GinSing Software Reference Guide

version 4.0

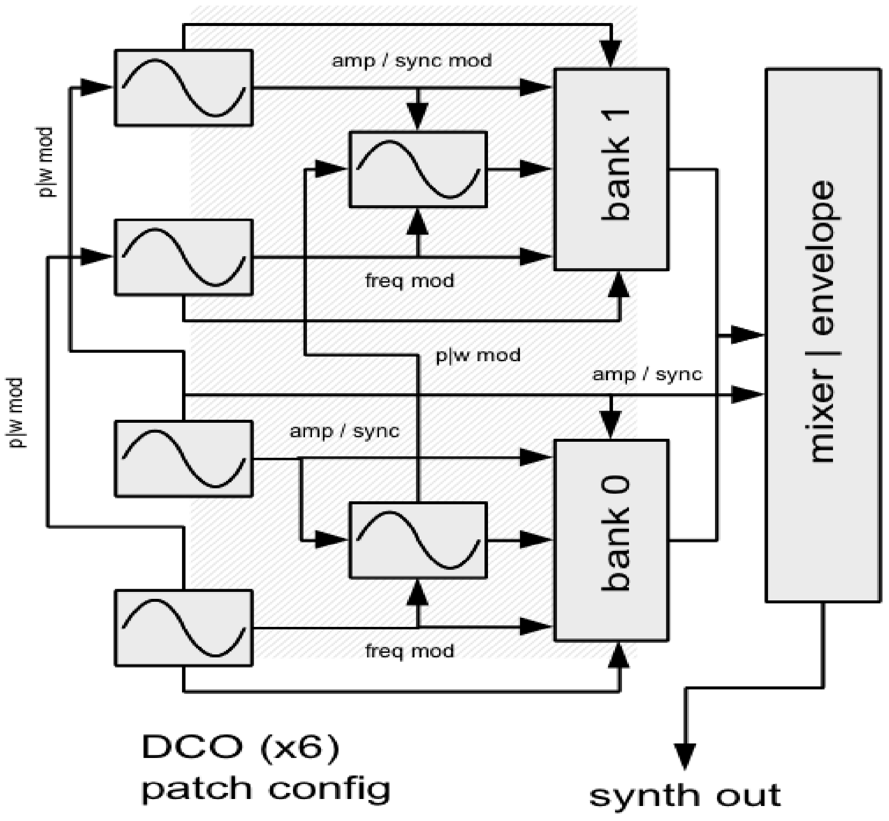


Table of Contents

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 **functional reference** of the interface from a programming perspective. You may find this document useful when you begin coding your applications on the Arduino. In addition to providing information on the functions and what they do, this document also contains an appendix that describe the constants and variables used throughout the library.

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.

starting up and shutting down

Once you have created the GinSing interface class (in the example above the interface class is called GS), you can **initialize the shield** by making one function call. After that, you can access all of the other functions based on what you want to do.

So to initialize the shield for use, you call the **begin()** function on the base class. This function (or method in C++ parlance) requires three arguments that correspond to the hardware configuration jumpers set up on the shield:

```
#define rcvPin  4       // pin used for receiving
#define sndPin  3       // pin used for transmitting
#define ovfPin  2       // pin used for overflow control

GS.begin( rcvPin , sndPin , ovfPin );   // start up GinSing
```

You can also shutdown the shield using **end()** if needed; which when called will mute the output of the board. You must again call begin() if you wish to use it again. Note that the GS class used in this code is your interface to all of the other functions available in the library. To make it easier you access groups of functions, you can get them from the GS class and refer to them on their own as illustrated below.

mode access

Once the system has been started up, you can access additional functionality based on the mode that you want to use. The modes are actually C++ classes themselves contained within the base class. For example, in order to use the voice mode (synthetic speech), we can get the voice mode functions from the base interface, and then call functions within it:

```
GinSingVoice *voice = GS.getVoice();  // get voice mode

voice->begin();                       // enter voice mode
voice->preview();                     // listen to a sample
```

Note that the **mode interfaces are pointers rather than static classes**; syntactically this means that you use an asterisk (*) to prefix the variable, and you use an arrow (→) instead of the dot (.) to access the functions within the class. The reason for using a class as a pointer is that you can throw away the interface when you are done using it so you don't need to keep around a global variable. This code for example could be wrapped into its own function. An alternative approach would be to avoid the local variable altogether if you wish, but does make the code more wordy:

```
GS.getVoice()->begin();             // enter voice mode
GS.getVoice()->preview();           // listen to a sample
```

Either way works so its your preference as to which way you want to implement the interface. Note that each mode also has a begin() method that you should call to s**et up the system to match the mode you want to use**. When you switch from one mode to another it is generally advisable to call begin() on the new mode.

synth mode punch-through

Because the operating modes are simply conceptual models, there is flexibility in how the models can operate together. On the actual hardware there are 144 status registers and 40 commands, and it is the setting of these registers and the sending of the commands that determines what sound is created, regardless of what software operating mode you are in.

Synth mode is the closest analogy to the hardware, and as such as the most functionality providing a near 1:1 relationship between the commands and registers and the functional interface. The **other operating modes are in effect simplifications of synth mode**, internally using synth mode to perform operations under its own function interface. Because of this, y**ou can always make synth mode calls in any operating mode** to extend functionality.

For example, if you are in poly mode, you can set the frequency using the synth.setFrequency() function; this function does not exist in the poly mode interface for simplicity, but you do have full control over the functionality of the chip in this way. Using synth mode functions while in another operating mode is called “punch-through” because you do not (or would wish) to actually switch to synth mode to make the changes; it is a way to extend the function interface in the other modes without duplicating the function set. Consider an example of using preset mode to load up a built-in effect; you can then use synth mode punch-through to dynamically modify the sound effect in ways that make the effect unique, but is still based on the original preset you load.

sample code

The GinSing software installation contains some sample source code programs that can help guide you through the coding process. In the Arduino IDE, you can compile and run these programs using the menu:

File → Sketchbook → GinSing

Before you write your first application using GinSing we recommend you take a quick run through the numbered programs under this menu list (in order) to firmly establish the concepts as well as the syntax presented here.

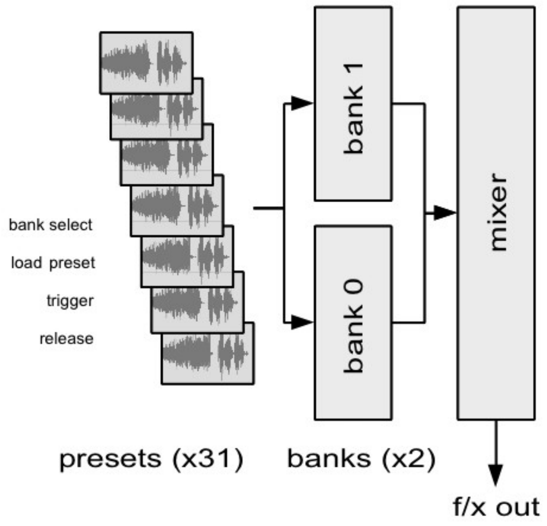
functional modes

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.

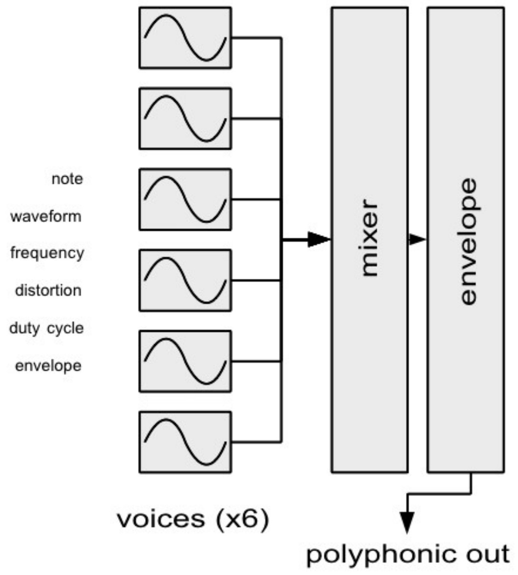
preset mode

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.



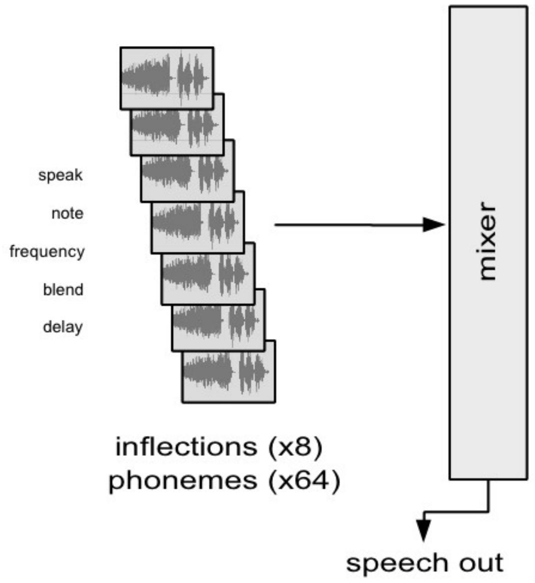
poly mode

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).



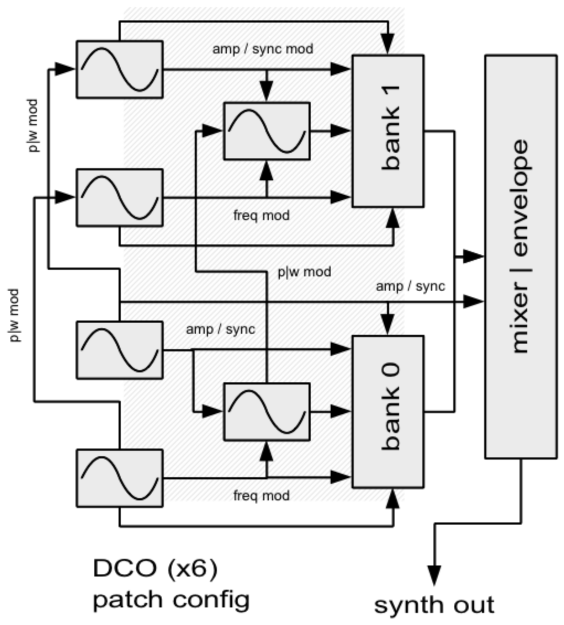
voice mode

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.



additional interfaces

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.

constants and definitions

For convenience, all of the **constants, variable types, and fixed argument parameters** used in the interface are contained in a single file called G**inSingDefs.h,** and are included in this document in **appendix A.** This file is a great resource for illustrating all of the functional abilities of the interface and the chip.

Although the library source files are split into each of the subclass interfaces (i.e. GinSingPreset.cpp / GinSingPreset.h), all the constants used by the interfaces are presented in this single file. Having this file open as a cut/paste reference will save valuable time manually typing in parameters throughout the interface.

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*

GinSing.begin

syntax

void begin (*int rcvPin* , *int sndPin* , *int ovfPin*)

description

Initializes the GinSing library, starts the communication between the Arduino and the GinSing shield, and configures the system into a default state. This function must be called before any other call can be made in the system, and only needs to be called once.

example

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

#define rcvPin  4       // pin used for receiving
#define sndPin  3       // pin used for transmitting
#define ovfPin  2       // pin used for overflow control

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

arguments

rcvPin

An integer value that matches the hardware jumper on the GinSing shield. This value can either be 4 (default), 12 (alternate), or -1 (no jumper) and designates which pin on the Arduino board will be allocated for receiving data from the shield. If the jumper is removed, the data receiving functions will not work, but it will free up both pins 4 and 12 for your application (or other shields).

sndPin

An integer value that matches the hardware jumper on the GinSing shield. This value can either be 3 (default), or 11 (alternate) and designates which pin on the Arduino board will be allocated for sending data to the shield. This jumper is required for the shield to function properly, and will utilize the hardware input designated by the jumper.

ovfPin

An integer value that matches the hardware jumper on the GinSing shield. This value can be either 2 (default), 10 (alternate), or -1 (no jumper) and designates which pin on the Arduino board will be allocated for flow control. If the jumper is not present then flow control will not be used and you must guard against overrunning the communications manually in your code. If the jumper is removed, flow control will not work, but it will free up both pins 2 and 10 for your application (or other shields).

