Keywords: network audio, ethernet audio, embedded audio, embedded networking

Jun 18, 2004

APPLICATION NOTE 3266

Using the DS80C400/TINIm400 for Remote Audio

Abstract: The versatile DS80C400 is capable of Ethernet-enabling just about anything, and is capable of doing it in several different ways. This application note describes using the TINIm400 reference module and the Java™ Runtime Environment as a remotely controlled speaker, as might be used in a distributed public address system.

Introduction

The versatile DS80C400 is capable of Ethernet-enabling just about anything, and can do it in several different ways. This application note describes using the TINIm400 reference module and the Java Runtime Environment as a remotely controlled speaker, as might be used in a distributed public address system.

Application note 609[1] also describes an Ethernet-enabled speaker, but implemented in a different way. That example uses a custom board programmed in 8051 assembly language to maximize performance. With that configuration, the DS80C400 was able to handle 16-bit audio streams at 44.1kHz, which is good enough to play quality music. The requirements for this application are less rigorous-voice data can be comfortably transmitted at 11kHz, so a custom board and the performance of assembly language is not required. Therefore, this demonstration will use a standard TINIm400, which is much more accessible to hobbyists and developers. In addition, it will use the Java Runtime Environment to simplify adding an HTTP server to the project.

System Overview

Software

In this example application, the TINIm400's spare 512 kilobytes of flash will be used to store pre-built audio messages in 16-bit, 11kHz format. The user interface will be a web page served by the TINIm400. The page has a list of audio messages on a form. When the user selects a message to play, an HTTP POST message is sent to the TINIm400, which must decide which audio sample was selected and start playing it.

In this application, there are two main pieces of software. One is the HTTP server (including the POST handling code), and one is the audio playing code. Since the code that must play the audio must run under an 11kHz timer interrupt, it should be lean and efficient. Therefore, the audio interface has been written in 8051, while the HTTP server code is written in Java.

In addition to the application software, a Java tool was written to read audio data from WAV files and rewrite it in the TBIN format that is understood by the DS80C400's ROM loader. Using this tool, several WAV files can be combined into one file suitable for easy loading into the TINIm400's flash.

Hardware

Figure 1 shows a diagram of an audio circuit that can be connected to a TINIs400 socket. The digital-to-analog converter provides an output of 0 to 2V in this configuration. Since line-level speaker input is ±1V, the speaker's ground is connected to 1V, which gives the speakers a line-level input.

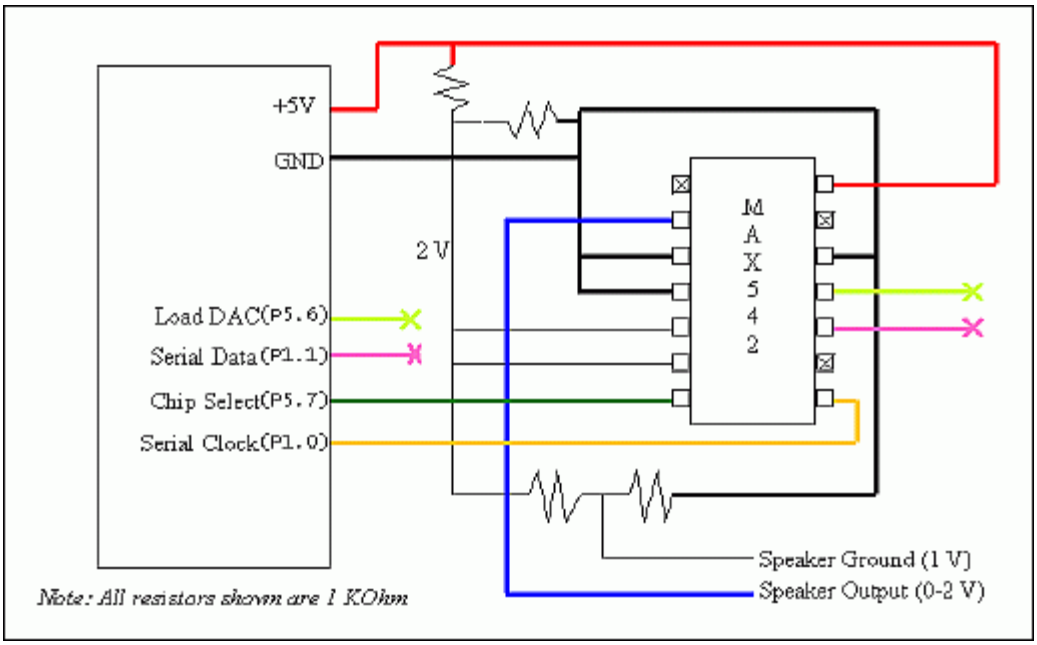

Figure 1. Hardware block diagram.

