous problems. When one treats x86 Extended Precision values as though they are standard IEEE-754 values, normal and denormal values will be calculated incorrectly because the explicit fraction MSB causes the fraction to be effectively divided by two. Infinity values will always be recognized as NaN due to the non-zero fraction field. Finally, NaN values will always be recognized as QNaN.

Solutions to this issue are not trivial. Some possible approaches to handle this issue are addressed in Chapter 6.

Chapter 4

Proposed Framework

This chapter presents a framework to allow for the portable serialization of floating-point numbers that attempts to address the issues presented in Chapter 3. In addition to the core framework, a number of helper utilities are provided, as well as details regarding integration with the Boost Serialization Library, and an example usage of the framework.

4.1 Framework Core

The framework consists of two major components: a set of standards dictating how floating-point numbers will be serialized, and a pair of abstract base classes that provide the interface for serialization and de-serialization. Users of the framework must implement these interfaces so that the serialization implementation saves floating-point values in a manner compliant with the framework requirements, while the de-serialization implementation loads framework compliant values and converts them to a format compatible with the user's platform.

Serialization Standards

When using this framework, users shall adhere to the following standards to ensure that floating-point values may be saved and loaded in a consistent manner:

1. Binary Representation

    (a) Within the framework, floating-point values shall always be treated as binary values.

i. The types `float`, `double`, and `long double` shall only be used to obtain the representation when a value enters the framework before serialization, and to set the final floating-point number as the value leaves the framework following de-serialization. In other words, no floating-point instructions or casts are to be used in the serialization or de-serialization code. This ensures that the underlying bit pattern is only changed when desired by the developer.

ii. The type `uint32_t` shall be used to hold the binary representation of a `float`, and the type `uint64_t` shall be used to hold the binary representation of a `double`.

iii. Since `long double` may vary in size, its binary representation shall be held in 128 bits using a pair of `uint64_t` values. True 128 bit integer types remain uncommon, therefore the choice of `uint64_t[2]` ensures greater portability.

(b) The integer types that hold the binary values shall be serialized using a byte ordering consistent with how the containing serialization library handles integer types. In this thesis, the Boost Serialization Library is used, and integer types are serialized using the built-in functionality of that library.

(c) When a `long double` is serialized, the integer representing the most significant portion of the binary value shall be written first, followed by the integer representing the least significant portion of the binary value.

2. Floating-Point Format

(a) All values shall be serialized using a bit pattern that represents a value conforming to the appropriate IEEE-754 format. `long double` values shall always use the 128 bit Extended Double Precision format.

(b) Whenever possible, values shall be serialized as a normal number.

(c) Any number that falls within the range of IEEE-754 denormal values shall be serialized using a denormal number representation. Zero always uses a denormal representation.

(d) Any number that falls outside the range of IEEE-754 normal values shall be serialized as either $+\infty$ or $-\infty$.

(e) All NaN values shall be considered to be QNaN.

(f) During de-serialization, an implementation must recognize all denormal, infinity, and NaN values, and convert them to an appropriate format as required by the target floating-point format.

i. If a denormal value is detected during de-serialization and the target floating-point format does not support denormal values or is otherwise incapable of representing the value, the value shall be replaced with the target format's representation of zero.

ii. When a NaN value is detected during de-serialization and the target floating-point format is IEEE-754, it is recommended to replace the loaded binary value with the target system's QNaN representation, obtained using the facilities of the C++ Standard Library[1].

iii. If an infinity or NaN value is detected during de-serialization and the target floating-point format is incapable of representing either infinity or NaN, it is recommended to signal an error condition or to raise an exception. If de-serialization is allowed to continue, the value returned in place of infinity or NaN is implementation-defined.

Serialization Interface

The abstract base class (interface) `FP_Serializer`, shown in Listing 4.1, provides the means by which the local floating-point type is converted to a binary value suitable for serialization. A serialization library using this framework shall require the user to provide an implementation of the `FP_Serializer` interface before serialization begins.

When the library encounters a floating-point value, it shall call the corresponding `convert()` method. The parameter `in` shall be the floating-point value to be serialized, passed using the floating-point format of the current platform. The implementation shall set

[1] `std::numeric_limits<T>::quiet_NaN()`; `T` is the floating-point type.

the parameter `out` to be the binary value that will be serialized, conforming to the requirements listed above. The library shall then perform serialization of the integer value containing the binary representation stored in `out`.

Listing 4.1: The `FP_Serializer` interface.

```
1  class FP_Serializer
2  {
3  public:
4      virtual ~FP_Serializer() {}
5
6      virtual void convert(const float in, uint32_t& out) const = 0;
7      virtual void convert(const double in, uint64_t& out) const = 0;
8      virtual void convert(const long double in, LongDouble& out) const = 0;
9  };
```

De-serialization Interface

The counterpart to `FP_Serializer` is the abstract base class (interface) `FP_Deserializer`, shown in Listing 4.2. It is used to convert the binary value loaded during de-serialization into a suitable floating-point value that is compatible with the current platform. A serialization library using this framework shall require the user to provide an implementation of the `FP_Deserializer` interface before de-serialization begins.

When the library receives a request that a floating-point value be loaded, it shall first de-serialize the appropriate integer type, which will contain the binary representation of the floating-point value. This integer shall then be passed to the appropriate `convert()` method as the parameter `in`. The `convert()` method determines what type of floating-point value the `in` parameter represents, and calls the appropriate method to handle the conversion of the loaded type. Four methods are provided for each type: `loadedNorm()`, `loadedDenorm()`, `loadedNan()`, `loadedInf()`. Each method takes a single parameter, which is the binary representation of the loaded value, performs a conversion of the binary representation to the appropriate floating-point type. This floating-point value shall then be assigned to the `out` parameter of the `convert()` method.