GinSing.end

syntax

void end ()

description

Mutes the GinSing shield, disables communication with the Babblebot IC, and terminates the interface library. This function can be used in any shutdown code in your application to ensure the shield is in a known and quiet state. No function calls other than begin() should be made after this call.

example

```
GS.end();
```

GinSing.reset

syntax

void reset()

description

Sends a reset command to the Babblebot IC to reset to default powerup state. All registers will be set to their default settings and the system will be muted. This command should be followed by an initialization (i.e. begin()) function call based on the desired operating mode to ensure proper configuration after reset.

example

```
GS.reset();
```

GinSing.isReady

syntax

bool ready = isReady ()

description

Polls the current state of the communication interface to determine if the system is ready to accept commands. This function is used internally as a means of flow control to avoid overrunning the interface between the Arduino and the Babblebot IC.

The Babblebot communicates using a serial interface with a 4 byte buffer on the chip. When this function is called, the state of the buffer is checked; if the buffer is empty it returns true, otherwise it returns false. Therefore any command (4 bytes or less) can be stored in the buffer without delay if this function returns true. If this function returns false, then further commands sent will be delayed until the data pending in the buffer is processed by the Babblebot IC first.

This function has no effect if the flow control jumper is not placed on the GinSing shield. In this case it is possible to overrun the communication link without warning unless you make accommodations in your code, such as making commands on a predetermined time interval. The interface operates at 9600 baud, or approximately 1000 bytes per second; given that most commands are 4 bytes in length or less, this allows for approximately 250 commands per second. Additional processing required by the chip for some commands can reduce this effective rate drastically, such as speaking an phrase, as it requires more time to generate than to transmit.

example

```
while ( !GS.isReady() ) {

      Serial.println ( “GinSing is busy ...” );
      delay ( 100 );
}
```

returns

ready

A boolean value that is true if the communication link is ready to accept commands, or false if it is busy processing pending commands.

GinSing.getVersion

syntax

int version = getVersion ()

description

Gets the version number for the GinSing library currently in use by the Arduino IDE. This version number can be used for compatibility issues between different versions of the GinSing library. Note that this version number relates only to the software library and has no information about the version of the GinSing shield you may be using.

example

```
Serial.print ( “GinSing Version = “ );
Serial.println ( GS.getVersion() , DEC );
```

returns

version

An integer value that represents the version of the software library currently in use by the IDE.

GinSing.getPoly

syntax

GinSingPoly * polyClass = getPoly ()

description

Gets the poly mode class interface. The poly mode interface contains methods that configure and control the system as a polyphonic musical instrument. This interface pointer is only valid once the system has been initialized and will remain valid from then on.

To ensure proper working of poly mode, the poly mode function begin() should be called prior to making other function calls in this class.

example

```
GinSingPoly *poly = GS.getPoly();  // get poly mode

poly->begin();                     // enter poly mode
poly->preview();                   // listen to a sample
```

returns

polyClass

A pointer to the poly mode interface. The variable returned by this type is a GinSingPoly class pointer. The address of the interface will not change and can be referenced at any time once assigned.

GinSing.getVoice

syntax

GinSingVoice * voiceClass = getVoice ()

description

Gets the voice mode class interface. The voice mode interface contains methods that configure and control the system as a synthetic voice generator. This interface pointer is only valid once the system has been initialized and will remain valid from then on.

To ensure proper working of voice mode, the voice mode function begin() should be called prior to making other function calls in this class.

example

```
GinSingVoice *voice = GS.getVoice();  // get voice mode

voice->begin();                       // enter voice mode
voice->preview();                     // listen to a sample
```

returns

voiceClass

A pointer to the voice mode interface. The variable returned by this type is a GinSingVoice class pointer. The address of the interface will not change and can be referenced at any time once assigned.

GinSing.getPreset

syntax

GinSingPreset * presetClass = getPreset ()

description

Gets the preset mode class interface. The preset mode interface contains methods that configure and control the system to allow for the loading and playing of built-in sound effects. This interface pointer is only valid once the system has been initialized and will remain valid from then on.

To ensure proper working of preset mode, the voice mode function begin() should be called prior to making other function calls in this class.

example

```
GinSingPreset *preset = GS.getPreset();  // get preset mode

preset->begin();                         // enter preset mode
preset->preview();                       // listen to a sample
```

returns

presetClass

A pointer to the preset mode interface. The variable returned by this type is a GinSingPreset class pointer. The address of the interface will not change and can be referenced at any time once assigned.

GinSing.getSynth

syntax

GinSingSynth * synthClass = getSynth()

description

Gets the synth mode class interface. The synth mode interface contains methods that configure and control the system to allow for complex waveform synthesis. This interface pointer is only valid once the system has been initialized and will remain valid from then on.

To ensure proper working of synth mode, the synth mode function begin() should be called prior to making other function calls in this class.

example

```
GinSingSynth *synth = GS.getSynth();    // get synth mode

synth->begin();                         // enter synth mode
synth->preview();                       // listen to a sample
```

returns

synthClass

A pointer to the synth mode interface. The variable returned by this type is a GinSingSynth class pointer. The address of the interface will not change and can be referenced at any time once assigned.

GinSing.getMaster

syntax

GinSingMaster * masterClass = getMaster()

description

Gets the master class interface. The master interface contains methods that manipulate global variables within the system, such as output volume, timing parameters, etc. This interface pointer is only valid once the system has been initialized and will remain valid from then on.

example

```
GinSingMaster *master = GS.getMaster();  // get master functions

master->setMasterAmplitude( 0.0f );      // mute master volume
```

returns

masterClass

A pointer to the master class interface. The variable returned by this type is a GinSingMaster class pointer. The address of the interface will not change and can be referenced at any time once assigned.

GinSing.sendCommand

syntax

void sendCommand (GSCommand cmd , ubyte a1 , ubyte a2 , ...)

description

Sends a low level command to the Babblebot IC. Commands can be sent to the Babblebot IC to control the execution state of the chip, such as to load and trigger sounds, and so on. A complete description of the commands can be found in the Babblebot IC data-sheet and are outlined in appendix A. The library uses this command internally, so this function is only needed if you want complete control over the low level commands a registers on the chip.

example

```
// set master amplitude to 50% ( through register set )

GS.sendCommand ( WriteOneByte ,        // send command to write 1 byte
                 MasterAmplitude ,     // write to the master amp register
                 127 );                // write a value of 127 ( 50% )
```

arguments

cmd

The command to be executed by the Babblebot IC. The list of valid commands is enumerated as the GSCommand variable type in appendix A. Note that when using this function the low level command header is sent automatically and is not needed to communicate as per the data sheet specification.

a1 - a4

The unsigned byte data arguments associated with the specified command. Commands can have from zero to 4 arguments, and the number of arguments must match exactly the requirements for the command to work properly. A complete list and description of both commands and registers can be found in appendix A.

GinSing.writeRegister

syntax

void writeRegister (GSRegister regIdx , ubyte value , ubyte mask)

description

Sets the value of specific register on the Babblebot IC. The Babblebot IC has 144 registers that set operating parameters of the chip. A complete description of the registers can be found in the Babblebot IC data-sheet and outlined in appendix A. The higher level operating modes use this command internally, so this function is only needed for low level custom user control.

example

```
// set bank A amplitude register to zero
GS.writeRegister ( A_Amplitude , 0 );
```

arguments

regIdx

The target register to set. The registers are enumerated in appendix A as the variable type GSRegister.

value

The unsigned byte value to assign to the register. To support bit operations, only bits in the value that have a matching bit position in the mask will be changed. Bits that do not have a matching position in the mask will not be changed (don't care).

mask

A bit-mask specifying which bits in the value will be set. If a bit in the mask has a value of 1, the corresponding value for that bit will be set in the register. If a bit in the mask has a value of 0, then the value for that bit will not be written.

GinSing.readRegister

syntax

ubyte readRegister (GSRegister regIdx)

description

Reads the contents of a specified register on the Babblebot IC. The Babblebot IC has 144 registers that set operating parameters of the chip. A complete description of the registers can be found in the Babblebot IC data-sheet and outlined in appendix A. This function will not work properly if the GinSing shield is not using the receive jumper on the board.

example

```
// dump the contents of all the registers

for ( int regIdx = (int) A_MixControl_0; regIdx <= (int) Output_B3; regIdx++ )
{
      ubyte value = GS.readRegister ( (GSRegister) regIdx );
      Serial.println ( value , DEC );
}
```

arguments

regIdx

The target register to read. The registers are enumerated in appendix A as the variable type GSRegister.

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*

GinSingPreset.begin

syntax

void begin ()

description

Configures the system for use in preset mode. This function should be called when first using the preset mode functions or when switching from a different operating mode to ensure that the system is in a known state.

example

```
GinSingPreset *preset = GS.getPreset();  // get preset mode

preset->begin();                         // enter preset mode
preset->preview();                       // listen to a sample
```

GinSingPreset.preview

syntax

void preview ()

description

Triggers a brief demonstration of this operating mode. This function can be called at any time to verify that the system is working and is in the proper state for additional calls that you may make to this class. When called, this function sets the bank amplitudes to 50%, loads the Carney and TipToe effects into the banks, and triggers the effects for 3 seconds.

example

```
GinSingPreset *preset = GS.getPreset();  // get preset mode

preset->preview();                       // preview this mode
```

GinSingPreset.load

syntax

void load (ubyte bankIdx , GSPreset presetIdx)

description

Loads one of 32 built-in preset sound effect settings from the Babblebot IC into the registers associated with the specified bank. The preset can then be triggered to play the sound effect. Up to two sound effects (one for each bank) can be loaded at any given time. The preset effects can provide a solid basis from which custom modifications can be made with your own code.

example

```
GinSingPreset *preset = GS.getPreset();  // get preset mode

preset->load    ( 0 , Carney );          // load Carney preset on bank A
preset->trigger ( 0 );                   // trigger the preset
```

arguments

bankIdx

An integer value that specifies which bank to load the preset sound effect into. If a zero is specified, the preset is loaded into bank A, if a one is specified the preset is loaded into bank B.

presetIdx

The preset definition to load. Preset definitions are enumerated as the GSPreset variable type in appendix A.

GinSingPreset.trigger

syntax

void trigger (ubyte bankIdx)

description

Triggers the preset currently loaded preset on the specified bank. When triggered, the amplitude envelope (ADSR) specified for the preset begins its sequence. For built-in presets, this envelope is constant and will run continuously, but can be modified by you if you wish to control the volume over time (see synth.setEnvelope for details). If trigger() is called while the effect is playing its amplitude envelope will restart.

Example

```
// cycle through the effects and load them into the first bank

GinSingPreset *preset = GS.getPreset();

for ( int preIdx = (int) SpaceWarp; preIdx < (int) AmpMod; preIdx++ )
{
    preset->load    ( 0 , (GSPreset) presetIdx );   // load preset on bank 0
    preset->trigger ( 0 );                          // trigger the preset
    delay ( 1000 );                                 // listen to it
    preset->release ( 0 );                          // release the preset
}
```

arguments

bankIdx

An integer value that specifies which bank to trigger. If a zero is specified, the preset is on bank A is triggered, if a one is specified the preset on bank B is triggered. Both banks can be triggered simultaneously.

GinSingPreset.release

syntax

void release (ubyte bankIdx)

description

Releases the preset currently loaded preset on the specified bank. When released, the amplitude envelope (ADSR) specified for the preset enters its release stage. For built-in presets, this release is instantaneous, but can be modified by you if you wish to control the volume over time (see synth.setEnvelope() for details). If release() is called while the effect is not playing the function will have no effect.

example

```
// cycle through the effects and load them into the first bank

GinSingPreset *preset = GS.getPreset();

for ( int preIdx = (int) SpaceWarp; preIdx < (int) AmpMod; preIdx++ )
{
    preset->load    ( 0 , (GSPreset) presetIdx );   // load preset on bank 0
    preset->trigger ( 0 );                          // trigger the preset
    delay ( 1000 );                                 // listen to it
    preset->release ( 0 );                          // release the preset
}
```

arguments

bankIdx

An integer value that specifies which bank to release. If a zero is specified, the preset is on bank A is release, if a one is specified the preset on bank B is release.

GinSingPreset.setAmplitude

syntax

void setAmplitude (ubyte bankIdx , float amplitude)

description

Sets the output amplitude of the sound effect preset loaded on the specified bank. The amplitude can be controlled continuously, and serves as an alternative to using the trigger/release methodology on the effect. If the preset has triggered using the trigger() function, the volume will be overwritten by the amplitude envelope (ADSR) immediately after this function is called.

example

```
GinSingPreset *preset = GS.getPreset();  // get preset mode
preset->load         ( 0 , Carney );     // load Carney preset on bank A
preset->setAmplitude ( 1.0f );           // hear the effect at full volume
```

arguments

bankIdx

An integer value that specifies which bank to set the volume for. If a zero is specified, the volume is set on bank A, if a one is specified, the volume is set on bank B.

amplitude

A floating point value between 0.0 and 1.0 that sets the relative output volume of the effect. A value of 0.0 will mute the sound, whereas a value of 1.0 will set the output amplitude at full volume.

