GinSing Programming Guide

version 4.0

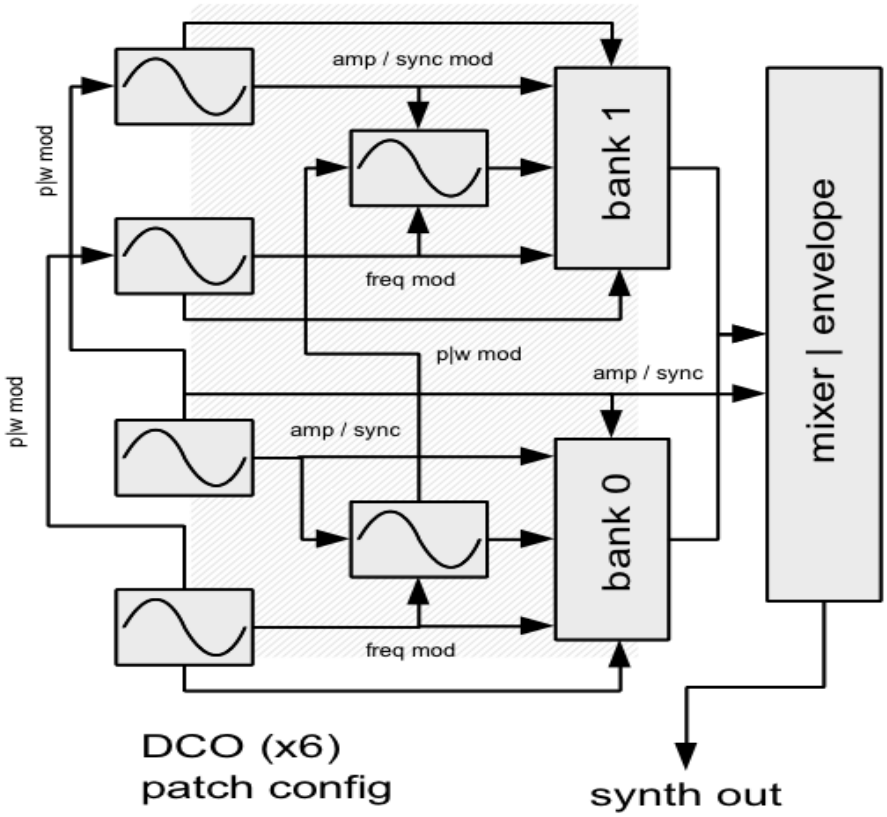


Table of Contents

overview

purpose

The GinSing software library is a **C++ class interface** that communicates with the GinSing Arduino Shield. The purpose of this document is to provide a **programming overview** of the interface so you can quickly integrate the GinSing functionality into your applications. In addition to this document, you can use the GinSing Software Reference guide to find out more specifics about what the library functions do.

what is the GinSing library?

The GinSing library is a collection of **source files** written in C++ that integrate into your Arduino applications through the Arduino Integrated Development Environment (IDE). When you compile your Arduino application in the IDE, the source code is included in the compilation process and linked into the executable program that is downloaded onto your Arduino board.

The library communicates with the GinSing shield using a simple command and register control communication interface as dictated by the GinSing's processing chip, known as the **Babblebot IC**. This low level interface can be called directly if you wish, but the library also provides **high level conceptual software models** that organize the functionality of the Babblebot to make it easier to understand and control the Babblebot based on the type of uses you may have for it.

software models

The software models in the library each target a specific element of functionality that is available on the shield. You can switch from one model (or mode) to another quickly and easily, allowing you to **expand functionality as you develop** your application. The four software modes are **preset**, **poly**, **voice**, and **synth** as outlined in more detail below.

code integration

The library is organized into **groups of functions (classes)** based on which mode you are using. Those familiar with the Arduino interface are most likely familiar with this as all of the Arduino interface functions (for example Serial) are structured in the same way (i.e. Serial.println()).

Unlike the Arduino, however, this library has a **global interface class** that you create explicitly; the Arduino creates its interface classes internally, which means they are always linked into your application whether you need them or not.

