

The UObject Tutorial for Urbi 1.x

(book compiled from 682M)

Matthieu Nottale

The UObject Tutorial for Urbi 1.x: (book compiled from 682M)

by Matthieu Nottale

Publication date

Copyright © 2005-2007 Gostai

This document is released under the Attribution-NonCommercial-NoDerivs 2.0 Creative Commons licence (<http://creativecommons.org/licenses/by-nc-nd/2.0/deed.en>).

Table of Contents

1. Introduction	1
Basic concepts	1
2. The UObject API	2
Creating a class, binding variables and functions	2
Creating new instances	3
Notification of a variable change or access	3
Timers	3
The special case of sensor/effector variables	3
Using URBI variables	3
Using binary types	4
Using hubs to group objects	4
Sending URBI code	4
3. Use cases	5
Writing a servomotor device	5
Caching	8
Using timers	8
Using hubs to group objects	9
Alternate implementation	10
Writing a camera device	11
Optimization in plugin mode	13
Writing a speaker or microphone device	13
Writing a softdevice: ball detection	13
A. Copyright	15

Chapter 1. Introduction

This document guides you through the various steps of writing an URBI C++ component using the UObject API. This API can be used to add new objects written in C++ to the URBI language, and to interact from C++ with the objects that are already defined. We cover the use cases of controlling a physical device (servomotor, speaker, camera...), and interfacing higher-level components (voice recognition, object detection,...) with URBI.

Basic concepts

The API defines the UObject class. Each instance of a derived class in your C++ code will correspond to an URBI object sharing some of its methods and attributes. The API provides methods to declare which elements of your object are to be shared. To share a variable with URBI, you have to give it the type UVar. This type is a container that provides cast and equal operators for all types known to URBI: double, string and char*, and the binary-holding structures UBinary, USound and UImage. This type can also read from and write to the liburbi UValue class. The API provides methods to set up callbacks functions that will be notified when a variable is modified or read from URBI code. Instance methods of any prototype can be rendered accessible from URBI, providing all the parameters types and the return type can be converted to/from UValue.

The next chapter is a walk through covering all the aspects of the UObject API, and the rest of the document describes many use cases in details.

Chapter 2. The UObject API

Creating a class, binding variables and functions

Let's illustrate those concepts by defining a simple object: `adder`. This object has one variable `v`, and a method `'add'` that returns the sum of this variable and its argument.

First the required include and namespace.

```
#include <uobject.hh>
```

Then we declare our `adder` class:

```
class adder : public urbi::UObject // must inherit from UObject
{
public:
    // the class must have a single constructor taking a string
    adder (const std::string&);

    // our variable
    urbi::UVar v;

    // our method
    double add (double);
};
```

Finally the implementation of the constructor and our `add` method.

```
// the constructor defines what is available from URBI
adder::adder (const std::string& s)
    : UObject (s) // required
{

    // macro used to bind variables
    UBindVar (adder,v);

    // macro used to bind methods
    UBindFunction (adder, add);
}

double
adder::add (double rhs)
{
    return v + rhs;
}

// register the class to the URBI kernel.
UStart (adder);
```

To summarize:

- Declare your object class as inheriting from `urbi::UObject`.

- Declare a single constructor taking a string, and pass this string to the constructor of `urbi::UObject`.
- Declare the variables you want to share with URBI with the type `urbi::UVar`.
- In the constructor, call `UBindVar (classname, varname)` for each `UVar` you want as an instance variable, and `UBindFunction (classname, functionname)` for each function you want to bind.
- Don't forget to call the macro `UStart` for each object.

Creating new instances

When you start an URBI server, an object of each class registered with `UStart` is created with the same name as the class. New instances can be created from URBI using the `new` command. For each instance created in URBI, a corresponding instance of the C++ object is created. You can get the arguments passed to the constructor by defining and binding a method named `init` with the appropriate number of arguments.

Notification of a variable change or access

You can register a function that will be called each time a variable is modified or accessed (for embedded components only) by calling `UNotifyChange` and `UNotifyAccess`, passing either an `UVar` or a variable name as first argument, and a member function of your `UObject` as second argument. This function can take zero or one argument: a `UVar &` pointing to the `UVar` being accessed or modified. The `notifyChange` callback function is called after the variable value is changed, whereas the `notifyAccess` callback is called before the variable is accessed, giving you the possibility to update its value.

