ChirpinoSing for Arduino

Version 1.0 May 2015

chirp.io

Overview

ChirpinoSing enables Arduinos (and UCL Engduinos) to output chirps as described at **chirp.io** through attached audio hardware.

You'll need an Arduino Uno (or a board with an equivalent processor such as the Arduino Ethernet) an Arduino Mega 2560 or an Engduino. For sound output you will need a compatible speaker or other device (we have found just connecting an earbud directly to the pins works, although it is likely to be overdriving the earbud), chirp codes from chirp.io (test codes are provided in the examples), and something such as smartphone running the free Chirp app to listen out for the chirps emitted.

Installing ChirpinoSing

ChirpinoSing is written for the Arduino IDE versions 1.0.6 and above, including 1.6.3 which is the most recent at the time of writing. Install **ChirpinoSing** as a library. For instructions, see

```
http://arduino.cc/en/Guide/Libraries
```

Once installed, you can access the example programs from the menu **File > Examples > ChirpinoSing > *example*** and you can include the headers to use **ChirpinoSing** in your own code by using **Sketch > Import Library > ChirpinoSing**.

Using ChirpinoSing

Hardware: Connect a speaker or other audio device to your board. Use ground plus pin 3 on the Uno, ground plus pin 9 on the Mega, or ground plus pin I/O_0 on the Engdiuno.

Software: create a **Beak** or **PortamentoBeak** and ask it to chirp. For instance, **File > Examples > ChirpinoSing > Simple** contains this complete program:

```
#include <ChirpinoSing.h>
Beak beak;
void setup() {}
void loop() {
    beak.chirp("ntdb982ilj6etj6e3l");
    delay(2000);
}
```

Much more detail follows. After notes on audio and chirps, a class reference outlines the beak classes for all and the synthesiser class for those wanting finer control. Then a guide to the example programs follows as an introduction to a range of techniques. **ChirpinoSing** is a source code library, so for the ultimate reference take a look at the code itself.

Audio output

On the **Uno**, the main sound signal is supplied on digital pin **3**, and on the **Mega** it's on digital pin **9**. Use pin **I/O_0** on the **Engduino**. Connect this pin and a ground pin to your compatible speaker or audio device.

The signal supplied is a 62.5kHz, pulse width modulated (PWM) approximation to an ideal audio waveform. It’s 2.7V on the Engduino and 5V on the other Arduinos. This PWM technique is explained further below but it's important to understand that it's up to your attached audio hardware to smooth this signal. Ideally, this smoothing would be performed by additional filter circuitry before the result is fed to your audio device, however in practice the 62.5kHz pulse train is at too high a frequency for audio hardware to pass unchanged, and so simple speakers will naturally smooth the signal.

Although amplification & smoothing circuitry would be ideal, if you want to keep things simple you can attach a small speaker or an earbud directly to the Arduino pins. The exact nature of the smoothing, and so the fine characteristics of the sound, will depend on the particular hardware that you attach. We have had success directly connecting the pins to a 3.5mm audio lead and plugging this into a device such as an old radio. Please be aware that such a signal won't match the official audio jack electrical specifications (so keep the volume low), and you will certainly be overdriving an earbud if you connect it this way (don’t put it in your ear!). Whatever hardware you connect to the Arduino you do at your own risk; it's your responsibility to check for compatibility.

Pulse Width Modulation

The signal generated approximates a sine wave that is changing continuously as appropriate for a chirp. However, the Arduino can't supply a true variable analog output; at any one time a pin may only ever be switched on (5V or 2.7V depending on device) or off (0V). So to approximate a sine wave the pin is pulsed rapidly, with 62500 pulses being issued every second (one every 16 microseconds) under the assumption that this pulse train will be smoothed (averaged over time). The targeted Arduinos run at 16MHz so a new pulse is issued every 256 clock cycles, the Engduino runs at 8MHz so issue one every 128 clock cycles. In each 16-microsecond period the pin is only turned on for a fraction of the time, from 0% (0V, not turned on at all) to 100% (5V on an Arduino, turned on the whole time). For instance, to approximate a 3.75V signal (75% of 5V) the pulse will be switched on for 192 clock cycles, that is 12 microseconds (75% of the time). Since we are approximating a continuously varying sinusoidal signal, each successive pulse is likely to be on for a different time than its predecessor. This description is provided only to help you to understand the output signal; the generation of these pulses is fully handled for you by the synthesiser.