So to get started using the library, you need first to **include the library** code in your project and then **create the GinSing interface** class in your app code. Note that nothing will happen at this point; you have simply included the library into your application:

```
#include <GinSing.h>    // include the GinSing library
GinSing  GS;            // create the GinSing interface class
```

Note that during the installation process the GinSing library files needed for this code to compile will have been copied into your Arduino sketch folder. You can therefore **examine any of the GinSing source code** by looking in that directory. This can be very useful to understand the inner workings of the library and how you can interact with the low level functions if you wish.

library structure

Although you may not ever need or want to examine the source code for the library, it is good to know that you can if you want to find out more about the inner workings. You can also modify the code as you see fit for your application. The source code is broken int C++ source files (.cpp) and header files (.h) as described here:

	library base class
GinSing.cpp GinSing.h	the GinSing base class. This header file is included in your application, and is the only one needed, as it includes all of the other header files inside it.
	constants and definitions
GinSingDefs.h	all the constants, enumerations, and variable types that are used in GinSing are contained in this one file. You can reference it when writing code that makes calls to the GinSing library. It is included automatically in GinSing.h
	preset mode
GinSingPreset.cpp GinSingPreset.h	implements the preset operating mode, which allows you load preconfigured synthesizer settings into memory and play and/or modify them in realtime.
	poly mode
GinSingPoly.cpp GinSingPoly.h	implements the polyphonic operating mode, which sets up the system to act as 6 identically configured musical voices.
	voice mode
GinSingVoice.cpp GinSingVoice.h	implements the voice mode, which simulates speech using built-in phonemes with realtime control over voice parameters

synth mode

GinSingSynth.cpp
GinSingSynth.h

implements the synthesizer mode, which contains the bulk of functions in the system for complete control over the sound generation.

master functions

GinSingMaster.cpp
GinSingMaster.h

implements the non-mode specific functions such as global volume, command and control, etc.

serial interface

GinSingSerial.cpp
GinSingSerial.h
GinSingSerialDefs.h

implements the low-level serial interface used by the library to communicate between the Arduino and the GinSing shield.

programming basics

bring in the ginsing

In order to use the GinSing library in your Arduino application, you need only include the single header file GinSing.h, and create an instance of the C++ interface class from which you call its functions.

GinSing uses the C++ class abstraction as a simple way to organize all of its functions in a convenient manner. If you are new to C++ you can consider the operation of creating (instancing) the GinSing class as creating a portal to the functions that GinSing offers. In concrete terms, you need to have the following code as a minimum in your application.

```
#include <GinSing.h>    // include the GinSing library
GinSing  GS;            // create the GinSing interface class

void setup()            // Arduino setup function called on powerup
{
}
void loop()             // Arduino loop function called repeatedly
{
}
```

Note that every Arduino program must define the setup() and loop() functions, and if you have working code you already have coded these up.

In this example, we have created a static class called GS that we will use to reference the GinSing library functions. We could have just as easily called it myGinSing, or any other name you prefer. If all goes well this program won't do anything except compile, but we do now have access to the GinSing library.

getting going

Lets begin by accessing the initialization function to start up the system. Before any other calls are made to the library, we must call **begin().** This function will initialize the serial connection to the chip and configure it in a default state. Likewise, when shutting down the system, we should call **end()** to quiet the chip and terminate the serial connection.This is typically done in your setup() function anywhere its appropriate. As this function does not actually produce any sound it can be placed wherever it is convenient, typically where you are initializing the rest of your application.

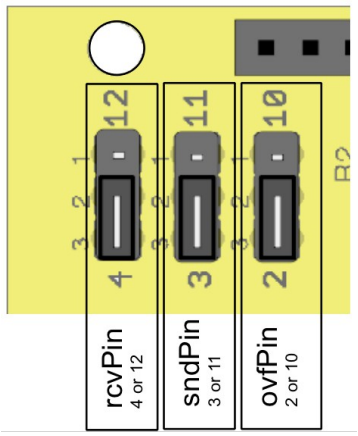
```
void setup()
{

 #define rcvPin  4       // receiving    ( can be either 4 or 12 )
 #define sndPin  3       // transmitting ( can be either 3 or 11 )
 #define ovfPin  2       // overflow control ( can be either 2 or 10 )

 GS.begin( rcvPin , sndPin , ovfPin );
}
```

Note how we access the base GS class to call the function. This allows the GinSing functions to exist in their own world (namespace) to avoid conflict with other code. Other base class functions exist and can be called the same way, as described later on in this document.

