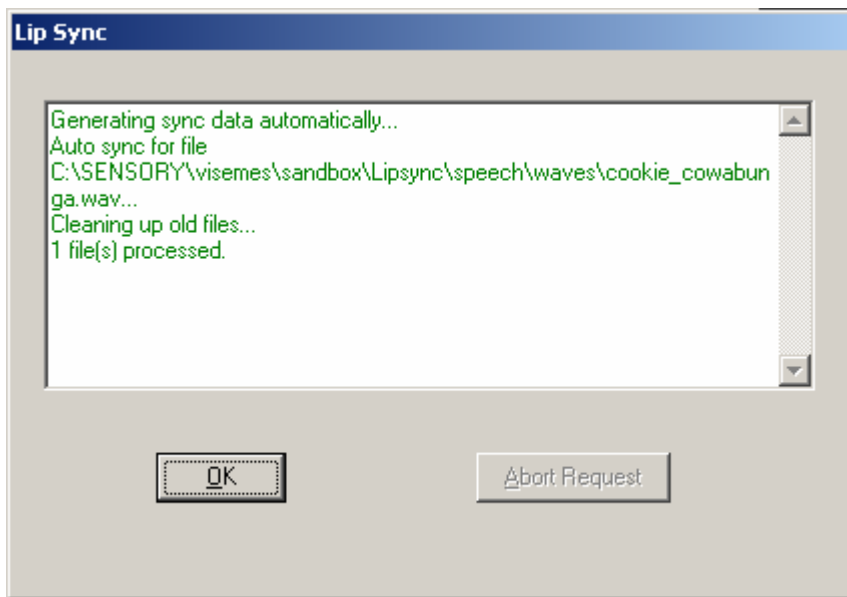
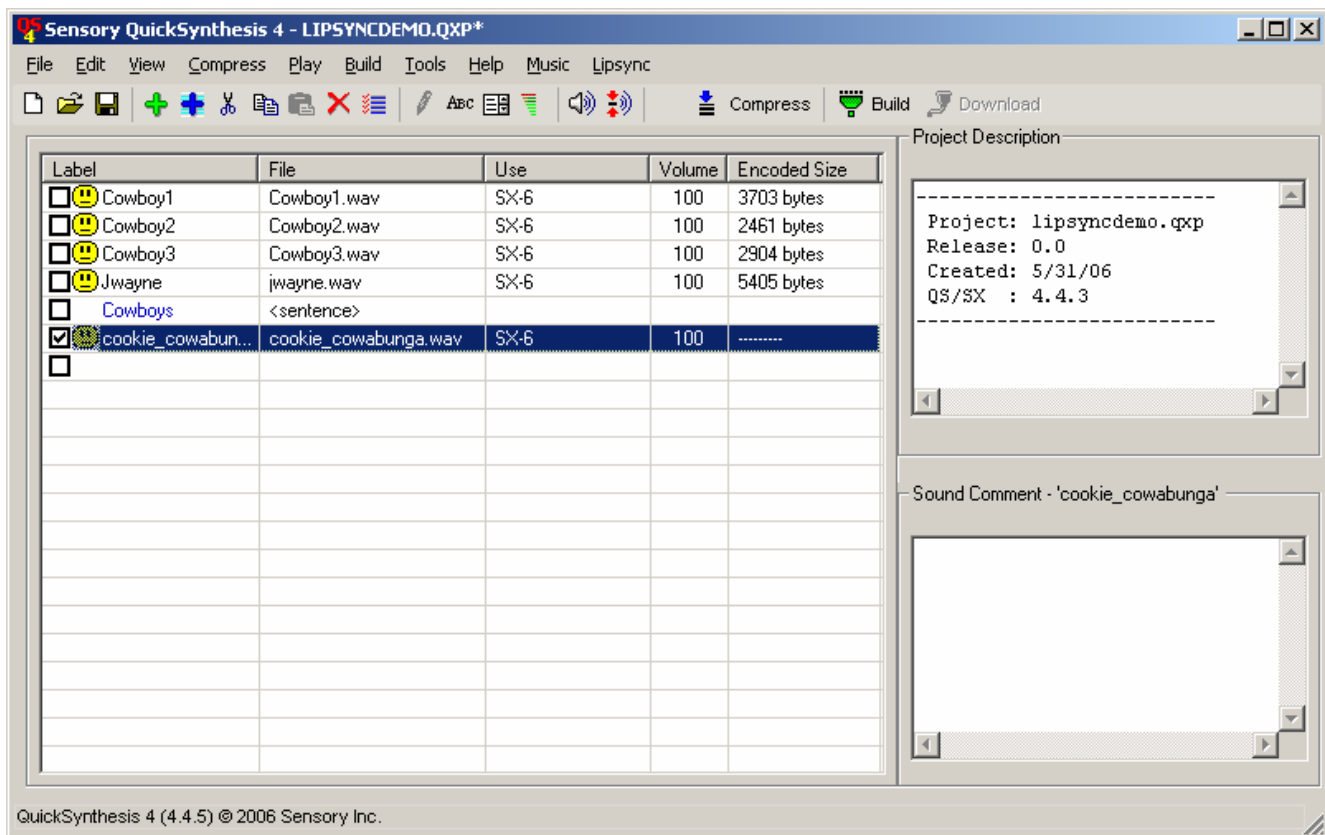


We now wish to add LipSync data for that file. With the file highlighted, open the LipSync menu, and click on “Auto Generate Sync”

When complete, the tool will display:

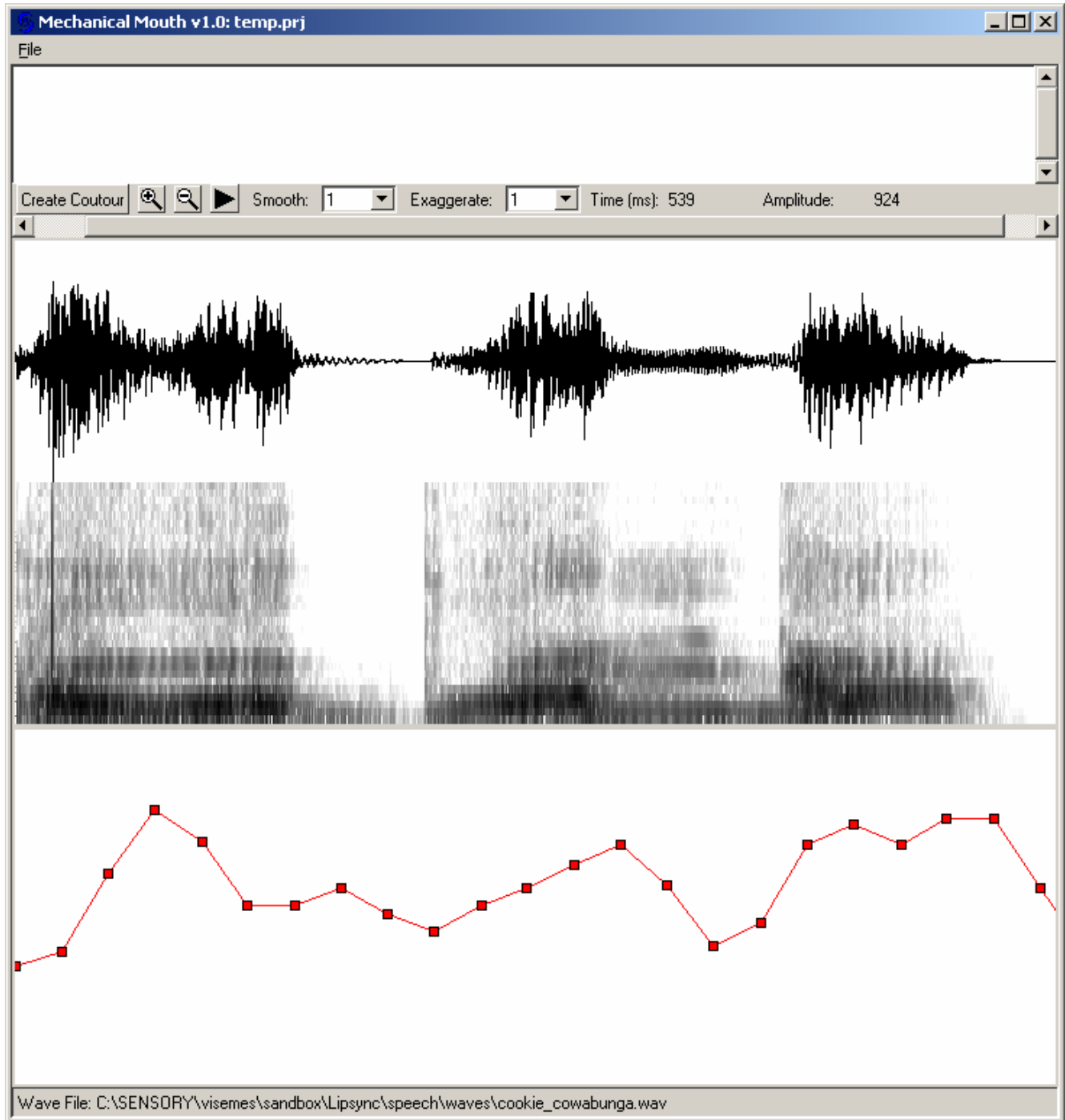


Note that the “cookie_cowabunga” label now has a face icon to its left.



