Column #108 April 2004 by Jon Williams:

Speak the Speech

"Speak the speech, I pray you..." starts Shakespeare's famous instruction to the actor. The essence of this admonition is for the actor to speak truthfully, easily, without fabrication or extended effort. This is important to me because, as many of you know, I lead two lives: one as a happy-go-lucky Parallax employee, the other as a professional actor. What does this have to do with BASIC Stamps? Have faith, friend, this is my cheesy introductory text and you know I'll get there!

Nearly every actor goes through a stage where nothing that comes out of our mouths sounds right. Believe it or not, it takes a lot of work to sound completely natural speaking words written by somebody else – especially in the surreal atmosphere of a stage or film set. The challenge is elevated for the film and television actor where conversations are rarely shot in a single continuous take.

So where am I going with this? Just as the actor struggles, frequently we techno-types struggle when adding speech to our electronic projects. Sure, there are lots of neat products out there, but most (allophone based) are more difficult to use than the quality of their output warrants. It may take an hour to string together the right collection of allophones to get decent speech, and still, it usually sounds very unnatural. Yes, we ultimately get there, but man is it a struggle.

Enter Winbond. The company responsible for the ChipCorder® products has created a true Text-To-Speech product called the WTS701 that makes converting plain English text into high-quality spoken speech fairly straightforward. Okay, "fairly straightforward" is a relative term – and the WTS701 itself is a bit tricky. The device contains a rules processor for handling English text and a memory that consists of actual speech fragments that – when strung together properly – produce surprisingly pleasing female speech. To my ear, the voice is a bit like the voice of "Mother" from the movie *Alien*.

The Emic TTS

Figure 108.1: Grand Idea Studios Text to Speech Module

If you happen to go to the Winbond site (www.isd.com) and get the datasheet for the WTS701, you'll probably say "Uh, oh..." out loud. Don't worry. A southern California company called Grand Idea Studio has created a product called the Emic Text-To-Speech Platform (Emic TTS) that shields us from the complexities of the WTS701, yet gives us access to its impressive features.

The Emic TTS comes in two flavors: an OEM version and prototyping-friendly SIP version. We're going to be using the SIP version here because it will plug right into a solderless breadboard, and it includes an onboard 300 mW amplifier – all we have to do is connect an 8-ohm speaker. The SIP version also allows us to route external audio (i.e., Stamp-generated sound effects) through the WTS701 and to the audio amplifier.

Another nice feature of the SIP version is that is has the same pin-out as the Quadravox QV306 modules. So, if you have a project that is using the QV306 with prerecorded speech, you can swap in the Emic TTS and update your code for direct text-to-speech output. There are advantages to both modules, and I like the ability to move back-and-forth between them.

Just Say It, Please....

What makes the Emic TTS so much fun to use is that it's just plain easy to make our project talk. For example:

```
  SEROUT Tx, Baud, [Say, "Nuts & Volts rocks!", EOM]
```

How easy as that? All we have to do is send our text through a serial connection. As you'll see in the demo program, *Say* is the command to speak the text that follows, and *EOM* is the end-of-message marker. When active, the Emic TTS will tell us by lighting a red LED and setting the Busy output high. When the current phrase is complete and we can send a new one, the LED will change to green and Busy will go low.

Before we get to the code, let me discuss the two operational modes supported by the Emic so you understand why I wrote the program the way I did.

There are a couple of configuration switches on the Emic TTS. SW1 sets the command mode for the device. When SW1 is on, the Emic TTS expects commands in text mode. In this mode, the command above would look like this:

