Understanding TTBlue
A Not So Short Introduction to the TTBlue Environment

Timothy Place
Electrotap, LLC.

October 12, 2009

Aknowledgements

The first version of TTBlue was originally written in the Autumn of 2003. A large portion of the initial environment was created in the lobby of the Augustin Hotel in Bergen, Norway during series of workshops hosted by BEK[1]. TTBlue was originally authored as the basis of Electrotaps[2] Tap.Tools 2, a set of plug-ins for Cycling 74s Max/MSP[3] environment. This in turn formed the basis of the Hipno[4] VST/AU/R-TAS plug-ins released by Electrotap and Cycling 74. TTBlue was open-sourced in the Spring of 2005.

As an open source initiative, TTBlue has received valuable input from a large number of contributors in mostly informal ways. I can't hope to acknowledge everyone here. Trond Lossius and Dave Watson are both active participants in the development of TTBlue and have added immeasurably to the environment, not just by the addition of code, but by their feedback and critique. Joshua Kit Clayton also provided a valuable critique of TTBlue while we demonstrated Hipno at the Winter 2005 NAMM Show in Anaheim, California.

Many contributions to TTBlue are fed through the Jamoma[5] project. In fact, many aspects of TTBlue were first developed by the Jamoma team and then ported to TTBlue, including the FunctionLib developed at a Jamoma workshop hosted by iMal[6]. Through its support of Jamoma, GMEA[7] has also helped indirectly in supporting TTBlue.

This document owes its creation in part Trond Lossius, who introduced me to LaTeX(with which it is written), the Not So Short Introduction to LaTeX(for which it is named), and also provided the basic structure for the document. He is also largely responsible for the initial creation of the [8] website, which is generously hosted by BEK.

[1]`http://bek.no/`
[2]`http://electrotap.com/`
[3]`http://cycling74.com/products/maxmsp/`
[4]`http://cycling74.com/products/hipno/`
[5]`http://jamoma.org/`
[6]`http://imal.org/`
[7]`http://gmea.net/`
[8]`http://ttblue.org`

Contents

Chapter 1

Introduction

TTBlue is a reflective... (copy and paste from website).
Discuss the vocabulary/jargon term-by-term.

Chapter 2

Data Types

We will begin our introduction of the TTBlue environment by looking at the basic data types. All other aspects of the environment, creating objects, writing extensions, etc., all require some knowledge about how data is represented.

2.1 Primitive Data Types

2.1.1 Numbers

To begin, TTBlue defines a variety of basic types for representing integers and a floating-point numbers with various degrees of resolution, and in signed or unsigned variants. Table 2.1 lists the basic numeric types according to their properties.

	Unsigned Integer	**Signed Integer**	**Floating-Point**
8-bit	`TTUInt8`	`TTInt8`	
16-bit	`TTUInt16`	`TTInt16`	
32-bit	`TTUInt32`	`TTInt32`	`TTFloat32`
64-bit	`TTUInt64`	`TTInt64`	`TTFloat64`

Table 2.1: Selected numeric data types in TTBlue.

2.1.2 Booleans

In addition to these types for representing numbers, there are basic types for representing booleans (true/false values), TTBoolean.

2.1.3 Strings and Symbols

Strings can be represented in several different ways in TTBlue. Of course, arrays of the standard char type is well understood way to work with text in C. The `TTString` type, at the time of this writing, is a typedef of the C++ std::string, and thus follows the conventions of the string provided by the C++ standard library.

In addition to `TTString`, there is also a `TTSymbol` type. A symbol is simply a wrapper around a string that is cached in a fast lookup table. While comparing `TTString` values is relatively slow, comparing `TTSymbol` values is extremely fast.

Symbols are never created directly. Rather, to use a symbol you lookup the symbol in the symbol table. If the symbol is there, then a pointer to the symbol is returned to you. If the symbol is not in the table, then it is created, added to the table, and the pointer is returned to you. Because looking up symbols is such a central operation in TTBlue, and because we will use it so often, the easy-to-type macro TT is defined to do this. For example:

```
TTSymbolPtr bearSymbol = TT("bear");
```

2.2 Composite Data Types

2.2.1 TTValue

While is important to have defined the basic data types for numbers, strings, and booleans, this is not the way that values are typically passed in TTBlue. Instead, values are passed as TTValues. `TTValue` is a generic type that can hold any of the number, string, boolean, or a few other types that we use in TTBlue. In fact, it can even hold an array of values made up of these various types. The following example shows several assignments using TTValues.

```
// Assigning numbers and symbols to TTValues
TTValue v = 3.1415;
TTValue s = TT("hog");

// Assigning TTValues to numbers
TTUInt16 i = v;
```

2.2.2 TTObject

Another type that can be represented with a `TTValue` is the `TTObject` type. Well discuss more about that in coming sections.