Timers

The API gives you two methods to have a function called periodically:

- Calling `USetUpdate (int period)` in your object sets up a timer that calls the virtual `UObject` method `update ()` with the specified period (in milliseconds)
- Calling `USetTimer` passing the period in milliseconds and a pointer to a method in your `UObject` will call this method at the specified period.

The special case of sensor/e#ector variables

In URBI, a variable can have a different meaning depending on whether you are reading or writing it: you can use the same variable to represent the target value of an e#ector and the current value measured by an associated sensor. This special mode is activated by the `UObject` defining the variable by calling `UOwned` after calling `UBindVar`. This call has the following effects

- When URBI code or code in other modules read the variable, they read the current value. When URBI code or code in other modules write the variable, they set the target value.
- When the module that called `UOwned` reads the variable, it reads the target value. When it writes the variable, it writes the current value.

If `UOwned` is not called, then the variable has only one associated value that is read or written by all modules and URBI code.

Using URBI variables

You can read or write any URBI variable by creating an `UVar` passing the variable name to the constructor. Change the value by writing any compatible type to the `UVar`, and access the value

by casting the UVar to any compatible type. However, some care must be taken in remote mode: changes on the variable coming from urbi code or an other module are only visible if you have called UNotifyChange on this variable. Otherwise the UVar is not synchronized. Alternatively, in remote mode, you can get the value on demand by calling UNotifyOnRequest (variable, function). Then to get an updated value, call the requestValue method on the UVar, and your callback function will be called as soon as the value is available. You can read and write all the URBI properties of an UVar by reading and writing the appropriate UProp object in the UVar.

Using binary types

URBI can store binary objects of any type in a generic container, and provides specific structures for sound and images. The generic containers is called UBinary and is defined in the urbi/uobject.hh header. It contains an enum field type giving the type of the binary (UNKNOWN, SOUND or IMAGE), and an union of a USound and UImage struct containing a pointer to the data, the size of the data and type-specific metainformations. Reading a UBinary from an UVar, and writing a UBinary to a UVar performs a deep-copy of the data. In plugin mode, you can access directly the buffer used by the kernel by casting the UVar to an UImage or a USound. You can then write to the buffer, but you can not change any other information.

Using hubs to group objects

Sometimes, you need to perform actions for a group of UObjects, for instances devices that need to be updated together. The API provides the UObjectHub class for this purpose. To create a hub, simply declare a subclass of UObjectHub, and register it by calling once the macro UStartHub (classname). A single instance of this class will then be created upon server startup. UObject instances can then register to this hub by calling URegister (hubclassname). Timers can be attached to UObjectHub the same way as to UObject (see Timers section above). The kernel will call the update () method of all UObject before calling the update () method of the hub. A hub instance can be retrieved by calling getObjectHub (string classname). The hub also holds the list of registered UObject in its members attribute.

Sending URBI code

If you need to send URBI code to the server, the URBI () macro is available, as well as the send () function. You can either pass it a string, or directly URBI code inside a couple of parenthesis:

```
send ("tag:1+1;");
```

```
URBI (( at (someevent (x)) { sometag:echo x; }; ));
```

Chapter 3. Use cases

Writing a servomotor device

Let's write a UObject for a servomotor device whose underlying API is:

- `bool initialize (int id); //initialize the servomotor with given ID`
- `double getPosition (int id); //read servomotor of given id position`
- `void setPosition (int id, double pos); //send a command to servomotor`
- `void setPID (int id, int p, int i, int d); //set PID arguments`

First our header. Our servo device provides an attribute named "val", the standard URBI name, and two ways to set PID gain: a method, and three variables.

```
class servo : public urbi::UObject //must inherits UObject
{
public:
    // the class must have a single constructor taking a string
    servo(const std::string&);

    // URBI constructor
    int init(int id);

    // main attribute
    urbi::UVar val;

    // position variables:
    // P gain
    urbi::UVar P;
    // I gain
    urbi::UVar I;
    // D gain
    urbi::UVar D;

    // callback for val change
    int valueChanged(UVar &v);
    //callback for val access
    int valueAccessed(UVar &v);
    // callback for P I D change
    int pidChanged(UVar &v);

    // method to change all values
    void setPID(int p, int i, int d);

    // motor ID
    int id_;
};
```

The constructor only registers init, so that our default instance "servo" does nothing, and can only be used to create new instances.

```
servo::servo (const std::string& s)
: urbi::UObject (s)
```

```

{
    // register init
    UBindFunction (servo, init);
}