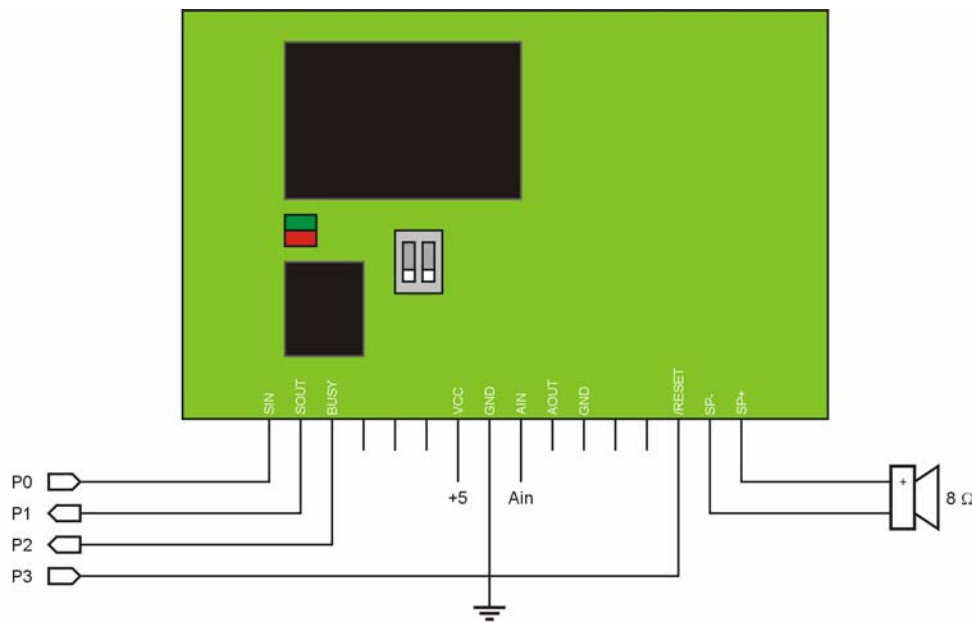
```
SEROUT Tx, Baud, ["say=Nuts & Volts rocks!;"]
```

This mode is very useful if you've got the Emic connected to a terminal program, but consumes a lot of program space when used in an embedded micro. So, we're going to set SW1 to off which puts the Emic into hex mode. In this mode, "say=" (four bytes) is replaced with $00 (one byte) so we will ultimately save code space. To make our program easy to read, we'll create a constant called *Say* that has a value of $00. What we get is conservation of code space without sacrificing the ability to read and understand the program.

The second switch, SW2, is used to select/deselect character echo. When on, SW2 will cause every character transmitted to the Emic TTS to be echoed back. Again, this is more useful when connected to a terminal than embedded micro. When SW2 is set to off, we don't get the echo, but we do still get status and other important information from the Emic TTS.

Okay, let's get to it. Grab your Emic TTS (SIP version) and plunk it into a breadboard. The circuit is straightforward and will only take a few minutes to connect.

Figure 108.2: TTS to BASIC Stamp Connections



Stamp-based Chatterbox

The purpose of our program this month is to put the Emic TTS through its paces so we can make decisions about text, volume, speed, and pitch before installing it into that currently-silent project that's just aching for a voice. The program as written will work on any BS2-family. Since the communication rate between the host processor and the Emic is a tame 2400 baud, you can even connect it to a BS1 – though you'll probably want to minimize the connections. Check the Parallax web site for BS1 and Javelin samples if those micros interest you.

Okay, here's what our program is going to do:

- Reset the Emic TTS
- Display a menu
- Accept and validate user input
- Run the selected demo item

The first thing we're going to do is reset the Emic TTS so that we can start in a known state. There are two ways to do this: hard and soft. Doing a hard reset requires an external control line. If you have a project that is short on IO, you can let the Emic RST\ line float and do a soft reset through the serial link. The only downside of the soft reset process is we have to wait if the Emic is busy. Here's our code for a hard reset and to preset a couple program variables to the Emic TTS defaults.

```
Hard_Reset:
  LOW Rst
  PAUSE 0
  INPUT Rst
  GOSUB Wait_OK
  vol = 4
  spd = 2
  ptch = 1
  RETURN
```

The first four lines handle the reset and confirmation process. As you can see, we just need to pull the RST\ line low briefly, then release it to the onboard pull-up. When the Emic TTS resets it will send an "okay" signal through the serial line. We'll need to watch for this from time-to-time, so waiting for that signal is handled in its own routine.

```
Wait_OK:
  SERIN  RX, Baud, 1000, TO_Error, [WAIT(OK)]
  RETURN
```

There's no magic here – we're just waiting for the *OK* byte. It should come right away, and if it doesn't then we can jump out to another routine (TO_Error) to deal with the lack of response. We might, for example, construct a robot program that works fine with or without the Emic TTS. If we don't get the *OK* signal after the hard reset we'll know the board is not installed and our code will proceed accordingly.