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*

GinSingPoly.begin

syntax

void begin ()

description

Configures the system for use in poly mode. This function should be called when first using the poly mode functions or when switching from a different operating mode to ensure that the system is in a known state.

example

```
GinSingPoly *poly = GS.getPoly();  // get poly mode

poly->begin();                     // enter poly mode
poly->preview();                   // listen to a sample
```

GinSingPoly.preview

syntax

void preview ()

description

Triggers a brief demonstration of this operating mode. This function can be called at any time to verify that the system is working and is in the proper state for additional calls that you may make to this class. When called, this function will play trigger each of the six voices in delayed sequence, then release all the voices simultaneously after 3 seconds. When called directly after begin() this will result in a playback of the A Major musical scale.

example

```
GinSingPoly *poly = GS.getPoly();  // get poly mode

poly->preview();                   // preview the basic features of this mode
```

GinSingPoly.trigger

syntax

void trigger (ubyte voiceIdx)

description

Triggers the specified voice. In poly mode there are six oscillators (or voices) that can play notes independently of each other, providing six note polyphony. This might be used for example for playing 2 part harmony with a 4 note base chord. When this function is called, the specified voice will begin executing its amplitude envelope (ADSR), and will sequence through to its sustain amplitude, where it will hold the amplitude until release() is called. If this function is called while the envelope is running it will start the envelope sequence over again.

example

```
// trigger the default voices in delayed sequence

GinSingPoly *poly = GS.getPoly();

for ( ubyte voiceIdx = 0; voiceIdx <= 5; voiceIdx++ )
{
      poly->trigger ( voiceIdx );
      delay ( 1000 );
}
```

arguments

voiceIdx

An integer value that specifies which voice (0-5) should be triggered. If the constant AllVoices is used (as defined in appendix A), all voices will be triggered.

GinSingPoly.release

syntax

void release (ubyte voiceIdx)

description

Releases the specified voice. In poly mode there are six oscillators (or voices) that can play notes independently of each other, providing six note polyphony. When this function is called, the specified voice will begin terminating its amplitude envelope (ADSR), and will sequence through to its release amplitude. If this function is called while the envelope is not active (no triggered), it will have no effect.

example

```
// release all the voices and wait for release

GinSingPoly *poly = GS.getPoly();

poly->release ( AllVoices );
delay ( 300 );
```

arguments

voiceIdx

An integer value that specifies which voice (0-5) should be release. If the constant AllVoices is used (as defined in appendix A), all voices will be released.

GinSingPoly.setNote

syntax

void setNote (ubyte voiceIdx , GSNote noteSel)

description

Sets the frequency of the voice based on a musical note. The Babblebot IC has a built in note-to-frequency table that spans an eight octave tempered musical scale range from C0 (16.352 Hz) to B7 (3,951.067 Hz). Note that musical notes below C0 are subsonic and will not be heard by most humans. When this function is called, the output frequency of the specified voice will be set to the corresponding frequency for the selected musical note.

example

```
// play concert A 440

GinSingPoly *poly = GS.getPoly();          // get poly mode

poly->setNote ( 0 , A_4  );                // set voice 0 to A440
poly->trigger ( 0 );                       // trigger envelope
```

arguments

voiceIdx

An integer value that specifies which voice (0-5) to set the note for. If the constant AllVoices is used (as defined in appendix A), all voices will be set to the same note.

noteSel

The musical note to set for the voice. The note is enumerated as a the variable type GSNote in the file appendix A, and is based on a concert A 440 (A_4) tempered musical scale.

GinSingPoly.setWaveform

syntax

void setWaveform (ubyte voiceIdx , GSWaveType waveSel)

description

Sets the waveform type for the specified voice. Initially all voices are set to the same waveform to make them sound tonally identical, but you can change any or all of the voices at any time using this function.

example

```
// change all voices to use triangle waves

GinSingPoly *poly = GS.getPoly();            // get poly mode

poly->setWaveform ( AllVoices , TRIANGLE );  // set all voices to triangle
```

arguments

voiceIdx

An integer value that specifies which voice (0-5) to set the waveform for. If the constant AllVoices is used (as defined in appendix A), all voices will be set to the same waveform.

waveSel

The waveform type for the selected voice. The waveform type is enumerated as the variable type GSWaveType in the file appendix A, and can be one of SINE, TRIANGLE, SAWTOOTH, RAMP, PULSE, or NOISE.

GinSingPoly.setFreqDist

syntax

void setFreqDist (ubyte voiceIdx , float distLevel)

description

Sets the relative frequency distortion factor for the specified voice. Frequency distortion can be used to change the timbre (tonal quality) of a voice by varying the output frequency slightly at a very fast rate. The effect adds a whistling or windy quality to the sound.

example

```
// set lead voice (0) to whistle

GinSingPoly *poly = GS.getPoly();            // get poly mode

poly->setFreqDist ( 0 , 0.2f );             // set 20% whistle effect
```

arguments

voiceIdx

An integer value that specifies which voice (0-5) to set the waveform for. If the constant AllVoices is used (as defined in appendix A), all voices will be set to the same waveform.

distLevel

A floating point value that represents the amount of relative frequency distortion to apply to the voice. A value of 0.0 will eliminate the effect, whereas a value of 1.0 will add the maximum effect.

GinSingPoly.setDutyCycle

syntax

void setDutyCycle (ubyte voiceIdx , float dutyCycle)

description

Sets the on/off pulse width ratio for the current voice. This function only has an effect if the currently selected waveform is PULSE. Duty cycle can change the timbre (tonal quality) of a voice by accentuating various harmonics in the waveform based on the ratio of on to off time in waveform cycle.

example

```
// set lead voice (0) to a square wave

GinSingPoly *poly = GS.getPoly();            // get poly mode

poly->setWaveform  ( 0 , PULSE );           // set voice to pulse wave
poly->setDutyCycle ( 0 , 0.0f );            // set the duty cycle to 50%
```

arguments

voiceIdx

An integer value that specifies which voice (0-5) to set the duty cycle for. If the constant AllVoices is used (as defined in appendix A), all voices will be set to the same duty cycle.

dutyCycle

A floating point value that specifies the relative time between the waveform output going from maximum to minimum in each cycle. A value of 0.0 will produce a square wave. Negative values (down to -1.0) will decrease the ratio of maximum to minimum time, whereas positive values (up to 1.0) will increase the ratio of maximum to minimum time.

GinSingPoly.setEnvelope

syntax

```
void setEnvelope   ( ubyte voiceIdx,
                     GSAttackDur attackDur , float attackAmp,
                     GSDecRelDur decayDur , float decayAmp,
                     GSDecRelDur releaseDur , float releaseAmp )
```

description

Sets the amplitude envelope parameters for the specified voice. Each voice has a table that determines how the output amplitude varies over time once the voice is triggered. This table has four sequential stages (Attack, Decay, Sustain, Release) known as an ADSR envelope, which define amplitude ramps within a time window. The sustain stage is unique in that its time window is variable based on the time between the completion of the decay stage until release() is called, and its amplitude is fixed at the decay stage level; for this reason it does not need to be specified.

example

```
GinSingPoly *poly = GS.getPoly();                // get poly mode

setEnvelope   ( AllVoices , AT_24MS ,  0.9f ,    // 24 ms attack to 90% vol
                            DR_2MS , 0.5f ,      // 2 ms decay to 50% vol
                            DR_575MS , 0.0f );   // .5 second release to 0%
```

arguments

voiceIdx

An integer value that specifies which voice (0-5) to set the duty cycle for. If the constant AllVoices is used (as defined in appendix A), all voices will be set to the same duty cycle.

attackDur | attackAmp

Attack stage settings. Duration time specified as a variable type GSAttackDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

decayDur | decayAmp

Decay stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

releaseDur | releaseAmp

Release stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

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*

GinSingVoice.begin

syntax

void begin ()

description

Configures the system for use in voice mode. This function should be called when first using the voice mode functions or when switching from a different operating mode to ensure that the system is in a known state.

example

```
GinSingVoice *voice = GS.getVoice();  // get voice mode
voice->begin();                       // enter voice mode
voice->preview();                     // listen to a sample
```

GinSingVoice.preview

syntax

void preview ()

description

Triggers a brief demonstration of this operating mode. This function ca be called at any time to verify that the system is working and is in the proper state for additional calls tat you may make to this class. When called, this function will speak the phrase “hello world” in the default pitch and speed.

example

```
GinSingVoice *voice = GS.getVoice();  // get voice mode
voice->begin();                       // enter voice mode
voice->preview();                     // listen to a sample
```

GinSingVoice.speak

syntax

void speak (GSAllophone * phrase)

description

Speaks the given phrase using the current voice settings. The phrase consists of an arbitrary length array of allophones that is terminated with the _ENDPRHASE token. Synthetic speech is produced by stringing together a sequence of short vocalizations, with optional inflection and pauses.
When called, this function will sequentially send each allophone to the Babblebot IC for processing. Depending on the buffer state of the communication interface and the amount of time to process (speak) the allophone, this function may block. It is generally recommended to use short phrases when blocking may be an issue in your application.

Example

```
// speak the phrase “I am GinSing”

GinSingVoice *voice = GS.getVoice();

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

voice->speak ( msg );
```

arguments

phrase

An array of allophones to speak, terminated by the _ENDPHRASE token. The allophones are enumerated in the variable type GSAllophone in appendix A.

Certain allophones are silent and are control commands that temporarily modify the speed, volume, and pitch of the single allophone that follows it. These are _SPEEDUP, _SPEEDDN , _VOLUP, _VOLDN, _PITCHUP, _PITCHDN, _BENDUP, and _BENDDN.

Certain allophones are for controlling timing for pauses and fixed delays. These are _PA0, _PA1, _PA2, _FD0, _FD1, and _FD2. Refer to appendix A for details on their meaning.

GinSingVoice.getMillis

syntax

int speakTime = getMillis (GSAllophone * phrase)

description

Computes the approximate amount of time that the given phrase will take to complete on the GinSing shield. This function can be useful to approximate how long the Arduino will be blocked waiting for speech to complete in advance. The time is approximate because it does not include time to transmit the phrase to the Babblebot IC and any other pending operations that may be underway when the phrase is spoken.

example

```
// say hello and wait for completion

GinSingVoice *voice = GS.getVoice();

GSAllophone  hello[] = { _HE , _E , _LO , _OE , _PA1 , _ENDPHRASE };

voice-> speak ( hello );
delay ( voice->getMillis( hello ) );
```

arguments

phrase

An array of allophones to compute speech time for, terminated by the _ENDPHRASE token. The allophones are enumerated in the variable type GSAllophone in appendix A.

returns

speakTime

The approximate amount of time (in milliseconds) that the phrase will take to speak (and potentially block) on the GinSing shield.

GinSingVoice.setNote

syntax

void setNote (GSNote noteSel)

description

Sets the speaking frequency for speech synthesis based on a musical note. The Babblebot IC has a built in note-to-frequency table that spans an eight octave tempered musical scale range from C0 (16.352 Hz) to B7 (3,951.067 Hz). Note that musical notes below E0 are subsonic and will not be heard by most humans. When this function is called, the speaking frequency will be set based on the selected musical note.

example

```
// sing a simple song

GSNote notes[10] = { C_4, E_4, D_4, F_4, E_4, G_4, E_4, F_4, D_4, E_4  };

GSAllophone phrase[] = { _LE , _AA , _ENDPHRASE };      // phrase to sing

GinSingVoice *voice = GS.getVoice();                    // get voice mode

for ( int noteCnt = 0; noteCnt < 10; noteCnt++ )        // run through notes
{
      voice->setNote ( notes [ noteCnt ] );             // set the musical note
      voice->speak   ( phrase );                        // sing the phrase

   delay ( 500 );                                       // wait 1/2 second
}
```

arguments

noteSel

The musical note to set for the voice. The note is enumerated as a the variable type GSNote in the file appendix A, and is based on a concert A 440 (A_4) tempered musical scale.

GinSingVoice.setFrequency

syntax

void setFrequency (float freqHz)

description

Sets the frequency for speech synthesis in Hertz (cycles per second). The speaking frequency can be set between 0.0009 Hz and 7,812.50 Hz, although frequencies less than 20 Hz are typically too low to be detected by normal humans.

