CpE5160 Experiment #6
Using the Simple Embedded Operating System for the MP3 Player

Introduction

The purpose of this experiment is to use the pieces created during the semester to implement the simple embedded operating system for the MP3 player. The SPI interface created in experiment #2 will be used to read files from the SD card and to write data values to the STA013 MP3 decoder. The TWI functions created in experiment #3 will be used to initialize the STA013 MP3 decoder. The file system created in experiment #4 will be used to select which MP3 file to play and to locate the clusters that hold the file data. The final experiment is to implement a method of reading values from the SD card and writing values to the MP3 decoder so that the data is sent in a continuous stream. The first priority of the microcontroller is to write values to the MP3 decoder when the DATA_REQ pin is active. The second priority is to keep data in internal buffers ready to be written to the MP3 decoder. The values in the buffers are read as blocks from the SD card. The microcontroller must schedule the correct time to read from the SD card because of the length of time that this task takes. The programmer must also keep in mind that sometimes two blocks must be read from the SD card in order to read from the FAT to find the next cluster number and then read the next sector. The programmer may choose to use a super loop or the simple embedded operating system solution to move the data at the required rate. Whichever method is used, an important step in creating the solution is defining all of the tasks the system must perform and placing them in a state diagram.

The super loop used to navigate the directories from experiment #4 can be used as a starting point for this experiment. When a song file is selected, the song will be played instead of printed as it was in experiment #4. The timer 2 interrupt and sEOS only needs to be enabled while the song is playing.

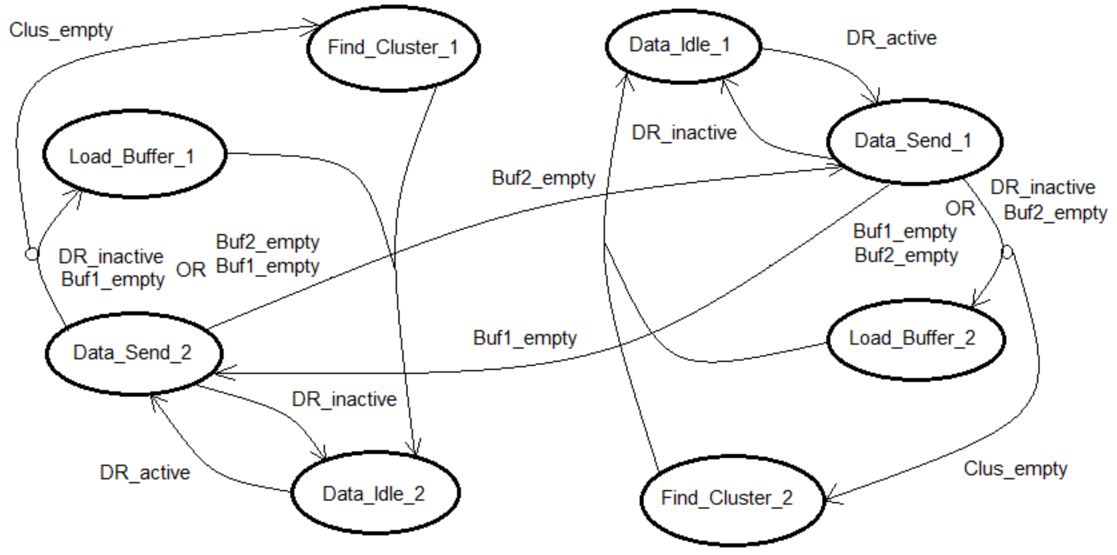
The MP3 Player State Diagram

The first step in creating a state diagram is to identify the tasks required in the system. The tasks listed in the introduction which were writing to the STA013 MP3 decoder and reading from the SD card form a partial list. One task will be writing to the STA013 when the DATA_REQ pin is active. The song file data should be sent as fast as possible until the DATA_REQ pin goes inactive, the buffer(s) in the 8051 are empty or the task takes longer than the interval for Timer 2. I used another timer to create a hardware timeout to force an exit if any task takes too long.

The DATA_REQ pin should be continually monitored. The only time that it is not required to monitor the DATA_REQ pin is immediately after it goes inactive indicating the STA013 buffer is full. The amount of time that it will remain inactive depends on the bit-rate of the MP3 file and it is somewhat predictable. Reading a block from the SD card should occur during this time period.

Only one block can be read from the SD Card in one timer 2 interval. This makes reading a data sector and reading a FAT sector two different tasks and states. If a new cluster number is needed then the state will be changed to read a FAT sector. Once the new cluster number is found then the next data sector can be found. The state diagram that is given does not go directly from reading a FAT sector to reading a data sector. I made the choice that reading an SD card block should only occur after the DATA_REQ pin has gone inactive and then only one block should be read. This allows me to monitor the DATA_REQ pin more often and begin sending data as soon as it goes active. The next data sector is read after the next time the DATA_REQ pin goes inactive. You may choose to read the next data sector immediately after finding the next cluster. At the bit-rates that our songs have in this experiment, there is typically enough time to read two sectors before the DATA_REQ pin goes active again.