But we do have a board installed, so let's go back to the program. The next stage is presenting a menu and processing the user input. We don't frequently need menu programs in embedded controllers, but when we do it's nice to be able to handle them effectively – and PBASIC gives us some neat tools that simplify input processing.

```
Main:
  DEBUG CLS,
        "Emic TTS Demo Menu", CR,
        "------------------", CR,
        "[V] Set Volume (", DEC1 vol, ")", CR,
        "[S] Set Speed (", DEC1 spd, ")", CR,
        "[P] Set Pitch (", DEC1 ptch, ")", CR,
        CR,
        "[1] Demo 1", CR,
        "[2] Demo 2", CR,
        "[3] Sound Effects (uses Ain)", CR,
        CR,
        "[A] Use Abbreviation", CR,
        "[J] Japanese (phonetic demo)", CR,
        CR,
        ">> "
```

The menu is simply several lines of text pumped out the programming port to the **DEBUG** window. Notice that the program variables for current volume, speed, and pitch are displayed so we can take note of our results when we get what we like.

The next step is command input and validation. Let's look at the code, then go through it line-by-line:

```
   DEBUGIN cmd
  LOOKDOWN cmd, ["vVsSpP112233aAjJ"], cmd
  cmd = cmd / 2
  IF (cmd > 7) THEN Main

  BRANCH cmd, [Set_Volume, Set_Speed, Set_Pitch,
               Play_Msg, Play_Msg, Play_SFX,
               Play_Msg, Ph_Demo]
```

The first thing we have to do is get a key from the user, so we do this with **DEBUGIN**. Since our input section is a single byte variable, we'll end up with a single key input. The next line is where the PBASIC magic takes place. **LOOKDOWN** is used to scan a table for the input variable, and if that value is found in the table the position will be reported in the output variable. As you can see, we've used the same input and output variable with **LOOKDOWN**, so what this does is convert the input key to its position in the table. What happens if the entry is not in the table? Nothing; the output variable will not be changed.

Take a look at the table and you'll see that each key is covered by two characters. This allows us to be user-friendly and treat lower- and upper-case letters in an intelligent manner. For the numeric characters, which have no case, we need to enter them twice. Let's go through an actual input to see why.

When we press the letter "P" on our keyboard **DEBUGIN** will put "P" into the variable *cmd*. **LOOKDOWN** takes *cmd* and hunts for it in the table and finds it in position five (the first position in the table is zero). Since "P" was found in the table, the output variable, *cmd,* will now hold five. The next line deals with our two-key situation by dividing the raw position value by two, and now *cmd* holds two (If we start counting from zero, we'll see that "P" is in position two of our menu). The next step is validation; we need to make sure the entry is in range. If yes, *cmd* will be passed on to **BRANCH** to run the selected code, otherwise the program redraws the menu and waits for another input.

If you haven't used **LOOKDOWN** before you may be wondering about the duplicated entries. What happens is that the first position is put into the output variable. So if we press "1" on the keyboard we will end up with a value of six in *cmd*. The second entry of each number is needed to correctly position the keys that follow, since the output position must be divided by two to correct the entry.

The first three items on our menu allow us to modify the sound of the Emic TTS speech by changing the volume, speed, and pitch. Since the code for each of these entries is identical, we'll just go through the first of them.

```
Set_Volume:
  DEBUG CLS, "Enter Volume (0 - 7): "
  DEBUGIN DEC1 response
  vol = response MAX 7
  SEROUT TX, Baud, [Volume, DEC1 vol, EOM]
  GOSUB Wait_OK
  GOTO Main
```

As you can see, we clear the screen, display a prompt, and then wait for a key. To help filter the input we use the DEC1 modifier. This will allow just one key, and force it to be "0" through "9." Like our other inputs, we have to validate it before moving on. In this case we're going to use **MAX** to make sure we don't send an illegal volume level (of 8 or 9) to the Emic. Then we send it with **SEROUT**. One thing to note is that we actually have to send the volume level as text, not as a numeric value. There's no problem here, we use DEC1 again and that will convert the numeric volume level back an ASCII character.