Example

```
// set the voice to a low frequency

GinSingVoice *voice = GS.getVoice();              // get voice mode

GSAllophone phrase[] = { _GO, _OO, _OO, _DE, _PA2, // “goodbye”
                         _BE, _IE, _ENDPHRASE };

voice->setFrequency ( 100.0f );                   // set freq to 100 Hz
voice->speak( phrase );                           // speak the phrase
```

arguments

freqHz

An floating point value that represents the speaking frequency in Hz.

GinSingVoice.setBlendSpeed

syntax

void setBlendSpeed (float relSpeed)

description

Sets the relative allophone blending speed for speech synthesis. The Babblebot IC speaks phrases based on speech fragments (phonemes). To make the speech smooth, the phoneme synthesis parameters are blended over time to create a continuous transition between phonemes. This function can be used to specify how much blending is performed on the allophones.

example

```
// slur the phonemes

GinSingVoice *voice = GS.getVoice();              // get voice mode

GSAllophone phrase[] = {_HE, _E, _LO, _PA1, _ENDPHRASE };

voice->setBlendSpeed ( 0.1f );                   // set blend speed to 10%
voice->speak( phrase );                          // speak the phrase
```

arguments

relSpeed

An floating point value that represents the relative blending speed between allophones. A value of zero will disable allophone blending, resulting in immediate transitions between phones in the spoken phrase. A value of 1.0 will result in continuous blending and highly slurred speech.

GinSingVoice.setDelay

syntax

void setDelay (float relDelay)

description

Sets the relative delay between allophone transitions. The delay can be used to speed up or slow down the rate at which allophones are spoken. This value in combination with blend speed can add unique character to the voice synthesis.

example

```
// slow down speech

GinSingVoice *voice = GS.getVoice();              // get voice mode

GSAllophone phrase[] = { _HE, _E, _LO, _OE, _PA1,          // “hello”
                         _W,_ER, _LE, _ED, _ENDPHRASE };   // “world”

voice->setDelay ( 0.1f );                        // set delay speed to 10%
voice->speak( phrase );                          // speak the phrase
```

arguments

relDelay

A floating point value that specifies the relative delay in sequencing allophones. A value of 0.0 will slow speech down to a single allophone, whereas a value of 1.0 will transition through the allophones one after another immediately.

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*

GinSingSynth.begin

syntax

void begin ()

description

Configures the system for use in synth mode. This function can be called to establish a known state for using functions in the class. When using synth mode for punch-through functions (calling while in a different mode) it should not be called to avoid changing the state of registers.

example

```
GinSingSynth *synth = GS.getSynth();    // get synth mode

synth->begin();                         // enter synth mode
synth->preview();                       // listen to a sample
```

GinSingSynth.preview

syntax

void preview ()

description

Triggers a brief demonstration of this operating mode. This function can be called at any time to verify that the system is working and is in the proper state for additional calls that you make to this class. When called, this function will demonstrate a simple modulation example for approximately six seconds.

example

```
GinSingSynth *synth = GS.getSynth();    // get synth mode
synth->preview();                       // listen to a sample
```

GinSingSynth.selectBank

syntax

void selectBank (GSSynthBank bankSel)

description

Selects the current bank to be targeted for DCO functions. To address a DCO, both the bank index and the DCO index are required. To simplify this, you can call this function to specify the bank once, which will target other DCO functions to target the bank without having to explicitly pass it each time.

example

```
GinSingSynth *synth = GS.getSynth();        // get synth mode

synth->selectBank   ( BANK_A );             // select bank A
synth->setPatch     ( OSC_1_TO_MIXER );     // patch DCO A1 to mixer
synth->setFrequency ( OSC_1 , 1000.0f );    // set the oDCO A1 to 1 kHz
```

arguments

bankSel

Specifies the bank will be used for as a target for DCO functions. The variable type GSSynyhBank can be either BANK_A or BANK_B.

GinSingSynth.selectPatch

syntax

void setPatch (uint patchSel)

description

Sets the DCO patch configuration for the currently selected bank. A patch defines a routing map among the DCOs that determines how they are connected to each other and the output mixer. For example, you can patch DCO 3 to frequency modulate DCO 1 with its output to produce a complex waveform on the current bank. The patch configuration argument is a bit-mask, which allows for multiple routing paths to be specified completely for the bank using the bitwise or operator.

example

```
// frequency sweep a base tone

GinSingSynth *s = GS.getSynth();            // get synth mode

s->selectBank   ( BANK_A );                 // select bank A

s->setPatch     (   OSC_1_TO_MIXER          // patch DCO 1 to mixer
                  | OSC_3_FRQMOD_OSC_1 );   // patch DCO 3 to FM DCO 1

s->setFrequency ( OSC_1 , 100.0f );         // set the base tone frequency
s->setFrequency ( OSC_3 , 1.0f );           // set the modulation frequency
```

arguments

patchSel

An unsigned integer (16 bit mask) created from the enumerated variable type GSSynthPatch. The patch options are described in detail in appendix A. The enumeration is such that the bits in the configuration mask can be combined using the bitwise or mechanism (|). Note that each bank has its own patch configuration and this function targets the currently selected bank (via selectBank).

GinSingSynth.setWaveform

syntax

void setWaveform (GSSynthOsc dcoSel , GSWaveType waveSel)

description

Sets the waveform type for the specified DCO on the currently selected bank.

example

```
GinSingSynth *s = GS.getSynth();         // get synth mode

s->selectBank   ( BANK_A );              // select bank A
s->setWaveform  ( OSC_ALL , SINE );      // set all DCOs on bank A to sine
```

arguments

dcoSel

The DCO to change the waveform for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

waveSel

The waveform type for the selected voice. The waveform type is enumerated as the variable type GSWaveType in the file appendix A, and can be one of SINE, TRIANGLE, SAWTOOTH, RAMP, PULSE, or NOISE.

GinSingSynth.setWavemode

syntax

void setWavemode (GSSynthOsc dcoSel , GSWaveMode modeSel)

description

Sets the waveform mode (bias) for the DCO on the currently selected bank. By default, DCOs have output values that are both positive and negative, providing a symmetrical output. However, for modulation purposes it can be useful to offset, or bias the output so that the range of values are only positive. For example, when using one DCO to amplitude modulate another with a square wave, a symmetrical output of the modulator would only cause a phase change rather and turning the amplitude on and off as a biased wave would.

example

```
// generate a 1 kHz on/off tone

GinSingSynth *s = GS.getSynth();           // get synth mode

s->selectBank   ( BANK_A );                 // select bank A functions

s->setPatch     (   OSC_1_TO_MIXER         // patch DCO 1 to mixer
                  | OSC_2_AMPMOD_OSC_1 );  // set DCO 2 to AM DCO 1

s->setFrequency ( OSC_1 , 1000.0f );       // set the tone at 1 kHz
s->setFrequency ( OSC_2 , 1.0f );          // turn on/off at 1 Hz
s->setWaveform  ( OSC_2 , PULSE );         // square wave = on/off

s->setWavemode  ( OSC_2 , POSITIVE );      // bias waveform zero to amp
```

arguments

dcoSel

The DCO to change the waveform mode for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

modeSel

The type of waveform bias to set. Waveform bias is enumerated as the variable type GSWaveMode, and can be one of SYMMETRICAL or POSITIVE. Note that if the amplitude for the DCO is negative, the bias will switch from positive to negative.

GinSingSynth.setNote

syntax

void setNote (GSSynthOsc dcoSel , GSNote noteSel)

description

Sets the frequency of the DCO based on a musical note. The Babblebot IC has a built in note-to-frequency table that spans an eight octave tempered musical scale range from C0 (16.352 Hz) to B7 (3,951.067 Hz). Note that musical notes below E0 are subsonic and will not be heard by most humans. When this function is called, the output frequency of the specified voice will be set to the corresponding frequency for the selected musical note.

example

```
GinSingSynth *s = GS.getSynth();         // get synth mode

s->setNote ( OSC_1 , A_4  );             // set DCO 1 to A440
s->trigger ( OSC_1 );                    // trigger envelope
```

arguments

dcoSel

The DCO to set the note for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

noteSel

The musical note to set for the voice. The note is enumerated as a the variable type GSNote in the file appendix A, and is based on a concert A 440 (A_4) tempered musical scale.

GinSingSynth.enableOverflow

syntax

void enableOverflow (GSSynthOsc dcoSel , bool enableOVF)

description

Enables or disables overflow processing on wavetable cycling. The Babblebot IC uses a 24-bit number to keep track of the current position in the wave table. On each sample (every 65.25 microseconds), the value in frequency register is added to this position. As the position runs off the end of the table, it can either truncate (with overflow enabled) or wrap (overflow disabled).
When overflow is disabled, the frequency is precise but can produce sampling artifacts as the table wraps at different wave positions on each cycle. When overflow is enabled, the frequency is quantized, but the output has no sampling artifacts as the table position is reset to the first sample on each cycle.

Example

```
// compare overflow methods

GinSingSynth *s = GS.getSynth();           // get synth mode

s->selectBank   ( BANK_A );                 // select bank A functions

s->setPatch     ( OSC_1_TO_MIXER  );       // send DCO 1 to mixer
s->setFrequency ( OSC_1 , 1000.0f );       // set the tone at 1 kHz
s->setAmplitude ( OSC_1 , 1.0f );          // set amplitude to 100%

s->enableOverflow ( OSC_1 , false );       // accurate frequency
delay ( 5000 );                            // listen for 5 seconds

s->enableOverflow ( OSC_1 , true );        // eliminate sampling artifacts
delay ( 5000 );                            // listen for 5 seconds
```

arguments

dcoSel

The DCO to set the wavetable overflow mode for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

enableOVF

A boolean value to enable or disable overflow truncation. When the value is true, aliasing artifact removal algorithm is enabled; when the value is false frequency accuracy algorithm is enabled.

GinSingSynth.setFrequency

syntax

void setFrequency (GSSynthOsc dcoSel , float freqHz)

description

Sets the frequency for the DCO in Hertz (cycles per second). The frequency can be set between 0.0009 Hz (1073 seconds) and 7,812.50 Hz (128 microseconds). Frequencies less than 20 Hz are typically too low to be detected by normal humans and can used for modulation effects.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch     ( OSC_1_TO_MIXER );        // patch DCO 1 to mixer
s->setFrequency ( OSC_1 , 100.0f );        // set frequency to 100 Hz
s->setAmplitude ( OSC_1 , 1.0f );          // set amplitude to full volume
```

arguments

dcoSel

The DCO to set the frequency for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

freqHz

A floating point value that specifies the DCO output frequency in Hz.

GinSingSynth.setFreqVal

syntax

void setFreqVal (GSSynthOsc dcoSel , ulong freqVal)

description

Sets the DCO frequency as an unsigned long integer. The Babblebot IC internally uses a 24-bit frequency counter, so using this function will result in the most accurate specification of frequency available and avoids the floating point conversion performed in setFrequency(). Note that to obtain a precise frequency on the DCO the overflow function should be disabled. To convert from frequency to frequency value, you can use this equation:

freqVal = (freqHz) * 1073.742487

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->enableOverflow ( OSC_1 , false );       // enable accurate frequency
s->setFreqVal     ( OSC_1 , 107374 );      // set freq precisely to 100 Hz
```

arguments

dcoSel

The DCO to set the frequency for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

freqVal

An unsigned long integer that represents the output frequency. A value of zero will disable the DCO. A value value of 1 will set the frequency at its minimum of 0.0009 Hz (1073 seconds), whereas a value of 8,388,613 will set the frequency at its maximum of 7,812.5 Hz (128 microseconds).

GinSingSynth.setAmplitude

syntax

void setAmplitude (GSSynthOsc dcoSel , float amplitude)

description

Sets the relative output amplitude of the DCO. Amplitude specifies both volume and phase as a single value because it can be either positive or negative.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setAmplitude ( OSC_1 , -0.5f );         // 50% volume inverted phase
```

arguments

dcoSel

The DCO to set the amplitude for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

amplitude

A floating point value that represents the relative output amplitude of the DCO. The amplitude value can range from -1.0 (full volume inverted phased) to 1.0 (full volume non-inverted phase), with ah value of 0.0 muting the output.

GinSingSynth.setAmplitudeVal

syntax

void setAmplitudeVal (GSSynthOsc dcoSel , sbyte ampVal)

description

Sets the output amplitude of the DCO as a signed byte. The Babblebot IC internally uses a signed byte to represent volume, so using this function will avoid the floating point conversion performed in setAmplitude().

example

```
GinSingSynth *s = GS.getSynth();            // get synth mode

s->setAmplitudeVal ( OSC_1 , -64 );         // 50% volume inverted phase
```

arguments

dcoSel

The DCO to set the amplitude for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

ampVal

A signed byte that represents the output amplitude for the DCO. The value can range from -127 (full volume inverted phase) to 127 (full volume non-inverted phase), which a value of 0 muting the output.

GinSingSynth.setFreqDist

syntax

void setFreqDist (GSSynthOsc dcoSel , float distLevel)

description

Sets the relative value of frequency distortion applied to the DCO. Frequency distortion can be used to change the timbre (tonal quality) of a DCO by randomly shifting the wavetable step interval on each sample (every 65.25 microseconds). The effect adds a whistling or windy quality to the sound.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setFreqDist ( OSC_1 , 0.2f );           // add a 20% whistle effect
```

arguments

dcoSel

The DCO to set the frequency distortion level for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

distLevel

A floating point value that represents the amount of relative frequency distortion to apply to the DCO. A value of 0.0 will eliminate the effect, whereas a value of 1.0 will add the maximum effect.