We have now created preliminary LipSync data for the SX file.

Next, click on LipSync:EditSync. This opens up the LipSync tool editor and its simulation window. We will use this editor to improve the accuracy of the LipSync data set:

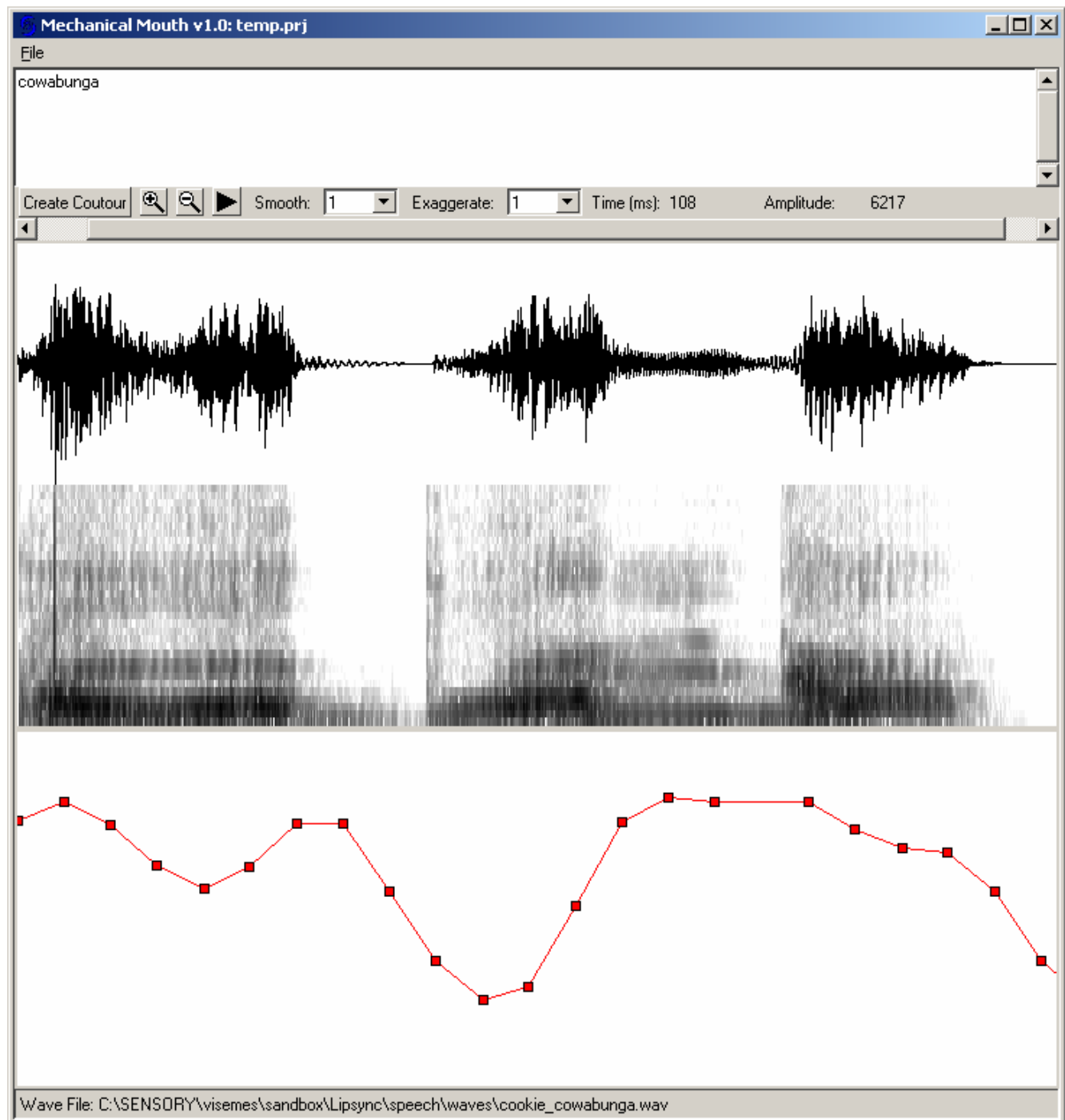




Note that the window opens with a wave file amplitude and spectral view and the preliminary LipSync data set (the red line). This red line represents how much the tool will open the mouth. It is scaled from 0% (mouth closed) at the minimum, to 100% (mouth fully open) at the highest possible amplitude. This set is how the LipSync tool interprets the wave file without text of the speech (in this case, the word "cowabunga"). Press the play button (▶), or right click in the wave/spectral view window. Note that the simulation mouth moves as the wave file plays.

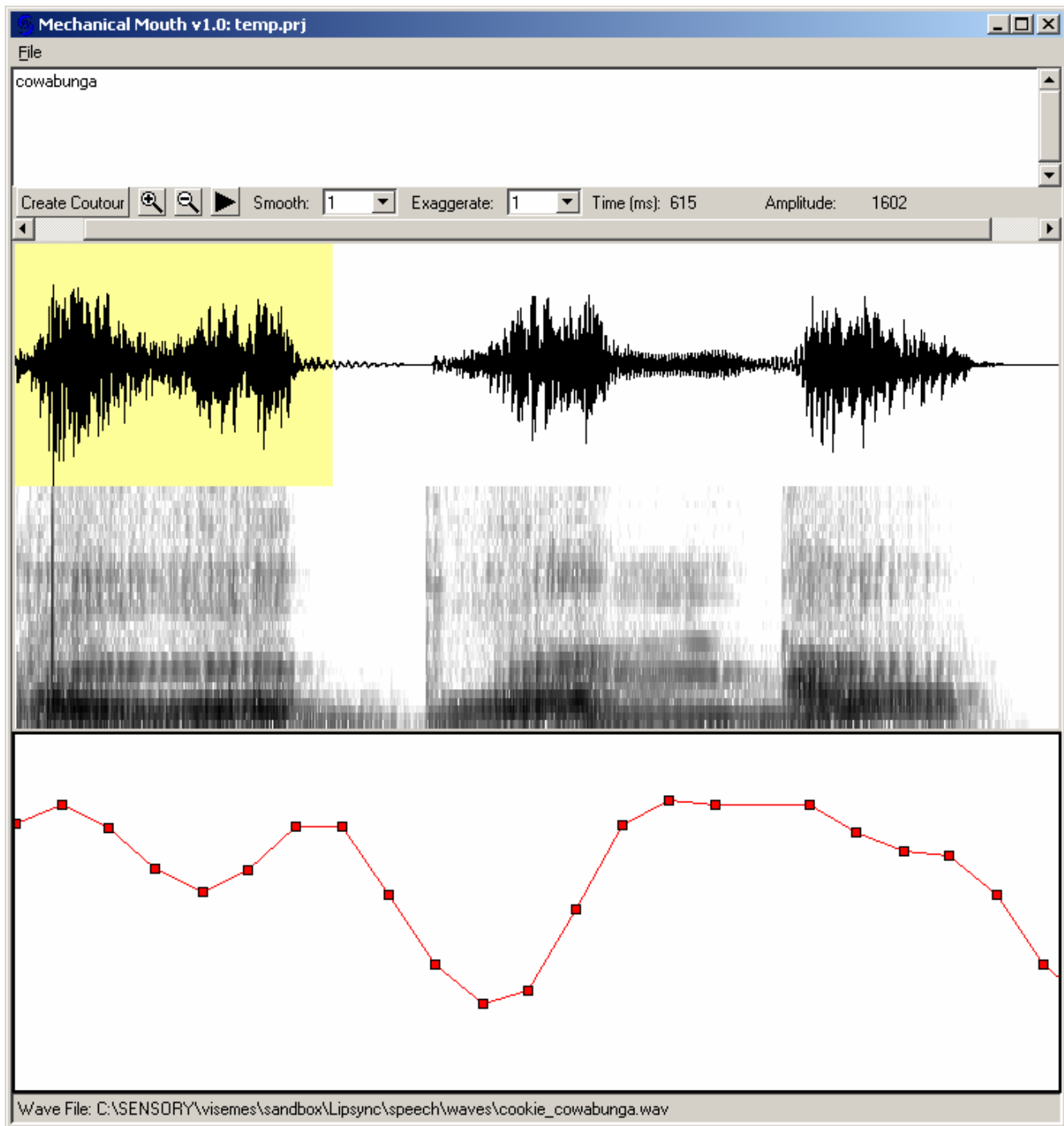
1B. Adding text for improved LipSync accuracy.