The begin() function requires three arguments which correspond to the hardware setting on the shield. On the board, there are three jumpers that can each be set to one of two options. These jumpers physically connect the Arduino to the GinSing shield and establish what pins on the Arduino will be allocated to support the communications. More information about what these pins do can be found in the reference guide. Unless you have need to modify the pins (due to other shields using the default pins), you can probably just keep them where they are and use the pin definitions in this example.



code the mode

The next step is to select the operating mode. The GinSing library operates in one of four modes, and you can switch from one mode to another mode at will; this allows you for example to switch from speaking a phrase to playing a musical melody instantly. Note that you cannot operate two modes simultaneously as each mode configures the hardware in a unique way (with the exception of punch-through defined later). Lets begin by entering voice mode, and then have it say something:

```
void loop()
{

 GSAllophone phrase[] = { _IE , _A , _M ,
                          _BENDDN , _J, _I , _NE ,
                          _SE ,  _PITCHDN , _I , _PITCHDN , _NGE ,
                          _PA0 , _ENDPHRASE };

 GS.getVoice().begin();
 GS.getVoice().setNote ( C_2 );
 GS.getVoice().speak ( phrase );

 delay ( GS.getVoice().getMillis ( phrase ) + 500 );
}
```

In this example, we have placed the speaking code in the loop() function. By doing this, it

will repeatedly speak over and over as the Arduino calls the loop() function repeatedly. You could also place this code in the setup() function if you just wanted to speak it once. It is important to understand the differences between these two functions as you develop your code.

As you can see above, each mode has its own set of functions within the GinSing class. In this example, we are using the getVoice() function in the base GS class to get the voice mode functions, and then make calls to that function. Note that the first function called when selecting a mode is to call begin(), which sets up the system in a way that lets other functions in the mode operate properly. What this code does is define a phrase to speak (“I am GinSing”), starts voice mode, sets the voice frequency to musical note C, speaks the phrase, and then waits for the phrase to finish before moving on.

accessorize

This method of accessing mode functions is perfectly valid, but you may also opt to store off the mode functions as its own variable to make it simpler to type. Here is the same example, but we store off the voice mode functions and reference them from the stored variable:

```
void loop()
{
 GingSingVoice *v = GS.getVoice();

GSAllophone phrase[] = { _IE , _A , _M ,
                         _BENDDN , _J, _I , _NE ,
                         _SE ,  _PITCHDN , _I , _PITCHDN , _NGE ,
                         _PA0 , _ENDPHRASE };

 v->begin();
 v->setNote ( C_2 );
 v->speak ( phrase );

 delay ( v->getMillis ( phrase ) + 500 );
}
```

This is a more compact way to access the mode, and can be done for any of the other modes; you just need to store off the mode class in your code in a variable that is accessible when you need it. In this example the variable v stores off the voice mode interface that we can use at any time once we have retrieved it form the base class.

We have purposely not discussed the functionality of this code; the idea here is to grok how you will be writing your application to call the GinSing functions in general.

what you've learned

To review:

1. include the GinSing header file outside of any code blocks
2. instance the base GinSing class in your startup code
3. get the operating mode interface class from the base class
4. call the begin() method when first starting or switching a mode
5. call functions in the mode interface that you want to use

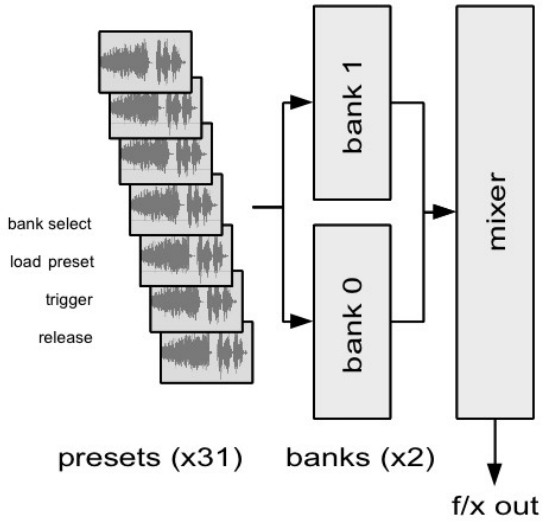
mode overview

There are four functional modes in the library – preset mode, poly mode, voice mode, and synth mode; each targeting a different type of application. You can quickly and easily switch between modes, allowing you to develop features as you learn more about how the library works.