Note that it is not strictly necessary to implement all five "loaded" methods for each type. For example, the binary representation of a `float` specified by this framework may be converted directly to the `float` type of a platform that conforms to IEEE-754 (see Section 4.4).

However, even in that case it is still recommended to implement `loadedNan()`, as previously discussed.

Listing 4.2: The `FP_Deserializer` interface.

```
1  class FP_Deserializer
2  {
3  public:
4      virtual ~FP_Deserializer() {}
5
6      virtual void convert(const uint32_t in, float& out) const = 0;
7      virtual float loadedNorm(const uint32_t v) const = 0;
8      virtual float loadedDenorm(const uint32_t v) const = 0;
9      virtual float loadedNan(const uint32_t v) const = 0;
10     virtual float loadedInf(const uint32_t v) const = 0;
11
12     virtual void convert(const uint64_t in, double& out) const = 0;
13     virtual double loadedNorm(const uint64_t v) const = 0;
14     virtual double loadedDenorm(const uint64_t v) const = 0;
15     virtual double loadedNan(const uint64_t v) const = 0;
16     virtual double loadedInf(const uint64_t v) const = 0;
17
18     virtual void convert(const LongDouble in, long double& out) const = 0;
19     virtual long double loadedNorm(const LongDouble v) const = 0;
20     virtual long double loadedDenorm(const LongDouble v) const = 0;
21     virtual long double loadedNan(const LongDouble v) const = 0;
22     virtual long double loadedInf(const LongDouble v) const = 0;
23 };
```

4.2 Provided Utilities

To facilitate the identification and conversion of floating-point types within implementations of `FP_Serializer` and `FP_Deserializer`, a set of utility functions and values are provided as part of the framework.

Floating-Point Field Masks

A number of integer constants are supplied that implementations may use when manipulating floating-point numbers as binary values. These constants are shown in Listing 4.3. Masks are supplied for the 32, 64, and 128 bit types that are used for serialization within the framework. For each type, five constants are provided:

`biasN` : The bias for the type, as listed in Table 2.1.

`signMaskN` : A mask that may be used to obtain the sign bit for the type.

exponentMaskN : A mask that may be used to obtain the exponent field for the type.

fractionMaskN : A mask that may be used to obtain the fraction field for the type.

nanTypeMaskN : A mask that may be used to obtain the QNaN/SNaN flag bit if the type conforms to IEEE-754-2008.

In the name of the constant, `N` indicates the size of the type, in bits. The 128 bit type actually defines a pair of constants for each item listed above, with `hi` or `lo` appended to the name, indicating whether that mask applies to the most significant or least significant portion of the value.

Listing 4.3: Floating-point field masks.

```
1  const uint32_t bias32 = 127;
2  const uint32_t signMask32 = UINT32_C(0x80000000);
3  const uint32_t exponentMask32 = UINT32_C(0x7f800000);
4  const uint32_t fractionMask32 = UINT32_C(0x007fffff);
5  const uint32_t nanTypeMask32 = UINT32_C(0x00400000);
6
7  const uint64_t bias64 = 1023;
8  const uint64_t signMask64 = UINT64_C(0x8000000000000000);
9  const uint64_t exponentMask64 = UINT64_C(0x7ff0000000000000);
10 const uint64_t fractionMask64 = UINT64_C(0x000fffffffffffff);
11 const uint64_t nanTypeMask64 = UINT64_C(0x0008000000000000);
12
13 const uint64_t bias128 = 16383;
14 const uint64_t signMask128hi = UINT64_C(0x8000000000000000);
15 const uint64_t signMask128lo = UINT64_C(0);
16 const uint64_t exponentMask128hi = UINT64_C(0x7fff000000000000);
17 const uint64_t exponentMask128lo = UINT64_C(0);
18 const uint64_t fractionMask128hi = UINT64_C(0x0000ffffffffffff);
19 const uint64_t fractionMask128lo = UINT64_C(0xffffffffffffffff);
20 const uint64_t nanTypeMask128hi = UINT64_C(0x0000800000000000);
21 const uint64_t nanTypeMask128lo = UINT64_C(0);
```

Floating-Point Type Functions

As discussed in Section 3.2, the C++ Standard Library has only recently added functionality to identify the various types of floating-point values. Therefore, the framework includes a selection of utility functions for identifying floating-point values. These functions are intended to be used with binary values within the framework, therefore all of them take the binary representation of the floating-point value as an integer parameter. The following functions are included, with overloads supplied for the each of the 32, 64, and 128 bit floating-point types:

`isNeg()` : Returns true if the value is negative, regardless of its type.

`isDenormal()` : Returns true if the value is a denormal number.

`isNormal()` : Returns true if the value is a normal number.

`isZero()` : Returns true if the value is either $+0$ or -0.

`isPosZero()` : Returns true only if the value is $+0$.

`isNegZero()` : Returns true only if the value is -0.

`isInf()` : Returns true if the value is either $+\infty$ or $-\infty$.

`isPosInf()` : Returns true only if the value is $+\infty$.

`isNegInf()` : Returns true only if the value is $-\infty$.

`isNaN()` : Returns true if the value is Not a Number. Does not distinguish between Quiet or Signaling NaN.

Listing 4.4 contains the prototypes for the included utility functions.