The accuracy of the LipSync data will be greatly improved by including the text of the synthesis utterance. So, we now add text to the LipSync tool editor window, and press the “Create Contour” window.



Note how the LipSync data set display (the red line) has changed. Play the wave file again, and watch the animation. You should note that the mouth movement is better synchronized with the utterance.

If you left “click and drag” on a section of the wave display, note that that section is highlighted. If you right click within that highlighted section, the tool will only play back that section.

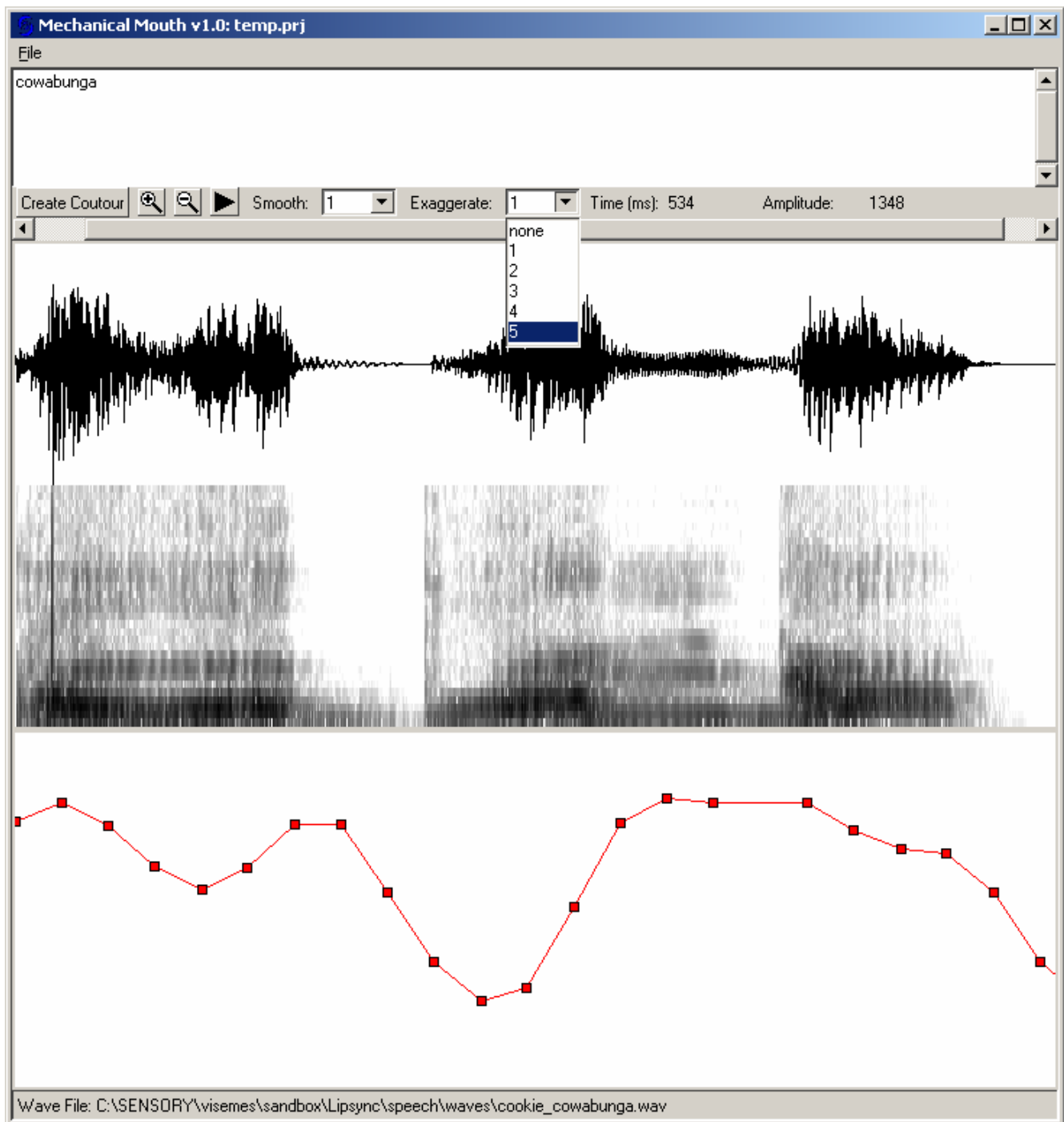


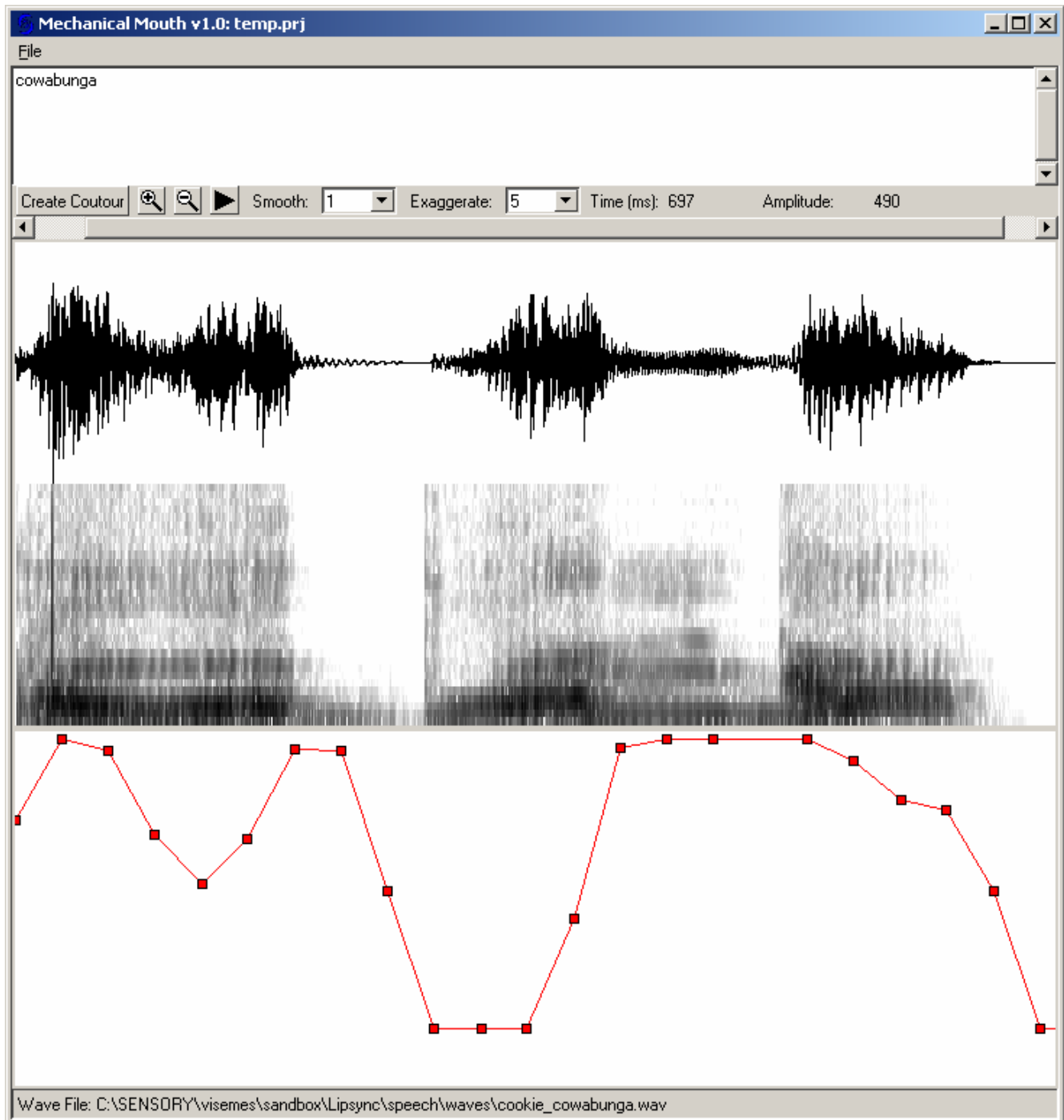
1C. Fine tuning the LipSync data set.

