Homework 9: Software Design Considerations

Team Code Name: Drink Mixer **Group No.** 2

Team Member Completing This Homework: Susanne Schmidt

E-mail Address of Team Member: schmidsm @ purdue.edu

Evaluation:

SCORE	DESCRIPTION
10	***Excellent*** *– among the best papers submitted for this assignment. Very few corrections needed for version submitted in Final Report.*
9	***Very good*** *– all requirements aptly met. Minor additions/corrections needed for version submitted in Final Report.*
8	***Good*** *– all requirements considered and addressed. Several noteworthy additions/corrections needed for version submitted in Final Report.*
7	***Average*** *– all requirements basically met, but some revisions in content should be made for the version submitted in the Final Report.*
6	***Marginal*** *– all requirements met at a nominal level. Significant revisions in content should be made for the version submitted in the Final Report.*
*	***Below the passing threshold*** *– major revisions required to meet report requirements at a nominal level.* ***Revise and resubmit.***

* *Resubmissions are due within* ***one week*** *of the date of return, and will be awarded a score of "6" provided all report requirements have been met at a nominal level.*

Comments:

1.0 Introduction

The Drink Mixer is an eight channel audio mixer that is capable of adding effects as well as saving and loading scene settings. The Hammer ARM9 accepts user input from the touch screen, communicates with each channel through that channel's ATmega32A, and communicates with the DSP. The ATmega32A's load scene settings as specified by the ARM9 and monitor changes made manually to the channel. The DSP accepts, processes, and outputs audio from each channel and updates the specific effects processing according to user input given through the ARM9.

2.0 Software Design Considerations

Hammer ARM9

The Hammer is a real time OS with a Linux 2.6.29.6 kernel [1]. We will be programming the device in C++. The Hammer interfaces with each of the ATmega32A's through the I2C connection which means the kernel must be configured to enable this part of the device. The I2C bus speed will be set slow at first, approximately 1kbps, to minimize error in the data. The SPI0 connection will be used to initialize the A/D for each channel as well as the D/A for the output. The same SPI0 connection will be used to communicate with the DSP and the two LED drivers. This interface should be able to run at a maximum speed of 25MHz, however due to the travel length in some areas this device may run at 4MHz. The SPI enable pins for each of these devices are as follows: GPB0, GPB2, GPH0, GPH7, and GPH6. After initializing the I2C connection with the ATmega32A's, the I2C connection will be tested. The QT interface will then be loaded before the Hammer begins its polling loop. The QT interface will be used to update the LCD screen via a frame buffer driver written for the Linux kernel. This interface requires connections to VM, VFRAME, VLINE, VCLK, and LCD_VD0-7 through the SFV20R (ZIF 20-pin connector). The touch screen is connected through the HFWR (ZIF 4-pin) to pins AIN0 and AIN1. This interface will communicate with the frame buffer drivers in the Linux kernel as well as interface directly with the AIN pins to get touch screen settings. The 100ms of sleep at the beginning of the code loop will keep the hammer from occupying 100% of its cycles with processing data and allow time for the QT events to occur. Within the loop, the I2C connection will be checked and the appropriate registers will be updated with changes that have been made to the individual channels. The DSP will then be updated with these changes over the SPI connection. The Hammer will use the amplitude sent by the DSP to update the LED drivers, also through SPI.

ATmega32A

The ATmega32A contains 32kB of flash memory, upon which the application program is stored. There are also 2143 bytes of SRAM, and 1024 bytes of data EEPROM memory [2]. It will be programmed using the AVR-GCC compiler and AVR Studio. Because the development environment manages memory independently from the programmer, a discussion of memory addressing and data storage locations is omitted. The ATmega32A is interrupt driven in order to prioritize the various tasks it has and control debouncing. Upon power up, initialization routines will initialize the A/D converter as well as set the GPIO settings. The processor will then sleep until an interrupt occurs. Within the timer interrupt, various peripherals will be checked and updated. The fader A/D (connected to pin ADC0) will be checked and the corresponding registers updated. Because it will be polled within the timer interrupt, the A/D will operate in a non-continuous mode. After that button presses and any changes to the RPG will be checked. The illuminated pushbutton is connected to T0, T1, and ADC1, which are configured as GPI pins. The RPG is connected to TCK and TMS, which are also configured for GPI. A TWI interrupt occurs when the Hammer is sending information over the I2C connection. The I2C connection is connected to SCL and SDA. If the ATmega32A is reading information from the Hammer, it will set flags for what needs to be changed and then use PWM to move the fader. OC0 and OC2 are used to communicate with an H bridge, which sets the fader position. On one microcontroller, an additional H-bridge is connected to OC1A and OC1B. A 10-segment LED bar graph, connected to TDO, TDI, TOSC1, TOSC2, and ADC2-7 (all configured to act as general-purpose output), will be updated as the Hammer commands. If the Hammer is requesting information, the register value containing current user-input information will be sent.