Each of the modes has its own set of functions that reflects its primary purpose. For this reason this reference is organized by mode. Some modes have functions with the same name (for example trigger()) and although may perform a similar function may require different arguments.

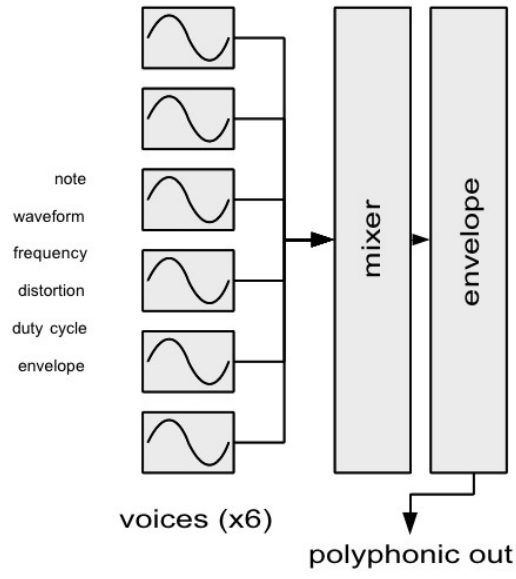
preset mode

Preset mode is the simplest of the four modes and has the simplest interface functions. In preset mode, you can **trigger on-board preset configurations** of the system to play sound effects. Up to two presets can be loaded and triggered at a given time (one for each of the two banks). Preset mode is a good place to start when adding sound to your application because it requires minimal code and knowledge about how the system works; it will get you up and running with sounds in a very short time.



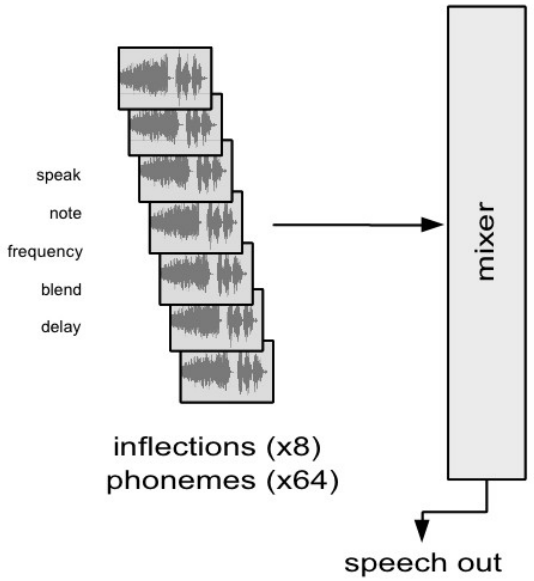
poly mode

Poly mode (or polyphonic mode) configures the system to operate as a **six channel musical instrument**. Each channel (or voice) operates independently allowing up to 6 simultaneous tones to be produced. This mode is a simplification of synth mode in that all the voices are configured identically and sent directly to the output, and allows for parameter changes to occur on all six voices using the same function call (i.e. change waveform type).



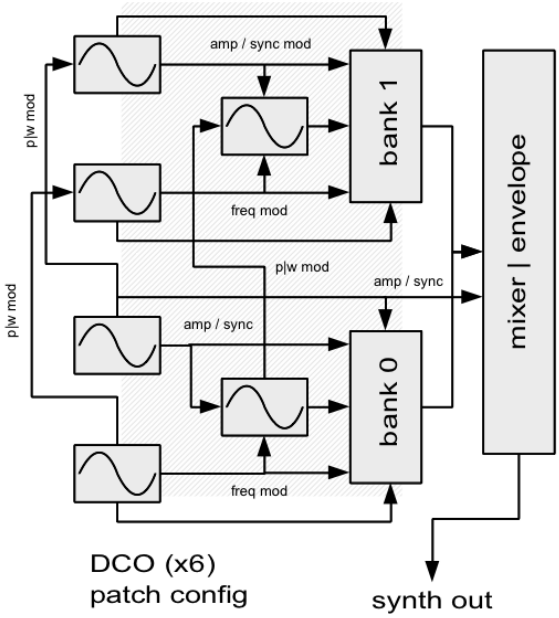
voice mode

Voice mode can be used to produce **artificial speech**. When voice mode is activated, all resources in the system are used internally for the purpose of generating human (or otherwise) voice synthesis. The interface provides the ability to string together basic speech fragments (called allophones) into phrases as well as control the tonal qualities of the synthesis. Voice mode provides a very simple way to add artificial voice to your Arduino project.