Listing 4.4: Floating-point type utility functions.

```
1  bool isNeg(const uint32_t v);
2  bool isNeg(const uint64_t v);
3  bool isNeg(const LongDouble v);
4
5  bool isDenormal(const uint32_t v);
6  bool isDenormal(const uint64_t v);
7  bool isDenormal(const LongDouble v);
8
9  bool isNormal(const uint32_t v);
10 bool isNormal(const uint64_t v);
11 bool isNormal(const LongDouble v);
12
13 bool isZero(const uint32_t v);
14 bool isZero(const uint64_t v);
15 bool isZero(const LongDouble v);
16
17 bool isPosZero(const uint32_t v);
18 bool isPosZero(const uint64_t v);
19 bool isPosZero(const LongDouble v);
20
21 bool isNegZero(const uint32_t v);
22 bool isNegZero(const uint64_t v);
23 bool isNegZero(const LongDouble v);
24
25 bool isInf(const uint32_t v);
26 bool isInf(const uint64_t v);
27 bool isInf(const LongDouble v);
28
29 bool isPosInf(const uint32_t v);
30 bool isPosInf(const uint64_t v);
31 bool isPosInf(const LongDouble v);
32
33 bool isNegInf(const uint32_t v);
34 bool isNegInf(const uint64_t v);
35 bool isNegInf(const LongDouble v);
36
37 bool isNaN(const uint32_t v);
38 bool isNaN(const uint64_t v);
39 bool isNaN(const LongDouble v);
```

`LongDouble` Type

`long double` may vary in width from 64 to 128 bits, as discussed in Section 3.4. At this time, integers larger than 64 bits are not commonly available. Therefore, the `LongDouble` union type shown in Listing 4.5 is provided for ease of accessing the binary representation of a `long double` number, as well as converting a binary value back to a `long double` type. Within the framework, `LongDouble` is always used to encapsulate the pair of `uint64_t` values that hold the binary representation of a `long double`. The individual bytes of the type are also

accessible, if needed.

Note that on big-endian systems, the integer `LongDouble::u64[0]` will refer to the most significant portion of the `long double` value, while on little-endian systems it will refer to the least significant portion of the value.

Listing 4.5: The `LongDouble` type.

```
1  union LongDouble
2  {
3      long double ld;
4      uint8_t u8[16];
5      uint64_t u64[2];
6  };
```

System Endianness Identification

Ideally all endianness concerns would be delegated to the containing serialization library. However, as indicated in the previous section, when dealing with the binary representation of `long double` values it is necessary for the developer to be aware of the endianness of the current system. The framework provides a utility function, shown in Listing 4.6, to detect the endianness of the platform.

Listing 4.6: The endianness identifier utility function.

```
1  bool isLittleEndian()
2  {
3      union
4      {
5          uint16_t u16;
6          uint8_t u8[2];
7      } endian;
8
9      endian.u16 = 0x1234;
10     return endian.u8[0] == 0x34;
11 }
```

`long double` Type Identifier.

As discussed in Section 3.4, 10 byte `long double` values may be stored using 16 bytes of memory. The framework includes a utility function to address this issue, shown in Listing 4.7. This is an implementation of the algorithm from Listing 3.1.

Listing 4.7: The `long double` type identifier utility function.

```
1  bool LDis10in16()
2  {
3      if (sizeof(long double) != 16)
4          return false;
5
6      LongDouble ld = { -1.0L };
7      if (isLittleEndian())
8          return !(ld.u64[1] & signMask128hi);
9      else
10         return !(ld.u64[0] & signMask128hi);
11 }
```

4.3 Integration With Boost the Serialization Library

The Boost Serialization Library, discussed in Section 2.3, is the serialization library used for testing the framework in this thesis. The example `Archive` classes provided with the library are `portable_binary_oarchive`, used for serialization, and `portable_binary_iarchive`, used for de-serialization. In order to accommodate this floating-point framework, these classes were modified as follows:

1. Addition of the floating-point interfaces.

   (a) A protected member was added to `portable_binary_oarchive`, which is used to hold an implementation of the `FP_Serializer` interface.

   (b) A protected member was added to `portable_binary_iarchive`, which is used to hold an implementation of the `FP_Deserializer` interface.

2. Each class was modified so that the methods that perform floating-point serialization function as described in Section 4.1.

   Listing 4.8 and Listing 4.9 show the changes made to these classes.

Listing 4.8: Modifications to the `portable_binary_oarchive` class.

```
class portable_binary_oarchive
{
protected:
    FP_Serializer& m_fp_serializer;
public:
    void save(const float& t)
    {
        uint32_t v;
        this->m_fp_serializer.convert(t, v);
        this->primitive_base_t::save(v);
    }
    void save(const double& t)
    {
        uint64_t v;
        this->m_fp_serializer.convert(t, v);
        this->primitive_base_t::save(v);
    }
    void save(const long double& t)
    {
        LongDouble out;
        this->m_fp_serializer.convert(t, out);
        this->primitive_base_t::save(out.u64[0]);
        this->primitive_base_t::save(out.u64[1]);
    }
};
```

Listing 4.9: Modifications to the `portable_binary_iarchive` class.

```
class portable_binary_iarchive
{
protected:
    FP_Deserializer& m_fp_deserializer;
public:
    void load(float& t){
        uint32_t v;
        this->primitive_base_t::load(v);
        this->m_fp_deserializer.convert(v, t);
    }
    void load(double& t){
        uint64_t v;
        this->primitive_base_t::load(v);
        this->m_fp_deserializer.convert(v, t);
    }
    void load(long double& t)
    {
        LongDouble in;
        this->primitive_base_t::load(in.u64[0]);
        this->primitive_base_t::load(in.u64[1]);
        this->m_fp_deserializer.convert(in, t);
    }
};
```

4.4 Example Usage on IEEE-754 platforms

To demonstrate the usage of this floating-point framework, an example implementation has been provided that operates on systems supporting IEEE-754. This implementation supports only true IEEE-754 platforms; specifically, platforms that use the x86 Extended Precision type (see Section 3.5) are not supported.