Let's have the Stamp say something, shall we? You've already seen how easy it is, and what we're going to do is change the code a bit so that we can store our text strings in **DATA** statements. This will allow us to call a single routine to speak any number of text phrases we want to say. This will also cut down on the number of **SEROUT** instructions. As I've told you before, **SEROUT** is a bit complex and consumes code space, so minimizing the number of **SEROUT** instructions will give us more room for operational code. If, for example, our robot has 50 different things it might want to say, we can cut down the number of **SEROUT**s from 50 to just one to say any of those phrases. Of course, this takes a little planning:

```
Say_String:
  DO
    READ eePntr, char
    SEROUT TX, Baud, [char]
    eePntr = eePntr + 1
  LOOP UNTIL (char = EOM)
  RETURN
```

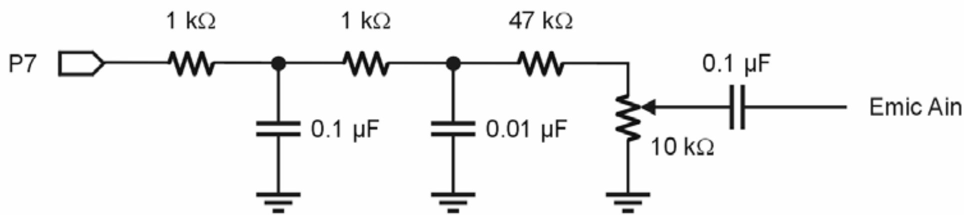
The Say_String code is really easy. The program will pass the first position of the string to the subroutine and the code will loop through, sending each character to the Emic until it hits the *EOM*. A no-brainer, right? You're right, it is. Here's what a stored phrase looks like:

```
Msg1            DATA    Say, "Nuts & Volts rocks!", EOM
```

Prefixing each phrase with the *Say* code may seem redundant, and in programs that use a lot of strings it just might be. For most, however, this is probably the most memory-efficient way to store our speech phrases.

Okay, before we wrap up let's cover a couple extra features that will come in handy. The first is the ability to amplify an external audio signal with the Emic TTS amplifier. To do this we have to enable the Ain pin, then we can apply a line-level audio signal. Keep in mind that so long as the Ain line is active, we can't send speech strings to the Emic (standard speech and external audio are mutually exclusive). Another thing to note is that the Emic TTS volume level does not affect the external audio signal.

Figure 108.3: Volume and Filter Circuit

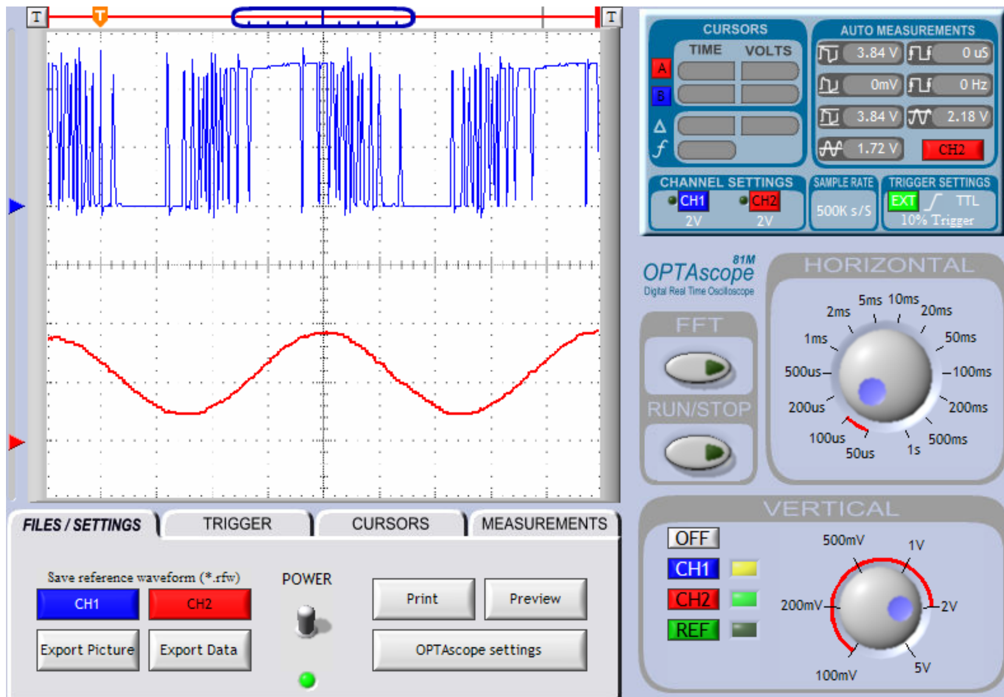


The circuit in Figure 108.3 serves two functions: first, it filters the digital output of the Stamp's **FREQOUT** and **DTMFOUT** instructions into a nice sine wave suitable for amplification and second, it attenuates the signal down to a manageable level for the Emic TTS. The 10K pot lets us set the level of Stamp-generated audio.