DSP

The DSP is interrupt driven in order to ensure that audio processing takes place as quickly as possible. Being interrupt driven is actually very common with DSP based devices and is considered one of the only ways to operate them correctly. Within the first interrupt, data will be added to the channel buffer and a filter will be applied. One of the great features of a DSP that gives it great speed improvements is the hardware implemented circular buffer. This buffer allows data to collect while processing is occurring in the main DSP. The data is received from the A/D's through an I2S connection on DAIP1-4 at 24 bits and 96kHz. It will then be sent to the output buffer and the FX interrupt will be fired. External SRAM has been interfaced using AD0-15, NWR, NRD, and ALE. The 512kB external SRAM will enable us the memory to process effects such as delay, which requires a fair amount of extra memory. Within the FX interrupt, effects processing will take place before the interrupt for the output being placed in the output buffer is fired. The output is sent to the D/A's through the I2S connection using DAIP11-14. The SPI interrupt is fired when information from the Hammer is being sent. The DSP receives the command, sends back the amplitude for the Hammer to display through the LED drivers and then updates the filter transforms and gain settings according to the user input that has been relayed. The DSP is programmed through the JTAG header, which is attached to TMS, TCK, NTRST, TDI, and NEMU.

3.0 Software Design Narrative

Hammer ARM9

At application start the system will first initialize the SPI bus and I2C interface by setting the appropriate registers. These registers are unimportant in mentioning here due to the fact that they are handled by the Linux kernel automatically. Once these interfaces are initialized the program will start a new process to handle the graphical user interface side of the application. The process prioritizing will be divided up based on the algorithms built into the Linux kernel.

After we have completed the initialization stage of the application a program loop will begin. In this loop a series of modules will be activated after a 100ms wait. The first of the modules will be to check to see if we have data to send out to the Atmels. This data will correspond to changing values on the individual channel peripherals (i.e. moving a fader). If this information exists the I2C module of the code will be instructed to update values of the necessary channels. After this is performed the I2C module will then request updated changes from the channels, which had no new information to go out. This information is then stored in a series of variables. All user interface values are stored on the hammer. After the I2C system has finished doing its job updated settings will be sent and requested from the QT process. This will allow the user interface to be properly updated as well as get changes that were made through the touch screen interface. This module is not yet completed.

After the communication with the QT interface is complete all the modified settings will be sent out over the SPI interface to the DSP. These settings will be sent out in a series of bytes corresponding to an 8-bit register followed by the assigned value. The DSP will interpret this information and update the effects processing filters. This module is started but not completed.

At the end of this process a request will be made for the amplitude of the left and right output channels from the DSP. These amplitudes will be sent to the Hammer over the SPI interface. The Hammer will then recycle this data and send it out to the LED drivers. The reason the Hammer is performing this task as opposed to the DSP has to do with limited GPIO pins on the DSP and the fact that the Hammer is operating as the master on the SPI bus. The system will then wait for a small amount of time and begin again. This module is not yet completed.

A separate operating process on the Hammer will be the QT user interface. While it is preferred that the QT interface process simply act as a thread of the main application, it will depend on issues with the QT interface and these tasks. In the event that they can, then the user settings variables will be mutually accessible and can be used between threads via mutex locking. This interface will display detailed information regarding values on settings of each individual channel as well as effects settings. This aspect of the code has a partial set of completed parts. The initialization code for the various data interfaces as well as the frame buffer driver for the LCD screen is mostly complete. The one complication that is occurring is an outdated embedded Linux support causing failed Linux builds. We have been working with the designer of the Hammer to update this documentation and patch the latest kernel versions.

ATmega32A

The ATmega32A program will operate in an interrupt-driven loop. Using the timer module of the device a series of procedures will be run every 2 ms. These procedures will check if the on/off button has been pressed and set the appropriate flag. It will check the value of the fader position and update the variable corresponding to its value. It will also check to see if the RPG has been rotated and if so which direction it has been rotated. For the RPG a count will be kept with a positive or negative integer. If the number in this variable is positive then the RPG has been turned clockwise that many times and if negative, CCW respectively. This information will be stored for the next interrupt of this application.