Chirps

Codes

Chirps are specified by strings containing 18 characters. Each character selects one of 32 tones; it must be either a digit (**0**-**9**) or one of the first 22 lower-case letters (**a**-**v**). Eg:

```
"ntdb982ilj6etj6e3l"
```

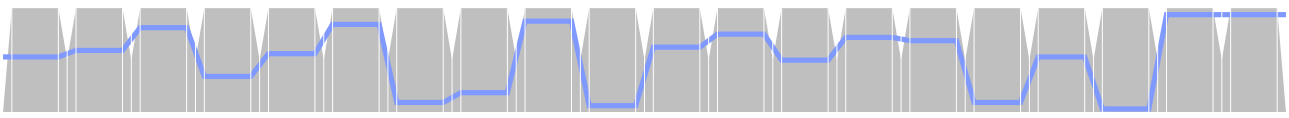
When chirps are played, two extra tones (corresponding to tones **h** and **j**) are prepended to the 18 characters you supply to make 20 in total. Chirp codes from **chirp.io** contain some data and some error correction characters based on the data. **ChirpinoSing** will play any string of the correct length and characters, but only strings from **chirp.io** contain the valid error correction codes that let listening devices receive them properly, and only very particular strings link to content.

Sounds

The **ChirpinoSing** library provides two chirp-generator classes: the plain and simple **Beak**, and the fancier-sounding **PortamentoBeak**.

This is what a **Beak** looks like. Grey depicts volume level, blue frequency (for the example chirp code above) and the white columns divide *frames* (described shortly).

Beak objects play the codes as 20 pure tones in succession. Each *block* (corresponding to one character in the code) lasts 87.2 milliseconds, comprises a single frame, and is played at the same volume (which may be specified). The frequency change between tones happens abruptly. **Beak**'s only frill is that the first tone is quickly faded in (the volume is raised from zero) and the last tone is similarly faded out since sudden volume changes tend to create unpleasant clicks.



The second picture shows a **PortamentoBeak**. These slide the frequency between each main tone and dip the volume briefly while the slides are occurring so they sound more characterful. Their blocks also last 87.2 ms, but each is divided into three *frames* and the frequency and volume are only constant during the central sustained frame. By default, 12 ms is spent *ramping* (sliding and fading) the values at the beginning of a block, and 12 ms ramping them at the end. This leaves the central 63.2 ms to play a constant tone at the specified volume. The timings and volume levels may be adjusted as desired.

Classes

Beak

A class providing the basic chirp functionality together with a simple volume control.

Creation

`Beak()`
Creates a **Beak** object initialised to output at full volume (255).

`Beak(volume)`
Creates a **Beak** object initialised to output at the specified volume (0 to 255).

The highest volume (255) provides greatest output range and so also the best potential sound quality. Reduced values might be preferred in cases where the higher ones overload your audio hardware. Very low levels should be avoided as the highly restricted output range degrades the sound quality. The exact effective working range depends on the sound output hardware used.

Methods

`chirp(chirpStr)`
The chirp described by **chirpStr** (see **Chirps / Codes** above) is played and the method returns when the synthesiser has finished playing, around 1.7 seconds later.

Accepted chirp strings are exactly 18 characters long and use only the digits **0**-**9** and/or the lowercase letters **a**-**v**.

Chirp returns 0 if all is well, else a `warnings` value to indicate the problem. See the **Trouble** example program for a demonstration of the use of `warnings`.

`chirp(chirpStr, enoughSpaceForFrames)`
The first version of chirp allocates temporary space for the frames on the stack and returns this for general use when its done. If you have a buffer large enough for the frames already you can pass it to this function and save requiring extra stack space. Be aware, that at least 793 bytes are required for portamento chirps.

`chirp(chirpStr, enoughSpaceForFrames)`
The final version allows you to allocate memory as a pre-allocated array of frames. It is primarily for use by the original simple chirp function,

`setVolume(volume)`
The volume (0-255) used by subsequent chirps.