The digital-to-analog converter used in this circuit is a MAX542[2], which has 16-bit precision. Serial data can be passed to the DAC through the DS80C400's serial port, which is much faster than programmatically toggling the clock and data pins. However, on the TINIs400, we only have access to serial transmit and receive pins after they have gone through a level shifter, which makes them useless for talking to the DAC. Therefore, in this application we will have to manually wiggle the data and clock lines. This will make the TINI®'s performance degrade when it is trying to play audio samples, since it will have fewer free cycles to process network data and take care of other tasks. In a real application, it would be highly desirable to connect the DAC directly to one of the DS80C400's serial port transmit and receive lines.

In addition to the clock and data lines, the TINI will also need to hold a chip select line low for the duration of the serial load, and pulse a load signal (LDAC) low after all serial data has been written.

Connecting to the TINIs400[3]

We have chosen to steal the 4 I²C pins (J27 on the socket board) because they give us 5V, ground, and two I/O pins (P1.0 and P1.1). Bits in the P1 SFR are bit addressable, which means we can set and clear them using the mov bit, c and setb/clr bit instructions instead of the anl/orl direct, #data instructions, saving us one cycle. In this case, one cycle is significant--each audio sample the clock must be pulled low 16 times and high 16 times, plus the data bit must be set 16 times, giving us a total savings of 48 cycles per sample.

Since the load signal (LDAC) and chip select (CS) are manipulated fewer times (twice each per sample), we do not lose as much if they are not bit addressable. We have chosen to use two SPI™ pins (J21 on the socket board). We are using pins P5.6 and P5.7, labeled as nPCE2 and nPCE3 on the schematic (the SPI pins also double as PCEs).

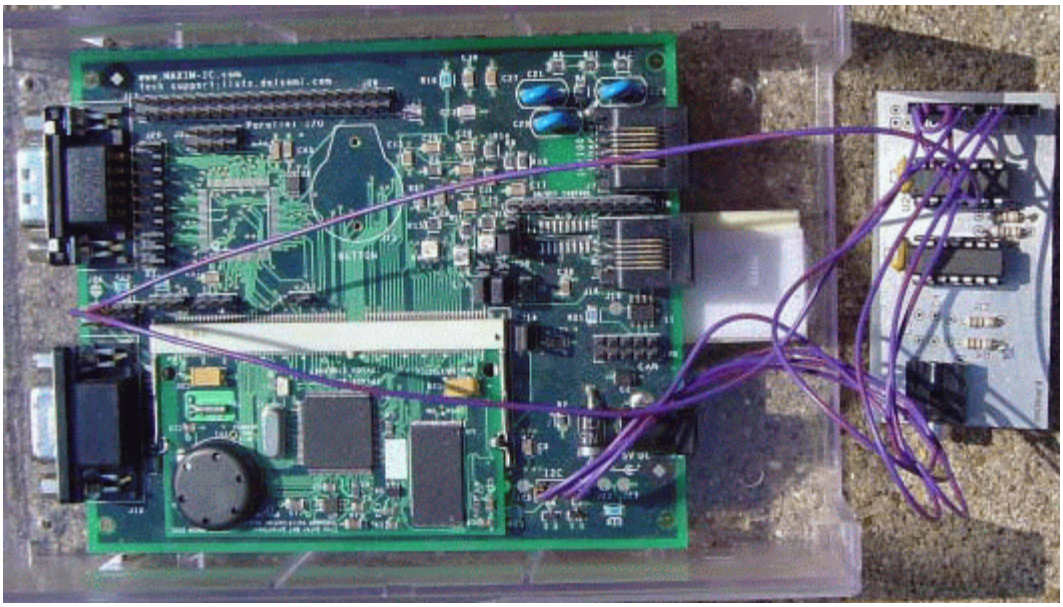

Figure 2. TINI connected to MAX542 DACs.