On a periodic basis the Hammer will require updated information from the Atmels. This information will be requested via the I2C interface (or in Atmel terms, TWI interface). When the interface receives a request with its corresponding device number an interrupt will be fired. The Atmel will then determine if it is a read request or a write request. If it is a read request the data stored in the variables from the previous module will be sent to the Hammer. If it is a write request, data sent from the Hammer will be stored in those variables. After this interrupt is performed a check will be made to compare the value of the fader with the desired value made by the Hammer. If these values differ, the PWM module will be initialized and instruct the fader to move until the desired value is asserted. The PWM interface is working with the H-bridge; the fader value can be read and asserted.

DSP

At the start of the program the circular buffer is enabled for the I2S interface on the DAI pins. The SPI interface is activated in slave mode to receive commands from the Hammer. Once the input pins are initialized to accept TDM data from the A/D's, the output pins are configured. After this is performed an interrupt table is activated.

The first interrupt is the data input ready interrupt. This interrupt is fired every time a series of 24-bits of data is ready for processing. This data is added to a buffer for its corresponding channel and a star-transform is applied to act as a 128th-order FIR filter. Once this filter is processed it is tacked into a channel output value ready to be added to the main output after all 8 channels are processed.

The second interrupt is the effects interrupt. This interrupt is triggered after all 8 inputs have been asserted. This will combine all the final channel filter values to create the output value and also tack it into an output effects buffer. This buffer will then be assessed and its output sent to the output buffer. After this second interrupt has fired an output ready interrupt will be triggered. This third interrupt will send the final set of output data to the output circular buffer to go to the D/A's.

The fourth and final interrupt involves the SPI interface. This interface will be triggered when a full byte of data is received from the SPI interface. Once this interrupt has accumulated the full number of bytes needed for the first byte instruction, some actions will be performed. If settings have been changed on channel equalizer settings then the FIR filter coefficients will need to be updated. This can be done with the DSP's built in FFT optimized functions. If gain settings need to be adjusted then the corresponding variables will be adjusted for processing. The SPI interrupt will be responsible for sending back the LED driver bits containing the amplitude of the left and right channel. So far the DSP code has successfully been able to handle basic gain control, panning, and delay via a talk through style interface. Once the PCB board is complete it will be significantly easier to develop.

4.0 Summary

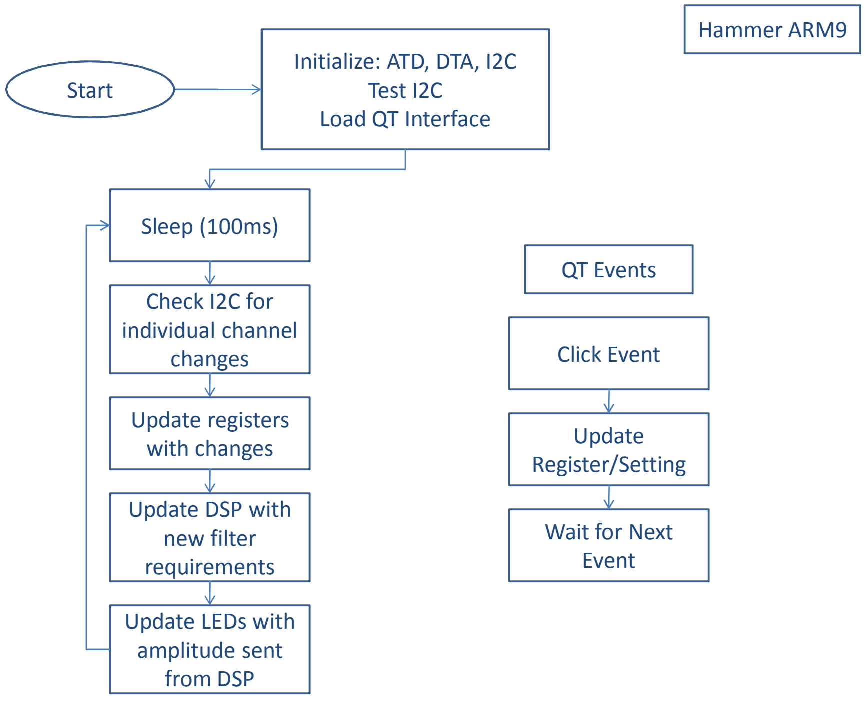
There is still a great deal of code to be written and tested for all of these devices, but much of it is simply a matter of establishing communication between two devices. With the outline we have created and the knowledge we have, we are confident that it can be coded.

