Animation Library

December 22, 2013

The Animation library is a simple parallel processing script engine for manipulating large number of LEDs through a number of chained ULK devices.

Chapter 1

Animation Concepts

The Animation library runs a number of *programs* in parallel, where each *program* is a list of special instructions and their associated parameters. Instructions are executed in blocks separated by *delay* instructions. When a *delay* instruction is encountered the program is released and the next *program* is executed. Each animation instance has its own context and no animation can affect another animation.

Programs are stored as arrays of 8-bit unsigned characters (typically *uint8_ t*, *byte* or *unsigned char*) and must be terminated by one of the *terminating* instructions, such as LOOP or END.

Simple counted repeats are supported, as well as infinite repeats with an external interrupt (known as a *nudge*) to force loop termination.

Operations can be grouped into three broad categories: *control*, *LED* and *flow*. *Control* instructions perform such facilities as LED group definitions, *LED* instructions directly manipulate the state of LEDs and groups of LEDs and *flow* instructions define how the program proceeds.

Each instruction is stored internally as an 8-bit unsigned value, or *op-code*, and the library provides a number of symbolic names for these *op-codes* to enable simpler and more readable programming.

Chapter 2

Animation Instructions

2.1 Control Instructions

2.1.1 GROUP (op-code 1)

The GROUP command defines a new (or replaces an existing) LED group definition. A group of LEDs can all be manupilated together with LED instructions through the use of a group number instead of an LED number.

The GROUP command takes a minimum of 3 parameters, but can take many more. The first parameter is the number of the group to define or redefine. Groups are numbered between 200 and 255 inclusive. The second parameter is the number of LEDs in the group. This is then followed by that number of parameters, one for each LED. For instance, to define group 200 with 3 LEDs (6, 7 and 8) in it, the command structure would be:

```
Animation::GROUP, 200, 3, 6, 7, 8
```

2.2 LED Commands

2.2.1 SET (op-code 2)

The set command is used to directly set the brightness of either a single LED or a group of LEDs defined with the GROUP command. The command takes 2 parameters - the number of the LED or GROUP to change, and the value to change it to. For instance, to set the LED number 7 to 50% brightness the command would be:

```
Animation::SET, 7, 128
```

Or to set the GROUP 200 to full brightness:

```
Animation::SET, 200, 255
```

2.2.2 FADE (op-code 3)

The FADE command, like the SET command, changes the brightness of an LED or group of LEDs. Unlike the SET command, however, the change isn't immediate. Instead the LED or GROUP fades gradually between the current brightness and the desired target brightness. The rate of the change is specified as a delay in milliseconds between each step of brightness.

The command takes 3 paramaters. The first is the number of the LED or GROUP to change. The second is the target brightness value (0-255) and the third is the number of milliseconds delay per step of the fade.

Example - fade LED 4 up to full brightness with a delay of 10ms per step:

```
Animation::FADE, 4, 255, 10
```

2.3 Flow Commands

2.3.1 DELAY (op-code 4)

The DELAY command pauses the program for the specified number of milliseconds. The command takes two parameters, which are the high and low bytes of a 16-bit value, most significant byte first. For instance, a 1 second delay, which is 1000 milliseconds, would be represented as the two bytes 3 and 232 (3 * 256 + 232 = 1000):

```
Animation::DELAY, 3, 232
```

A preprocessor macro PAIR is provided to perform this splitting for you:

```
Animation::DELAY, PAIR(1000)
```

2.3.2 WAITEQ (op-code 5)

The WAITEQ command delays the program until any FADE instructions have finished processing (it stands for WAIT until EQual). It takes no parameters:

```
Animation::WAITEQ
```

2.3.3 RDELAY (op-code 6)

The RDELAY command performs a random length delay. Like the DELAY command the delay is measured in milliseconds, and is provided as a pair of bytes representing a 16-bit value. Two values are provided as parameters representing the minimum and maximum delay times to use. Again, the PAIR macro can be used. For example, for a random delay between 100 and 1000 milliseconds:

```
Animation::RDELAY, PAIR(100), PAIR(1000)
```

2.3.4 REPEAT (op-code 7)

This command repeats the following block of code, up to the first LOOP command found, a specified number of times, up to 255. Only one parameter is provided, which is the number of iterations to perform:

```
Animation::REPEAT, 10
... code ...
Animation::LOOP
```

2.3.5 FOREVER (op-code 8)

FOREVER acts like the REPEAT command, however a number of iterations is not specified. The following block repeats indefinitely until a *nudge* is sent to the animation at which point the loop finishes and the program continues:

```
Animation::FOREVER
... code ...
Animation::LOOP
```

2.3.6 LOOP (op-code 254)

This is one of the instructions classed as a terminating instruction and as such is one of the few instructions which may be used as the final instruction of a program.