Figure 2 shows the TINIm400/TINIs400 pair connected to the audio board. The wires coming off the socket board on the left are the LDAC and CS signals. On the lower right of the TINI socket board are the clock, data, 5V, and ground. To the right is a simple DAC board[4] that gives us a nice 3.5mm phono jack where we can connect our amplified speakers. Note there are two DACs on this board-it was designed to handle stereo output. However, we are tying the DAC inputs together for the purposes of the demonstration, resulting in mono sound.

The Web Interface

In our application, we have chosen to use a web page to interface the audio program running on TINI. A form on the web page contains a list of audio messages that are stored in the extra flash on the TINIm400. Other applications might choose to send audio messages over the network or retrieve audio messages from a database.

Figure 3 shows the web page as served from the TINI. In addition to the index page, 3 images are also served. On the left side of the page is a form that allows the user to select a message and play it. When the PLAY IT button is clicked, a POST HTTP message is sent to the TINI, which will be handled by the class named PlayPostedAudio, as seen in this HTML snippet:

```
<FORM METHOD="POST" ACTION="PlayPostedAudio">
```

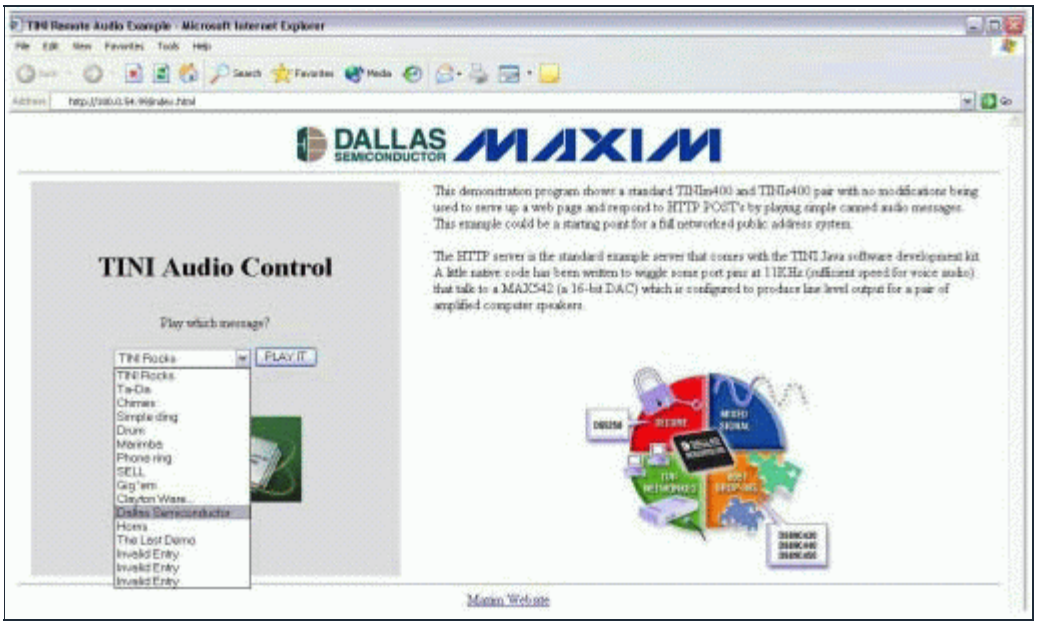

Figure 3. Audio application website served from TINI.

Pre-Generating the Audio Messages

Part of our application includes putting pre-chewed audio messages into the extra flash on the TINIm400. To make it easy to load and organize these audio samples, we created an application to convert several WAV files into one TBIN file that could be loaded by the JavaKit application.

The TBIN Format

The bootstrap loader of the DS80C400 understands how to load messages in HEX and TBIN formats. However, we usually want to load TBIN files because it at least twice as fast.