Constants

`BAD_CHIRP_CODE_WARNING`
If the bit `(warnings & BAD_CHIRP_CODE_WARNING)` is non-zero then a string that includes an invalid character has been supplied to **chirp**. Valid chirp strings use only the digits **0**-**9** and/or the letters **a**-**v**.

`CHIRP_STRING_LENGTH_WARNING`
If the bit `(warnings & CHIRP_STRING_LENGTH_WARNING)` is non-zero then a string that isn't exactly 18 characters long has been supplied to **chirp**.

`TOO_LITTLE_RAM_WARNING`
If the bit `(warnings & TOO_LITTLE_RAM_WARNING)` is non-zero then there is not enough space on the stack to allocate working storage for the chirp. Beak requires at least 555 bytes to be available, PortamentoBeak requires 1049.

PortamentoBeak

A subclass of **Beak** that provides a more sophisticated sounding chirp. **PortamentoBeak** objects feature all of their superclass **Beak**'s methods, attributes and constants as well as those listed below. **setVolume** affects only **maxVolume** when used on a **PortamentoBeak**.

Creation

`PortamentoBeak()`
creates a **PortamentoBeak** initialised to the default parameter settings. Equivalent to `PortamentoBeak(128, 255, 750)`.

`PortamentoBeak(minVolume, maxVolume, rampTime)`
minVolume (0-255) specifies the volume that the sound output momentarily ramps to between blocks (see the second diagram in **Chirps / Sounds** above). Although normally expected to be, it doesn't have to be smaller than the **maxVolume**.

maxVolume (0-255) specifies the volume used for the central sustained frame of each block. Although normally expected to be, it doesn't have to be larger than the **minVolume**.

rampTime (0-1362) specifies the duration of the frames surrounding the central sustained frame as a pulse count. Since pulses are sent every 16 microseconds you can consider these to be 16-microsecond time units. So to calculate the length of **rampTime** in milliseconds multiply it by 0.016. To calculate the length of the remaining sustained frame subtract twice the **rampTime** from the total block time.

For example, each block is 5450 pulses long (87.2 ms) and comprises a ramped frame either side of a sustained frame. If you set **rampTime** to 1000 (16 ms) then the sustained frame will be 3450 pulses (= 5450 - 2*1000), which is 55.2 ms. Alternatively, a **rampTime** of zero leaves the sustained frame to fill the block and so creates a **Beak**-style chirp.

Methods

`setParameters(minVolume, maxVolume, rampTime)`
Allows these values (as described for the constructor above) to be changed for subsequent chirps.

Synth

Synth provides a general purpose sine-wave synthesiser that generates the PWM pulses on the output pin. Direct interaction with the synthesiser is optional, and is only required for more advanced applications.

The beak classes feed instructions to the synthesiser as a sequence of *frames*. Each frame describes a section of sound in which the amplitude and frequency are changing from starting to finishing values in a linear manner. Frames supplied by the user program are stored in sequence in an array. A frame whose duration is given as 0 (an *end frame*) signals the end of the sequence. When a sound description sequence is complete, **play** is called to actually generate the sound.

Synth is the only class within **ChirpinoSing** to interact directly with timer and output pin hardware of the processor. It uses a hardware timer to begin a pulse on the output pin every 16 microseconds. The length of each pulse is adjusted according to the phase of the waveform and position of the playhead within the frame sequence.

Creation

Synth is a singleton class; the one and only **Synth** object is available to you in the global variable **TheSynth**. Don't create any other **Synth** objects.

Methods

`beginFrameSequence(nFrames, theFrames)`
Call this to start a new **SynthFrame** sequence, supplying the location of space to store the frames in **theFrames**, and size of this space as the maximum number of frames it has capacity for in **nFrames**. Each **SynthFrame** requires 13 bytes.

`addFrame(duration, startPhaseStep, endPhaseStep, startVolume, endVolume)`
A frame describes a chunk of sound based on a sine wave where the frequency and volume may be changing steadily from start to end over the specified duration. This method lets you add a frame to the sound description.

duration is specified as a pulse count (see **PortamentoBeak rampTime**).

Frequencies are specified by phase step values **startPhaseStep** and **endPhaseStep**. The phase step tells the synthesiser how far to cycle through its sine wave look-up table between successive pulses. Multiply a frequency in hertz by 536.870912 and round the

result to obtain the corresponding phase step value. The derivation of this number is a little complex, but look for a block comment inside **Beak.cpp** for clues.