GinSingSynth.setFreqDistVal

syntax

void setFreqDistVal (GSSynthOsc dcoSel , ubyte distVal)

description

Sets the relative value of frequency distortion applied to the DCO as an unsigned byte. The Babblebot IC internally uses an unsigned byte to represent frequency distortion, so this function avoids the floating point conversion performed in setFreqDist(). Frequency distortion can be used to change the timbre (tonal quality) of a DCO by randomly shifting the wavetable step interval on each sample (every 65.25 microseconds). The effect adds a whistling or windy quality to the sound.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setFreqDistVal ( OSC_1 , -64 );         // add a 50% whistle effect
```

arguments

dcoSel

The DCO to set the frequency distortion for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

distVal

An unsigned byte value that represents the amount of relative frequency distortion to apply to the DCO. A value of 0 will eliminate the effect, whereas a value of 127 will add the maximum effect.

GinSingSynth.setDutyCycle

syntax

void setDutyCycle (GSSynthOsc dcoSel , float dutyCycle)

description

Sets the symmetry of the DCO pulse wave output. This function only has an effect when the selected waveform is PULSE. A pulse wave is a two stage output that instantly switches from maximum to minimum once per cycle. The duty cycle specifies where in the cycle the switch occurs. Changing the duty cycle can vary the frequency harmonics in the output as well as provide timing variation when the DCO is used as a modulator.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setDutyCycle ( OSC_1 , -0.5 );          // 25% on , 75% off
```

arguments

dcoSel

The DCO to set the duty cycle for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

dutyCycle

A floating point value that represents the relative position of where the pulse changes phase in the cycle. The duty cycle value can vary from -1.0 (all negative phase) to 1.0 (all positive phase). A value of 0.0 specifies the center of the waveform cycle (square wave).

GinSingSynth.setDutyCycleVal

syntax

void setDutyCycleVal (GSSynthOsc dcoSel , sbyte dcVal)

description

Sets the symmetry of the DCO pulse wave output. This function only has an effect when the selected waveform is PULSE. The Babblebot IC internally uses a signed byte to represent duty cycle, so using this function will avoid the floating point conversion performed in setDutyCycle().
A pulse wave is a two stage output that instantly switches from maximum to minimum once per cycle. The duty cycle specifies where in the cycle the switch occurs. Changing the duty cycle can vary the frequency harmonics in the output as well as provide timing variation when the DCO is used as a modulator.

example

```
GinSingSynth *s = GS.getSynth();          // get synth mode

s->setDutyCycle ( OSC_1 , 0 );            // set up a square wave
```

arguments

dcoSel

The DCO to set the duty cycle for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

dutyCycle

A signed byte value that represents the relative position of where the pulse changes phase in the cycle. The duty cycle value can vary from -127(all negative phase) to 127 (all positive phase). A value of 0 specifies the center of the waveform cycle (square wave).

GinSingSynth.enableFreqTarget

syntax

void enableFreqTarget (GSSynthOsc dcoSel , bool enable)

description

Enables or disables frequency targeting on the DCO. For frequency targeting to function, frequency ramping must also be enabled (enableFreqRamp()).
Frequency targeting allows the DCO to smoothly ramp from its current frequency to a specified target frequency at a specified rate. This can be useful for smooth frequency changes in the DCO, such as portamento.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );           // send DCO 1 to mixer

s->setFrequency    ( OSC_1 , 100.0f );     // set initial frequency 100 Hz
s->setFreqTarget   ( OSC_1 , 1000.0f ,     // set frequency target 1 kHz
                             0.008f );     // at a slow rate

s->enableFreqRamp  ( OSC_1 , true );       // enable frequency ramp

s->enableFreqTarget( OSC_1 , enable );     // enable frequency target
```

arguments

dcoSel

The DCO to enable or disable frequency targeting for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

enable

A boolean value that enables or disables frequency targeting.

GinSingSynth.setFreqTarget

syntax

void setFreqTarget (GSSynthOsc dcoSel , float freqHz , float relRate)

description

Sets the parameters for frequency targeting. This function will only have an effect if frequency targeting is enabled (enableFreqTarget). Frequency targeting allows the DCO to smoothly ramp from its current frequency to the specified target frequency at the specified relative rate. This can be useful for smooth frequency changes in the DCO, and can be called continuously to provide pitch bend functionality.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );           // send DCO 1 to mixer

s->setFrequency    ( OSC_1 , 100.0f );     // set initial frequency 100 Hz
s->setFreqTarget   ( OSC_1 , 1000.0f ,     // set frequency target 1 kHz
                             0.008f );     // at a slow rate

s->enableFreqRamp  ( OSC_1 , true );       // enable frequency ramp

s->enableFreqTarget( OSC_1 , enable );     // enable frequency target
```

arguments

dcoSel

The DCO to set the frequency targeting parameters for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

freqHz

A floating point value that specifies the target frequency in Hz. The frequency can be set between 0.0009 Hz and 7,812.50 Hz, although frequencies less than 20 Hz are typically too low to be detected by normal humans.

relRate

A floating point value that specifies he relative rate at which the DCO ramps from the current frequency to the target frequency. Values can range from 0.0 (disables targeting) to 1.0 (instantaneous targeting).

GinSingSynth.setFreqTargetVal

syntax

void setFreqTargetVal (GSSynthOsc dcoSel , uint freqVal , uint rateVal)

description

Sets the parameters for frequency targeting as unsigned integers. The Babblebot IC internally uses unsigned integers for frequency targeting so using this function will avoid the floating point conversion performed in setFreqTarget(). This function will only have an effect if frequency targeting is enabled (enableFreqTarget).
Frequency targeting allows the DCO to smoothly ramp from its current frequency to the specified target frequency at the specified relative rate. This can be useful for smooth frequency changes in the DCO, and can be called continuously to provide pitch bend functionality.

example

```
GinSingSynth *s = GS.getSynth();          // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );          // send DCO 1 to mixer

s->setFrequency    ( OSC_1 , 100.0f );    // set initial frequency 100 Hz
s->setFreqTarget   ( OSC_1 , 1000.0f ,    // set frequency target 1 kHz
                             0.008f );    // at a slow rate

s->enableFreqRamp  ( OSC_1 , true );      // enable frequency ramp

s->enableFreqTarget( OSC_1 , enable );    // enable frequency target
```

arguments

dcoSel

The DCO to set the frequency targeting parameters for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

freqVal

An unsigned two byte integer that represents the target frequency. The frequency can be converted using the equation: freqVal = (freqHz) * 4.194304547. Values can range from 1 (4.2 Hz) to 32,768 (7,812.50 Hz).

rateVal

The relative rate at which the DCO ramps from the current frequency to the target frequency. Values can range from 0.0 (disables targeting) to 1.0 (instantaneous targeting). Note that both targeting and ramping share this rate variable.

GinSingSynth.enableFreqRamp

syntax

void enableFreqRamp (GSSynthOsc dcoSel , bool enable)

description

Enables or disables frequency ramping on the DCO. Frequency ramping allows the DCO to smoothly ramp from minimum frequency to maximum frequency at a specified rate. When frequency targeting is enabled as well it will ramp to the target frequency, otherwise it will free run until disabled.

example

```
GinSingSynth *s = GS.getSynth();            // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );            // send DCO 1 to mixer

s->setFrequency    ( OSC_1 , 100.0f );      // set initial frequency 100 Hz
s->setFreqRamp     ( OSC_1 , 0.01f );       // ramp at a slow rate

s->enableFreqRamp  ( OSC_1 , true );        // enable frequency ramp
```

arguments

dcoSel

The DCO to enable or disable frequency ramping for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

enable

A boolean value that enables or disables frequency ramping.

GinSingSynth.setFreqRamp

syntax

void setFreqRamp (GSSynthOsc dcoSel , float relRate)

description

Sets the relative ramping rate parameter for frequency ramping. This function will only have an effect if frequency ramping is enabled (enableFreqRamp).Frequency ramping allows the DCO to smoothly ramp from minimum frequency to maximum frequency at the specified rate.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );           // send DCO 1 to mixer

s->setFrequency    ( OSC_1 , 100.0f );     // set initial frequency 100 Hz
s->setFreqRamp     ( OSC_1 , 0.01f );      // ramp at a slow rate

s->enableFreqRamp  ( OSC_1 , true );       // enable frequency ramp
```

arguments

dcoSel

The DCO to set the frequency ramping rate for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

relRate

A floating point value that specifies he relative rate at which the DCO ramps from the minimum frequency to the maximum frequency. Values can range from 0.0 (disables ramping) to 1.0 (maximum ramping). Note that both targeting and ramping share this rate variable.

GinSingSynth.setFreqRampVal

syntax

void setFreqRampVal (GSSynthOsc dcoSel , uint rateVal)

description

Sets the relative ramping rate parameter for frequency ramping as an unsigned integer. The Babblebot IC internally uses an unsigned integer for frequency ramping so using this function avoids the floating point conversion performed in setFreqRamp(). This function will only have an effect if frequency ramping is enabled (enableFreqRamp).Frequency ramping allows the DCO to smoothly ramp from minimum frequency to maximum frequency at the specified rate.

example

```
GinSingSynth *s = GS.getSynth();          // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );          // send DCO 1 to mixer

s->setFrequency   ( OSC_1 , 100.0f );     // set initial frequency 100 Hz
s->setFreqRampVal ( OSC_1 , 2 );          // ramp at a slow rate

s->enableFreqRamp ( OSC_1 , true );       // enable frequency ramp
```

arguments

dcoSel

The DCO to set the frequency ramping rate for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

rateVal

An unsigned integer value that specifies he relative rate at which the DCO ramps from the minimum frequency to the maximum frequency. Values can range from 0 (disables ramping) to 255 (maximum ramping). Note that both targeting and ramping share this rate variable.

GinSingSynth.enableAmpTarget

syntax

void enableAmpTarget (GSSynthOsc dcoSel , bool enable)

description

Enables or disables amplitude targeting on the DCO.For amplitude targeting to function, amplitude ramping must also be enabled (enableAmpRamp()).
Amplitude targeting allows the DCO to smoothly ramp from its current amplitude to a specified target amplitude as a specified rate. This can be useful for smooth amplitude changes in the DCO.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );           // send DCO 1 to mixer

s->setAmplitude    ( OSC_1 , 0.0f );       // set initial amplitude to 0%
s->setAmpTarget    ( OSC_1 , 1.0f ,        // set amplitude target to 100%
                             0.008f );     // at a slow rate

s->enableAmpRamp   ( OSC_1 , true );       // enable amplitude ramp

s->enableAmpTarget ( OSC_1 , enable );     // enable amplitude target
```

arguments

dcoSel

The DCO to enable or disable amplitude targeting for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

enable

A boolean value that enables or disables amplitude targeting.

GinSingSynth.setAmpTarget

syntax

void setAmpTarget (GSSynthOsc dcoSel , float amp , float relRate)

description

Sets the parameters for amplitude targeting. This function will only have an effect if amplitude targeting is enabled (enableAmpTarget). Amplitude targeting allows the DCO to smoothly ramp from its current amplitude to the specified target amplitude at the specified relative rate. This can be useful for smooth amplitude changes in the DCO, and can be called continuously to provide amplitude fade functionality.

example

```
GinSingSynth *s = GS.getSynth();          // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );          // send DCO 1 to mixer

s->setAmplitude    ( OSC_1 , 0.0f );      // set initial amplitude to 0%
s->setAmpTarget    ( OSC_1 , 1.0f ,       // set amplitude target to 100%
                             0.008f );    // at a slow rate

s->enableAmpRamp   ( OSC_1 , true );      // enable amplitude ramp

s->enableAmpTarget ( OSC_1 , enable );    // enable amplitude target
```

arguments

dcoSel

The DCO to set the amplitude targeting parameters for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

amp

A floating point value that specifies the target amplitude. The amplitude can range from 0.0 (muted) to 1.0 (full volume).

relRate

A floating point value that specifies he relative rate at which the DCO ramps from the current amplitude to the target amplitude. Values can range from 0.0 (disables targeting) to 1.0 (instantaneous targeting).