Reasons to fine tune the data set include compensating for slow mechanical systems, or reducing the effect of extraneous noise in the original .WAV recording (in a case where it is not practical to correct the .WAV file).

First, note the “Exaggerate” and “Smooth” pull down menus. The Exaggerate default value is 1, and the Smooth default is 0. This corresponds to a small amount of exaggeration, and no smoothing.

The “Exaggerate” pull down menu adjusts the dynamic range of the data set. This adjustment allows you to compensate for mechanical systems that, for example, don’t open or close the mouth sufficiently for this utterance. The default is 1, but you can either remove exaggeration or set it to a value from 1 to 5. For example, let’s set it to 5 and recreate the contour.

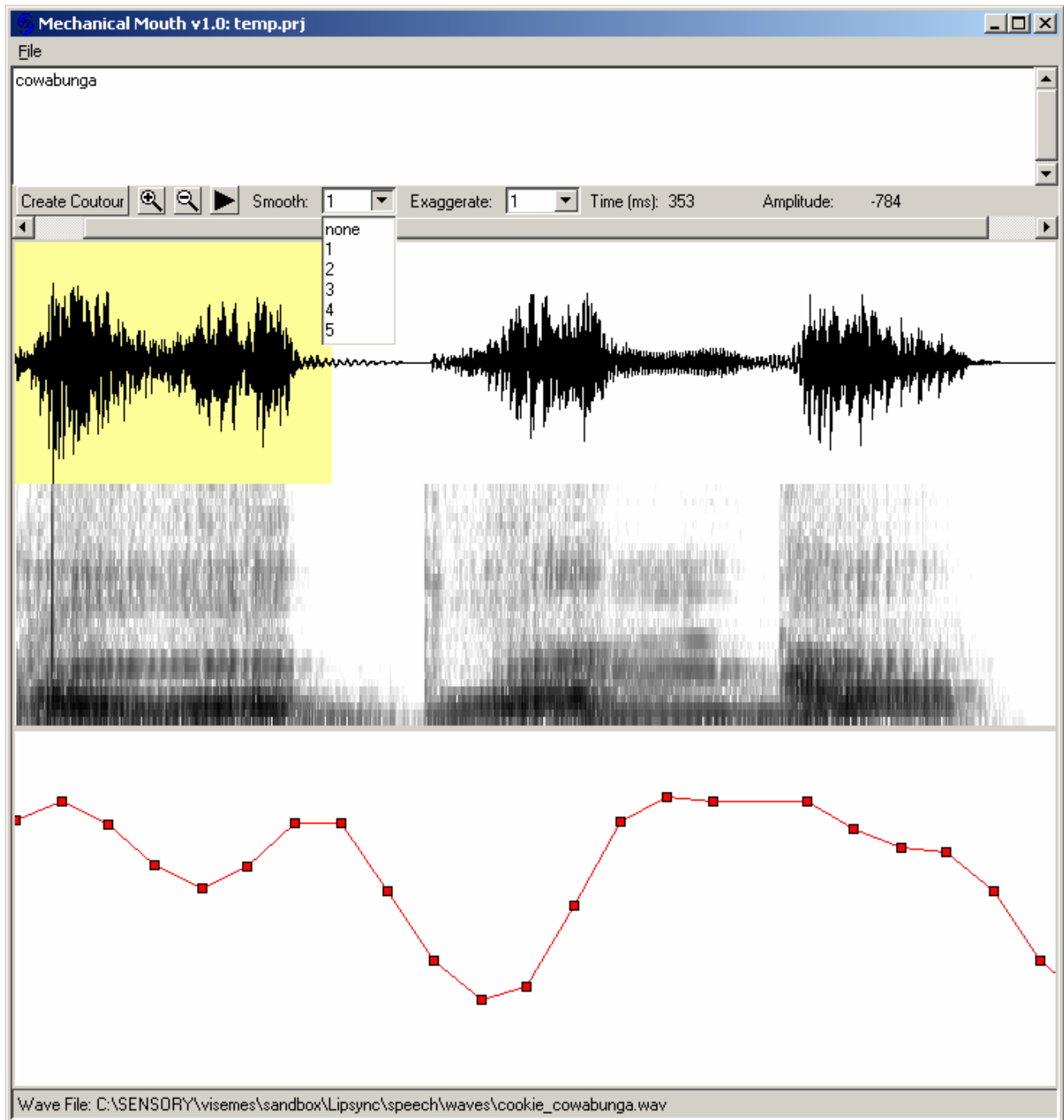




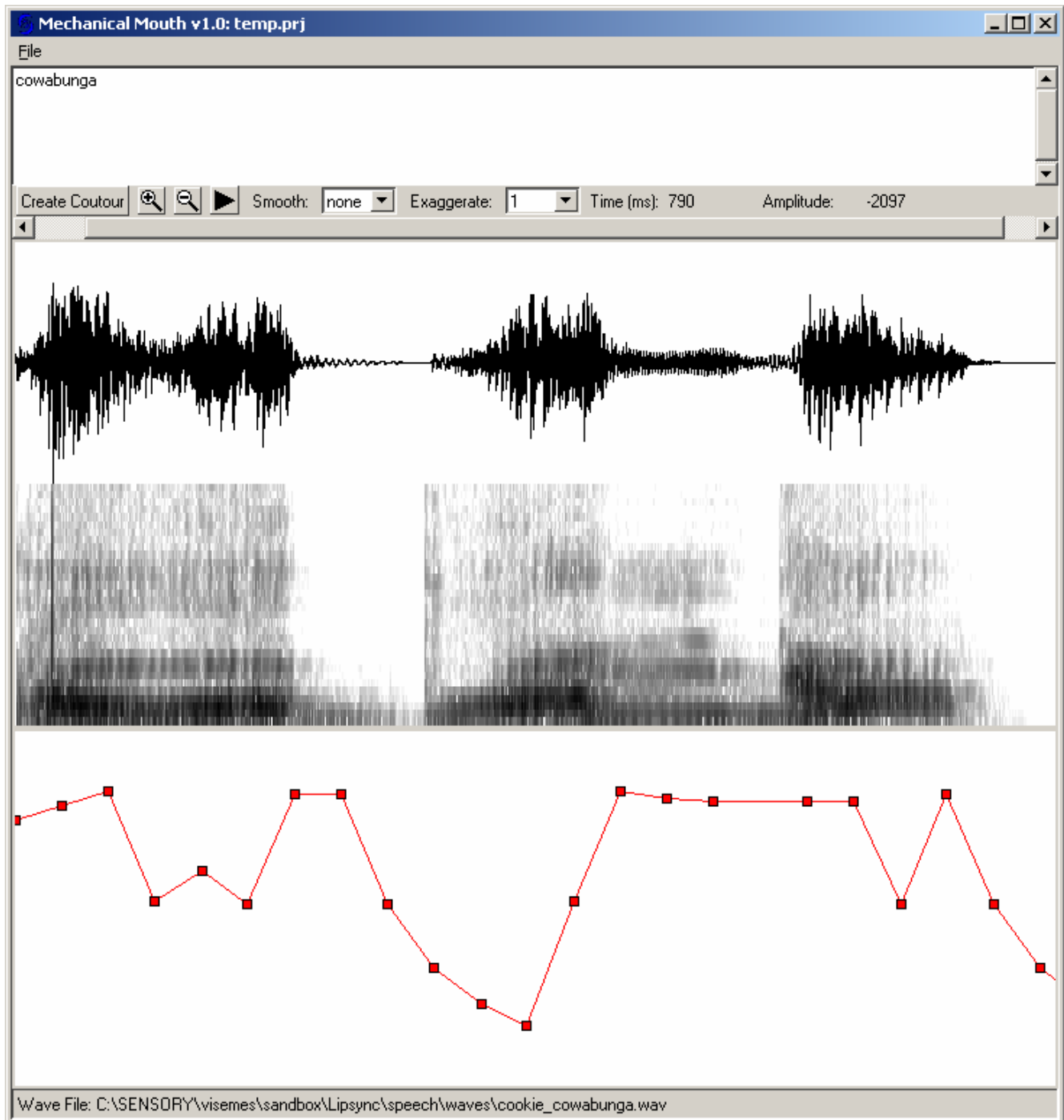
Note that in this example, we are over exaggerating the control, and causing the lipsync data to clip. Some amount of clipping may be acceptable, depending on your mechanical system. You can always use the play button (▶) and the Simulation window to preview the change.