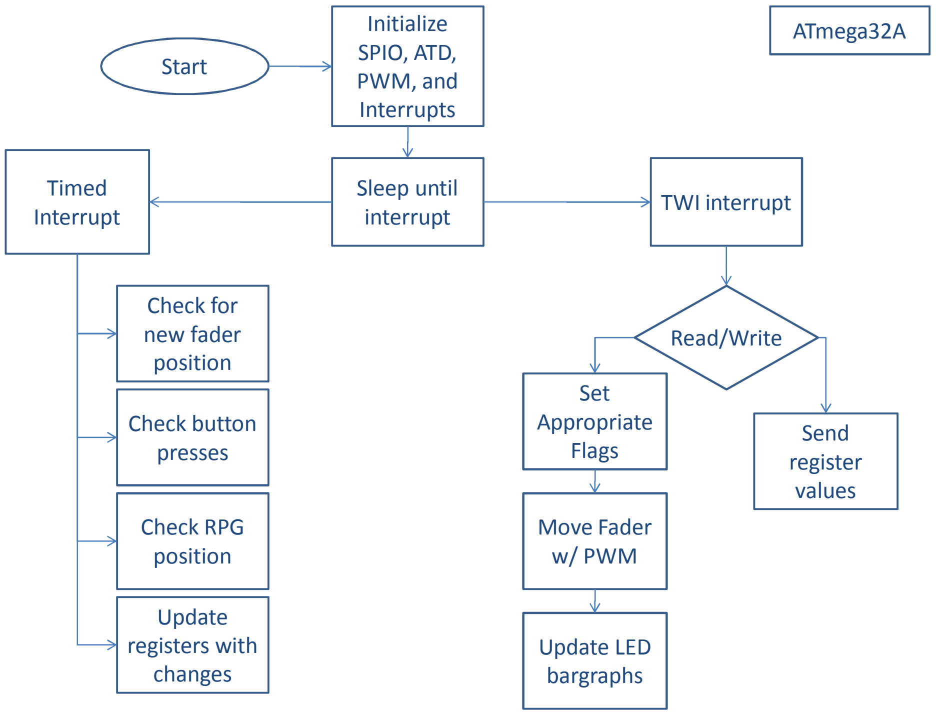
List of References

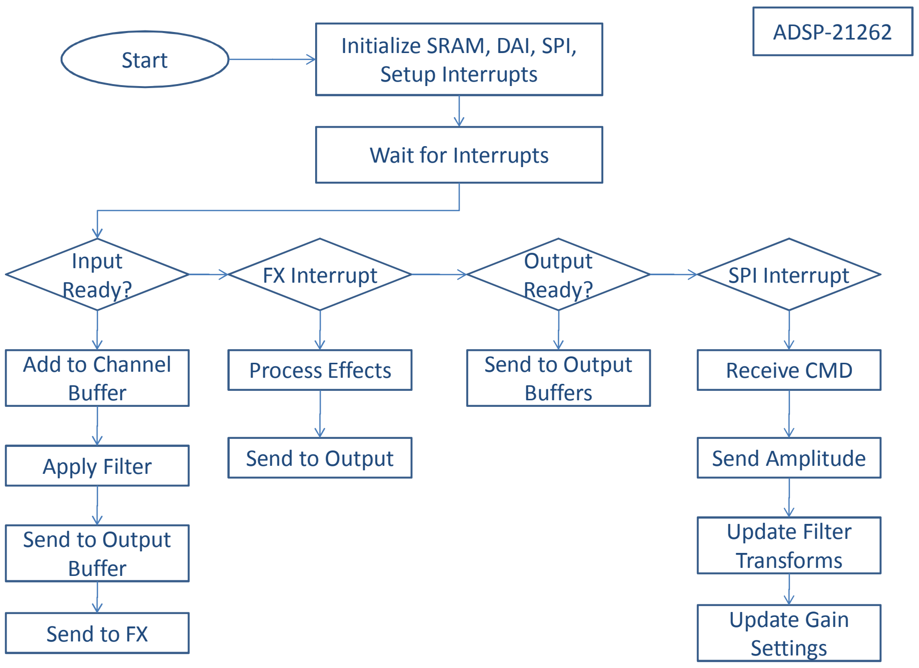
[1] "Hammer," [Online], Available: http://tincantools.com/product.php?productid=16133&cat=0&page=1. [Accessed: Sept 15, 2009].

