App Note 609: Internet Speaker with the DS80C400 Silicon Software

The networking capabilities of the DS80C400 microprocessor make it a natural choice for designing a simple Ethernet-enabled speaker. By using the TCP/IP stack built into the processor's ROM, an application written in 8051 assembly can easily read streaming audio data from the network and use that data to drive a digital-to-analog converter (DAC) that provides line-level output for a set of speakers. This application note presents the hardware design and the software necessary to run a simple Ethernet-enabled speaker.

System Overview

Software

At the top level, the application consists of a host computer sending uncompressed audio (such as data from a WAV file) over a network connection to a DS80C400, which listens and plays the audio data. Figure 1 shows a block diagram of this system.

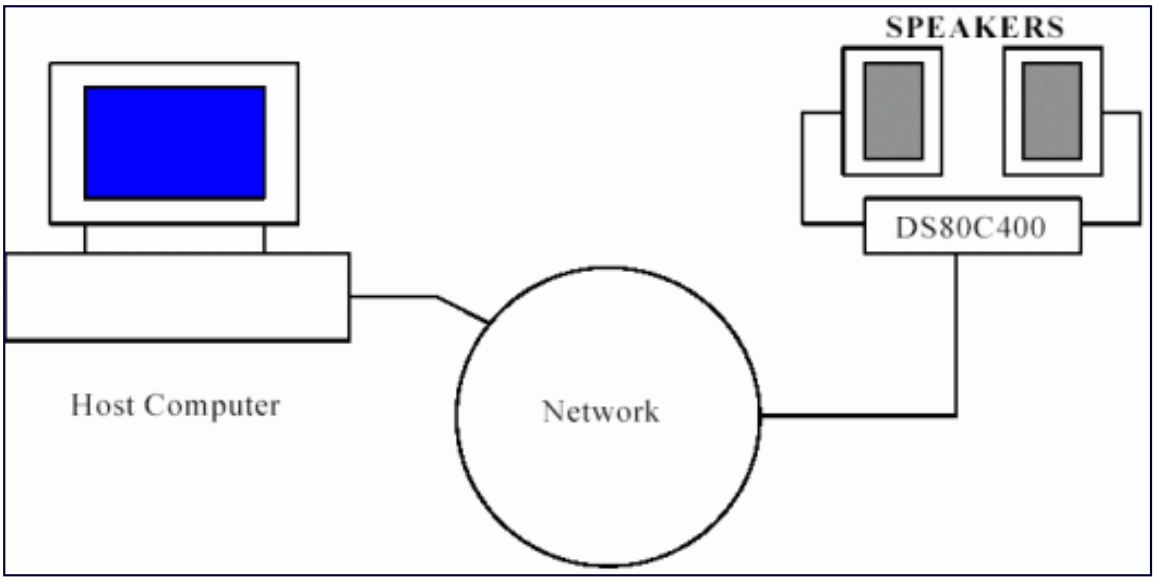

Figure 1. System block diagram.

There must be two software applications to make this system work. One application must run on the host computer and send audio data to the DS80C400. The other application must run on the DS80C400 and play audio data.

The host application has an easy job in this system. It must read raw audio data from a WAV file and send it over the network. Since there is a lot of processing power not being used on the host, it does some other jobs too, like flow control and simple data formatting.

The application on the DS80C400 is a little more complicated. It needs to receive audio data over the network, and push that data to an audio circuit at the specified sample rate.

Receiving the audio data is implemented in a loop that waits for audio data and writes it to a circular buffer when it becomes available. As it receives new data, it must also maintain a pointer to the end of the valid data in the buffer, so the application does not play invalid data.

The second portion of the speaker application is the part that pushed data to the audio circuit. The audio data is fed into a digital-to-analog converter, which in turn drives a normal computer speaker. Since regular timing is critical to audio applications, this portion of the application is implemented as a timer interrupt. Figure 2 shows how the loop and timer portions of the application interact through the circular audio buffer.

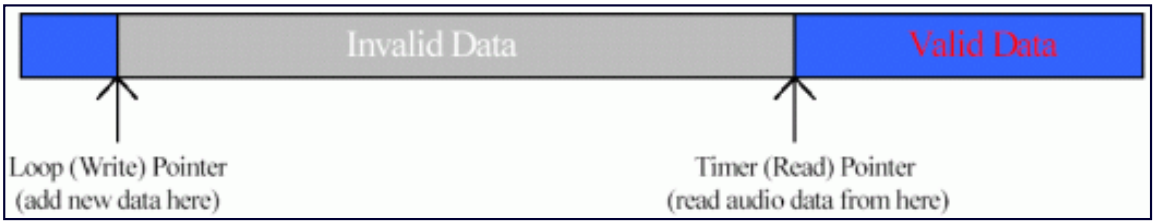

Figure 2. Circular audio buffer.