One issue that occurs is that at the beginning of a song the MP3 decoder discards several sectors of data and the DATA_REQ pin never goes inactive. This is because the beginning of most MP3 files contains information about the file in the ID3 tag. The MP3 decoder discards this data and looks for the header of the first block of music data. The result of this discarding of data is that we must include some manner of dealing with the buffers being emptied out and the DATA_REQ pin not going inactive. In other words, there must be some method to force the next data sector to be read from the SD Card if both buffers in the 8051 are empty. Since music is not being played yet, the timing of reading sectors from the SD card and keeping the STA013 buffer full is not important. The diagram below shows a possible state diagram.



An initialization function loads both buffers with data and sets the state to be Data_Send_1. Timer 2 is then started with an interval of at least 11ms to give ample time to load a sector from the SD card. The interval will need to be longer if you are not using the 18.432MHz crystal and x2 mode. Data is sent to the STA013 until either the DATA_REQ pin goes inactive or buffer 1 is empty. If the DATA_REQ pin goes inactive, then a check is made to see if the alternate buffer is empty. This allows this buffer to be filled during a predictable idle time. Otherwise, the state is switched to the Data_Idle_1 state. If buffer 1 is empty, then the state is switched to Data_Send_2 which will send data from buffer 2 during the next interrupt. If both buffers are empty, then the state is switched to one of the Load Buffer states. After a buffer is loaded, then the state is switched to a Data Idle state.

When all of the sectors of a cluster have been read, then the next cluster must be located. An “if statement” can be used when the Load Buffer state is set to see if the number of sectors loaded is greater than or equal to the sectors per cluster. If so, then the state is switched to one of the Find Cluster states. After a new cluster value is found, the state is switched to one of the Data Idle states or one of the Load Buffer states.

The Data Idle state can monitor the DATA_REQ pin continuously (with a hardware timeout) or once per ISR. When the DATA_REQ pin is active, the system switches to a Data Send state. For monitoring DATA_REQ continuously, a while loop is used outside of the switch-case statement. If DATA_REQ is active, the state is changed to a Data Send state and then exit the while loop. Otherwise, exit the loop with a timeout.

One final item to note is how to determine when the song is over and return to the print directory loop so the user can select the next song to play. When the last cluster of the file is reached, the FAT entry for that cluster is End of File (0x0FFFFFFF for FAT32 or 0xFFFF for FAT16). The song will likely not continue all the way to the end of the cluster, but all of the sectors can still be sent to the mp3 decoder. Any data that is not part of an mp3 frame will just be discarded by the mp3 decoder. My implementation used a “while(1) loop” with an instruction to place the microcontroller into sleep mode at the end of the loop. At the beginning of the loop, an “if statement” that checks a song complete status with a “break statement” to exit the loop when the song was complete.

The Simple Embedded Operating System

The simple embedded operating system is created using a timer interrupt. The interrupt is caused by the timer overflow flag being set. The amount of time that it takes from the timer start value to the overflow determines the sEOS interval. It is important to note that the interrupt and the sEOS only need to be operating during the time that a song is being played. One of the requirements of the sEOS that we are using is that no other interrupts are used in the system. Another requirement is that the interval be longer that the longest task in our system. The longest task in the system is reading a block (sector) from the SD card. In my experimentation I found that using an 18.432MHz crystal and switching to the x6 clock mode allowed the block to be read in about 10ms. I set my interval to about 11ms or 12ms to have a little extra processing time before an SD card sector is read.

Another requirement for our system is that there is enough time after the DATA_REQ pin goes inactive to read a block from the SD card and that there is enough time to read two blocks from the SD card before the active buffer is emptied. This allows our system to have time to read a FAT sector and read the next sector of a file into the inactive buffer. The rate at which data is required by the STA013 MP3 decoder is determined by the bit-rate of the file which makes slower songs easier for our system to handle. The slower bit rate songs are in the root directory and they have an average bit-rate of

30Kbps. The Posies subdirectory has two songs which are at an average bit-rate of 10Kbps. There are two subdirectories which have the same selection of songs at an average bit-rate of 70Kbps and the original 192Kbps.

I ran into an issue when trying to implement the simple embedded operating system. Any functions that are called by the interrupt service routine and from the main program must be reentrant functions or a warning will occur. A reentrant function can be interrupted during execution from the main program and executed by the interrupt service routine without affecting the results for either function. Our functions have not been written in that manner. One possible solution is to create copies of the functions for use only by the interrupt service routine. This will require a little bit more code memory usage. Another solution is to make the functions non-overlayable. This means that the local variables have a specific location in RAM instead of just in the DATA_GROUP. This may cause some memory problems if you are already running out of RAM memory. It is also recommended, that if you are using this solution that interrupts are disabled whenever you are executing these functions from the main program. Another solution is to ignore the warnings because our system is setup so that interrupts are only used during a specific portion of the program. I choose the first solution by creating copies of functions that were used in the interrupt service routine.