GinSingSynth.setAmpTargetVal

syntax

void setAmpTargetVal (GSSynthOsc dcoSel , sbyte ampVal ,
uint relRateVal)

description

Sets the parameters for amplitude targeting as unsigned integers. The Babblebot IC internally uses unsigned integers for amplitude targeting so using this function will avoid the floating point conversion performed in setAmpTarget(). This function will only have an effect if amplitude targeting is enabled (enableAmpTarget). Amplitude targeting allows the DCO to smoothly ramp from its current amplitude to the specified target amplitude at the specified relative rate. This can be useful for smooth amplitude changes in the DCO, and can be called continuously to provide amplitude fade functionality.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );           // send DCO 1 to mixer

s->setAmplitude    ( OSC_1 , 0.0f );       // set initial amplitude to 0%
s->setAmpTargetVal ( OSC_1 , 127 ,         // set amplitude target to 100%
                             2 );          // at a slow rate

s->enableAmpRamp   ( OSC_1 , true );       // enable amplitude ramp

s->enableAmpTarget ( OSC_1 , enable );     // enable amplitude target
```

arguments

dcoSel

The DCO to set the amplitude targeting parameters for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

ampVal

An signed byte value that represents the target amplitude. The amplitude can range from 0 (muted) to 127 (full volume).

relRateVal

An unsigned integer that represents the relative targeting rate. The relative rate can range from 0 (targeting disabled) to 32767 (instantaneous targeting). Note that both targeting and ramping share this rate variable.

GinSingSynth.enableAmpRamp

syntax

void enableAmpRamp (GSSynthOsc dcoSel , bool enable)

description

Enables or disables amplitude ramping on the DCO. Amplitude ramping allows the DCO to smoothly ramp from minimum amplitude to maximum amplitude at a specified rate. When amplitude targeting is enabled as well it will ramp to the target amplitude, otherwise it will free run until disabled.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );           // send DCO 1 to mixer

s->setAmpRamp      ( OSC_1 , 0.08f );      // ramp at a slow rate
s->enableAmpRamp   ( OSC_1 , true );       // enable amplitude ramp
```

arguments

dcoSel

The DCO to enable or disable amplitude ramping for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

enable

A boolean value that enables or disables amplitude ramping.

GinSingSynth.setAmpRamp

syntax

void setAmpRamp (GSSynthOsc dcoSel , float relRate)

description

Sets the relative ramping rate parameter for amplitude ramping. This function will only have an effect if amplitude ramping is enabled (enableAmpRamp). Amplitude ramping allows the DCO to smoothly ramp from minimum amplitude to maximum amplitude at the specified rate.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );           // send DCO 1 to mixer

s->setAmpRamp      ( OSC_1 , 0.08f );      // ramp at a slow rate
s->enableAmpRamp   ( OSC_1 , true );       // enable amplitude ramp
```

arguments

dcoSel

The DCO to set the amplitude ramping rate for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

relRate

A floating point value that specifies he relative rate at which the DCO ramps from the minimum amplitude to the maximum amplitude. Values can range from 0.0 (disables ramping) to 1.0 (maximum ramping). Note that both targeting and ramping share this rate variable.

GinSingSynth.setAmpRampVal

syntax

void setAmpRampVal (GSSynthOsc dcoSel , uint relRateVal)

description

Sets the relative ramping rate parameter for amplitude ramping as an unsigned integer. The Babblebot IC internally uses an unsigned integer for amplitude ramping so using this function avoids the floating point conversion performed in setAmpRamp(). This function will only have an effect if amplitude ramping is enabled (enableAmpRamp). Amplitude ramping allows the DCO to smoothly ramp from minimum amplitude to maximum amplitude at the specified rate.

example

```
GinSingSynth *s = GS.getSynth();           // get synth mode

s->setPatch ( OSC_1_TO_MIXER  );           // send DCO 1 to mixer

s->setAmpRampVal   ( OSC_1 , 2 );          // ramp at a slow rate
s->enableAmpRamp   ( OSC_1 , true );       // enable amplitude ramp
```

arguments

dcoSel

The DCO to set the amplitude ramping rate for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

rateVal

An unsigned integer value that specifies he relative rate at which the DCO ramps from the minimum amplitude to the maximum amplitude. Values can range from 0 (disables ramping) to 255 (maximum ramping). Note that both targeting and ramping share this rate variable.

GinSingSynth.trigger

syntax

void trigger (GSSynthOsc dcoSel)

description

Triggers the specified DCO. When this function is called, the DCO will begin executing its amplitude envelope (ADSR), and will sequence through to its sustain amplitude, where it will hold the amplitude until release() is called. If this function is called while the envelope is running it will start the envelope sequence over again.

example

```
GinSingSynth *s = GS.getSynth();          // get synth mode

s->setPatch     ( OSC_1_TO_MIXER  );      // patch DCO 1 to mixer
s->setFrequency ( OSC_1 , 100.0f );       // set the base tone frequency

s->trigger      ( OSC_1 );                // trigger the ADSR
delay ( 1000 );                           // wait 1 second
s->release      ( OSC_1 );                // release the ADSR
```

arguments

dcoSel

The DCO to trigger. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

GinSingSynth.release

syntax

void release (GSSynthOsc dcoSel)

description

Releases the DCO. When this function is called, the specified voice will begin terminating its amplitude envelope (ADSR), and will sequence through to its release amplitude. If this function is called while the envelope is not active (no triggered), it will have no effect.

example

```
GinSingSynth *s = GS.getSynth();          // get synth mode

s->setPatch     ( OSC_1_TO_MIXER  );      // patch DCO 1 to mixer
s->setFrequency ( OSC_1 , 100.0f );       // set the base tone frequency

s->trigger      ( OSC_1 );                // trigger the ADSR
delay ( 1000 );                           // wait 1 second
s->release      ( OSC_1 );                // release the ADSR
```

arguments

dcoSel

The DCO to release. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

GinSingSynth.setEnvelope

syntax

```
void setEnvelope   ( GSSynthOsc dcoSel ,
                     GSAttackDur attackDur , float attackAmp ,
                     GSDecRelDur decayDur , float decayAmp ,
                     GSDecRelDur releaseDur , float releaseAmp )
```

description

Sets the amplitude envelope parameters for the DCO. Each DCO has a table that determines how the output amplitude varies over time once the DCO is triggered. This table has four sequential stages (Attack, Decay, Sustain, Release) known as an ADSR envelope, which define amplitude ramps within a time window. The sustain stage is unique in that its time window is variable based on the time between the completion of the decay stage until release() is called, and its amplitude is fixed at the decay stage level; for this reason it does not need to be specified.

example

```
GinSingSynth *s = GS.getSynth();                   // get synth mode

s->setEnvelope   ( OSC_ALL , AT_1000MS , 0.25f ,   // 1s attack to 25% vol
                             DR_2MS    , 0.25f ,   // 2ms decay to 25% vol
                             DR_1500MS , 0.0f  );  // 1.5s release to 0% vol
```

arguments

dcoSel

The DCO to set the ADSR envelope for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

attackDur | attackAmp

Attack stage settings. Duration time specified as a variable type GSAttackDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

decayDur | decayAmp

Decay stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

releaseDur | releaseAmp

Release stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

GinSingSynth.setEnvelopeVal

syntax

void setEnvelopeVal (GSSynthOsc dcoSel ,
GSAttackDur attackDur , ubyte attackAmpVal ,
GSDecRelDur decayDur , ubyte decayAmpVal ,
GSDecRelDur releaseDur , ubyte releaseAmpVal)

description

Sets the amplitude envelope parameters for the DCO as unsigned bytes. The Babblebot IC internally uses unsigned bytes for ADSR so using this function will avoid floating point conversion performed using setEnvelope(). Each DCO has a table that determines how the output amplitude varies over time once the DCO is triggered. This table has four sequential stages (Attack, Decay, Sustain, Release) known as an ADSR envelope, which define amplitude ramps within a time window.

example

```
GinSingSynth *s = GS.getSynth();                // get synth mode

s->setEnvelopeVal( OSC_ALL , AT_1000MS , 127 ,   // 1s attack to 100% vol
                             DR_2MS    , 64  ,   // 2ms decay to 50% vol
                             DR_1500MS , 0   );  // 1.5s release to 0% vol
```

arguments

dcoSel

The DCO to set the ADSR envelope for. The DCO selection is enumerated as the variable type GSSynthOsc, and can be one of OSC_1, OSC_2 , or OSC_3, or OSC_ALL on the currently selected bank (via selectBank).

attackDurVal | attackAmpVal

Attack stage settings. Duration time specified as a variable type GSAttackDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

decayDurVal | decayAmpVal

Decay stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

releaseDurVal| releaseAmpVal

Release stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

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*

GingSingMaster.enableUserOutput

syntax

void enableUserOutput (bool enable)

description

Enables or disables the user programmable output on the shield. On the shield, there are two solder pads labeled “Q” that can be used to power external circuits controlled by this function. The Q output is a 5 volt 40 mA connection to the Babblebot IC, and can be useful when adding additional features such as blinking eyes, etc.

example

```
GinSingMaster *m = GS.getMaster();              // get master interface

m->enableUserOutput ( true );                   // turn on Q output
delay ( 1000 );                                 // wait one second

m->enableUserOutput ( false );                  // turn off Q output
delay ( 1000 );                                 // wait one second
```

arguments

enable

A boolean value that determines the output state of Q on the shield. When the value is true, the Q output is enabled (5V), when the value is false, the Q output is disabled (0V).

GinSingMaster.setAmplitude

syntax

void setAmplitude (GSMasterMixer mixerIdx , float amplitude)

description

Sets the relative output amplitude of the selected mixer. The two available mixers correspond to each of the two respective banks, and allow a way to control the overall bank volume independently. The bank A mixer will scale the output amplitudes of DCOs A1, A2, and A3, whereas the bank B mixer will scale the output amplitudes of DCOs B1, B2, and B3.

example

```
GinSingMaster *m = GS.getMaster();             // get master interface
m->setAmplitude ( MIX_A , 0.5f );              // scale bank A DCO vol by 50%
```

arguments

mixerIdx

The mixer to set the amplitude for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

amplitude

A floating point value that represents the amplitude for the mixer. The amplitude value can range from 0.0 (muted) to 1.0 (full volume. Note that DCO output is also scaled by its own amplitude and the master amplitude as well.

GinSingMaster.setAmplitudeVal

syntax

void setAmplitudeVal (GSMasterMixer mixerIdx , ubyte ampVal)

description

Sets the relative output amplitude of the selected mixer as an unsigned byte. The Babblebot IC internally uses an unsigned byte to represent volume, so using this function will avoid the floating point conversion performed in setAmplitude().

example

```
GinSingMaster *m = GS.getMaster();              // get master interface
m->setAmplitudeVal ( MIX_A , 64 );              // 50% volume on bank A
```

arguments

mixerIdx

The mixer to set the amplitude for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

ampVal

An unsigned byte that represents the output amplitude for the mixer. The value can range from 0 (muted) to 127 (full volume).

GinSingMaster.setMasterAmplitude

syntax

void setMasterAmplitude (float amplitude)

description

Sets the overall (global) amplitude of the shield. This amplitude is a software equivalent to a volume control on the shield, and scales the output from the DCOs and bank mixers by the given value.

example

```
GinSingMaster *m = GS.getMaster();             // get master interface
m->setMasterAmplitude ( 0.5f );                 // set global output to 50%
```

arguments

amplitude

A floating point value that represents the global amplitude output. The amplitude value can range from 0.0 (muted) to 1.0 (full volume. Note that DCO output is also scaled by its own amplitude and the bank amplitude as well.

GinSingMaster.setMasterAmplitudeVal

syntax

void setMasterAmplitudeVal (ubyte ampVal)

description

Sets the overall (global) amplitude of the shield as an unsigned byte. The Babblebot IC internally uses a unsigned byte to represent volume, so using this function will avoid the floating point conversion performed in setMasterAmplitude().

example