```

The init function, called in a new instance each time a new URBI instance is created, registers the three variables (val, P, I and D), and sets up callback functions.

```

// URBI constructor
int
servo::init (int id)
{
    id_ = id;

    if (!initialize (id))
        return 1;

    UBindVar (servo, val);

    // val is both a sensor and an actuator
    Uowned (val);

    // set blend mode to mix
    val.blend = urbi::UMIX;

    // register variables
    UBindVar (servo, P);
    UBindVar (servo, I);
    UBindVar (servo, D);

    // register functions
    UBindFunction (servo, setPID);

    // register callbacks on functions
    UNotifyChange (val, &servo::valueChanged);
    UNotifyAccess (val, &servo::valueAccessed);
    UNotifyChange (P, &servo::pidChanged);
    UNotifyChange (I, &servo::pidChanged);
    UNotifyChange (D, &servo::pidChanged);

    return 0;
}

```

Then we define our callback methods. `servo::valueChanged` will be called each time the `val` variable is modified, just after the value is changed: we use this method to send our servo command. `servo::valueAccessed` is called just before the value is going to be read. In this function we request the current value from the servo, and set `val` accordingly.

```

//called each time val is written to
int
servo::valueChanged (urbi::UVar & v)
{
    // v is a ref. to val
    setPosition (id, (double)val);

    return 0;
}

```

```
// called each time val is read
int
servo::valueAccessed(urbi::UVar & v)
{
    // v is a ref. to val
    val = getPosition (id);

    return 0;
}
```

`servo::pidChanged` is called each time one of the PID variables is written to. The function `servo::setPID` can be called directly from URBI.

```
int
servo::pidChanged (urbi::UVar &v)
{
    setPID(id, (int)P, (int)I, (int)D);

    return 1;
}
```

```
void
servo::setPID (int p, int i, int d)
{
    setPID (id, p, i, d);
    P = p;
    I = i;
    D = d;
}
```

```
//register servo class to the URBI kernel
UStart (servo);
```

That's it, compile this module, and you can use it within URBI:

```
//creates a new instance, and calls init (1)
headPan = new servo (1);

//calls setPID ()
headPan.setPID (8,2,1);

// calls valueChanged ()
headPan.val=13;

// calls valueAccessed ()
headPan.val * 12;

// periodically calls valueChanged ()
headPan.val = 0 sin:1s ampli:20,

// periodically calls valueAccessed ()
at (headPan.val < 0)
    echo "left";
```

The sample code above has one problem: `valueAccessed` and `valueChanged` are called each time the value is read or written from URBI, which can happen quite often. This is a problem if sending the

actual command (setPosition in our example) takes time to execute. There are two solutions to this issue:

Caching

One solution is to remember the last time the value was read/written, and not apply the new command before a #xed time. Note that the kernel is doing this automatically for Uowned()'d variables that are in a blend mode di#erent than "normal". So the easiest solution to the above problem is likely to set the variable to the "mix" blending mode. The unavoidable drawback is that commands are not applied immediately, but only after a small delay.

Using timers

Instead of updating/fetching the value on demand, you can chose to do it periodically based on a timer. A small di#erence between the two API methods comes in handy for this case: the update () virtual method called periodically after being set up by USetUpdate (interval) is called just after one pass of URBI code execution, whereas the timers set up by USetTimer are called just before one pass of URBI code execution. So the ideal solution is to read your sensors in the second callback, and write to your actuators in the #rst. Our previous example (ommiting PID handling for clarity) can be rewritten. The header becomes:

```
// inherits from UObject
class servo : public urbi::UObject
{
public:
    //the class must have a single constructor taking a string
    servo (const std::string&)

    // URBI constructor
    int init (int id);

    // called periodically
    virtual int update ();
    // called periodically
    int getVal ();

    // our position variable
    urbi::UVar val;

    // motor ID
    int id_;
};
```

Constructor is unchanged, init becomes:

```
int
servo::init (int id)
{
    // urbi constructor
    id_ = id;

    if (!initialize (id))
        return 0;

    UBindVar (servo,val);
    //val is both a sensor and an actuator
```

```

UOwned(val);

// will call update () periodically
USetUpdate(1);
// idem for getVal ()
USetTimer (1, &servo::getVal);

return 0;
}

```

valueChanged becomes update and valueAccessed becomes getVal. Instead of being called on demand, they are now called periodically. The period of the call cannot be lower than the server period (which is chosen when starting the robot, or #xed by the underlying architecture), so you can set it to 0 to mean "as fast as is usefull".