The LOOP command returns the program either back to the previous REPEAT or FOREVER command (if one has been used) or to the start of the program. If a REPEAT or FOREVER has not been previously issued the LOOP will always loop the program regardless of any nudges:

```
Animation::LOOP
```

2.3.7 END (op-code 255)

The END command terminates the program. Any LEDs affected by the program remain at their final settings until changed by another animation. The animation may be manually restarted by the user:

```
Animation::END
```

2.4 Example Program

This small script will fade an RGB LED (the pins are defined by the preprocessor macros RGBR, RGBG adnd RGBB and should be provided by the calling program) to yellow, cause it to throb gently for a while, then fade it to blue and back to yellow. Extra line breaks have been added to aid readability:

```
Animation::FADE, RGBR, 64, 10,
Animation::FADE, RGBG, 64, 10,
Animation::FADE, RGBB, 0, 10,
Animation::WAITEQ,

Animation::REPEAT, 50,

Animation::FADE, RGBR, 80, 10,
Animation::FADE, RGBG, 80, 10,
Animation::FADE, RGBB, 0, 10,
Animation::WAITEQ,

Animation::FADE, RGBR, 64, 10,
Animation::FADE, RGBG, 64, 10,
Animation::FADE, RGBB, 0, 10,
Animation::WAITEQ,

Animation::LOOP,

Animation::FADE, RGBR, 0, 10,
Animation::FADE, RGBG, 10, 10,
Animation::FADE, RGBB, 64, 10,
Animation::WAITEQ,
Animation::DELAY, PAIR(2000),

Animation::LOOP
```

Chapter 3

API

The API contains both static and instance-based operations for controlling animations.

A new animation sequence is created by calling the static createAnimation() function and passing it a pointer to the program array. The animation is started by calling it's start() function, stopped by calling its stop() function, etc.

All animation processing is performed by a repeated call to the static process() function.

3.1 Static Functions

3.1.1 Animation *Animation::createAnimation([[const] uint8_t *program]);

This function creates a new Animation object, optionally assigns the program to it, and returns the new object to the user. The object is also internally added to the animation processing list.

Animation *myAnimation = Animation::createAnimation(myProgram);

3.1.2 void Animation::process();

Iterate through the animation processing list and execute blocks of instructions up to a delay class instruction for each animation registered. This function needs to be called repeatedly (as fast as possible ideally) from loop().

Animation::process();

3.1.3 void Animation::addAnimation(Animation *anim);

Add an animation constructed elsewhere to the internal processing list. Under normal circumstances this function should not be used; instead create the objects with the Animation::createAnimation() function.

3.2 Instance-based functions

3.2.1 void setAnimation([const] uint8_t *program);

Replace (or set) the program for the current animation. If the animation is running the new program is started from the beginning. Care should be taken using this function as any LEDs will retain the state they had at the time the program was replaced.

myAnimation->setAnimation(myOtherProgram);

3.2.2 void start();

Begin animating the animation. The program wil run, from the beginning, either until it reaches an END instruction or the program is manually terminated with stop().

myAnimation->start();

3.2.3 void stop();

Terminate a running animation. All LEDs retain their existing states.

myAnimation->stop();

3.2.4 void nudge();

Indicate to a program in a FOREVER loop that it should leave the loop at the end of the current iteration.

myAnimation->nudge();

3.3 Example Program

This small program executes the example animation from the previous chapter.

```
#include <ULK.h>
#include <Animation.h>

#define RGBG 0
#define RGBB 1
#define RGBR 2

const uint8_t newLEDs[] = {
        Animation::FADE, RGBR, 64, 10,
        Animation::FADE, RGBG, 64, 10,
        Animation::FADE, RGBB, 0, 10,
        Animation::WAITEQ,
        Animation::REPEAT, 50,
        Animation::FADE, RGBR, 80, 10,
        Animation::FADE, RGBG, 80, 10,
        Animation::FADE, RGBB, 0, 10,
        Animation::WAITEQ,
        Animation::FADE, RGBR, 64, 10,
        Animation::FADE, RGBG, 64, 10,
        Animation::FADE, RGBB, 0, 10,
        Animation::WAITEQ,
        Animation::LOOP,
        Animation::FADE, RGBR, 0, 10,
        Animation::FADE, RGBG, 10, 10,
        Animation::FADE, RGBB, 64, 10,
        Animation::WAITEQ,
        Animation::DELAY, PAIR(2000),
        Animation::LOOP
};

void setup() {
        ULK.begin(1, 10);
        Animation *anim1 = Animation::createAnimation(newLEDs);
        anim1->start();
}

void loop() {
        Animation::process();
}
```