Hardware
Figure 3 shows a diagram of an audio circuit that can be connected to a TINIm400 Verification Module or a custom design based on the DS80C400. For this project, the speaker application was developed on a board originally designed for a networked camera, with some small modifications.

The schematics for this board can be found at:
ftp://ftp.dalsemi.com/pub/tini/reference_designs/netspeaker/networkspeaker.pdf.

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. The digital-to-analog converter used in this circuit is a MAX542[1], which has a precision of 16 bits. 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. The MAX542 has a chip select line that must be held low for the duration of the serial load, and a load signal (Load DAC) that must be pulsed low after all serial data has been written.

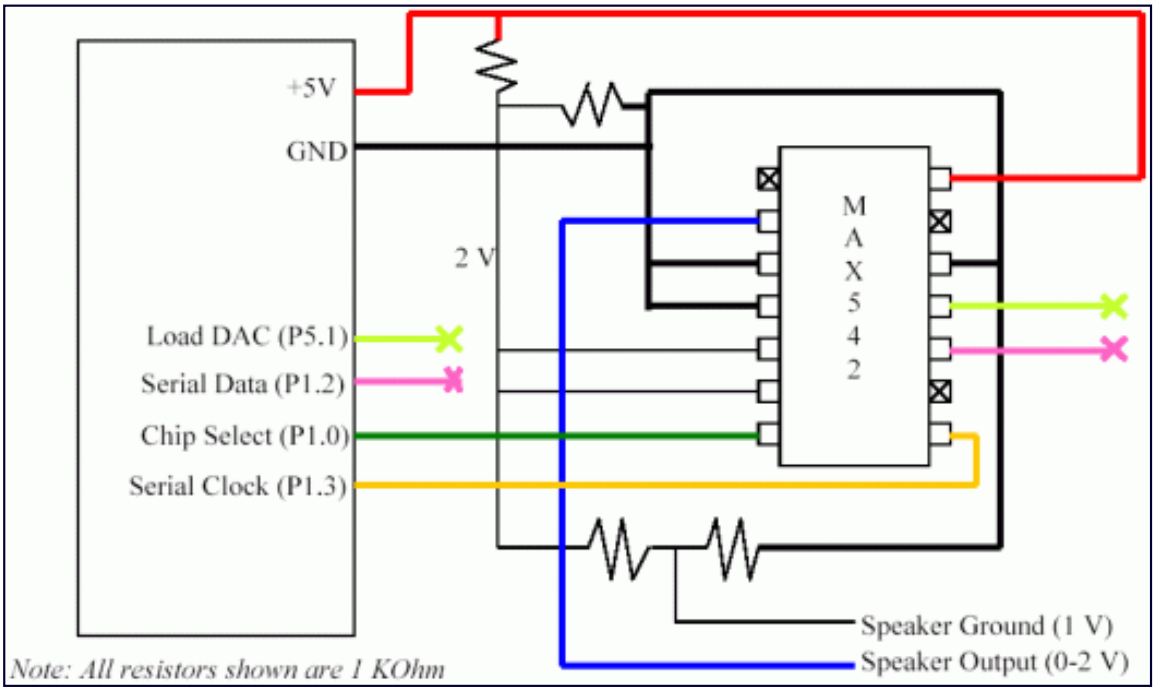

Figure 3. Hardware block diagram.

The Host Application: Sending Uncompressed Audio

The host application is a Java™class called `SendDataTCP`. It is a Java application that reads a PCM encoded WAV

file, performs some simple formatting, and sends blocks of audio samples to the DS80C400 through a TCP connection.

The program assumes the WAV file being read contains stereo, 16-bit data to be played at a sample rate of 44.1kHz. However, the application supports sending sample rates of 44.1, 22.05, and 11.025kHz, therefore the audio data may need some reformatting. The data in the WAV file is assumed to be 16 bit stereo, therefore each sample consists of 4 bytes (2 bytes for channel 1, 2 bytes for channel 2). If the DS80C400 is expecting mono data rather than stereo data, only one channel is extracted from the WAV file. If the sample rate is lower than 44.1kHz, some samples are skipped. For example, if the DS80C400 expects stereo data with a sample rate of 22.05kHz, the `SendDataTCP` program would send 2 bytes of channel 1 data, send 2 bytes of channel 2 data, and then skip the next sample. If mono data at 22.05kHz is expected, the `SendDataTCP` program would send 2 bytes of channel 1 data, skip the channel 2 portion, and then skip the entire next sample.