Chapter 3

Using Objects

3.1 Object Life Cycle

3.1.1 Creating and Destroying

To create an instance of a TTBlue object, use the `TTObjectInstantiate()`. This will look up the class in TTBlues registry of objects and return a pointer to the new instance.

An example below creates a stereo instance of an allpass filter, and two stereo audio signals. They are stereo because the argument given when instantiating these types of objects define the initial number of audio channels as two. You can safely ignore the fact that one of the variables is a TTAudioObjectPtr instead of a `TTObjectPtr`. TTAudioObject is simply a specialized version of `TTObject`.

```
TTAudioObjectPtr  myObject       = NULL;
TTObjectPtr       myAudioInput   = NULL;
TTObjectPtr       myAudioOutput  = NULL;

TTObjectInstantiate(TT("allpass")), &myObject, 2);
TTObjectInstantiate(TT("audiosignal")), &myAudioInput, 2);
TTObjectInstantiate(TT("audiosignal")), &myAudioOutput, 2);
```

When you are all done using the objects, you need to release them. You can do this with the `TTObjectRelease()` function. For example:

```
TTObjectRelease(myObject);
TTObjectRelease(myAudioInput);
TTObjectRelease(myAudioOutput);
```

3.1.2 Retaining and Reference Counts

TTObjects are reference counted. This means that you can create references to existing objects, and the object will not be freed until all references have been released. This is demonstrated in the following example:

```
TTObjectPtr myObject = NULL;
TTObjectPtr aReferenceToMyObject = NULL;

TTObjectInstantiate(TT( noise )), &myObject, 2);
// reference count is now 1
aReferenceToMyObject = TTObjectReference(myObject);
// reference count is now 2

TTObjectRelease(myObject);
// The object is not actually deleted, but the reference count is now 1
TTObjectRelease(aReferenceToMyObject);
// Now, because the reference count fell to zero, the object is deleted.
```

3.2 Querying the Environment

As alluded to in section 3.1, TTBlue maintains a registry of available classes that may be instantiated. Some of these classes are implemented internally in the TTBlue library, and some are implemented externally as TTBlue Extensions. The registry does not differentiate between these, but simply provides a list of everything that is loaded in the system.

3.2.1 Getting All Class Names

To obtain a list of all class in the TTBlue registry, you call the `TTGetRegisteredClassNames` function as in the following example:

```
TTErr err;
TTValue classNames;

err = TTGetRegisteredClassNames(classNames);
if(!err){
  // The classNames value now contains an array of TTSymbolPtrs,
  // one for each class in the TTBlue registry.
  // For example, we can print them all to console:

  TTUInt16 numClassNames = classNames.getSize();

  for(TTUInt16 i=0; i<numClassNames; i++){
    TTSymbolPtr aClassName;

    classNames.get(i, aClassName);
    TTLogMessage("class␣name:␣%s", aClassName->getCString());
  }
}
```

3.2.2 Searching For Classes Based on Tags

In addition to retrieving all class names, it is also useful to be able to retrieve a limited number of class names based on criteria that you specify. For example, you may wish to only list classes that generate their own audio. Or perhaps only those class which implement some sort of lowpass filter.

In this case we call `TTGetRegisteredClassNamesForTags()`, and process the results in the same manner as we did for `TTGetRegisteredClassNames()`. This is demonstrated in the example below:

```
TTErr err;
TTValue classNames;
TTValue searchTags;

searchTags.clear();
searchTags.append(TT("audio"));
searchTags.append(TT("filter"));
searchTags.append(TT("lowpass"));

err = TTGetRegisteredClassNamesForTags(classNames, searchTags);
if(!err){
  // The classNames value now contains an array of TTSymbolPtrs,
  // one for each class in the TTBlue registry.
  // For example, we can print them all to console:

  TTUInt16 numClassNames = classNames.getSize();

  for(TTUInt16 i=0; i<numClassNames; i++){
    TTSymbolPtr aClassName;

    classNames.get(i, aClassName);
    TTLogMessage("class␣name:␣%s", aClassName->getCString());
  }
}
```

For a list of common tags and what they mean in TTBlue see Appendix A. To get a list of all tags in use at any time, call `TTGetRegisteredTags()`.

Tag Searching and Instantiation in Action

Here is an example that searches based on tags for a lowpass filter using a Butterworth algorithm, and then creates an instance of that class.

```
// In this case there are multiple matches returned.
// Since more specific information was not provided,
// we just instantiate the first one.

TTErr             err;
TTValue           classNames;
TTValue           searchTags;
TTAudioObjectPtr  butterworthFilter = NULL;

searchTags.clear();
searchTags.append(TT("audio"));
searchTags.append(TT("filter"));
searchTags.append(TT("lowpass"));
searchTags.append(TT("butterworth"));
```

```
err = TTGetRegisteredClassNamesForTags(classNames, searchTags);
if(!err){
  // by passing classNames, the TTObjectInstantiate() function will take the first
  err = TTObjectInstantiate(classNames, &butterworthFilter, 1);
}
// now do something with the filter.
```

3.3 Sending Messages

Having created an instance of an object, we must now do something with that object. We do things with objects by sending them messages. A message defines an action to be performed.