Procedure
There is no step by step procedure for this experiment. The programmer may choose to implement a super loop or the simple embedded operating system approach. The state diagram given in this handout is a guide to get started. The programmer may also wish to choose a different approach than either one of these. If the programmer wants to use the simple embedded operating system approach described in this handout, then it is recommended to first implement the timer interrupt without trying to send any data. Use a simple state diagram to flash LEDs at a defined rate and verify that it is working. Next implement sending just the first two buffers to the STA013. It is unlikely that anything can be heard when sending just two buffers, but you can test if the interrupt function exits okay to the main program. Then try implementing sending one cluster of data. A few of the files do not have an ID3 tag at the beginning and it is possible that you will hear some music if these sectors are successfully sent. Then try locating more than one cluster and sending all of those sectors. A logic analyzer could be useful at some point to verify that the system is working. The programmer could view the BITEN pin to see when data is written to the STA013 and the SD_select pin to see when a data block is read from the SD card. The programmer would be able to see each interval of writing data and reading data blocks. Using LEDs as visual indicators of what action is occurring is another possible debugging tool. If you need more LEDs, feel free to stop by my office.
Important: Do not try to use a “printf statement” in the interrupt service routine because printing one character at 9600 baud takes about 1ms and print phrases will likely take longer than 11ms to execute. A print buffer can be used to print one character at a time. Using a print buffer is described in chapter 9 of “Embedded C” by Pont.

Grading:

Functonality: Implements the state machine described in the handout.	15pts	
Uses the simple Embedded Operating System	5pts	
Submitted code does not compile	**-8pts**	
Organization: Separates microcontroller specific code from application code to make code more reusable.	5pts	
Readability: Code is commented. Defined public and private constants are used as needed.	5pts	
Correctness: Source code does not have any patches to make it work and works as described in the comments.	10pts	
Demonstration: Demonstration of working or non-working MP3 player.	4pts	
The MP3 Player attempts to play song and sound is generated.	4pts	
The MP3 player attempts to play the entire song.	4pts	
The MP3 player exits back to the directory and another song can be selected to play.	4pts	
Playback of the song is good.	4pts	
Total Points		60pts

Up to 20 possible bonus points can be earned by doing one or two of the bonus parts given below.

Bonus #1: (10pts) Display the name of the current song being played and the elapsed time of song playback on the LCD Module. The name should be displayed on line 1 of the LCD and this can be done before the Timer 2 interrupt is started. The time can be displayed by simply keeping track of how many sEOS “ticks” have occurred while playing a song. Use a variable to keep track of the total time and add the amount of the tick interval during each interrupt. When the total time is greater than 1000ms then subtract 1000ms and increment the seconds variable by 1. When the seconds variable is equal to 60, then set it to 0 and increment the minutes variable. Convert the minutes and seconds variables into printable characters and format into a clock display. Care must be taken to only update the LCD Module during idle times (not sending data to the mp3 decoder or loading sectors from the SD card) so that it does not interfere with the playing of the song. Also make sure that sending data to the LCD module does not exceed the sEOS interval.

Bonus #2: (10pts) Implement at least two switch inputs into the sEOS to allow control of the song playback. The simplest switch input would be a stop button which when pressed would cause the playback to exit just as if it reached the end of a song. A pause button could also be implemented. One method is to just stop sending data with a button press and resume sending data with another button press. However this will lead to noise because of the partial frame being sent. You may try using the I2C write function to change the values of the play and mute registers to see if the play back can be paused without noise.

Bonus #3: (20pts) Implement a second simple embedded operating system to replace the super loop used for song selection and directory navigation. All printing would need to be done using a print buffer. The Print_Directory and Read_Directory_Entry functions would need to be re-written, so that only one sector at a time is read and parsed during an sEOS interval. The long_serial_in function could not be used because it uses the UART_Receive function that waits until a character is received. Instead a receive function that just checks to see if a character is received and exits would be needed. The received characters would then need to be parsed to determine the user input. Once a song is selected to play, this sEOS could exit and the sEOS for playing the song could be initialized and started. Another option is to use the same sEOS and have the song selection be another set of states.

Bonus #4: (20pts) Implement a second simple embedded operating system or use a super loop to read the switches and print one directory entry at a time to the LCD module. This can be accomplished by reading in one sector of a directory into XRAM. Use a variation of the read entry function to locate an entry in this sector and then print the 8.3 file name of the entry to the LCD module using a variation of the LCD print function created in experiment #1. Two of the switches can be used to scroll up or down to the next entry. If the end of the sector is reached before the last directory entry is found, then the next directory sector is loaded. Limits should be placed so that the user cannot scroll beyond the first or last entry. Instead of limits the programmer may choose to roll over to the first or last entry. Care should be taken to debounce the switches and not to exceed the tick interval when loading a new sector. Another switch can be used to start playing a song, which would then switch to the sEOS code written for this experiment.