Listing 4.10 shows the implementation of the `FP_Serializer` interface. In this implementation, `float` and `double` types are both serialized directly, as their formats are already compliant with the requirements listed in Section 4.1. `long double` types are modified to conform to the framework specifications by modifying the binary representation as described in Section 3.4.

Listing 4.10: The `IEEE754Serializer` class.

```
1  class IEEE754Serializer : public FP_Serializer
2  {
3  public:
4      virtual void convert(const float in, uint32_t& out) const
5      {
6          out = *reinterpret_cast<const uint32_t*>(&in);
7      }
8
9      virtual void convert(const double in, uint64_t& out) const
10     {
11         out = *reinterpret_cast<const uint64_t*>(&in);
12     }
13
14     virtual void convert(const long double in, LongDouble& out) const
15     {
16         const int hiIdx = isLittleEndian() ? 1 : 0;
17         const int loIdx = !hiIdx;
18
19         uint64_t& outHi = out.u64[hiIdx];
20         outHi = in.u64[hiIdx];
21         uint64_t& outLo = out.u64[loIdx];
22         outLo = in.u64[loIdx];
23
24         if (sizeof(long double) == 8)
25         {
26             // have to tweak the exponent of normal values
27             if (isNormal(outLo))
28             {
29                 // fix the exponent
30                 uint64_t exponent = (outLo & exponentMask64) >> 52;
31                 exponent += (bias128 - bias64);
32
33                 outHi = 0;
34                 // grab the sign
35                 outHi = (outLo & signMask64);
36                 // set the new exponent
```

```
37                  outHi |= (exponent << 48);
38                  // the most significant 48 bits of the fraction are
39                  // moved to the least significant part of outHi
40                  outHi |= ((outLo & fractionMask64) >> 4);
41                  // pad 60 bits into the fraction
42                  outLo <<= 60;
43              }
44              // denormal are fine just padding the fields into place
45              else if (isDenormal(outLo))
46              {
47                  // long double is an alias for double.  outLo contains
48                  // the entire double
49                  // set the sign bit
50                  outHi = outLo & signMask64;
51                  // get the exponent and left pad by 4 bits
52                  outHi |= ((outLo & exponentMask64) >> 4);
53                  // the most significant 48 bits of the fraction are moved
54                  // to the least significant part of outHi
55                  outHi |= ((outLo & fractionMask64) >> 4);
56                  // pad 60 bits into the fraction
57                  outLo <<= 60;
58              }
59              else if (isNaN(outLo) || isInf(outLo))
60              {
61                  outHi = 0;
62                  // grab the sign bit
63                  outHi |= (outLo & signMask64);
64                  // set all the exponent bits
65                  outHi |= exponentMask128hi;
66                  // the most significant 48 bits of the fraction are
67                  // moved to the least significant part of outHi
68                  outHi |= ((outLo & fractionMask64) >> 4);
69                  // pad 60 bits into the fraction
70                  outLo <<= 60;
71              }
72              else
73                  throw FP_UnknownType();
74          }
75          else if (sizeof(long double) == 10 || LDis10in16())
76          {
77              // shift the sign and exponent into place
78              outHi <<= 48;
79              // move the most significant 48 bits of lo to the
80              // least significant part of hi
81              outHi |= (outLo >> 16);
82              // pad in 48 bits to the least significant part of lo
83              outLo <<= 48;
84          }
85          else if (sizeof(long double) != 16)
86              throw FP_LongDoubleSizeException();
87          // else 16 byte type, already formatted correctly
88
89          // swap to big endian if needed
90          if (isLittleEndian())
91              std::swap(outHi, outLo);
92      }
93  };
```

Listing 4.11 shows the implementation of the `FP_Deserializer` interface. In this implementation, the binary values loaded for `float` and `double` types are converted directly to their respective floating-point types. `long double` values are modified as described in Section 3.4 to match the size of that type on the current platform.