We now reset the exaggeration to the default value of one and discuss the Smooth function.

The Smooth function is a low pass filter. This reduces rapid, fine detail changes in the lipsync data. If your mechanical system is very responsive, this fine detail can make the lipsync appear jittery. The default is none (no smoothing), but you can set it to a value from 1 to 5.



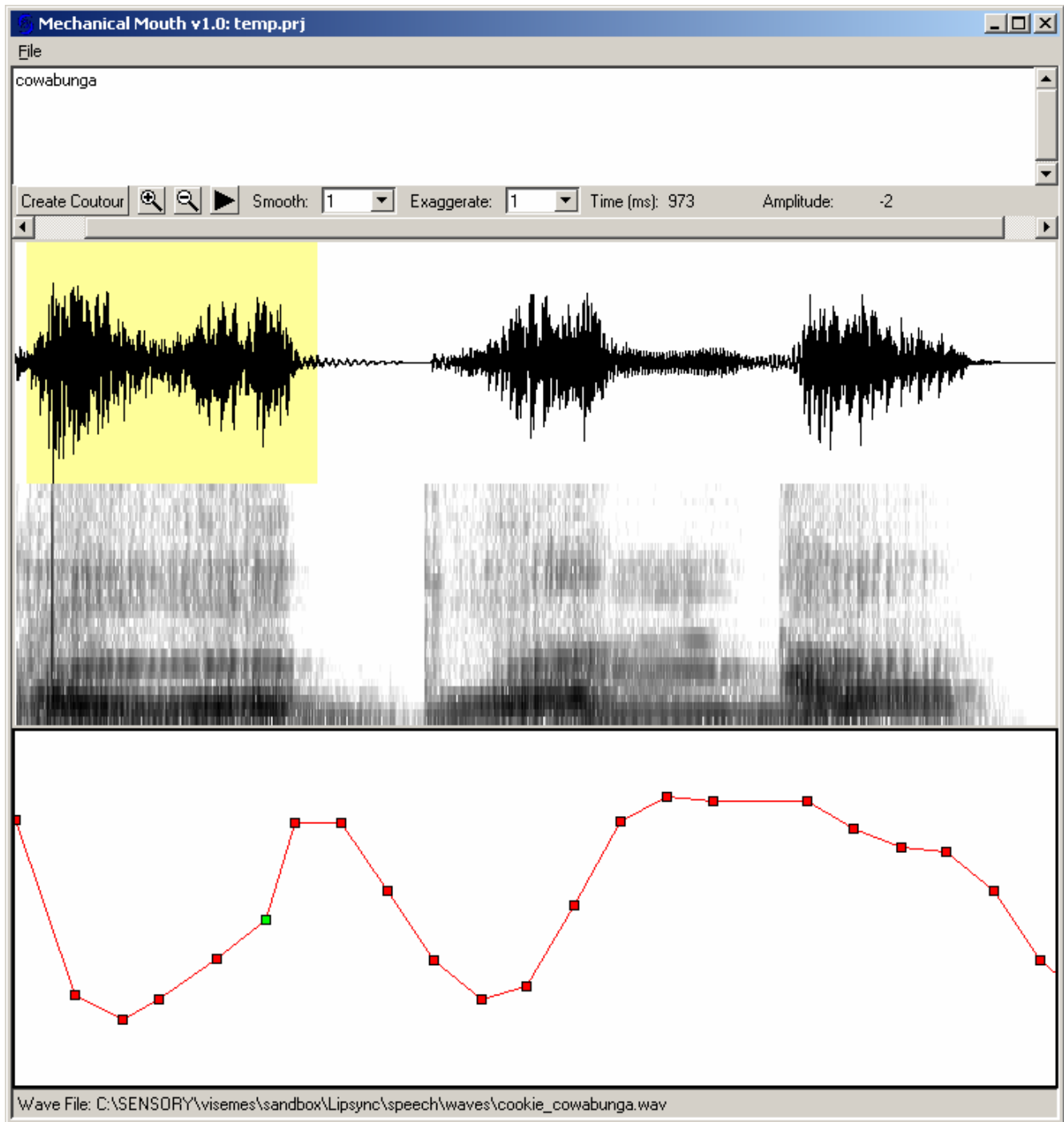
In the above example, smoothing had been set to one. If you now set the value to “none” (which removes filtering) and recreate the contour, note the change to the lipsync data set:



Note that the Smooth and Exaggerate functions work on the entire utterance.

If, for some reason, you need to manually edit how much the mouth opens on just part of the utterance, you can drag the LipSync tool's edit points. These are the small squares on the red LipSync data curve. Click on a point and drag it up for a greater mouth opening, and down for a smaller mouth opening.

In the following example, note that we have reduced how much the mouth opens in part of the "cowa" (after setting Smooth and Exaggerate to "1"). If you now play the sound, you will see that the animation reflects the changes.



To undo your manual edits, just press “Create Contour”, making sure that the Smooth and Exaggerate values are where you want them.

To save your changes, click the File:Save menu.