Amplitudes are specified by **startVolume** and **endVolume** using the 0-255 scale.

`addSustainFrame(duration, phaseStep, volume)`
A convenience method, for use where the amplitude and frequency remain unchanged over the duration of the frame. Equivalent to `addRampFrame(duration, phaseStep, phaseStep, volume, volume)`.

`endFrameSequence()`
Adds a frame of zero duration to the frame store to signal the end of the frame sequence.

`play()`
This requests the synthesiser to play the sound sequence described by the supplied frames. Interrupts are turned off and the synthesiser is given 100% control over the processor during sound generation. Chirps last around 1.7 seconds.

The method returns a **warnings** value describing any problems that have been encountered. If this is zero then everything was fine. See the **Trouble** example program for an extensive demonstration of the use of warnings.

Constants

`MISSING_END_FRAME_WARNING`
A non-zero `(warnings & MISSING_END_FRAME_WARNING)` bit signals that you have attempted to play a sound sequence before you have called **endFrameSequence**.

`NO_FRAMES_TO_PLAY_WARNING`
A non-zero `(warnings & NO_FRAMES_TO_PLAY_WARNING)` bit signals that the sound sequence contained *only* an end-frame.

`LATE_SAMPLE_PULSE_WARNING`
A non-zero `(warnings & LATE_SAMPLE_PULSE_WARNING)` bit signals that the tight synthesiser pulse generation loop has been delayed. It should only be possible if you modify the loop to add extra tasks.

Examples

Once **ChirpinoSing** is installed as a library the examples supplied are available in the IDE menu **File > Examples > ChirpinoSing > *example***. They are designed to be explored in approximately the following order:

1) Simple
2) Variation
3) Trouble
4) CustomBeak

Simple

A minimal program. A **Beak** object is created and is used to chirp a specific code over and over, with two-second gaps. A good test to check your library is installed and functioning.

Variation

PortamentoBeak is introduced and contrasted with **Beak** as variants are played to demonstrate how the sound shaping parameters may be used. The main action is within the **loop** function. **nextChirpString** is provided as a means to cycle through the list of chirp codes.

Trouble

Demonstrates using the warning codes to help troubleshoot problems.

chirp returns a **warnings** value that you can ignore if you wish, or can pay attention to if you are investigating problems. While all is well warnings is zero, but whenever a problem is spotted a bit is set in the corresponding field.

Trouble and the subsequent example write to the Arduino IDE serial monitor at 9600 baud so you should open the monitor and set it to this speed to see the messages. Serial output is sent asynchronously using interrupts after the print command returns, but chirping turns off these interrupts so serial output may be delayed or lost. As demonstrated here, the simplest way to avoid this is to use **delay** to wait a while to allow the serial output to finish before issuing a chirp.

Only a few warnings might be encountered in general use. The others are only possible if you make your own custom beaks or attempt to modify the synthesiser code.

CustomBeak

This example shows how to subclass **Beak** as a straightforward way to make your own personalised beak to incorporate your own sound design. It's the same technique that's used within the library to make the **PortamentoBeak**. Of course, there's nothing forcing you to subclass **Beak**; you could interact with the synthesiser directly or even write your own synthesiser should you wish to do more work.

If your beak requires more frames to describe your sound sequence than the superclass (here, **Beak**) then you must provide this number in a virtual **numberOfFrames** method. This dictates the amount of storage that will be set aside for the sound description.

Beak supplies three other virtual methods that you can override; **head**, **append** and **tail**. **head** is called at the start of a chirp, and **tail** is called at the end. In between, **append** is called once for each of the 20 blocks (the 2 standard start blocks then the 18 supplied to **chirp**). These methods are each given a phase step value (which specifies the core sound frequency for the block), **head** is given the phase step of the first block, **tail** that of the last block, and append is given the phase step of the block it relates to.

It's the beak's job to take this information and feed the synthesiser with the appropriate frames to shape the sound. Any number of frames may be used to represent a block, but each block should last 5450 pulses (87.2ms) and you will need to hold the core frequency for a substantial portion of this time to give listening devices a chance to interpret the chirp.

The **CustomBeak** example creates a beak that plays around with pitch slides. Once defined it is used just like a built-in beak.