Using hubs to group objects

Now, suppose that, for our previous example, we can speed things up by sending all the servomotor commands at the same time, using the method setPositions (int count, int *ids, double * positions) that takes two arrays of ids and positions. A hub is the perfect way to handle this task. The UObject header stays the same. We add a hub declaration:

```

class servohub : public urbi::UObjectHub
{
public:
//the class must have a single constructor taking a string
servohub (const std::string&);

// called periodically
virtual int update ();

// called by servo
void addValue (int id, double val);

int* ids;
double* vals;
int size;
int count;
};

```

servo::update becomes a call to the addValue method of the hub:

```

int
servo::update()
{
((servohub*)getUObjectHub ("servohub"))->addValue (id, (double)val);
};

```

The following line can be added to the servo init method, although it has no use in our speci#c example:

```
URegister(servohub);
```

Finally, the implementation of our hub methods is:

```
servohub::servohub (const std::string& s)
```

```

    : UObjectHub (s),
      ids   (0),
      vals  (0),
      size  (0),
      count (0)
  {
    // setup our timer
    USetUpdate (1);
  }

  int
  servohub::update ()
  {
    // called periodically
    setPositions (count, ids, vals);

    // reset position counter
    count = 0;

    return 0;
  }

  void
  servohub::addValue (int id, double val)
  {
    if (count + 1 < size)
    {
      // allocate more memory
      ids = (int*)realloc (ids, (count + 1) * sizeof (int));
      vals = (double*)realloc (vals, (count + 1) * sizeof (double));
      size = count + 1;
    }
    ids[count] = id;
    vals[count++] = val;
  }

  UStartHub (servohub);

```

Periodically, the update method is called on each servo instance, which adds commands to the hub arrays, then the update method of the hub is called, actually sending the command and resetting the array.

Alternate implementation

Alternatively, to demonstrate the use of the members hub variable, we can entirely remove the update method in the servo class (and the USetUpdate() call in init), and rewrite the hub update method the following way:

```

int servohub::update()
{
  //called periodically
  for (UObjectList::iterator it = members.begin ();
       it != members.end ();
       it++)
  {
    addValue (((servo*)*it)->id,
              (double)((servo*)*it)->val);
  }
}

```

```

    }
    setPositions(count, ids, vals);
    // reset position counter
    count=0;

    return 0;
}

```

Writing a camera device

A camera device is an `UObject` whose `val #eld` is a binary object. The urbi kernel itself doesn't differentiate between all the possible binary formats and data type, but the API provides `image-spec#c` structures for convenience. You must be careful about memory management. The `UBinary` structure handles its own memory: copies are deep copies, and the destructor frees the associated `bu#er`. The `UImage` and `USound` structures do not.

Let's suppose we have an underlying camera API with the following functions:

- `bool initialize (int id); // initialize the camera with given ID`
- `int getWidth (int id); // read image width`
- `int getHeight (int id); // read image height`
- `char* getImage (int id); // get image bu#er of format RGB24. The bu#er returned is always the same and doesn't have to be freed.`

Our device code can be written as follows:

```

//inherit from UObject
class Camera : public urbi::UObject
{
public:
    // the class must have a single constructor taking a string
    Camera(const std::string&);

    // URBI constructor
    int init (int id);

    // our image variable and dimensions
    urbi::UVar val;
    urbi::UVar width;
    urbi::UVar height;

    // called on access
    int getVal (UVar &);

    // called periodically
    virtual int update ();

    //frame counter for caching
    int frame;
    //frame number of last access
    int accessFrame;
    //camera id UBinary bin;
    int id_;
};

```

The constructor only registers init:

```
//the constructor registers init only
Camera::Camera (const std::string& s)
  : urbi::UObject (s),
    frame (0)
{
  // register init
  UBindFunction (Camera, init);
}
```

The init function binds the variable, a function called on access, and sets a timer up on update. It also initialises the UBinary structure.

```
int
Camera::init (int id)
{
  //urbi constructor
  id_ = id;
  frame = 0;
  accessFrame = 0;

  if (!initialize (id))
    return 0;

  UBindVar (Camera, val);
  UBindVar (Camera, width);
  UBindVar (Camera, height);
  width = getWidth (id);
  height = getHeight (id);

  UNotifyAccess (val, &Camera::getVal);

  bin.type=BINARY_IMAGE;
  bin.image.width = width;
  bin.image.height = height;
  bin.image.imageFormat = IMAGE_RGB;
  bin.image.size = width * height * 3;

  //will call update () periodically
  USetUpdate (1);

  return 0;
}
```