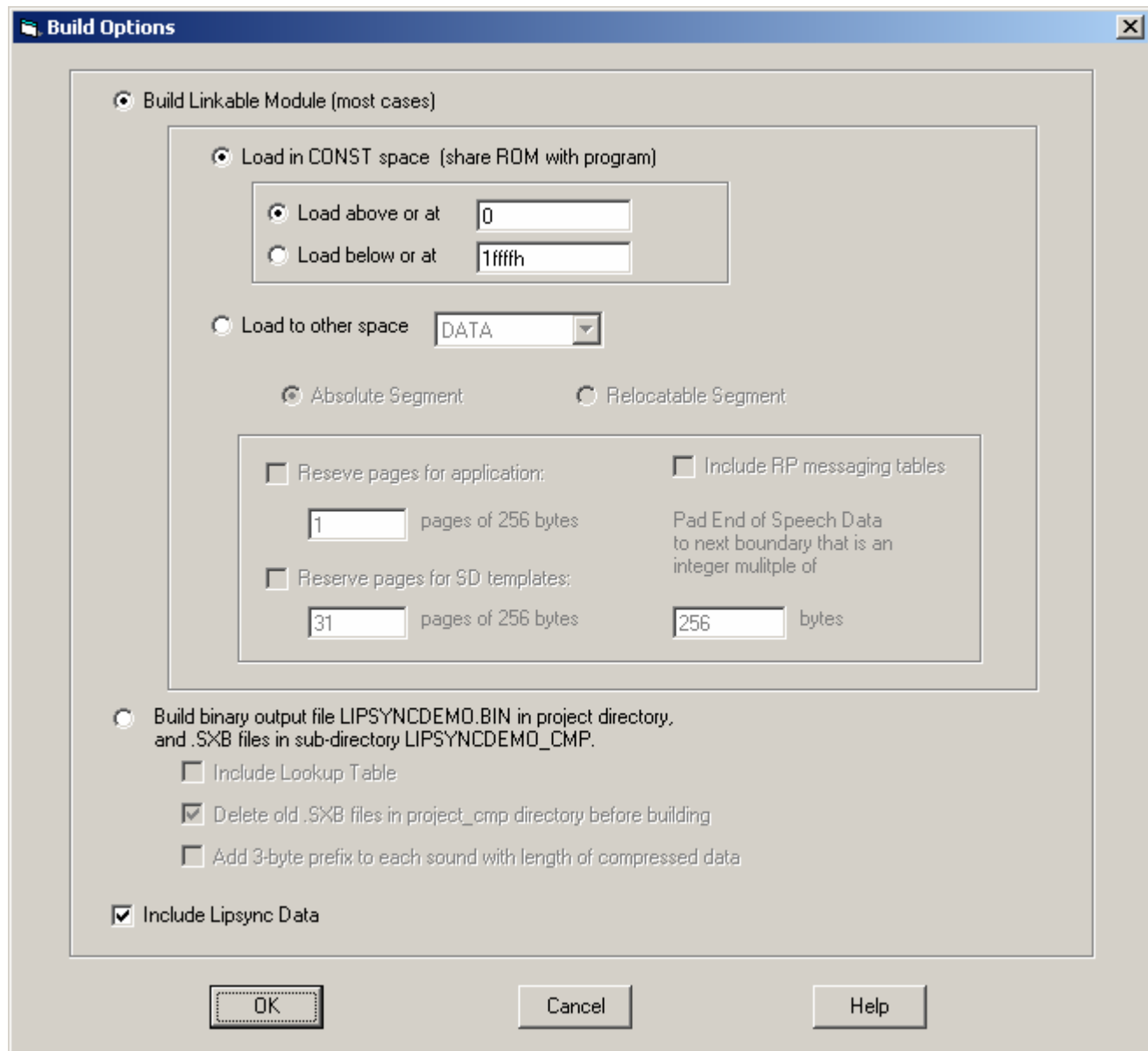
When you are done editing this LipSync tool set, exit the tool (File:Exit or close button).

A query window will ask if you want to save your changes.

You will now be back in the main QS4 window. In the LipSync menu, you now have the options to regenerate the original LipSync tool, reedit your file, rerun the animation, or delete your LipSync tool data.

If you choose to retain your data, the LipSync tool information will be included in the SX file when you rebuild the QS4 project. Note the face icon added to the left of the synthesis' label column. You must always rebuild the project in order to update the edited data.

When you rebuild, the “Build Options” window opens. Make sure the “Include Lipsync Data” box is checked.



2. Application code

A LipSync application includes a main program which plays SX phrases, as well as a customized SX handler and a Timer3 Interrupt Service routine. `_PlaySnd` calls out to the SX handler every 33 milliseconds to read the LipSync data and to set up the values needed to drive the mechanical mouth with a software PWM. Timer3 is used to keep track of time and to implement the duty cycle that drives the mouth to the specified amplitude for the specified time. A byte of flags is used to coordinate the main code, the SX handler and the Timer3 Interrupt Service Routine. The main code is very straight-forward, with more complicated logic in the SX handler and the Timer ISR. All of these pieces are in the application code and are described in detail below, using the C-version of the FluentChip LipSync sample. The assembly version of the samples implements the same algorithm. An actual application should start by copying the sample code, then customizing it, if needed.

2A. Main program

The main program needs to provide the following definitions, perform initialization and then play SX phrases. Note that the definitions of `MouthOpen` and `MouthClose` might change depending on the application. In the sample code, the code uses the same IO pin for the mouth as for the red LED.

```
//macros
#define MouthOpen RedOn // NOTE: Red LED is PWM output for mouth
#define MouthClose RedOff

//global variables
uchar itemCtr; // frames/item counter used by Timer3
uchar frameCtr; // frameWidth counter used by Timer3
uchar pulseCtr; // pulseWidth counter used by Timer3

uchar pulseWidth; // duty cycle = pulseWidth(Varies)/frameWidth(fixed)
uchar pulseReload; // reload values calculated by handler
uchar itemReload; // and loaded by timer3ISR at end of sync item

uchar scale; // scale for voltage compensation (nominal = 100)
uchar lsbits=0; // bits to coordinate SX handler and Timer3ISR:
#define InitialLoad 0x01 // 1 for beginning of sync data
#define ReloadNeeded 0x02 // 1 if new sync item should be read
#define Syncing 0x04 // 1 if syncing with speech segment
#define LastItemRead 0x08 // 1 if last sync item has been read

#define FRAME_WIDTH 40 // fixed frame width in pulses
#define TIMER_KHZ 2 // adjustment for 2Khz
#define FRAME_TIME (FRAME_WIDTH/TIMER_KHZ) // fixed frame width in msecs

//-----
// When started, this program just plays a few standard phrases with LipSync,
// When finished talking, it goes to sleep and a reset is needed to wake it up.
// Pressing Button A stops speech and LipSync and puts the RSC in sleep mode.
void main(void) {
    Timer3Init(); // set up Timer3 to implement a PWM
    VoltageCompInit(); // set up comparator for VBAT (see discussion below)
    Speak(SND_Cowboys); // LipSync a sentence of 3 phrases
    Speak(SND_Jwayne); // LipSync a single phrase
    GoToSleep();
}

void GoToSleep(void) {
    MouthClose;
    _Sleep(0,0,0); // enter low power sleep state
}

//-----
// Initialize LipSyncing, then speak a phrase, allowing abort.
void Speak(uint sound) {
    lsbits = InitialLoad; // Tell SX handler to load initial values
    scale = VoltageComp(); // Use battery voltage to scale PWM output

    if (_PlaySnd(sound, (long)&SNDTBL_LIPSYNCDemo, SX_FULL_VOL))
        GoToSleep(); // Abort if Button A pressed
    lsbits &= ~Syncing; // Turn off LipSync
}
```

2B. SX callout handler

During speech, `_PlaySnd()` makes a callout to `_SxTalkHandler()` approximately every 33 msec. The handler file, `sx_hc.c`, is provided in the FC source code folder and also in `csamples/lipsync` folder. The standard `_SxTalkHandler()` just checks for button abort. The customized LipSync sample handler also calls the routine `LipSyncAction()`, defined in `lipsync.c`, to coordinate the LipSync.