A TBIN file consists of multiple TBIN records, which contain a 24-bit starting address, 16-bit length, up to 64kB of data, and a CRC-16. The 16-bit field length is the data length minus 1 to allow for 64kB blocks of data. Therefore, a 0-length TBIN record is not possible. Figure 4 shows the format of a TBIN file. Note that a TBIN file need not contain multiple TBIN records. A TBIN file contains 1 or more TBIN records.

u3 address (low)(mid)(high)	u2 (length-1) (low)(high)	[length] data bytes	u2 CRC-16 (low)(high)
u3 address (low)(mid)(high)	u2 (length-1) (low)(high)	[length] data bytes	u2 CRC-16 (low)(high)
u3 address (low)(mid)(high)	u2 (length-1) (low)(high)	[length] data bytes	u2 CRC-16 (low)(high)
u3 address (low)(mid)(high)	u2 (length-1) (low)(high)	[length] data bytes	u2 CRC-16 (low)(high)
more TBIN entries...			
u3 address (low)(mid)(high)	u2 (length-1) (low)(high)	[length] data bytes	u2 CRC-16 (low)(high)

Figure 4. TBIN file format.

Data Format for the Audio Data Stored in Flash®

Within the TBIN format we will need to format our audio data to make it easy for our application to process quickly. Our application will be hard coded to play 16-bit mono audio samples at 11kHz, which is sufficient quality for voice audio. The web interface will provide the audio application with an index of the audio sample that needs to be played. We need a simple table that will allow us to translate from index number to a starting address and a length. It would also be nice if we knew whether or not the index was a valid one. The structure we have chosen is as follows:

```
u2 Number of audio samples in this archive (msb, lsb)
Number_of_audio_samples address & length
{
  u3 starting address of this audio sample (xsb, msb, lsb)
  u3 length of the audio sample (xsb, msb, lsb)
}
Number_of_audio_samples audio data
{
  [length * 2 bytes] audio data
}
```

Note that the length reported in this structure is the number of 16-bit samples, or half the number of bytes that are in the audio sample. With this structure, as long as we know the starting address for the table, we can quickly and easily figure out where our audio data is and how much to play.

Included with the source for this application[5] is a Java application called Wav2TBIN, which can be used to convert multiple WAV files to the format required for this application. Note that WAV files often represent audio data with different number of channels,

bytes per channel, and sample rate. The Wav2TBIN application handles several of the more common formats (for example: 44.1kHz, 16-bit stereo) but may not handle all possible formats.

The Audion Application

The application running on the TINI consists of 2 Java classes and a native library. The Java class that implements public static void main(String[ ]) is called HTTPAudioServer. The main function loads the native library and starts the HTTP server.

The second Java class is called PlayPostedAudio, which is called dynamically when our web interface generates a POST message. This class relies heavily on our native library and provides a high level interface to the audio data we have stored in our flash. The following lines in PlayPostedAudio.java define the functions implemented in our native library:

```
  // how many canned audio messages do we have in flash?
  public static native int getNumberOfAudios(int address);

  // get the address of the indexed audio record
  public static native int getAudioAddress(int address, int index);

  // get the length of the indexed audio record
  public static native int getAudioLength(int address, int index);

  // start playing the audio file at 'address' for 'length' bytes
  public static native int startAudio(int address, int length);

  // check to see if anything is playing right now
  public static native boolean isAudioPlaying();
```

After the PlayPostedAudio class parses the index from the POST data, it checks the validity of the index. If the requested index is invalid, it generates a web page reporting the error.

```
  int num_files = getNumberOfAudios(AUDIO_FLASH_ADDRESS);
  if (cannedindex >= num_files)
  {
    strBuff.append("Sorry, there are only "+num_files+ " audio files in my archive
and you have selected to play number "+cannedindex);
    strBuff.append("

That wasn't very nice of you.
");
  }
```

If the index was valid, next we check to make sure that an audio sample is not currently playing.

```
  if (isAudioPlaying())
  {
    strBuff.append("Sorry, but there is currently audio playing.
");
    strBuff.append("It would be rude of you to go now. Wait your turn.
");
  }
```

Otherwise, we start playing the audio sample?

```
  int audioaddress = getAudioAddress(AUDIO_FLASH_ADDRESS, cannedindex);
  int audiolength = getAudioLength(AUDIO_FLASH_ADDRESS, cannedindex);
  startAudio(audioaddress, audiolength);
```

At this point the native library takes over playing the audio, and the Java application returns to servicing HTTP requests.