Before the data is sent, two more transformations must be performed. First, the sample must be converted from signed data to unsigned data. WAV files contain signed data meant to represent voltages between -1 and 1, but the MAX542 accepts unsigned data representing voltages between 0 and 2. Note that since the circuit gives the speaker a virtual ground of 1V, the desired transformation is to simply add 1 V to the voltage defined in the WAV file. Since the input value 8000 hex represents 1V of output from the MAX542, we need to add 8000 hex to every 16-bit sample. Note that this is the same operation as toggling the high bit of the sample. Table 1 shows the relationship between a single 16-bit sample from a WAV file, the desired voltage, the voltage that would be produced by the unaltered sample, and the voltage that would be produced by the altered sample.

Table 1. Altering the audio samples to achieve the desired voltage output

16 bit audio sample (hex)	Desired Voltage	Voltage from unaltered sample	Altered Sample	Voltage from altered sample
0000	1.00	0.00	8000	1.00
7FFF	2.00	1.00	FFFF	2.00
8000	0.00	1.00	0000	0.00
4000	1.50	0.50	C000	1.50
C000	0.50	1.50	4000	0.50

The second transformation that must occur is a bit flip operation. The serial ports on the DS80C400 write the least significant bit first, but the MAX542 expects data most significant bit first. This operation is performed with a simple lookup table.

Data is sent to the DS80C400 in 1400 byte blocks - a size found to offer the best performance. Data flow control is performed by tracking how much data has been sent within the last second and comparing that to the amount of data expected to be sent every second. For instance, mono data at a sample rate of 22.05kHz would produce 44,100 bytes per second. If the `SendDataTCP` program has sent 44,500 bytes in the last 800 milliseconds, it sleeps for about 200 milliseconds. The DS80C400 uses a buffer that is more than 400kB, which equates to several seconds of audio data. Therefore, accurate timing is important in the `SendDataTCP` program, but not critical. Some variation is acceptable.

Note that the `SendDataTCP` program generally sends data as fast as it can. If the program never pauses because it has sent too much data in the last second, it is likely that the data rate is too much for the application to handle. This can be the result of excessive network traffic.

The DS80C400: Initializing the Speaker Application

The application for the DS80C400 is written entirely in 8051 assembly. Note that it would have also been possible to realize the application in C using Keil's compiler[2], or in Java using the TINI® Runtime Environment[3]. The application is small enough that writing it in assembly was not a daunting task.

Where possible, the speaker application has made use of resources that are not otherwise occupied or altered by the functions in the ROM. The DS80C400 has 4 data pointers, of which only one is not altered by the operating system. The first two data pointers are used extensively by all functions, especially for copy operations. The fourth data pointer is used in some network routines, but is always preserved. The third data pointer is never used. Since the interrupt to drive the speaker will need to be a high priority interrupt, the fourth data pointer is not suitable for use, leaving only the third data pointer available. The DS80C400 also has four timers. The ROM uses timer 0 as a clock tick and timer 2 for serial port output. This leaves timer 1 and timer 3 for the speaker application.

The speaker application uses timer 3 to generate interrupts for loading the MAX542 digital-to-analog converters. Timer 3 is selected to run in 16-bit timer mode. There is no auto-reload for timer 3 in 16-bit mode, although the hardware does clear the interrupt bit automatically. The timer 3 interrupt runs as high-priority, since the timing of loading the MAX542 is critical to the audio quality.

Before the application starts, the ROM has set up some of the special features of the DS80C400. The processor is already in 24-bit addressing mode, allowing easy code and data access across 64kB boundaries. The extended stack has also been enabled, making use of the DS80C400's dedicated 1024-byte stack space. This leaves the indirect memory space available for application use, without fear of stack usage destroying its contents. After the application starts, the clock quadrupler is enabled, yielding a single cycle instruction time of about 54ns. Next, timer 3 is initialized, which must be done before the ROM is initialized and process swapping begins. This is because the ROM preserves the interrupt-enable bits on process swaps. Since the timer interrupt needs to run all the time, it should be enabled before processes themselves are enabled.