`LipSyncAction()` extracts the LipSync tool information from the SX data files by calling the `_SxGetSyncItem()` function. The details are described below:

```
// This routine is called by the SX Handler to read the sync data and
// calculate the values needed by the Timer3 ISR.
//
// Each sync item consists of:
//   uint time = cumulative time in msec into the phrase for this item
//   uchar amp = 0:100 = % open for the mouth at this time (duty cycle for a PWM)
// End of sync data is signified by time = 0xffff
// NOTE: In sentence files, there will be intermediate EOFs between phrases
// NOTE: If speech files don't have sync data, _SxGetSyncItem returns 0
//
// The mouth is driven by a PWM implemented with Timer3 using a fixed frame width.
// The number of frames per sync item (itemCtr) depends on the time
// between two sync items (typically ~41ms, so 2 repetitions @2KHz, 40ms frames)
// and the pulseWidth comes from the amp stored in the sync item (scaled).
// Since deltaTime is not usually an even multiple of frameWidth, rem is used
// to accumulate the remaining unsynced time; when it gets large enough an extra
// frame is generated to get back in sync.
//
// The handler reads ahead in order to have the next set of reload values ready
// when Timer3 needs them; the reloadNeeded flag is used to coordinate this timing.

void LipSyncAction(void)
{
static uint lastTime; // previous time from _SxGetSyncItem
static uchar lastAmp; // previous amp from _SxGetSyncItem
static uchar rem; // accumulates the unsynced time for items
uint time; // current time from _SxGetSyncItem
uint deltaTime; // keep this as uint for calculations
uchar amp; // current amp from _SxGetSyncItem

// On initialLoad, get the first two sync data items and
// calculate pulseWidth based on a scaled amp and
// itemCtr and rem based on current time - last time (in msec)
// Exit without syncing if no sync data or end of sync data
if (IsBits & InitialLoad)
{
// Read the first and second items to calculate initial PWM values
if (_SxGetSyncItem(&lastTime, &lastAmp) && (lastTime!=0xffff))
{
if (_SxGetSyncItem(&time, &amp) && (time!=0xffff))
{
pulseWidth = (uchar)(FRAME_WIDTH * lastAmp / scale);
deltaTime = time - lastTime;
itemCtr = (uchar)(deltaTime)/FRAME_TIME;
rem = deltaTime - (itemCtr * FRAME_TIME);

// load these into variables that timer3 will count down
frameCtr = FRAME_WIDTH;
pulseCtr = pulseWidth;

// save current values as previous values
lastAmp = amp;
lastTime = time;

// duplicate as the initial reload values and start syncing
itemReload = itemCtr;
pulseReload = pulseWidth;
IsBits = ReloadNeeded + Syncing;
}
}
}
}
```

```

// After initialLoad, read next item as soon as timer3 has loaded current values
// (timer3 signals this with reloadNeeded). Signal EOF with the lastItemRead flag.
// Calculate the reload values for pulseWidth and itemCtr
else if (ISBits & ReloadNeeded)
{
    uchar tempPulseWidth;
    uchar tempItemCtr;

    // if final item has already been read, no more reloads
    if (lastTime == 0xffff)
        ISBits |= (LastItemRead + InitialLoad);
    else
    {
        // read next item
        _SxGetSyncItem(&time, &amp;);
        if (time==0xffff) { // i.e. reached end of all items
            // generate one final PWM at final amp
            tempPulseWidth = (uchar)(FRAME_WIDTH * lastAmp / scale);
            tempItemCtr = 1;
        }
        else {
            // calculate next pulseCtr and itemCtr into temp variables
            // then load them into reload variables with interrupts disabled
            tempPulseWidth = (uchar)(FRAME_WIDTH * lastAmp / scale);
            deltaTime = (time - lastTime);
            tempItemCtr = (uchar)((deltaTime + rem)/FRAME_TIME);
            rem += deltaTime - (tempItemCtr * FRAME_TIME);
        }
        // save current values as previous values
        lastAmp = amp;
        lastTime = time;

        // update reload values with interrupts disabled
        _cli_();
        itemReload = tempItemCtr;
        pulseReload = tempPulseWidth;
        ISBits &= ~ReloadNeeded;
        _sti_();
    }
}
}
}

```

2C. Timer 3 Interrupt Service Routine

The application keeps track of time in order to drive the mouth to the amplitude value for the specified duration. Typically this is done with a Timer3 interrupt based timer running at 2KHz. The sample application assumes that the T3 is only used for LipSync and is initialized at system start.

```
//-----
// Timer3 implements a PWM with a variable pulseWidth and fixed frameWidth.
// NOTE: Running T3 at 1KHz is not fast enough to avoid "jitter" in the
// mechanical mouth. Running at 2KHz avoids this. Running faster caused
// problems in the C version - not enough cycles.

#define T3_MASK 0x10

void Timer3Init(void)
{
    t3ctl = 0x82;          // t3 on, divide by 4
    t3r = 256-56-56;     // setup for 2 KHz
    imr = imr | T3_MASK; // enable t3irq
}
```

The irq handler implements the software PWM control, communicating with the SX handler through flags.

```
//-----
// Timer3 is used to implement a PWM based on the following variables which
// have been set up by the SX handler:
//   frameWidth is fixed (40)
//   pulseWidth varies (0:40) for a duty cycle based on the lipsync amp data.
//   itemCtr determines how many frames are needed for the current sync item
//   based on the lipsync time data (typically 2:5)
// Timer3 counts down the following variables, reloading as needed:
//   pulseCtr has pulseWidth counter for PWM (MouthOpen until counted down)
//   frameCtr has frameWidth counter for PWM
//   itemCtr has # of frames per current item
//
// At the end of an item, pulseCtr and itemCtr are reloaded from
// pulseReload and itemReload, which have been setup by the handler.
// As soon as these are reloaded, the ISR sets the ReloadNeeded flag,
// which causes the handler to read the next item
// and recalculate pulseReload and itemReload
// NOTE: Syncing flag says whether lipsyncing is occurring
// LastItemRead flag is clear until handler has read last item of sync data
void T3ISR(void) // 2KHz IRQ
{
    // is sync data setup?
    if (IsBits & Syncing) {
        // are we in the MouthOpen part of the duty cycle?
        if (pulseCtr) { // open mouth until pulseWidth is done
            MouthOpen;
            pulseCtr--; // do this here so pulseCtr doesn't go below 0
        }
        else // pulseWidth is done, close mouth til end of frame
            MouthClose;

        // at end of frameWidth?
        if (--frameCtr == 0) {
            // yes, at end of sync item?
            if (--itemCtr == 0) {
                // yes, at end of all items?
                // if not, replace current values with reload values and
                // signal SX handler to read next sync item
                if (!(IsBits & LastItemRead)) {
                    pulseWidth = pulseReload;
                    itemCtr = itemReload;
                    IsBits |= ReloadNeeded;
                }
                // if so, turn off lipsync
            }
            else {
                IsBits &= ~Syncing;
            }
        }
        // at end of frame, reload with current values
        frameCtr = FRAME_WIDTH;
        pulseCtr = pulseWidth;
    }
}
irq = ~T3_MASK; // clear the interrupt
```

2D. Voltage compensation

The sample includes a voltage compensation scheme using a voltage divider from the motor drive voltage to get a value to compare with the comparator's internal reference. Since the internal reference can be set to 16 different values, it simply steps through the reference values until the comparator output switches, then uses that step as a pointer into a table that sets the scale value for the PWM (used by [LipSyncAction\(\)](#)). Note that the comparator internal reference is taken from the RSC's Vdd. This must be a well regulated supply. In our sample schematic, we show separate battery supplies for RSC's Vdd and the motor drive (Vbat). Alternately, the RSC's Vdd could be a regulated tap from Vbat.

IMPORTANT NOTE: The [ScaleTable\[\]](#) table values and the voltage divider ratio are dependent on the mechanical design and system voltage. It is the developer's responsibility to implement an appropriate [ScaleTable\[\]](#) or to develop their own voltage compensation scheme. NOTE: see the schematic below for the hardware design that used for this software discussion.

```
// Set up comparator to measure VBAT on P2.0, with internal reference
// Call this routine once at initialization
void VoltageCompInit(void) {
    cmpCtl = 3;           // Set up RSC comparator
    p2ctlA &= 0xfe;      // Make P2.0 input, no pull up
    p2ctlB |= 01;
}

//-----
// NOTE: This implementation assumes nominal 6V Vbat and a minimum Vbat of 4.5V.
// It also assumes the voltage divider feeding the p2.0 comparator is a 1:1 ratio.
// Call this routine before speaking each phase, e.g. scale = VoltageComp();
uchar VoltageComp(void) {
    static uchar ScaleTable[] = {40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 50, 60, 70, 80, 90, 100};
    uchar *ptr = ScaleTable;

    // Adjust reference voltage up through 15 steps (after that use 100%)
    // End loop when VBAT is lower than reference voltage
    cmpRef = 0;
    while ((cmpCtl & 0x80) && (cmpRef < 15)) {
        cmpRef++;
        ptr++;
    }
    return *ptr;
}
```

2E. Gain control

Since [scale](#) is a value used to modify the PWM drive strength, it can be considered a gain control. This can come in handy if the normal PWM range is inappropriate for a particular mechanical design, especially if the mechanics are non linear, or do not work well over the full LipSync table's dynamic range.