In Figure 108.4 you can see the effect of this circuit. The top trace is the digital output from the BASIC Stamp. The lower trace is the filtered signal. As you can see, the filter output is a nice clean sine wave that is free of unwanted harmonics and will amplify cleanly.

When you play the sound effects demo you'll hear the Emic TTS say, "Dialing 1-916-624-8333" and follow it with the DTMF tones of the phone number. A simple bit of code allows us to pass the telephone number to the Say_String and DTMF dialing subroutines with the expected results from each. Neato.

Figure 108.4: Digital Output and Filtered Circuit




```
Dial_Phone:
  DO
    READ eePntr, char
    IF (char >= "0") AND (char <= "9") THEN
      DTMFOUT AOut, 200 */ TmAdj, 50, [char - "0"]
    ENDIF
    eePntr = eePntr + 1
  LOOP UNTIL (char = EOM)
  RETURN
```

The only thing of significance in the Dial_Phone routine is the use of a constant called TmAdj. This value is used to scale the DTMF on-time for the Stamp is use, so any Stamp that runs the program will output a 200 millisecond DTMF tone. A conditional compilation section at the top of the program takes care of setting the TmAdj value for the specific Stamp.

And finally ... there will be times when the English rules engine of the WTS701 doesn't quite meet our requirements for a given word. What we can do on those infrequent occasions is pass the word phonetically, just like we used to do with the old SP0256-AL2 and similar devices.

As an example, I wanted my Emic TTS to say "Hello" in Japanese. The Romanized spelling is konnichiwa – pronounced *cone-nee-chee-wah*. This word is stored in our program like this:

```
Nihongo         DATA     Say, PhT, "konniCiwa ", EOM
```

The *PhT* byte tells the WTS701 that what follows is phonetic text and to use those rules until a space is encountered. Note that each phonetic sound is represented by a single letter, hence the *ch* sound is represented by the single letter capital C. We can also tell the WTS701 which syllable gets stress by preceding the vowel in that syllable with a 1. Japanese is fairly evenly stressed, so we don't need it here. You won't need to do phonetic spelling very often, but it's nice to have for those out-of-the-ordinary words.

Abbreviated Flash

Oops, I wasn't quite done. There is a final feature I want to share with you – and along with it comes a big caution. The WTS701 has the ability to store abbreviations; like this:

```
   SEROUT Tx, Baud, [AddAbbr, "dC,degrees Celsius", EOM]
```

After this command we can have the Emic say "degrees Celsius" by telling it to say "dC." Now, here's the caveat: when we store an abbreviation it consumes a bit of the WTS701 flash. And even though there is a "delete abbreviation" command, it doesn't free the flash, it simply marks the abbreviation as not used. Even worse, if we tell it to store the same abbreviation more than once then we just lose flash – the WTS does not overwrite the original abbreviation as we would hope. Even if it has multiple copies of the same abbreviation it will use the first. So if you need to correct an abbreviation you have to delete the original. Just remember that you don't recover any flash in the process so be careful. These are "features" of the WTS701, not the Emic TTS module – so if you decide to go with another WTS701-based product these issues are in place.

I learned this the hard way. While working on my demo program for this article – and NOT knowing about the flash thing – I ended up consuming about half the available flash space in my Emic TTS by running the program over and over. I changed the code to use one of the built-in abbreviations so that the feature could be demonstrated.

Don't let this sway you from using custom abbreviations, but do let it encourage you to do so with some planning in mind. And since you can't recover the flash space with the DelAbbr command, don't do it (unless you're making a correction) because you never know when you're going to need that abbreviation in the future. My current strategy for dealing with custom abbreviations is to create a separate program to download them. In this program, I keep track of what abbreviations are currently downloaded to the Emic TTS. Note that if you connect the Emic TTS directly to a terminal program (you'll need a level shifter) you can use the "list abbreviations" command to see what has been stored.

All right, I think that's about enough for this month. You've got a great part and the code to take advantage of, so go make something talk!

And until next time – Happy Stamping.