To finish initializing the system, a number of ROM functions are called. The first ROM function called is `rom_init`, which initializes the memory manager, the process manager, and the network stack. Network parameters are set next, giving the DS80C400 a static IP address.

The system is now initialized and ready to create a listening socket. The network functions are assembly versions of the traditional Berkeley-style sockets. The application creates a new TCP socket handle by calling `create_socket`, and assigns it to a port number by calling `bind_socket`. The function `setup_listen` sets up the socket as a server socket, and `accept_connection` waits for a socket connection.

Before the program enters the main loop, the read and write pointers are initialized. Incoming data from the network connection will be written to the `EndBuffer` pointer, which is stored in the indirect memory area, since there are no directs that are free to use and safe across process swaps. The third data pointer is used to read the next valid sample from the buffer. This pointer is used exclusively by timer 3's interrupt service routine (ISR). Before the ISR reads the sample data, it checks to see if it is reading too close to the `EndBuffer` pointer. If the two pointers are in the same bank (the same 64kB area of memory), the timer ISR will simply exit without playing audio data. This not only prevents the ISR from reading invalid data beyond the end of the buffer, but also provides some amount of buffering in case the application is not receiving data quickly enough. If the application stops playing audio data, it will not start again until at least 64,000 bytes are available. The tradeoff here is that longer gaps in the audio can be heard if the application is not receiving data fast enough, but the audio will be discernable.

The Loop: Waiting for Data from the Network

Before the main loop of the application begins waiting for data, it checks the `EndBuffer` pointer to see if it has wrapped around the end of the circular buffer, adjusting the pointer to the start of the circular buffer if necessary. It then calls the `recv_data` function, which reads any available data or blocks until data is available. The network data received is read directly into the circular buffer. This prevents the application from having to copy data after the `recv_data` function returns. If the `EndBuffer` pointer is close to the end of the circular buffer, the `recv_data` function only requests enough data to reach the end of the buffer. This means that occasionally the application may request to receive a small amount of data, but the benefit is that the application can read data directly into the circular buffer with no intermediate copy. After the read, the `EndBuffer` pointer is updated and control returns to the top of the loop.

If an error occurs while reading, the application closes its socket and waits for another socket connection. Usually, a

detected error really means that the host closed the sending socket. This allows the sender to start and stop the host program at any time, and to play multiple WAV files one after another.

The Timer Interrupt: Playing Audio Data

Before doing any tasks, the interrupt service routine (ISR) for timer 3 must reload the timer registers. The timer registers are always reloaded with the same value. This reload value is tied to the rate at which audio samples are played. A higher reload value (meaning less time for the counter to roll over) means faster audio sample playback. A lower reload value means slower audio sample playback.

After the timer registers are reloaded, the ISR checks to see if it is reading data too close to the `EndBuffer` pointer. Checking only the bank number (the highest byte of the pointers) has two benefits. One has been discussed earlier in the section *Initializing the Speaker Application* - preventing short, unintelligible bursts of audio when data is not received fast enough by the application. Another benefit is a quicker comparison in the ISR. The ISR runs several thousand times a second, so cutting cycles from the ISR is extremely important. By only checking the high address byte, two extra comparisons for the middle and low address bytes are avoided.

If there is valid audio data ready to be played, the sample is read and the data pointer incremented to the next sample. Data is loaded into the MAX542 digital-to-analog converter by first setting the chip select line low, loading 2 bytes into the serial port, setting the chip select line high, then pulsing the load DAC line low. The serial port handles the correct toggling of the serial clock and data lines. Several `nop` instructions are inserted after each load of the serial port, allowing the hardware to finish shifting out the byte. Last, the ISR checks the pointer that reads the audio data to see if it has wrapped around the end of the circular buffer, and corrects it if necessary.