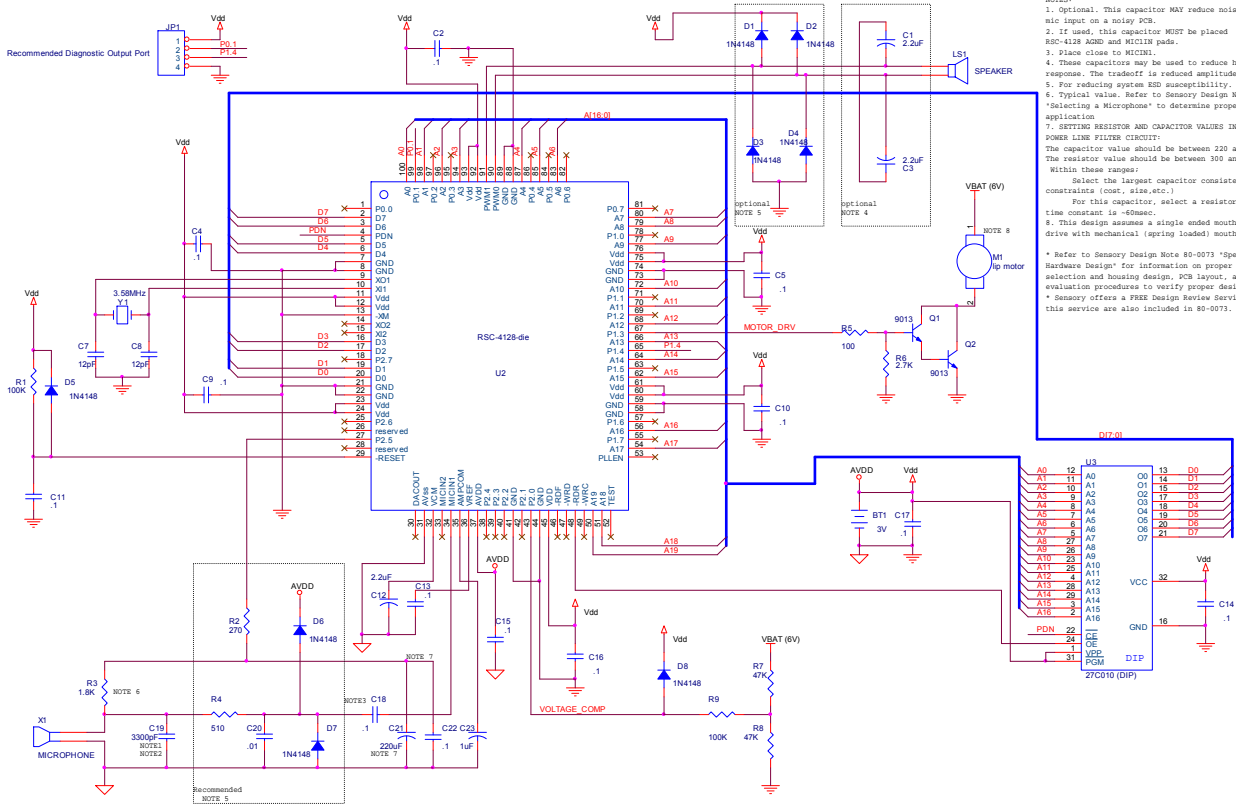
For example, we have seen a mechanical design that bobs the entire head before the mouth movement becomes visible. This can be compensated for by overdriving the PWM. That forces the system more quickly to a point such that that the mouth movement becomes visible. One could accomplish this by changing the [ScaleTable\[\]](#) to a set of smaller values.

Even if the application does not use voltage compensation, it is also possible to change the gain of the system by picking a constant value for [scale](#) that is not 100. Since [scale](#) is a divisor, a smaller value results in a higher gain system.

In this case, the voltage compensation function could be as simple as:

```
uint VoltageComp(void) {
#define SCALE 60
    return SCALE;
}
```

Sample schematic:



- NOTES:
1. Optional. This capacitor MAY reduce noise coupled into the mic input on a noisy PCB.
 2. If used, this capacitor MUST be placed close to the RSC-4128 AGND and MICIN pads.
 3. Place close to MICIN1.
 4. These capacitors may be used to reduce high frequency response. The tradeoff is reduced amplitude.
 5. For reducing system ESD susceptibility.
 6. Typical value. Refer to Sensory Design Note 80-0299 "Selecting a Microphone" to determine proper value for your application.
 7. SETTING RESISTOR AND CAPACITOR VALUES IN THE MICROPHONE POWER LINE FILTER CIRCUIT: The capacitor value should be between 220 and 30 ufd. The resistor value should be between 300 and 2000 ohms. Within these ranges: select the largest capacitor consistent with other constraints (cost, size, etc.) For this capacitor, select a resistor such that the RC time constant is $\le 60\text{ms}$.
 8. This design assumes a single ended mouth opening motor drive with mechanical (spring loaded) mouth closure.
- * Refer to Sensory Design Note 80-0073 "Speech Recognition Hardware Design" for information on proper microphone selection and housing design, PCB layout, as well as test and evaluation procedures to verify proper design and operation.
 * Sensory offers a FREE Design Review Service. Details of this service are also included in 80-0073.

The Interactive Speech™ Product Line

The Interactive Speech line of ICs and software was developed to “bring life to products” through advanced speech recognition and audio technologies. It is designed for cost-sensitive consumer-electronic applications such as home electronics, home automation, toys, and personal communication. The line includes the award-winning RSC-4x family of mixed signal processors and tools, the *VR Stamp™* 40-pin DIP module and tools, and the SC series of speech and music synthesis microcontrollers. It also includes our suite of software development kits, which are designed to run on non-Sensory processors and DSPs and support most popular operating systems.

RSC Microcontrollers and Tools

The RSC product family contains low-cost 8-bit speech-optimized microcontrollers designed for use in consumer electronics. All members of the RSC family are fully integrated and include A/D, pre-amplifier, D/A, ROM, and RAM circuitry. The RSC family can perform a full range of speech/audio functions including speech recognition, speaker verification, speech and music synthesis, and voice recording/playback. The family is supported by a complete suite of evaluation and development toolkits.

Speech Recognition Modules and Tools

The *VR Stamp™* is a complete speech recognition module based on the RSC-4x and is ideal for fast design and easy production. A low-noise audio channel and standardized 40-pin DIP footprint allow rapid prototyping, less debugging, and shorter time to market. The *VR Stamp Toolkit* includes everything needed to get started today, including VR Stamps, Module Programming Board, sample applications, and a complete set of development tools featuring the Phyton IDE and limited-life C compiler, QuickSynthesis™ 4 and Quick T2SI-Lite™ speech tools.

SC Microcontrollers and Tools

The SC-6x product family features the highest quality speech synthesis ICs at the lowest data rate in the industry. The line includes a 12.32 MIPS processor for high-quality, low data-rate speech compression and MIDI music synthesis, with plenty of power left over for other processing and control functions. Members of the SC-6x line can store as much as 37 minutes of speech on-chip and include as many as 64 I/O pins for external interfacing. Integrating this broad range of features into a single chip enables developers to create products with high quality, long duration speech at very competitive price points.

FluentSoft™ Technology

FluentSoft™ Recognizer is the engine powering the FluentSoft™ SDK. It provides a noise-robust, large-vocabulary, speaker-independent solution with continuous digit recognition and word-spotting capabilities. This small-footprint software recognizes up to 5,000 words; runs on non-Sensory processors including Intel XScale, TI OMAP, and ARM9 platforms; and supports operating systems such as MS Windows, Linux, and Symbian.

3Dmsg™ Technology

3Dmsg's (www.3Dmsg.com) Animated Speech technology offers animated avatars with advanced speech recognition and synthesis capabilities for use in smartphones, language trainers, and kiosk applications. Facial expressions can be configured to show emotions and lip synchronization can be automatically driven from voice or text data.

Important notices:

Sensory Incorporated (Sensory, Inc.) reserves the right to make changes, without notice, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Sensory, Inc. assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. Sensory, Inc. makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Safety Policy:

Sensory, Inc. products are not designed for use in any systems where malfunction of a Sensory, Inc. product can reasonably be expected to result in a personal injury, including but not limited to life support appliances and devices. Sensory, Inc. customers using or selling Sensory Incorporated products for use in such applications do so at their own risk and agree to fully indemnify Sensory, Inc. for any damages resulting from such improper use or sale.



S E N S O R Y®

575 N. Pastoria Ave., Sunnyvale, CA 94085
Tel: (408) 625-3300 Fax: (408) 625-3350

© 2007 SENSORY, INC. ALL RIGHTS RESERVED.
Sensory is registered by the U.S. Patent and Trademark Office.
All other trademarks or registered trademarks are the property of
their respective owners.

www.sensoryinc.com