```
GinSingMaster *m = GS.getMaster();              // get master interface
m->setMasterAmplitudeVal ( 64 );                 // set global output to 50%
```

arguments

ampVal

An unsigned byte that represents the global output amplitude. The value can range from 0 (muted) to 127 (full volume).

GinSingMaster.enableAmpTarget

syntax

void enableAmpTarget (GSMasterMixer mixerIdx , bool enable)

description

Enables or disables amplitude targeting on the bank output.For amplitude targeting to function, amplitude ramping must also be enabled (enableAmpRamp()).
Amplitude targeting allows the bank output to smoothly ramp from its current amplitude to a specified target amplitude as a specified rate. This can be useful for smooth amplitude changes in the bank output.

example

```
GinSingMaster *m = GS.getMaster();            // get master interface

m->setAmplitude       ( MIX_ALL , 0.0f );      // set initial amplitude to 0%
m->setAmpTarget       ( MIX_ALL , 1.0f ,       // set amplitude target to 100%
                                0.008f );      // at a slow rate

s->enableAmpRamp      ( MIX_ALL , true );      // enable amplitude ramp

s->enableAmpTarget    ( MIX_ALL , enable );    // enable amplitude target
```

arguments

mixerIdx

The mixer to control amplitude targeting for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

enable

A boolean value that enables or disables amplitude targeting.

GinSingMaster.setAmpTarget

syntax

void setAmpTarget (GSMasterMixer mixerIdx, float amp, float relRate)

description

Sets the parameters for amplitude targeting on the bank output. This function will only have an effect if amplitude targeting is enabled (enableAmpTarget). Amplitude targeting allows the global output to smoothly ramp from its current amplitude to the specified target amplitude at the specified relative rate. This can be useful for smooth amplitude changes in the bank output, and can be called continuously to provide amplitude fade functionality.

example

```
GinSingMaster *m = GS.getMaster();             // get master interface

m->setAmplitude       ( MIX_ALL , 0.0f );      // set initial amplitude to 0%

m->setAmpTarget       ( MIX_ALL , 1.0f ,       // set amplitude target to 100%
                                  0.008f );    // at a slow rate

s->enableAmpRamp      ( MIX_ALL , true );      // enable amplitude ramp

s->enableAmpTarget    ( MIX_ALL , enable );    // enable amplitude target
```

arguments

mixerIdx

The mixer to set amplitude targeting parameters for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

amp

A floating point value that specifies the target amplitude. The amplitude can range from 0.0 (muted) to 1.0 (full volume).

relRate

A floating point value that specifies he relative rate at which the mixer ramps from the current amplitude to the target amplitude. Values can range from 0.0 (disables targeting) to 1.0 (instantaneous targeting).

GinSingMaster.setAmpTargetVal

syntax

void setAmpTargetVal (GSMasterMixer mixerIdx, ubyte ampVal ,
uint relRateVal)

description

Sets the parameters for amplitude targeting as unsigned integers. The Babblebot IC internally uses unsigned integers for amplitude targeting so using this function will avoid the floating point conversion performed in setAmpTarget(). This function will only have an effect if amplitude targeting is enabled (enableAmpTarget).

example

```
GinSingMaster *m = GS.getMaster();            // get master interface

m->setAmplitude       ( MIX_ALL , 0.0f );      // set initial amplitude to 0%

m->setAmpTargetVal    ( MIX_ALL , 127 ,        // set amplitude target to 100%
                                    2 );       // at a slow rate

s->enableAmpRamp      ( MIX_ALL , true );      // enable amplitude ramp

s->enableAmpTarget    ( MIX_ALL , enable );    // enable amplitude target
```

arguments

mixerIdx

The mixer to set the amplitude targeting parameters for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

ampVal

An signed byte value that represents the target amplitude. The amplitude can range from 0 (muted) to 127 (full volume).

relRateVal

An unsigned integer that represents the relative targeting rate. The relative rate can range from 0 (targeting disabled) to 32767 (instantaneous targeting). Note that both targeting and ramping share this rate variable.

GinSingMaster.enableAmpRamp

syntax

void enableAmpRamp (GSMasterMixer mixerIdx , bool enable)

description

Enables or disables amplitude ramping on the bank. Amplitude ramping allows the bank to smoothly ramp from minimum amplitude to maximum amplitude at a specified rate. When amplitude targeting is enabled as well it will ramp to the target amplitude, otherwise it will free run until disabled.

example

```
GinSingMaster *m = GS.getMaster();             // get master interface

m->setAmpRamp        ( MIX_ALL , 0.08f );      // ramp at a slow rate
s->enableAmpRamp     ( MIX_ALL , true );       // enable amplitude ramp
```

arguments

mixerIdx

The mixer to control amplitude ramping for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

enable

A boolean value that enables or disables amplitude ramping.

GinSingMaster.setAmpRamp

syntax

void setAmpRamp (GSMasterMixer mixerIdx , float relRate)

description

Sets the relative ramping rate parameter for amplitude ramping. This function will only have an effect if amplitude ramping is enabled (enableAmpRamp). Amplitude ramping allows the bank to smoothly ramp from minimum amplitude to maximum amplitude at the specified rate.

example

```
GinSingMaster *m = GS.getMaster();            // get master interface

m->setAmpRamp        ( MIX_ALL , 0.08f );      // ramp at a slow rate
s->enableAmpRamp     ( MIX_ALL , true );       // enable amplitude ramp
```

arguments

mixerIdx

The mixer to set amplitude ramping parameters for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

relRate

A floating point value that specifies he relative rate at which the bank ramps from the minimum amplitude to the maximum amplitude. Values can range from 0.0 (disables ramping) to 1.0 (maximum ramping). Note that both targeting and ramping share this rate variable.

GinSingMaster.setAmpRampVal

syntax

void setAmpRampVal (GSMasterMixer mixerIdx , uint relRateVal)

description

Sets the relative ramping rate parameter for amplitude ramping as an unsigned integer. The Babblebot IC internally uses an unsigned integer for amplitude ramping so using this function avoids the floating point conversion performed in setAmpRamp(). This function will only have an effect if amplitude ramping is enabled (enableAmpRamp). Amplitude ramping allows the bank to smoothly ramp from minimum amplitude to maximum amplitude at the specified rate.

example

```
GinSingMaster *m = GS.getMaster();             // get master interface

m->setAmpRampVal     ( MIX_ALL , 2 );          // ramp at a slow rate
s->enableAmpRamp     ( MIX_ALL , true );       // enable amplitude ramp
```

arguments

mixerIdx

The mixer to set amplitude ramping parameters for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

rateVal

An unsigned integer value that specifies he relative rate at which the bank ramps from the minimum amplitude to the maximum amplitude. Values can range from 0 (disables ramping) to 255 (maximum ramping). Note that both targeting and ramping share this rate variable.

GinSingMaster.trigger

syntax

void trigger (GSMasterMixer mixerIdx)

description

Triggers the specified bank envelope. When this function is called, the bank mixer will begin executing its amplitude envelope (ADSR), and will sequence through to its sustain amplitude, where it will hold the amplitude until release() is called. If this function is called while the envelope is running it will start the envelope sequence over again.

example

```
GinSingMaster *m = GS.getMaster();           // get master interface

m->trigger ( MIX_A );                         // trigger ADSR on bank A
delay ( 1000 );                               // wait one second
m->release ( MIX_A );                         // release ADSR on bank A
```

arguments

mixerIdx

The mixer to trigger the envelope for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

GinSingMaster.release

syntax

void release (GSMasterMixer mixerIdx)

description

Releases the specified bank envelope. When this function is called, the specified bank will begin terminating its amplitude envelope (ADSR), and will sequence through to its release amplitude. If this function is called while the envelope is not active (no triggered), it will have no effect.

example

```
GinSingMaster *m = GS.getMaster();              // get master interface

m->trigger ( MIX_A );                            // trigger ADSR on bank A
delay ( 1000 );                                  // wait one second
m->release ( MIX_A );                            // release ADSR on bank A
```

arguments

mixerIdx

The mixer to release the envelope for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

GinSingMaster.setEnvelope

syntax

```
void setEnvelope   ( GSMasterMixer mixerIdx ,
                     GSAttackDur attackDur , float attackAmp ,
                     GSDecRelDur decayDur , float decayAmp ,
                     GSDecRelDur releaseDur , float releaseAmp )
```

description

Sets the amplitude envelope parameters for the bank. Each bank has a table that determines how the output amplitude varies over time once the bank enveloped is triggered. This table has four sequential stages (Attack, Decay, Sustain, Release) known as an ADSR envelope, which define amplitude ramps within a time window. The sustain stage is unique in that its time window is variable based on the time between the completion of the decay stage until release() is called, and its amplitude is fixed at the decay stage level; for this reason it does not need to be specified.

example

```
GinSingMaster *m = GS.getMaster();             // get master interface

m->setEnvelope   ( MIX_ALL , AT_1000MS , 0.25f ,   // 1s attack to 25% vol
                             DR_2MS    , 0.25f ,   // 2ms decay to 25% vol
                             DR_1500MS , 0.0f  );  // 1.5s release to 0% vol
```

arguments

mixerIdx

The mixer to set the ADSR parameters for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

attackDur | *attackAmp*

Attack stage settings. Duration time specified as a variable type GSAttackDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

decayDur | *decayAmp*

Decay stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

releaseDur | *releaseAmp*

Release stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

GinSingMaster.setEnvelopeVal

syntax

void setEnvelopeVal (GSMasterMixer mixerIdx ,
GSAttackDur attackDur , ubyte attackAmpVal ,
GSDecRelDur decayDur , ubyte decayAmpVal ,
GSDecRelDur releaseDur , ubyte releaseAmpVal)

description

Sets the amplitude envelope parameters for the bank mixer as unsigned bytes. The Babblebot IC internally uses unsigned bytes for ADSR so using this function will avoid floating point conversion performed using setEnvelope().

example