The Tick: Overwriting the System Timer

In order to receive data at a rate that will allow for quality audio playback, the operating system's timer tick function will need to be altered. Altering the timer tick will allow more control over I/O performance. Below is the original timer tick code as it runs in the DS80C400 ROM:

```
IOPOLL_TICK_MS    equ    4

WOS_Tick:
       ; The timer is running in divide by 12 mode.
       push    psw
       push    acc

       clr     tr0
       clr     tf0
       mov     a, sched_reload_lsb
       add     a, tl0
       mov     tl0, a
       mov     a, sched_reload_msb
       addc    a, th0
       mov     th0, a
       setb    tr0

       inc     ms_count_0
       mov     a, ms_count_0
       jnz     wos_tick_check_sched        ; Check for byte 0 roll.
       inc     ms_count_1
       mov     a, ms_count_1
       jnz     wos_tick_check_sched        ; Check for byte 1 roll.
       inc     ms_count_2
       mov     a, ms_count_2
       jnz     wos_tick_check_sched        ; Check for byte 2 roll.
       inc     ms_count_3
```

```
            mov    a, ms_count_3
            jnz    wos_tick_check_sched        ; Check for byte 3 roll.
            inc    ms_count_4                  ; If this wraps, we are in trouble

      wos_tick_check_sched:
            jb     need_sched, wos_tick_check_critical_section

            mov    a, ms_count_0               ; See if it's time to run the
            anl    a, #IOPOLL_TICK_MS-1        ;    scheduler/iopoll routines.
            jnz    wos_timer_reload            ; If not, don't do scheduler stuff.

      wos_tick_check_critical_section:
            clr    ea                          ; Make sure nobody interrupts
                                               ; us before we want to

            mov    a, STATUS                   ; Check for low priority interrupts
            jb     acc.5, wos_tick_low_priority_in_progress
                                               ; If low priority interrupts are being
                                               ; serviced, don't run the scheduler.
                                               ; If we don't do this, we'll start
running
                                               ; the scheduler as a low priority
interrupt.

            mov    a, wos_crit_count           ; Check the critical section count.
            jz     wos_tick_not_critical_section
                                               ; If we're not in a critical section,
                                               ; go ahead, jump and run the scheduler.

      wos_tick_low_priority_in_progress:
            setb   need_sched                  ; Signal to ourselves, or whoever, that
                                               ; we need to run the scheduler next time
            sjmp   wos_timer_reload            ; Going to blow off this tick.

      wos_tick_not_critical_section:
            WOS_ENTER_CRITICAL_SECTION
            pop    acc                         ; Clean up stack.
            pop    psw
            pop    curr_pc_x                   ; Return address to get out of
interrupt.
            pop    curr_pc_h
            pop    curr_pc_l
            PUSH_DPTR1
            push   dps
            mov    dps, #0
            mov    dptr, #WOS_IOPoll           ; Get address of IOPoll
            mov    sched_l, dpl
            mov    sched_h, dph
            mov    sched_x, dpx
            pop    dps
            POP_DPTR1

            push   sched_l                     ; Push address of IOPoll
            push   sched_h
            push   sched_x
            reti                               ; Run IOPoll
```

```
wos_timer_reload:
       ; Interrupts must have been on when the interrupt handler
       ; was called.
       setb   ea                          ; Enable interrupts
       pop    acc
       pop    psw
       reti
```

The most important changes to this tick function are using an `IOPOLL_TICK_MS` value of 1 instead of 4, and changing the `sched_reload_xxx` values so that the tick function runs more often. Note that by using an `IOPOLL_TICK_MS` value of 1, the line *anl a*, *#IOPOLL_TICK_MS-1* always evaluates to 0, allowing some code and logic reduction. Also, since we are not worried about accurate system clock timing, we can reduce the clock reload code.