synth mode

Synth mode can be used to directly control all aspects of complex waveform synthesis on the Babblebot. The system is configured into 2 banks of 3 digitally controlled oscillators (DCOs) that are patched in such as way as to allow DCOs to modulate each other creating complex waveform patterns and tonal qualities. Synth mode operates in much the same way as analog synthesizers do, but does so with complete digital control. It is the most complicated interface, but also has the most user functionality.



master interface

In common with all of the above modes is a common master interface that is available regardless of what mode you are currently in. The master interface controls the **global aspects of the system**, such as overall output volume, timing functions, and command and control functions. The master functions are available via the getMaster() method in the base GinSing class an can be called at any time after the system has been initialized.

punch-through

One thing to note about the operating modes that they vary in complexity, with synth mode being the most complex by far. This is important because the other modes (preset, voice, and poly) are just simplifications to synth mode; they set up the system in such as way to provide a simpler conceptual interface for the task at hand.

For example, poly mode configures the synthesizer's six DCOs with the same waveform, envelopes, and other parameters so that it looks to you like a six voice polyphonic tone generator, but is actually only making calls to synth mode internally.

By taking advantage of this relationship of the other modes to synth mode, you can enhance any of the other modes by calling synth mode functions while not actually in synth mode. This act of reaching into synth mode is called “punch-through” because you can punch directly into the synth mode functions without having to switch modes.

As an example of why you might want to do this, let's consider the simplest mode – preset mode. When you enter preset mode and trigger a preset, there appears not much more to do, but in fact you can start making synth mode calls after you trigger the preset to change all of its operating parameters, such as pitch of the modulation DCOs etc. Although we don't want to bog you down too much with the details, we encourage to you experiment with punch-through once you get your app up and running.

function overview

Hopefully the concepts presented so far have given you an understanding on how to syntactically write code for GinSing, and give you an idea on what generally the system can do. This section provides an overview of the functions that are available in each of the classes in the system with a brief description to help guide you through the planning of your project. For a complete description of the functions, arguments, and features please refer to the reference manual.

base class functions

The base class functions provide the ability to initialize the system and gain access other sections of the interface. It also provides for the low level command interface used internally. This class (GingSing) must be explicitly created in your application with any variable name you prefer; this document assumes the name “GS”.

state control

begin	*connect and initialize GinSing shield*
end	*mute and disconnect GinSing shield*
reset	*reset GinSing shield to power up state*
isReady	*test if communication link is available*
getVersion	*get the version of this library*

subclass accessors

getPoly	*get the poly mode interface*
getVoice	*get the voice mode interface*
getPreset	*get the preset mode interface*
getSynth	*get the synth mode interface*
getMaster	*get the master interface*

command & register control

sendCommand	*send a command to the Babblebot IC*
writeRegister	*write to a Babblebot IC register*
readRegister	*read from a Babblebot IC register*

preset mode functions

Preset mode is the simplest of the four modes and has the simplest interface functions. In preset mode, you can trigger on-board preset configurations of the system to play sound effects. Up to two presets can be loaded and triggered at a given time (one for each of the two banks). Preset mode is a good place to start when adding sound to your application because it requires minimal code and knowledge about how the system works; it will get you up and running with sounds in a very short time.

The presets stored on the Babblebot IC are simply predefined register sets that are copied into the registers when loaded. Once loaded, you can optionally modify the registers using the synth mode functions; this allows you to form base effects that you can customize and modify in real-time.