The Native Library

Our native library uses timer 3 (otherwise unused by the TINI OS) to generate interrupts at roughly 11kHz. On library initialization (in audioplayer_Init), we need to install our interrupt handler. We also make timer 3 a high priority interrupt to reduce the skew of our 11kHz interrupts. Note that we explicitly disable timer 3 interrupts at this time. We will only enable timer 3 interrupts when we have an audio sample to play. When we are at the end of our audio sample, we will disable the timer interrupt. This way, our interrupt service routine does not have to figure out if it has audio to play or not. Since our interrupt routine is the critical path for performance in this application, we should keep it as lean as possible.

Another interesting function in the native library is Native_startAudio. This function takes as input the starting address and number of

audio samples to play. We need a place where our interrupt service routine can access and change these values as it plays audio data. Indirects provide the fastest access for this purpose, so our application has decided to use indirects F0h to F5h. As long as no other applications that we are running use these indirects, they will be a safe location to store our data. Before exiting, this function also enables the timer 3 interrupt and enables timer 3 to run.

At this point, our Java application has returned to servicing HTTP requests, but now the timer interrupt is running and our interrupt service routine should fire about 11,000 times a second. The timer 3 ISR first gets the pointer out of the indirect RAM, which points to the next audio sample we want to play. We reload the timer counters and clear the timer interrupt, and then read our two-byte sample. Notice that we complement the top byte of the audio sample. Since the audio data in the WAV file is signed, we need to convert it to unsigned data for our DAC. Mathematically, we want to add 8000h to our audio sample (half of the maximum 16-bit value). The complement of the top bit performs the same operation.

Next we bit bang our audio sample out to the DAC. It would be highly preferable to use the DS80C400's serial port mode 0 (synchronous clock and data on the RX and TX pins) for this purpose. However, on an unaltered TINIm400 module and TINIs400 socket there aren't any serial pins that are easily accessible that have not been level shifted.

After the audio sample is banged out, we store the new value of the data pointer in the indirect RAM, and then decrement the remaining length, which is also stored in the indirect RAM. If there is no more data available, we disable the timer interrupt and turn the timer off.

Building and Running

To build the native libraries, you will need the standard include files that come with every TINI firmware distribution[6]: tini_400.inc, ds80c400.inc, tinimacro.inc, and apiequ.inc. These files can be found in the native/lib directory of the firmware distribution. You will also need the programs macro and a390 (macro pre-processor and assembler), available in the native/bin directory of the firmware distribution.

To build the native library, use the following commands:

```
macro audioplayer.a51
a390 -f 1.12 -p 400 -l audioplayer.mpp
```

Once the native library is built (it will be called audioplayer.tlib), you can build the entire application:

```
# compile the Java classes
javac -bootclasspath c:\tini1.12\bin\tiniclasses.jar

      -classpath c:\tini1.12\bin\modules.jar
      HTTPAudioServer.java PlayPostedAudio.java

# build the .tini file, including the native library
java BuildDependency -n audioplayer.tlib -f HTTPAudioServer.class
                     -f PlayPostedAudio.class -o server.tini
                     -d c:\tini1.12\bin\tini.db -add HTTPSERVER
                     -p c:\tini1.12\bin\modules -x
c:\tini\tini1.12\bin\owapi_dep.txt
```

Assuming that the audio data has already been loaded using JavaKit, you will need to FTP the application file server.tini, the web page index.html, and the images ds80c400.gif, logo.gif, and micro4.gif to your TINI. To run the application, the command is simply:

```
TINI> java server.tini
```

Note that this requires you to already have configured the IP address of your TINI. For help with getting started on your TINI setup, try the application note "Getting Started with the TINIm400 Verification Module".

This application was developed with TINI firmware 1.12, but any later 1.1x version of firmware should work.

Conclusion

This application note provides a good starting point for a networked public address system or similar application. A real, commercial application could go in several directions from here. It could provide a microphone to record short audio messages, and then broadcast those messages to every network node for playback. Or it could target specific nodes to receive messages-if a message is for the electronics department at the store, it doesn't need to be broadcast to every department. It could also become a smart intercom system, allowing instant communication between network nodes. TINI gives us a powerful, flexible platform from which all

these projects and more are possible.

References

1. Application note 609, "Internet speaker with the DS80C400 silicon software"
2. MAX542 data sheet
3. Schematics for the TINIs400 socket (PDF)
4. Schematics on page 4 (PDF)
5. Source code for this application note
6. The TINI firmware