One complication is introduced by overwriting the timer tick function. The speaker code should run on any future revision of the DS80C400 ROM, so we cannot hard code an address for the `WOS_IOPoll` function. Luckily, the address of the `WOS_IOPoll` function is in the ROM export table. The speaker program reads this address on startup and stores it in indirect memory, and is then used by the timer tick function to call the `WOS_IOPoll function`. Below is the timer tick function, customized for the speaker application:

```
speaker_wos_tick:
       ; The timer is running in divide by 12 mode.
       push   psw
       push   acc

       ;
       ; We know what we want our timer reload to be.
       ; And our millisecond count doesn't have to be too
       ; accurate, so we can just straight load the
       ; timer registers.
       ;
       mov    tl0, #TICK_RELOAD_LOW       ; TICK_RELOAD_LOW  = 80h
       mov    th0, #TICK_RELOAD_HIGH      ; TICK_RELOAD_HIGH = FDh

       inc    ms_count_0
       mov    a, ms_count_0
       jnz    wos_tick_check_sched        ; Check for byte 0 roll.
       inc    ms_count_1
       mov    a, ms_count_1
       jnz    wos_tick_check_sched        ; Check for byte 1 roll.
       inc    ms_count_2
       mov    a, ms_count_2
       jnz    wos_tick_check_sched        ; Check for byte 2 roll.
       inc    ms_count_3
       mov    a, ms_count_3
       jnz    wos_tick_check_sched        ; Check for byte 3 roll.
       inc    ms_count_4                  ; If this wraps, we are in trouble

wos_tick_check_sched:
       clr    ea                          ; Make sure nobody interrupts
                                          ; us before we want to
       mov    a, STATUS                   ; Check for low priority interrupts
       jb     acc.5, wos_tick_low_priority_in_progress
                                          ; If low priority interrupts are being
                                          ; serviced, don't run the scheduler.
                                          ; If we don't do this, we'll start
```

```
running
                                        ; the scheduler as a low priority
interrupt.
            mov    a, wos_crit_count    ; Check the critical section count.
            jz     wos_tick_not_critical_section
                                        ; If we're not in a critical section, go
                                        ; ahead, jump and run the scheduler.

      wos_tick_low_priority_in_progress:
            setb   need_sched           ; Signal to ourselves, or whoever, that
we
                                        ; need to run the scheduler next time
            sjmp   wos_timer_reload     ; Going to blow off this tick.

      wos_tick_not_critical_section:
            WOS_ENTER_CRITICAL_SECTION
            mov    psw, #0
            push   r0_b0
            mov    r0, #wos_iopoll_x
            mov    a, @r0               ; xhigh byte of wos_iopoll address
            inc    r0
            mov    sched_x, a
            mov    a, @r0               ; high byte of wos_iopoll address
            inc    r0
            mov    sched_h, a
            mov    a, @r0               ; low byte of wos_iopoll address
            inc    r0
            mov    sched_l, a
            pop    r0_b0

            pop    acc                  ; Clean up stack.
            pop    psw

            pop    curr_pc_x            ; Return address to get out of
interrupt.
            pop    curr_pc_h
            pop    curr_pc_l

            push   sched_l              ; Push address of IOPoll
            push   sched_h
            push   sched_x

            reti                        ; Run IOPoll

      wos_timer_reload:
            ; Interrupts must have been on when the interrupt handler
            ; was called.
            setb   ea                   ; Enable interrupts
            pop    acc
            pop    psw
            reti
```

The timer reload value of FD80h was determined through experimentation. This reload value allows audio data being sent at about 88,000 bytes per second to be played smoothly with minimal interruption, depending on other network traffic. This translates to playing mono audio data at 44.1kHz, or stereo audio data at 22.05kHz.

The Application: Building and Running

The host application is a Java application, and thus needs the Java Development Kit to build and run it. Version 1.3.1 was used during development, but the code in the `SendDataTCP` program is simple enough that any released version of the Java Development Kit should be sufficient. The command line to build is simply:

```
javac SendDataTCP.java
```

To run the `SendDataTCP` program, use a command line like the following:

```
java SendDataTCP 10.0.0.1 5555 some_song.wav
```

In this example, `10.0.0.1` is the IP address of the DS80C400, which is listening for connections on port `5555`. The WAV file `some_song.wav` will be used to send audio samples to the DS80C400. Note that the WAV files used are assumed to contain 44.1kHz stereo data samples. There are several tools available to generate WAV files from MP3 files. One free tool is included in the JavaLayer MP3 suite[4]. Most, but not all MP3 files contain 44.1kHz stereo data, so be aware of what kind of WAV file any tool generates.

The speaker application to run on the DS80C400 is written in 8051 assembly, and requires tools available freely in the TINI Software Development Kit[5]. The speaker application was developed for a board that has flash at addresses 400000-47FFFF (hex), and RAM at addresses 00000-7FFFF and 60000-67FFFF (hex). Figure 4 describes the memory configuration of this board.