Listing 4.11: The `IEEE754Deserializer` class.

```
1  class IEEE754Deserializer : public FP_Deserializer
2  {
3  public:
4      virtual void convert(const uint32_t in, float& out) const
5      {
6          if (isNormal(in) || isDenormal(in) || isInf(in))
7              out = *reinterpret_cast<const float*>(&in);
8          else if (isNaN(in))
9              out = loadedNan(in);
10         else
11             throw FP_UnknownType();
12     }
13
14     virtual float loadedNorm(const uint32_t v) const {}
15     virtual float loadedDenorm(const uint32_t v) const {}
16
17     virtual float loadedNan(const uint32_t v) const
18     {
19         return std::numeric_limits<float>::quiet_NaN();
20     }
21
22     virtual float loadedInf(const uint32_t v) const {}
23
24     virtual void convert(const uint64_t in, double& out) const
25     {
26         if (isNormal(in) || isDenormal(in) || isInf(in))
27             out = *reinterpret_cast<const double*>(&in);
28         else if (isNaN(in))
29             out = loadedNan(in);
30         else
31             throw FP_UnknownType();
32     }
33     virtual double loadedNorm(const uint64_t v) const {}
34     virtual double loadedDenorm(const uint64_t v) const {}
35
36     virtual double loadedNan(const uint64_t v) const
37     {
38         return std::numeric_limits<double>::quiet_NaN();
39     }
40
41     virtual double loadedInf(const uint64_t v) const {}
42
43     virtual void convert(const LongDouble in, long double& out) const
44     {
45         if (isNormal(in))
46             out = loadedNorm(in);
47         else if (isDenormal(in))
48             out = loadedDenorm(in);
49         else if (isNaN(in))
```

```
50              out = loadedNaN(in);
51          else if (isInf(in))
52              out = loadedInf(in);
53          else
54              throw FP_UnknownType();
55      }
56
57      virtual long double loadedNorm(const LongDouble v) const
58      {
59          uint64_t& hi = v.u64[0];
60          uint64_t& lo = out.u64[1];
61
62          if (sizeof(long double) == 8)
63          {
64              // drop 60 bits from the fraction
65              lo >>= 60;
66              // move 48 bits from the hi to the low portion
67              lo |= (hi & fractionMask128hi) << 4;
68              // fix the exponent
69              uint64_t exponent = (hi & exponentMask128hi) >> 48;
70              exponent -= (bias128 - bias64);
71              // set the exponent in place
72              lo |= (exponent << 52);
73              // set the sign
74              lo |= (hi & signMask128hi);
75          }
76          else if (sizeof(long double) == 10 || LDis10in16())
77          {
78              // drop the least significant 48 bits of lo
79              lo >>= 48;
80              // the least significant 48 bits of hi become the most significant 48
81              // bits of lo
82              lo |= (hi << 16);
83              // shift the sign and exponent into position
84              hi >>= 48;
85          }
86          else if (sizeof(long double) != 16)
87              throw FP_LongDoubleSizeException();
88          // else no modification necessary
89
90          // swap to little endian, if needed
91          if (isLittleEndian())
92              std::swap(v.u64[0], v.u64[1]);
93          return v.ld;
94      }
95
96      virtual long double loadedDenorm(const LongDouble v) const
97      {
98          uint64_t& hi = v.u64[0];
99          uint64_t& lo = out.u64[1];
100
101         if (sizeof(long double) == 8)
102         {
103             // drop 60 bits from the fraction
104             lo >>= 60;
105             // move the rest of the fraction into place
106             lo |= (hi & fractionMask128hi) << 4;
107             // trim the exponent and move it into place
108             lo |= (hi & exponentMask128hi) << 4;
```

```
109             // set the sign bit
110             lo |= (hi & signMask128hi);
111         }
112         else if (sizeof(long double) == 10 || LDis10in16())
113         {
114             // drop the least significant 48 bits of lo
115             lo >>= 48;
116             // the least significant 48 bits of hi become the most significant 48
117             // bits of lo
118             lo |= (hi << 16);
119             // shift the sign and exponent into position
120             hi >>= 48;
121         }
122         else if (sizeof(long double) != 16)
123             throw FP_LongDoubleSizeException();
124         // else no modification necessary
125 
126         // swap to little endian, if needed
127         if (isLittleEndian())
128             std::swap(v.u64[0], v.u64[1]);
129         return v.ld;
130     }
131 
132     virtual long double loadedNan(const LongDouble v) const
133     {
134         return std::numeric_limits<long double>::quiet_NaN();
135     }
136 
137     virtual long double loadedInf(const LongDouble v) const
138     {
139         // works the same as denormal numbers
140         return loadedDenorm(v);
141     }
142 };
```

Chapter 5

Testing

In order to show the viability of the framework presented in this thesis, a number of tests were performed on both the framework utilities and the framework itself. This section details the methodology used in those tests and their results.

5.1 Utilities

The utility functions detailed in Section 4.2 were tested using several different methods.

The functions `isLittleEndian()` and `LDis10in16()` were not considered viable for automated testing due to the nature of the information they retrieve. `isLittleEndian()` was used during the generation of the floating-point feature reports listed in Appendix A, and the output was compared against the known endianness of the processors used in those systems. `LDis10in16()` was run on the test systems and the results were manually compared to the `long double` values that were also generated as part of the feature reports.

Unit tests were created for the floating-point type identification functions using the Boost Test Library[1]. In each test case, at least one valid value was tested along with invalid values from all of the applicable floating-point types. Since all of the testing platforms used IEEE-754 as their native floating-point format, test values were cast directly to their binary representation from the `float` and `double` types. None of the available test platforms used

[1] `http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html/index.html`

the 128 bit `long double` format, so those version of the utility functions were not included in this test.

Listing 5.1 demonstrates the test cases for the 32 and 64 bit versions of the `isNeg()` utility function. Each function is first tested with a valid negative value. It is then tested with a series of invalid values: a positive number, zero, infinity, and NaN.