```
GinSingMaster *m = GS.getMaster();             // get master interface

m->setEnvelopeVal( MIX_ALL , AT_1000MS , 127 ,   // 1s attack to 100% vol
                             DR_2MS    , 64  ,   // 2ms decay to 50% vol
                             DR_1500MS , 0   );  // 1.5s release to 0% vol
```

arguments

mixerIdx

The mixer to set the ADSR parameters for. The mixer selection is enumerated as the variable type GSMasterMixer in appendix A and can be one of MIX_A , MIX_B , or MIX_ALL.

attackDurVal | attackAmpVal

Attack stage settings. Duration time specified as a variable type GSAttackDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

decayDurVal | decayAmpVal

Decay stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

releaseDurVal| releaseAmpVal

Release stage settings. Duration time specified as a variable type GSDecRelDur enumerated in appendix A. Amplitude specified as a floating point value from 0.0 (mute) to full (1.0).

Appendix A - enumeration types

Primitives

primitive	type	size (bytes)	range
ubyte	unsigned char	1	0 to 255
sbyte	signed char	1	-127 to 127
uint	unsigned int	2	0 to 65,535
ulong	unsigned long	4	0 to 4,294,967,296

GSPreset

SpaceWarp	Waba	RandomThoughts	Gong
Pwang	Wow	Rananana	Twarty
Telly	Pulsator	Bound	TipToe
Spokes	Chopper	Phazer	PowerLines
HeavyMetal1	HeavyMetal2	ACMotor	YaYa
March	NoiseChatter	BlipChatter	Carney
EarthQuake	MindProbe	Siren	Squaba
SteamLoco	FreqMod	AmpMod	

GSWaveType

SINE	sinusoidal (no harmonics)
TRIANGLE	linear rise, linear fall (odd harmonics A = $1/n^2$)
SAWTOOTH	instant rise, linear fall (all harmonics A = 1/n)
RAMP	linear rise, instant fall (all harmonics A = 1/n)
PULSE	instant rise, instant fall (odd harmonics A = 1/n @ 50%)
NOISE	random amplitude noise

LEVEL	fixed DC offset

GSWaveMode

SYMMETRIC	waveform oscillates symmetrically around zero
POSITIVE	waveform oscillates from zero to maximum only

GSNote

C_0	CS_0	D_0	DS_0	E_0	F_0	FS_0	G_0	GS_0	A_0	AS_0	B_0
C_1	CS_1	D_1	DS_1	E_1	F_1	FS_1	G_1	GS_1	A_1	AS_1	B_1
C_2	CS_2	D_2	DS_2	E_2	F_2	FS_2	G_2	GS_2	A_2	AS_2	B_2
C_3	CS_3	D_3	DS_3	E_3	F_3	FS_3	G_3	GS_3	A_3	AS_3	B_3
C_4	CS_4	D_4	DS_4	E_4	F_4	FS_4	G_4	GS_4	A_4	AS_4	B_4
C_5	CS_5	D_5	DS_5	E_5	F_5	FS_5	G_5	GS_5	A_5	AS_5	B_5
C_6	CS_6	D_6	DS_6	E_6	F_6	FS_6	G_6	GS_6	A_6	AS_6	B_6
C_7	CS_7	D_7	DS_7	E_7	F_7	FS_7	G_7	GS_7	A_7	AS_7	B_7

GSAllophone

inflections

_SPEEDUP	speed up next allophone
_SPEEDDN	slow down next allophone
_ VOLUP	increase volume on next allophone
_VOLDN	decrease volume on next allophone
_PITCHUP	increase pitch of next allophone
_PITCHDN	decrease pitch of next allophone
_BENDUP	bend up pitch of next allophone

_BENDDN	bend down pitch of next allophone

phonemes

_A	hat, fast, fan
_AA	father, fall
_AE	gate, ate, ray
_AIR	hair, stair, repair
_AU	hot, clock, fox
_BE	bear, bird, bead
_BO	bone, book, brown
_EB	cab, crib, web
_OB	bob, sub, tub
_CH	church, feature, march
_DE	deep, date, divide
_DO	do, dust, dog
_ED	could, bird
_OD	bud, food
_E	met, check, red
_EE	see, even, feed
_ER	fir, bird, burn
_F	food, effort, off
_GE	get, gate, guest
_GO	got, glue, god
_HE	help, hand, hair
_HO	hoe, hot hung
_I	sit, fix, pin
_IE	mice, fight, white
_J	dodge, jet, savage
_KE	can't, clown, key
_KO	comb, quick, coast
_EK	speak, task
_OK	nook, took, october
_LE	lake, alarm, lapel
_LO	clock, plus, hello
_M	milk, famous, broom

_NE	nip, danger, thin
_NO	no, snow, on
_NGE	think, ping
_NGO	hung, song
_OE	go, hello, snow
_OI	boy, toy, voice
_OO	book, could, should
_OU	our, ouch, owl
_OR	corn, four, your
_PE	people, computer
_PO	pow, copy
_R	ray, brain, over
_SE	see, vest, plus
_SO	so, sweat
_SH	ship, fiction, leash
_T	part, little, sit
_TH	thin, month
_THH	there, that, this
_TS	parts, costs, robots
_U	luck, jump, plus
_UE	food, june
_V	vest, even, twelve
_W	wool, sweat
_Y	yes, yard, million
_Z	zoo, zap
_ZH	azure, treasure
_PA0	12 millisecond pause
_PA1	48 millisecond pause
_PA2	62 millisecond pause
_FD0	1 millisecond delay
_FD1	100 millisecond delay
_FD2	600 millisecond delay

terminator

_ENDPHRASE	phrase terminator

GSAttackDur

AT_2MS	2 milliseconds	AT_100MS	0.1 seconds
AT_8MS	8 milliseconds	AT_250MS	0.25 seconds
AT_16MS	16 milliseconds	AT_500MS	0.50 seconds
AT_24MS	24 milliseconds	AT_800MS	0.8 seconds
AT_38MS	38 milliseconds	AT_1000MS	1.0 seconds
AT_56MS	56 milliseconds	AT_2800MS	2.8 seconds
AT_68MS	68 milliseconds	AT_5600MS	5.6 seconds
AT_80MS	80 milliseconds	AT_11200MS	11.2 seconds

GSDecRelDur

DR_2MS	2 milliseconds	DR_59MS	59 milliseconds
DR_6MS	6 milliseconds	DR_145MS	145 milliseconds
DR_10MS	10 milliseconds	DR_292MS	0.292 seconds
DR_15MS	15 milliseconds	DR_455MS	0.455 seconds
DR_23MS	23 milliseconds	DR_575MS	0.575 seconds
DR_34MS	34 milliseconds	DR_1500MS	1.50 seconds
DR_40MS	40 milliseconds	DR_2785MS	2.785 seconds
DR_48MS	48 milliseconds	DR_4873MS	4.873 seconds

GSSynthOsc

OSC_1	current bank DCO 1 (default base oscillator)
OSC_2	current bank DCO 2 (default amplitude modulator)
OSC_3	current bank DCO 3 (default frequency modulator)
OSC_ALL	all oscillators in current bank

GSSynthBank

BANK_A	DCO A1, A2, A3, mixer A
BANK_B	DCO B1, B2, B3, mixer B

GSSynthPatch

DCO mixer send

OSC_1_TO_MIXER	sends DCO 1 to bank mixer
OSC_2_TO_MIXER	sends DCO 2 to bank mixer
OSC_3_TO_MIXER	sends DCO 3 to bank mixer

DCO amplitude modulation

OSC_2_AMPMOD_OSC_1	amplitude modulate DCO 1 using DCO 2 output
OSC_2_AMPMOD_OSC_1_50PERC	reduce amplitude modulation effect by 50%
OSC_2_RINGMOD_OSC_1	ring modulate DCO 1 using DCO 2 output

DCO frequency modulation

OSC_3_FRQMOD_OSC_1	frequency modulate DCO 1 using DCO 3 output
OSC_3_FRQMOD_OSC_1_50PERC	reduce frequency modulation effect by 50%

mixer modulation

OSC_B1_AMPMOD_MIXER	amplitude modulate mixer using DCO B1 output
OSC_B1_AMPMOD_MIXER_50PERC	reduce amplitude modulation effect by 50%
OSC_B1_RINGMOD_MIXER	ring modulate bank mixer using DCO B1 output

pulse width modulation

OSC_B1_PWM_OSC_1	pulse width modulate DCO 1 using DCO B1 output
OSC_B2_PWM_OSC_2	pulse width modulate DCO 2 using DCO B2 output

OSC_B3_PWM_OSC_3	pulse width modulate DCO 3 using DCO B3 output

GSMasterMixer

MIX_A	bank A mixer
MIX_B	bank B mixer
MIX_ALL	both mixers

GSCommand

command	**size / args**		**function**
data send & receive			
ReadOneByte	1	reg	request 1 byte read from register
WriteOneByte	2	reg, val	write 1 byte value to register
WriteTwoBytes	3	reg, b1, b2	write 2 byte value to register
WriteThreeBytes	4	reg, b1, b2, b3	write 3 byte value to register
WriteOneByteWithMask	3	reg, val, mask	write 1 masked byte to register
speech control			
SetVoiceNote	1	note value	set speech frequency to note
SetVoiceFrequency	2	freq value	set speech frequency to value
SetVoiceDelay	1	delay value	set speech delay to value
SetVoiceDefaults	0		set speech values to defaults
TurnQOn	0		set Q output to high state
TurnQOff	0		Set Q output to low state
mixer control			
ClearMixersAB	0		clear bank mixer registers
ClearMixerAndOsc_A ClearMixerAndOsc_B	0		set bank mixer & DCO registers
RampToTargetsAB	0		activate ramp and target options

RampToTargets_A RampToTargets_B	0		activate bank A ramp and targets
LoadSoundMixer_A LoadSoundMixer_B	1	preset value	load sound preset into registers
DCO control			
LoadNoteOsc_A1 LoadNoteOsc_A2 LoadNoteOsc_A3 LoadNoteOsc_B1 LoadNoteOsc_B2 LoadNoteOsc_B3	1	note value	set DCO frequency by note value
LoadFreqOsc_A1 LoadFreqOsc_B2 LoadFreqOsc_A3 LoadFreqOsc_B1 LoadFreqOsc_B2 LoadFreqOsc_B3	3	f1, f2, f3	set DCO frequency by value
TriggerOsc_A1 TriggerOsc_A2 TriggerOsc_A3 TriggerOsc_B1 TriggerOsc_B2 TriggerOsc_B3	0		start the DCO amplitude envelop
ReleaseOsc_A1 ReleaseOsc_A2 ReleaseOsc_A3 ReleaseOsc_B1 ReleaseOsc_B2 ReleaseOsc_B3	0		release the DCO amplitude envelope
LoadPlayNoteOsc_A1 LoadPlayNoteOsc_A2 LoadPlayNoteOsc_A3 LoadPlayNoteOsc_B1 LoadPlayNoteOsc_B2 LoadPlayNoteOsc_B3	3	f1, f2, f3	set DCO frequency by value and trigger amplitude envelop
CmdHeader	-		Command prefix (internal)

GSRegister

bank

A_MixControl_0 B_MixControl_0	mixer control bits #1
A_MixControl_1 B_MixControl_1	mixer control bits #2
A_Amplitude B_Amplitude	output amplitude
A_AmplitudeTarget B_AmplitudeTarget	amplitude target value
A_AmplitudeXLow A_AmplitudeXHigh B_AmplitudeXLow B_AmplitudeXHigh	amplitude ramp rate
A_EnvelopeControl B_EnvelopeControl	envelope control bits
A_EnvelopeAttack B_EnvelopeAttack	ADSR attack value
A_EnvelopeDecay B_EnvelopeDecay	ADSR decay value
A_EnvelopeRelease B_EnvelopeRelease	ADSR release value
DCO	
A_OscDistort_1 A_OscDistort_2 A_OscDistort_3 B_OscDistort_1 B_OscDistort_2 B_OscDistort_3	frequency distortion
A_OscPWM_1 A_OscPWM_2 A_OscPWM_3 B_OscPWM_1 B_OscPWM_2 B_OscPWM_3	pulse width
A1_Control A2_Control A3_Control B1_Control B2_Control B3_Control	DCO control bits
A1_FrequencyFine A1_FrequencyLow A1_FrequencyHigh	frequency

A2_FrequencyFine A2_FrequencyLow A2_FrequencyHigh A3_FrequencyFine A3_FrequencyLow A3_FrequencyHigh B1_FrequencyFine B1_FrequencyLow B1_FrequencyHigh B2_FrequencyFine B2_FrequencyLow B2_FrequencyHigh B3_FrequencyFine B3_FrequencyLow B3_FrequencyHigh	
A1_FrequencyTargetLow A1_FrequencyTargetHigh A2_FrequencyTargetLow A2_FrequencyTargetHigh A3_FrequencyTargetLow A3_FrequencyTargetHigh B1_FrequencyTargetLow B1_FrequencyTargetHigh B2_FrequencyTargetLow B2_FrequencyTargetHigh B3_FrequencyTargetLow B3_FrequencyTargetHigh	frequency target
A1_FrequencyXLow A1_FrequencyXHigh A2_FrequencyXLow A2_FrequencyXHigh A3_FrequencyXLow A3_FrequencyXHigh B1_FrequencyXLow B1_FrequencyXHigh B2_FrequencyXLow B2_FrequencyXHigh B3_FrequencyXLow B3_FrequencyXHigh	frequency ramp rate
A1_Amplitude A2_Amplitude A3_Amplitude B1_Amplitude B2_Amplitude A3_Amplitude	output amplitude
A1_AmplitudeTarget A2_AmplitudeTarget A3_AmplitudeTarget	amplitude target value

B1_AmplitudeTarget B2_AmplitudeTarget B3_AmplitudeTarget	
A1_AmplitudeXLow A1_AmplitudeXHigh A2_AmplitudeXLow A2_AmplitudeXHigh A3_AmplitudeXLow A3_AmplitudeXHigh B1_AmplitudeXLow B1_AmplitudeXHigh B2_AmplitudeXLow B2_AmplitudeXHigh B3_AmplitudeXLow B3_AmplitudeXHigh	amplitude ramp rate
A1_EnvelopeControl A2_EnvelopeControl A3_EnvelopeControl B1_EnvelopeControl B2_EnvelopeControl B3_EnvelopeControl	ADSR control bits
A1_EnvelopeAttack A2_EnvelopeAttack A3_EnvelopeAttack B1_EnvelopeAttack B2_EnvelopeAttack B3_EnvelopeAttack	ADSR attack value
A1_EnvelopeDecay A2_EnvelopeDecay A3_EnvelopeDecay B1_EnvelopeDecay B2_EnvelopeDecay B3_EnvelopeDecay	ADSR decay value
A1_EnvelopeRelease A2_EnvelopeRelease A3_EnvelopeRelease B1_EnvelopeRelease B2_EnvelopeRelease B3_EnvelopeRelease	ADSR release value

speech

SpeechControl	speech control bits
SpeechFrequencyLow SpeechFrequencyHigh	frequency
TransitionSpeed	transition speed

PitchBend_A PitchBend_B	pitch bend rated
miscellaneous	
MiscellaneousControl	miscellaneous control bits
PortControl	port control bits
MasterAmplitude	master output amplitude
Output_A1 Output_A2 Output_A3 Output_B1 Output_B2 Output_B3	current output value