The update function simply updates the frame counter:

```
int
Camera::update ()
{
  frame++;
  return 0;
}
```

The getVal updates the camera value, only if it hasn't already been called this frame, which provides a simple caching mechanism to avoid performing the potentially lengthy operation of acquiring an image too often.

```
int
Camera::getVal(urbi::UVar &)
{
    if (frame == accessFrame)
        return 1;

    bin.image.data = getImage (id);
    //assign image to bin
    val = bin;
}

UStart(Camera);
```

The image data is copied inside the kernel when proceeding this way.

Be careful, suppose that we had created the UBinary structure inside the getVal method, our bu#er would have been freed at the end of the function. To avoid this, set it to 0 after assigning the UBinary to the UVar.

Optimization in plugin mode

In plugin mode, it is possible to access the bu#er used by the kernel by casting the UVar to a UImage. You can modify the content of the kernel bu#er but no other argument.

Writing a speaker or microphone device

Sound handling works similarly to image manipulation, the USound structure is provided for this purpose. The recommended way to implement a microphone is to #ll the UObject val variable with the sound data corresponding to one kernel period. If you do so, the urbi code "loop tag:micro.val," will produce the expected result.

Writing a softdevice: ball detection

Algorithms that require intense computation can be written in C++ but still be useable within URBI: they acquire their data using UVar referencing other modules's variables, and output their results to other UVar. Let's consider the case of a ball detector device that takes an image as input, and outputs the coordinates of a ball if one is found.

The header is de#ned like:

```
// inherit from UObject
class BallTracker : public urbi::UObject
{
public:
    //the class must have a single constructor taking a string
    BallTracker (const std::string&);

    // URBI constructor
    int init (const std::string& varname);

    // is the ball visible ?
    urbi::UVar visible;

    // ball coordinates
    urbi::UVar x;
```

```
    urbi::UVar y;  
};
```

The constructor only registers init:

```
//the constructor registers init only  
BallTracker::BallTracker (const::string& s)  
    : urbi::UObject (s)  
{  
    //register init  
    UBindFunction (BallTracker, init);  
}
```

The init function binds the variables and a callback on update of the image variable passed as a argument.

```
int  
BallTracker::init (const std::string& cameraval)  
{  
    UBindVar (BallTracker, visible);  
    UBindVar (BallTracker, x);  
    UBindVar (BallTracker, y);  
    UNotifyChange (cameraval, &BallTracker::newImage);  
  
    visible = 0;  
  
    return 0;  
}
```

The newImage function runs the detection algorithm on the image in its argument, and updates the variables.

```
int  
BallTracker::newImage (urbi::UVar &v)  
{  
    //cast to UImage  
    urbi::UImage i = v;  
    int px,py;  
    bool found = detectBall (i.data, i.width, i.height, &px, &py);  
  
    if (found)  
    {  
        visible = 1;  
        x = px / i.width;  
        y = py / i.height;  
    }  
    else  
        visible = 0;  
  
    return 0;  
}
```

Appendix A. Copyright

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

1. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License. 2. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License. 3. "Licensor" means the individual or entity that offers the Work under the terms of this License. 4. "Original Author" means the individual or entity who created the Work. 5. "Work" means the copyrightable work of authorship offered under the terms of this License. 6. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

1. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the

Collective Works; 2. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Derivative Works. All rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Sections 4(d) and 4(e).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

1. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. 2. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works. 3. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; and to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit. 4.

For the avoidance of doubt, where the Work is a musical composition: 1. Performance Royalties Under Blanket Licenses. Licensor reserves the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work if that performance is primarily intended for or directed toward commercial advantage or private monetary compensation. 2. Mechanical Rights and Statutory Royalties. Licensor reserves the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions), if Your distribution of such cover version is primarily intended for or directed toward commercial advantage or private monetary compensation. 5. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor reserves the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions), if Your public digital performance is primarily intended for or directed toward commercial advantage or private monetary compensation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License. 2. Subject to the above terms and conditions, the license granted here is perpetual (for

the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

1. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License. 2. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable. 3. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent. 4. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.