Listing 5.1: Testing the `isNeg()` function.

```
1  #define BINARY_F(val) \
2      do { f = val; fb = *reinterpret_cast<uint32_t*>(&f); } while (0)
3
4  #define BINARY_D(val) \
5      do { d = val; db = *reinterpret_cast<uint64_t*>(&d); } while (0)
6
7  BOOST_AUTO_TEST_CASE(test_isNeg32)
8  {
9      float f;
10     uint32_t fb;
11
12     // valid values
13     BINARY_F(-1.0f);
14     BOOST_CHECK(isNeg(fb));
15
16     // invalid values
17     BINARY_F(1.0f);
18     BOOST_CHECK(!isNeg(fb));
19     BINARY_F(0.0f);
20     BOOST_CHECK(!isNeg(fb));
21     BINARY_F(numeric_limits<float>::infinity());
22     BOOST_CHECK(!isNeg(fb));
23     BINARY_F(numeric_limits<float>::quiet_NaN());
24     BOOST_CHECK(!isNeg(fb));
25 }
26
27 BOOST_AUTO_TEST_CASE(test_isNeg64)
28 {
29     double d;
30     uint64_t db;
31
32     // valid values
33     BINARY_D(-1.0);
34     BOOST_CHECK(isNeg(db));
35
36     // invalid values
37     BINARY_D(1.0);
38     BOOST_CHECK(!isNeg(db));
39     BINARY_D(0.0);
40     BOOST_CHECK(!isNeg(db));
41     BINARY_D(numeric_limits<double>::infinity());
42     BOOST_CHECK(!isNeg(db));
43     BINARY_D(numeric_limits<double>::quiet_NaN());
44     BOOST_CHECK(!isNeg(db));
45 }
```

5.2 Framework Implementation

The IEEE-754 implementation of the framework, detailed in Section 4.4, was tested on the systems listed in Appendix A. Listing 5.2 shows a sample test program, in which a simple `struct` containing several floating-point numbers may be serialized or de-serialized. The serialization output was created using Windows, Linux, and Mac OS. Each file was then loaded on the other two platforms and the loaded values were compared to the original values, using both a debugger and standard output. Test programs on all platforms were compiled using version 1.55.0 of the Boost Libraries.

Listing 5.2: Testing the framework.

```
1  struct TestStruct
2  {
3      float f;
4      float fnan;
5      float finf;
6      double d;
7      double dnan;
8      double dinf;
9      long double ld;
10     long double ldnan;
11     long double ldinf;
12
13     template<typename Archive>
14     void serialize(Archive& ar, const unsigned int version)
15     {
16         ar & f;
17         ar & fnan;
18         ar & finf;
19         ar & d;
20         ar & dnan;
21         ar & dinf;
22         ar & ld;
23         ar & ldnan;
24         ar & ldinf;
25     }
26 };
27
28 ostream& operator<<(ostream& os, const TestStruct& t)
29 {
30     os << t.f << endl
31        << t.fnan << endl
32        << t.finf << endl
33        << t.d << endl
34        << t.dnan << endl
35        << t.dinf << endl
36        << t.ld << endl
37        << t.ldnan << endl
38        << t.ldinf << endl;
39     return os;
40 }
```

```
41
42 void load()
43 {
44     string filename;
45     cout << "Filename: ";
46     cin >> filename;
47
48     TestStruct t2;
49     ifstream inFile(filename.c_str());
50     IEEE754Deserializer ieee;
51     portable_binary_iarchive inArchive(inFile, ieee);
52     inArchive >> t2;
53     cout << t2 << endl;
54 }
55
56 void save()
57 {
58     string filename;
59     cout << "Filename: ";
60     cin >> filename;
61
62     TestStruct t1 = {
63         123.456F,
64         numeric_limits<float>::quiet_NaN(),
65         numeric_limits<float>::infinity(),
66         5678.01234,
67         numeric_limits<double>::quiet_NaN(),
68         numeric_limits<double>::infinity(),
69         12345678.901234567L,
70         numeric_limits<long double>::quiet_NaN(),
71         numeric_limits<long double>::infinity()
72     };
73     ofstream outFile(filename.c_str());
74     IEEE754Serializer ieee;
75     portable_binary_oarchive outArchive(outFile, ieee);
76     outArchive << t1;
77     cout << t1 << endl;
78 }
79
80 int main()
81 {
82     char action;
83     cout << "Load or save? [l/s] ";
84     cin >> action;
85
86     if (action == 'l')
87         load();
88     else
89         save();
90
91   return 0;
92 }
```

Boost Serialization Issues

Testing revealed several undocumented issues with the portability of files created using the example `portable_binary_archive` classes supplied with the Boost Serialization Library. These issues appear to be related to the internal versioning information used by the archive classes; there was no indication that they are directly related to the floating-point framework presented in this thesis. However, the issues are documented because they required manual modification of the serialization output so that testing could be completed.

Initial testing found that files created on Linux platforms would load on Windows platforms, however, when attempting to open a file created on Windows using a Linux platform, the Boost Serialization Library would fail to open the file, throwing an exception indicating that the file version was unsupported. Examination of the two files using a hex editor revealed that the first 0x18 bytes of the files were identical. The first difference in the file occurred at offset 0x19; in the file created using Windows the byte at this offset contained the value 0x0B, while the file created on Linux had the value 0x0A. Editing the Windows file so that offset 0x19 had the value 0x0A allowed the file to be loaded on both Windows and Linux. This problem was consistent regardless of the actual data being serialized, therefore, all files created using Windows were manually edited to change the byte at offset 0x19 to the value 0x0A.

A similar problem was discovered when attempting to load files created on Windows using Mac OS; loading of test files failed with an exception reporting an input stream error. The Windows file was compared to a file generated using Mac OS using a hex editor; however, a consistent fix for this problem was not found because the cause of the error could not be determined conclusively.

Testing Results

`float` and `double` values were found to de-serialize accurately on all tested platforms. This is expected, as all of the available test systems use IEEE-754 conforming `float` and `double` representations that did not require modification during serialization.