Preset mode functions are accessed via the GinSingPreset class obtained through the base class getPreset() function.

state control

begin	*configure system for preset mode*
preview	*play preset mode demo*

envelope control

load	*load a preset from built-in memory*
trigger	*trigger the amplitude envelope*
release	*release the amplitude envelope*
setAmplitude	*set the amplitude*

poly mode functions

Poly mode (or polyphonic mode) configures the system to operate as a six channel musical instrument**.** Each channel (or voice) operates independently allowing up to 6 simultaneous tones to be produced. This mode is a simplification of synth mode in that all the voices are configured identically and allows for parameter changes to occur on all six voices using the same function call (i.e. change waveform type).

Poly mode is essentially a simplification of synth mode in that it configures all six digitally controlled oscillators (DCOs) identically with no modulation, and patches them to directly to the output mixers. For complete customization of voices you punch-through to the synth mode functions.

Poly mode functions are accessed via the GinSingPoly class obtained through the base class getPoly() function.

state control

begin	*configure system for poly mode*
preview	*play poly mode demo*

envelope control

trigger	*trigger the amplitude envelope*
release	*release the amplitude envelope*
setEnvelope	*set the amplitude envelope parameters*

DCO parameters

setNote	*set the frequency as a musical note*
setWaveform	*set the waveform type*
setFreqDist	*set the frequency distortion level*
setDutyCycle	*set the duty cycle for pulse wave*

voice mode functions

Voice mode can be used to produce **artificial speech**. When voice mode is activated, all resources in the system are used internally for the purpose of generating human (or otherwise) voice synthesis. The interface provides the ability to string together basic speech fragments (called phonemes) into phrases as well as control the tonal qualities of the synthesis. Voice mode provides a very simple way to add artificial voice to your Arduino project.

Voice mode is in essence a set of built-in register configurations (one per allophone) that are loaded into the registers when an allophone is processed, and blended as the allophones change. Due to the complex nature of patching, mixing, and modulation to model each allophone there are limited but interesting uses of punch-through to synth mode.

Voice mode functions are accessed via the GinSingVoice class obtained through the base class getVoice() function.

state control

begin	*configure system for voice mode*
preview	*play voice mode demo*

voice mode control

speak	*speak a phrase*
getMillis	*compute phrase duration*

voice mode parameters

setNote	*set the speech frequency as a musical note*
setFrequency	*set the speech frequency for speech*
setBlendSpeed	*set the relative blending speed between phonemes*
setDelay	*set the relative delay between phonemes*

synth mode functions

Synth mode can be used to directly control all aspects of complex waveform synthesis on the Babblebot. The system is configured into 2 banks of 3 digitally controlled oscillators (DCOs) that are patched in such as way as to allow one DCO to modulate others (i.e. amplitude, frequency, pulse width), creating complex waveform patterns and tonal qualities. Synth operates in much the same way as analog synthesizers do, but does so with complete digital control.

Synth mode functions are accessed via the GinSingSynth class obtained through the base class getSynth() function.

state control

begin	*configure system for synth mode*
preview	*play synth mode demo*

bank & patch

selectBank	*select the current bank*
setPatch	*specify the DCO patch routing*

DCO parameters

setWaveform	*set the waveform type*
setWavemode	*set the waveform mode*
setNote	*set the frequency as a musical note*
enableOverflow	*enable or disable wavetable overflow handling*
setFrequency	*set the frequency in Hz*
setFreqVal	*set the frequency as an integer*
setAmplitude	*set the output amplitude*
setAmplitudeVal	*set the amplitude as an integer*
setFreqDist	*set the frequency distortion level*
setFreqDistVal	*set the frequency distortion as an integer*
setDutyCycle	*set the duty cycle for pulse wave*
setDutyCycleVal	*set the duty cycle for pulse wave as an integer*

targeting & ramping

enableFreqTarget	*enable or disable frequency ramp targeting*
setFreqTarget	*set the frequency ramp target and rate*
setFreqTargetVal	*set the frequency ramp and target as integers*
enableFreqRamp	*enable or disable frequency ramping*
setFreqRamp	*set the frequency ramp rate*
setFreqRampVal	*set the frequency ramp rate as an integer*
enableAmpTarget	*enable or disable amplitude ramp targeting*
setAmpTarget	*set the amplitude ramp target and rate*
setAmpTargetVal	*set the amplitude ramp target and rate as integers*
enableAmpRamp	*enable or disable amplitude ramping*
setAmpRamp	*set the amplitude ramp rate*
setAmpRampVal	*set the amplitude ramp rate as an integer*

envelope control

trigger	*trigger the amplitude envelope*
release	*release the amplitude envelope*
setEnvelope	*set the amplitude envelope parameters*
setEnvelopeVal	*set the amplitude envelope parameters as integers*

master functions

The master interface controls the **“global” aspects of the system**, such as overall output volume, timing functions, and command and control functions. The master functions are available via the getMaster() method in the base GinSing class an can be called at any time after the system has been initialized.