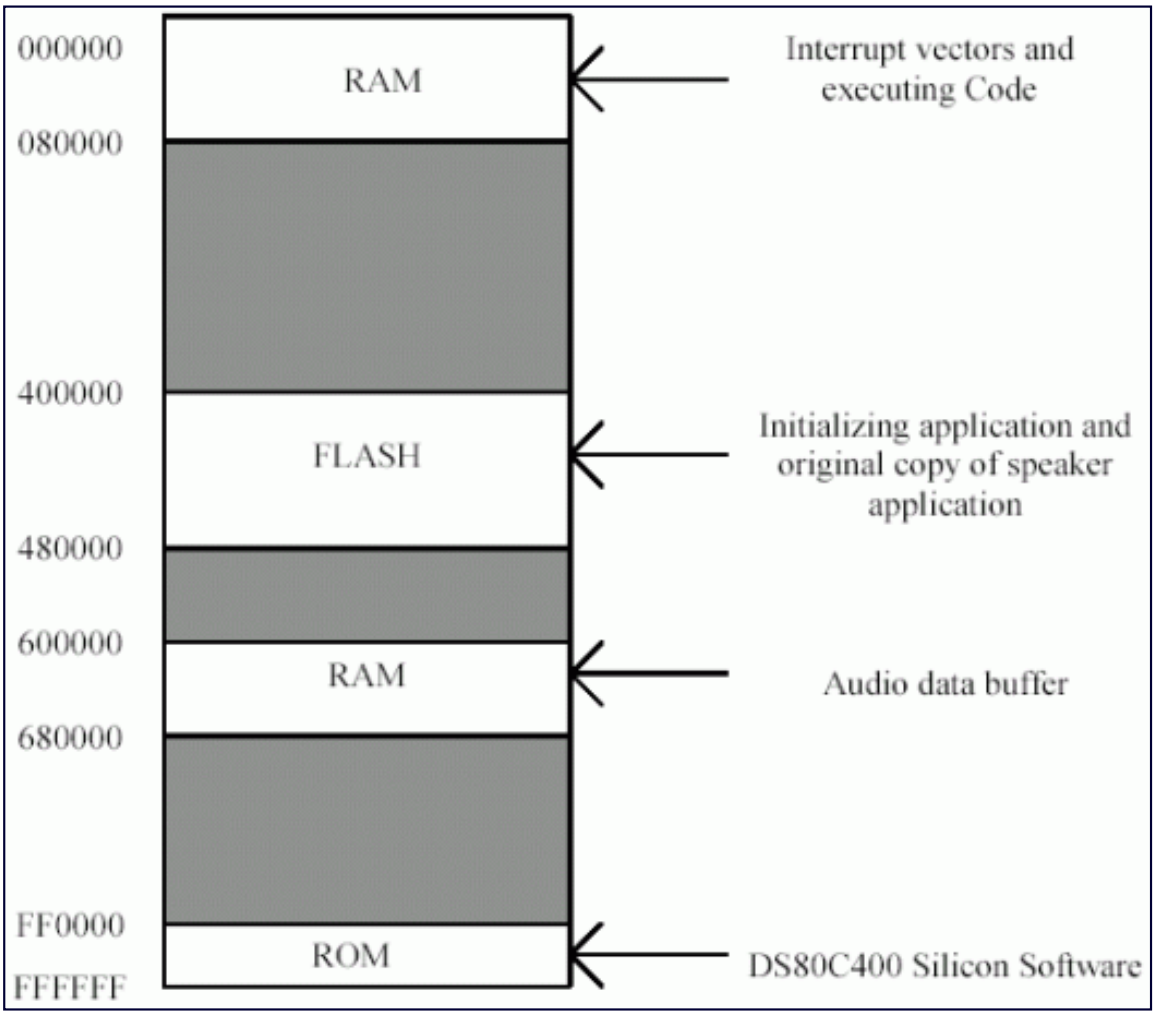

Figure 4. Board memory configuration.

Developers for other boards will need to bear in mind that addresses may need to be changed to match their board

configurations. The following is the build script that was used to create the speaker application:

```
macro speaker.a51
a390 -l -Ftbin -d -p 390 speaker.mpp
java fixBankNum speaker.tbin 66
```

The tools `macro` and `a390` are part of the TINI SDK. The `fixBankNum` program is a small Java application that changes that target memory bank for loading the application, and is included in the source files for this application note on the Dallas Semiconductor FTP site[6]. Note that '66' is decimal, so the *speaker.tbin* file will be targeted for bank 42 (hex), which lies in the flash.

The `fixBankNum` program is necessary because the speaker application cannot run out of flash, but storing the program in flash is the only way to make sure it is not erased when power to the DS80C400 is cut. The speaker application cannot run out of flash because with the clock quadrupled, it exceeds the specified access time for the flash. Therefore, a small initializing application runs, copying the speaker application out of the flash and into the RAM. Control then jumps to the copy of the speaker application in the RAM, which then enables the clock quadrupler and begins running normally. The source for this initializing application is called *init.a51*, and is also included in the source files for this application note. Build the initializing application with the following script:

```
macro init.a51
a390 -l -Ftbin -d -p 390 init.mpp
```

To run the speaker application, the initialization and speaker files must be loaded onto the DS80C400. This is done using JavaKit, another application included in the TINI SDK. The document *Running_JavaKit.txt* (also part of the TINI SDK) details how to run JavaKit. The build scripts above produces files called *speaker.tbin* and *init.tbin*. Use JavaKit to load these files into the DS80C400. The files should load into banks 41 and 42 (hex). To run the speaker application, type the following at the JavaKit loader prompt:

```
B41
X
```

The initializing application should copy the speaker application to memory, some debug is printed, and the speaker application has started. Run the `SendDataTCP` program to send audio data. After a second or two of audio buffering, the music should start.