`long double` values presented mixed results. `long double` values were successfully de-serialized on similar platforms. Test files created on Windows would de-serialize accurately

on Windows, but would de-serialize inaccurately on Linux. Meanwhile, test files created using Linux or Mac OS would de-serialize accurately on those platforms, but would be inaccurate following de-serialization on Windows. It was at this time that additional research was conducted and the x86 Extended Precision format was discovered. It was determined that the explicit leading fraction bit used by that format was the cause of the inaccurate values, as described in Section 3.5. Additional attempts were made to modify the implementation to support this format seamlessly alongside the standard IEEE-754 format; however, this was unsuccessful. A separate implementation will be required for systems that use the x86 Extended Precision format.

Ultimately, this testing demonstrates that the framework is conceptually sound. The successful serialization and de-serialization of these values, requires the conversion of the local floating-point format to the standardized format. De-serialization requires that the process be reversed. While this process could not be shown while moving between platforms due to the limitations described above, even serialization and de-serialization on the same platform involved a conversion that is fundamentally the same as the conversion used when de-serializing on another platform.

Chapter 6

Conclusion

6.1 Future Work

At this time, the IEEE-754 implementation is essentially complete for `float` and `double` values. Developers wishing to support cross-platform serialization of only those types may do so with minimal additional work. Support for `long double` values, however, is more complex.

Two primary issues affect the serialization of `long double` values. First, the detection of 10 byte `long double` values that are stored in 16 bytes of memory is somewhat frail. The algorithm proposed in Section 3.4 and implemented in Section 4.2 relies on the assumption that these implementations to not use the most significant bit of the storage for implementation-specific information. All of the systems tested as part of this thesis did not use that bit; however, implementations are free to use this space for whatever purpose it wishes. By definition, the extra six bytes of the memory location are not part of the IEEE-754 standard. An alternative mechanism of detecting this format would make the framework more robust.

The second issue is support for systems using the x86 Extended Precision format for `long double` values. Ideally a system should be implemented so that the usage of this type may be detected dynamically at runtime. The usage of this type is tied to the Intel x87 Hardware Floating-Point Unit. The x86 processor architecture provides the CPUID instruction, which may be used to query for the presence of an x87 FPU [5]. However, the physical presence of a FPU does not mean that the operating system and/or compiler has chosen to implement the x86 Extended Precision format. For example, a Lenovo X220 laptop was used to provide

two of the testing platforms in this thesis. This laptop uses an Intel Core i7 Mobile processor, which includes an x87 FPU[1]. When running the Windows operating system, `long double` is aliased to `double`; when using Linux, the x86 Extended Precision format is used. It seems unlikely that the CPUID instruction alone would be sufficient to detect this format, but perhaps that runtime information could be combined with compile time detection of common operating systems and compilers.

Additional work also remains to be done by implementing the framework for platforms that do not use some form of the IEEE-754 standard.

6.2 Summary

The framework described in this thesis lays the groundwork that developers may use to allow for the serialization of floating-point numbers in a portable manner.

Floating-point numbers that are serialized using this framework are saved to a standard format based on the IEEE-754 floating-point standard. This allows these values to be more easily converted into the format used by a specific target platform. Presented along with the framework is an implementation that supports serialization and de-serialization of floating-point numbers on platforms that support the IEEE-754 standard. In addition, numerous utility functions are provided to assist developers in creating additional implementations.

The variations in floating-point format mean that compromises are inherit in any serialization system. This framework minimizes those compromises by avoiding loss of precision, unless it is required by the target system, as well as maintaining IEEE-754 special values throughout the serialization process.

[1] `http://www.intel.com/content/www/us/en/processors/core/CoreTechnicalResources.html`

Bibliography

[1] Boost C++ Libraries. `http://www.boost.org/`.

[2] IEEE Standard for Binary Floating-Point Arithmetic. Technical Report ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA, 1985.

[3] 80C187 80-BIT MATH COPROCESSOR. Technical Report 80C187, Intel Corporation, 1992.

[4] IEEE Standard for Binary Floating-Point Arithmetic. Technical Report ANSI/IEEE Std 754-2008, The Institute of Electrical and Electronics Engineers, Inc, 3 Park Avenue, New York, NY 10016-5997, USA, 2008.

[5] CPUID Specification. Technical Report 25481, Advanced Micro Devices, 2010.

[6] Information technology — Programming languages — C++. Technical Report ISO/IEC 14882:2011(E), International Organization for Standardization, Geneva, Switzerland, 2011.

[7] Intel 64 and IA-32 Architectures Software Developers Manual. Technical Report 253665-050US, Intel Corporation, 2014.

[8] ACCU Conference. *Cross-Platform Issues With Floating-Point Arithmetics in C++*, 2006.

[9] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1), March 1991.

Appendices

Appendix A

Floating-Point Feature Tests

During the research for this thesis a number of computers and operating systems were tested to determine the floating-point features of each platform. System information was queried in the following manner:

Windows: Operating system and version as well as CPU model number was taken from the "System Information" application.

Mac OS X: Operating system and version as well as CPU model number was taken from the "About This Mac" dialog.

Linux: Operating system and version was determined by running the terminal command `cat /etc/*-release`. CPU model number was taken from the file `/proc/cpuinfo`.

All other information regarding floating-point features was determined by querying the `std::numeric_limits` class that is included in the C++ Standard Library.