Given the Butterworth filter example in **??**, we can now send the filter the 'clear' message to zero its sample history[1].

```
butterworthFilter->sendMessage(TT("clear"));
```

Some messages, like the 'clear' message for the Butterworth filter, require no additional information to perform the requested action. Other messages, however, do require additional information. This information can be provided using an optional argument to the sendMessage method. Here are some samples:

```
someObject->sendMessage(TT("foo"), 3.14);
someObject->sendMessage(TT("draw"), TT("circle"));

// create a TTValue that holds a list and pass the TTValue
TTValue v(1.0, 2.0));
someObject->sendMessage(TT("w"), v);

// when you send a message, it can return a value as well.
TTValue whatIsInThere;
TTErr   err;

err = someObject->sendMessage("getContents", whatIsInThere);
if(!err){
  // Assuming that someObject understands the message,
  // this results in whatIsInThere being set to contain something meaningful.
}
else if(err == kTTErrMethodNotFound){
  // The object didn't understand this message'
}
else{
  // There was some other problem.
  // Probably the object you sent the message to knows this message and is returnir
  // an error specific to the action you requested.
}
```

[1]The Butterworth filter is an IIR filter, meaning that it stores the results of previous calculations to perform future calculations. This feedback can sometimes get out of control, and thus the necessity for a 'clear' message.

3.4 Setting and Getting Attributes

Sending messages is great for performing actions, however these actions do not represent the state of the object. The state of the object is represented as the object's 'attributes'.

3.5 Querying Objects for Available Messages and Attributes

3.6 Processing Audio

Not all objects can process audio, however TTBlue objects that have the tag audio associated with them are able process audio. These objects derive from TTAudioObject, a subclass of `TTObject` which implements the process method.

Chapter 4

Writing Objects

Chapter 5

Extending Objects

5.1 Subclassing Objects

The classical methodology for extending functionality in object-oriented designs is to subclass an object. In other words, we derive a new object that inherits all of the functionality of a parent class. This is also possible in TTBlue, though you may wish to keep a few guidelines in mind when subclassing.

...

5.2 Decorating Objects

Because TTBlue is a dynamic and reflective API for creating objects, we can extend objects by means other than subclassing. One specific way of doing this is by decorating a class.

For example, let us assume that we have a bandpass filter for processing audio. Further, this object understands how to control its center frequency using an attribute specified in Hertz. What are we to do if we want to communicate with the object using the Bark scale?

One option is to write a conversion routine and always call that routine to convert the value, then send a message the filter. A second option is subclass the filter, creating another filter which has the required attribute specified using Barks. A third option is to decorate the existing filter.

5.2.1 Two Kinds of Decoration

The Decorator Pattern, in the classical sense, is where an object is passed a second object and this second object uses the first object to help get its work done (**?**; **?**). However, the work done by the class is typically fixed and does not change. For example, an HTML rendering engine may have the ability to take a decorator object that will provide a custom footer for a page, but the page structure is fixed. Another example is in writing tests (**?**; **?**).

Another kind of decoration is to actually expand a class by adding messages and attributes to an instance after it has been created. While the former means of passing in a decorator class for a pre-determined purpose is useful in some contexts in TTBlue, this later means of creating entirely new functionality is what we are looking to do for solving the problem defined in 5.2

5.2.2 An Example

by adding a new attribute to it at runtime.

...

5.3 Keeping An Eye On Objects

If you are working with objects, extending them, or creating a graph of connected objects, you will likely need to keep tabs on some objects or certain aspects of those objects. TTBlue provides an easy way to monitor the state of objects using an Observer strategy (**?**; **?**).

In the example below we have an object and two observers. The observers provide a way to view any changes to the state of our object.

```
TTObjectPtr ourObject;
TTObjectPtr firstView;
TTObjectPtr secondView;
```

Chapter 6

Unit Testing

Note: This chapter currently serves as the specification for the unit testing facilities in TTBlue. The unit testing facilities are only partially (if at all) implemented. Thus, this chapter is detailing how it should work in the future, and not necessarily how it works right now.

Chapter 7

Object Wrappers

Appendix A

Common Tags

audio - indicates that this object derived from TTAudioObject can can process audio

filter - indicates that the object is a processor that implements a standard type of filtering whereby the frequencies of the input are attenuated or boosted to produce a spectrally transformed version at the output.

generator - indicates that this object generates its own audio, and may not respond to audio input.

processor - indicates that this object uses an audio input source to produce a transformed audio output.

Bibliography