state control

enableUserOutput	*enable or disable the user output (Q)*

mixer control

setAmplitude	*set the bank mixer output amplitude*
setAmplitudeVal	*set the bank mixer amplitude as an integer*
setMasterAmplitude	*set the system output amplitude*
setMasterAmplitudeVal	*set the system amplitude as an integer*

targeting & ramping

enableAmpTarget	*enable or disable amplitude ramp targeting*
setAmpTarget	*set the amplitude ramp target and rate*
setAmpTargetVal	*set the amplitude ramp target and rate as integers*
enableAmpRamp	*enable or disable amplitude ramping*
setAmpRamp	*set the amplitude ramp rate*
setAmpRampVal	*set the amplitude ramp rate as an integer*

envelope control

trigger	*trigger the amplitude envelope*
release	*release the amplitude envelope*
setEnvelope	*set the amplitude envelope parameters*
setEnvelopeVal	*set the amplitude envelope parameters as integers*

examples overview

In the hopes of getting you up and running ASAP, we have created 5 specific source code examples that will guide you through the rest of the programming within the Arduino IDE itself. The idea here is that you can learn, copy, alter, and mix & match code from these examples in the environment that you code from. For this reason, we'll go through a general idea of these sample programs and let you experiment with working code in the IDE. The example programs can be found in your **Arduino/GinSing** folder, and are directly loadable in the Arduino IDE via the **File -> Sketchbook -> GinSing** menu option.

1.welcome	this program illustrates the basic concept of how to get the GinSing functionality into your own code. It does 4 iterations that go through the basic operating modes of the software in order (poly , preset , voice , synth) and run a built-in preview of each mode. of most importance is the understanding of how the C++ **class is created, what header file you need, and how to access the methods** (functions) in each mode.
2.presetmode	this program demonstrates the **basics of preset sound effects mode**. the chip contains 30+ built-in sound effects that are easy to trigger and fun to use. it illustrates how to access the preset class methods to cycle through all of the available preset sound effects and play them in order. It also demonstrates the concept of **load, trigger, and release methods** and utilizes only one of the two available banks for playing effects.
3.polymode	this program demonstrates the **basics of polyphonic mode**. In poly mode, the system is configured to create 6 identical voices that can be triggered simultaneously. in this example we create a simple note sequence and play using the **musical scale** in 3 part harmony. It demonstrates the basics of how to use GinSing as the basis for simple musical instruments.
4.voicemode	this program demonstrates the **basics of voice synthesis mode**. In voice mode, all resources in the chip are allocated to perform synthetic speech. this example contains several small routines that demonstrate **how to create simple phrases** and how to control pitch, inflection, and speed.
5. synthmode	this program demonstrates the **basics of waveform synthesis mode**. In synth mode, each of the 6 waveform generators can be patched in unique ways such as mixing and modulation to create complex waveforms. this example demonstrates some of the **basic techniques** used in waveform synthesis such as amplitude modulation, frequency modulation, frequency distortion, and more.

getting help

We hope that between this document, the source examples, and the reference manual you should be able to get all of the functionality you want from the GinSing library. However, you may come across some sort of weirdness or otherwise troubling behavior that is not explainable through documentation alone. If this happens, consider yourself among the ranks of a full-fledged programmer; welcome to the club!

To help you get through any problems, we have created a support forum on our website that you can use to post questions about troubles or issues you are having, or just to get more information from others that have gone down the path you are taking. We encourage you to check out the forum and participate in the discussion; you only need to register on the site to begin posting questions and getting answers. We welcome comments, suggestions, rants, raves and ideas about all things GinSing; and appreciate your input and letting us know what projects you are using GinSing for.

So please consider the support forum as your conduit to help you get through any troubles you have when using GinSing; we hope to hear from you soon!

http://www.ginsingsound.com/support