Listing A.1: Floating-point features of Windows 7.

```
Lenovo X220 laptop
Windows 7 x64
Intel Core i7-2640M

Endianness: little

Feature report for float:
Size (bits): 32
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x40490fdb
Epsilon: 1.19209e-007
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7f800000
Has QNaN: true
QNaN: 0x7fc00000
Has SNaN: true
SNaN: 0x7fc00001

Feature report for double:
Size (bits): 64
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x400921fb54442d18
Epsilon: 2.22045e-016
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7ff0000000000000
Has QNaN: true
QNaN: 0x7ff8000000000000
Has SNaN: true
SNaN: 0x7ff8000000000001

Feature report for long double:
Size (bits): 64
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x400921fb54442d18
Epsilon: 2.22045e-016
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7ff0000000000000
Has QNaN: true
QNaN: 0x7ff8000000000000
Has SNaN: true
SNaN: 0x7ff8000000000001
```

Listing A.2: Floating-point features of Windows 8.

```
TSS Lenovo ThinkCentre
Windows 8 x64
Intel Core2 Quad Q9400

Endianness: little

Feature report for float:
Size (bits): 32
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x40490fdb
Epsilon: 1.19209e-007
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7f800000
Has QNaN: true
QNaN: 0x7fc00000
Has SNaN: true
SNaN: 0x7f800001

Feature report for double:
Size (bits): 64
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x400921fb54442d18
Epsilon: 2.22045e-016
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7ff0000000000000
Has QNaN: true
QNaN: 0x7ff8000000000000
Has SNaN: true
SNaN: 0x7ff0000000000001

Feature report for long double:
Size (bits): 64
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x400921fb54442d18
Epsilon: 2.22045e-016
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7ff0000000000000
Has QNaN: true
QNaN: 0x7ff8000000000000
Has SNaN: true
SNaN: 0x7ff0000000000001
```

Listing A.3: Floating-point features of Mac OS X.

```
TSS iMac, late 2011 27"
OS X 10.9.2
Intel Core i7

Endianness: little

Feature report for f:
Size (bits): 32
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x40490fdb
Epsilon: 1.19209e-07
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7f800000
Has QNaN: true
QNaN: 0x7fc00000
Has SNaN: true
SNaN: 0x7fa00000

Feature report for d:
Size (bits): 64
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x400921fb54442d18
Epsilon: 2.22045e-16
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7ff0000000000000
Has QNaN: true
QNaN: 0x7ff8000000000000
Has SNaN: true
SNaN: 0x7ff4000000000000

Feature report for e:
Size (bits): 128
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x00000000000004000c90fdaa22168c235
Epsilon: 1.0842e-19
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x0000000000007fff8000000000000000
Has QNaN: true
QNaN: 0x00007fff750d7fffc000000000000000
Has SNaN: true
SNaN: 0x0000000000007fffa000000000000000
```

Listing A.4: Floating-point features of Manjaro Linux on a virtual machine.

```
Lenovo X220 laptop
Manjaro Linux 0.8.9
Intel Core i7-2640M

Endianness: little

Feature report for f:
Size (bits): 32
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x40490fdb
Epsilon: 1.19209e-07
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7f800000
Has QNaN: true
QNaN: 0x7fc00000
Has SNaN: true
SNaN: 0x7fa00000

Feature report for d:
Size (bits): 64
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x400921fb54442d18
Epsilon: 2.22045e-16
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7ff0000000000000
Has QNaN: true
QNaN: 0x7ff8000000000000
Has SNaN: true
SNaN: 0x7ff4000000000000

Feature report for e:
Size (bits): 128
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x00000000000004000c90fdaa22168c235
Epsilon: 1.0842e-19
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x00007fac1e217fff8000000000000000
Has QNaN: true
QNaN: 0x00007fff1a587fffc000000000000000
Has SNaN: true
SNaN: 0x00007fac1e217fffa000000000000000
```

Listing A.5: Floating-point features of CentOS Linux.

```
TSS pub2 server
CentOS 6.5
Intel Xeon E5645

Endianness: little

Feature report for f:
Size (bits): 32
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x40490fdb
Epsilon: 1.19209e-07
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7f800000
Has QNaN: true
QNaN: 0x7fc00000
Has SNaN: true
SNaN: 0x7fa00000

Feature report for d:
Size (bits): 64
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x400921fb54442d18
Epsilon: 2.22045e-16
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7ff0000000000000
Has QNaN: true
QNaN: 0x7ff8000000000000
Has SNaN: true
SNaN: 0x7ff4000000000000

Feature report for e:
Size (bits): 128
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x00000000000004000c90fdaa22168c235
Epsilon: 1.0842e-19
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x00007f3d6be37fff8000000000000000
Has QNaN: true
QNaN: 0x00007f3d6b5e7fffc000000000000000
Has SNaN: true
SNaN: 0x00007f3d6c097fffa000000000000000
```

Listing A.6: Floating-point features of Red Hat Linux.

```
student.cs.appstate.edu
Red Hat Enterprise 6.5
AMD Opteron 8222 SE

Endianness: little

Feature report for f:
Size (bits): 32
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x40490fdb
Epsilon: 1.19209e-07
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7f800000
Has QNaN: true
QNaN: 0x7fc00000
Has SNaN: true
SNaN: 0x7fa00000

Feature report for d:
Size (bits): 64
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x400921fb54442d18
Epsilon: 2.22045e-16
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x7ff0000000000000
Has QNaN: true
QNaN: 0x7ff8000000000000
Has SNaN: true
SNaN: 0x7ff4000000000000

Feature report for e:
Size (bits): 128
IEEE-754: true
Sample value of pi (3.14159265358979323846):
        0x00000000000004000c90fdaa22168c235
Epsilon: 1.0842e-19
Denormal: present
Rounding style: towards nearest
Has infinity: true
Infinity: 0x0000003a31897fff8000000000000000
Has QNaN: true
QNaN: 0x0000003a27867fffc000000000000000
Has SNaN: true
SNaN: 0x0000003a31af7fffa000000000000000
```