The Application: Changing Program Parameters

The speaker application and host code support playing mono data at 44.1kHz, 22.05kHz, or 11.025 kHz. The trade-off to consider when selecting a data rate is audio quality versus network interruptions. On a low traffic network, the application may be able to play data at 44.1kHz without interruptions. On a high traffic network, audible blips in the audio may become apparent. Follow these steps to change the sample rate:

1) Find the equate `RELOAD_44_1_at_18` near the top of the file *speaker.a51*. Change this value to 390 for 44.1kHz, 800 for 22.05kHz, and 1600 for 11.025kHz.
2) Find the variable `static int audio_quality` near the top of the file *SendDataTCP.java*. Change this value to `MONO_44100`, `MONO_22050`, or `MONO_11025`.
3) Recompile and rebuild both portions of the application, and reload the speaker application on the DS80C400.

Data is stored on a music compact disc in stereo, 44.1kHz 16-bit samples. Mono data only means that one channel is played, instead of two. Adequate quality for music is 22.05kHz; 11.025kHz is adequate for voice data.

The IP address and parameters of the speaker application are also configurable. Near the bottom of the *speaker.a51* file is the following declaration:

```
network_parameters:
    db  0, 0, 0                                                 ; 3 bytes overhead
    db  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 1         ; ip address
    db  255, 255, 0, 0                                          ; subnet mask
    db  16                                                      ; ipv4 netmask len
    db  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 2         ; gateway
```

The format of this structure is described in the DS80C400 User's Guide. However, note that the current IP address used in the speaker application is `10.0.0.1`, and the current gateway is set to `10.0.0.2`. Changing to make the application use a different IP address should be trivial. A little lower in the source file, the server socket's port number is specified:

```
address:
    db  0, 0, 0                                                 ; overhead
    db  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 1     ; address
    db  55h, 55h                                                ; port
    db  0                                                       ; family
```

Note that the port number passed to the `SendDataTCP` program is assumed to be a hex value.

The Future: Adding a Second Channel and Other Improvements

There are a number of improvements and modifications that could be made to the speaker application. Multicast UDP might replace TCP, allowing one server to broadcast messages to several DS80C400's. DHCP might be used to dynamically obtain an IP address, allowing for a self-configuring install. A configuration byte might tell the speaker application what the audio quality was, so it could easily play audio data at 11kHz, 22kHz, or 44kHz on the fly. Controlling the flow of data from the host to the DS80C400 could also stand to be improved.

Another key improvement would be the addition of another audio channel, allowing for stereo sound. The trick here is to make sure that adding another channel does not make the timer 3 interrupt routine run too long. The best solution might be to use serial port 0 to output the other audio channel. The application would lose the ability to send debug messages over the serial port, but the additional overhead to the timer interrupt would be minimal.

Conclusion

The DS80C400 is the perfect choice for an internet-enabled speaker. The DS80C400's ROM gives applications the ability to communicate through the network at speeds capable of transmitting raw audio data. With the addition of a 16-bit DAC, some resistors, and a little solder work, the DS80C400 becomes an Internet speaker.

References

[1] Information on the MAX542 can be found at www.maxim-ic.com/quick_view2.cfm/qv_pk/1419.

[2] More information on the Keil compiler can be found at www.keil.com.

[3] More information on the TINI Runtime Environment can be found at www.maxim-ic.com/TINI.

[4] The JavaLayer MP3 suite can be downloaded from www.javazoom.net/javalayer/javalayer.html and is distributed under a GNU public license.

[5] The TINI Software Development Kit can be downloaded from ftp://ftp.dalsemi.com/pub/tini/index.html .

[6] Source code for this application note can be found at ftp://ftp.dalsemi.com/pub/tini/reference_designs/netspeaker/networkspeaker_source.zip.

More Information

DS80C390:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS80C400:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DSTINIm400:	QuickView	-- Full (PDF) Data Sheet	