[2] "8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash: ATmega32A," [Online], Available: http://www.atmel.com/dyn/resources/prod_documents/doc8155.pdf. [Accessed: Oct 29, 2009].

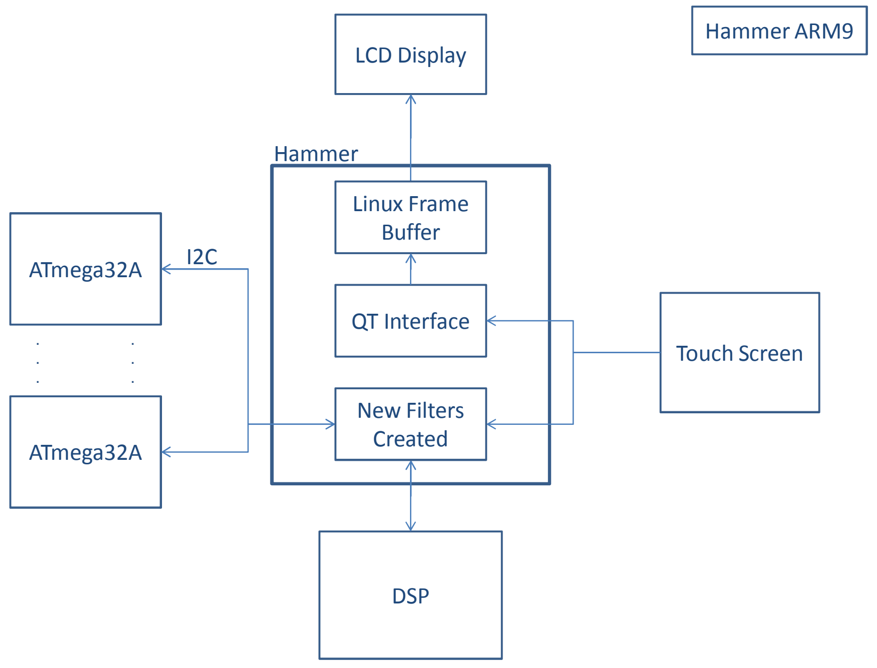
Appendix A: Flowchart/Pseudo-code for Main Program

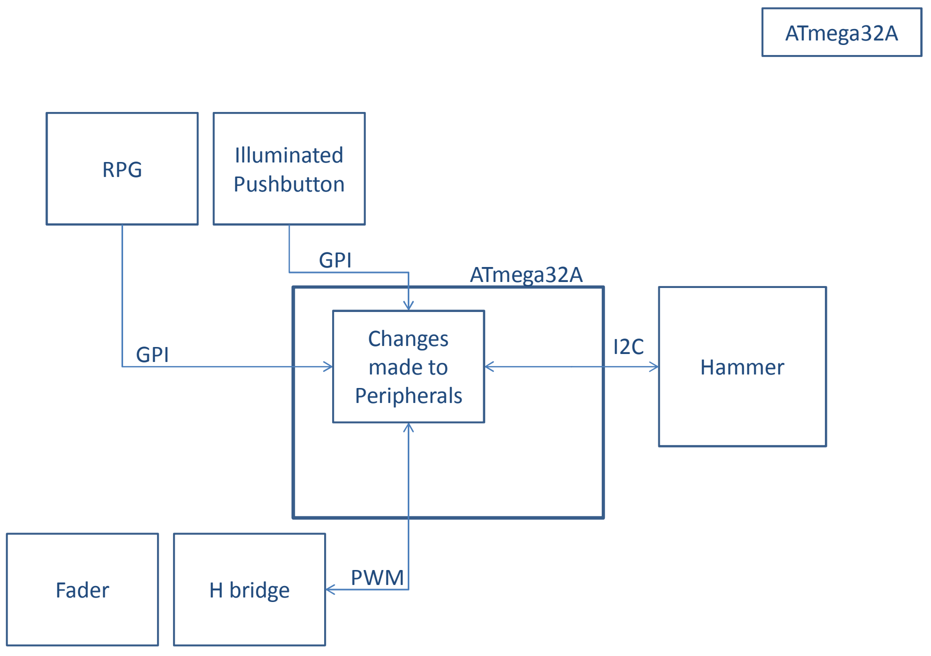

Hammer ARM9
Start
Initialize: ATD, DTA, I2C
Test I2C
Load QT Interface
Sleep (100ms)
Check I2C for individual channel changes
Update registers with changes
Update DSP with new filter requirements
Update LEDs with amplitude sent from DSP
QT Events
Click Event
Update Register/Setting
Wait for Next Event

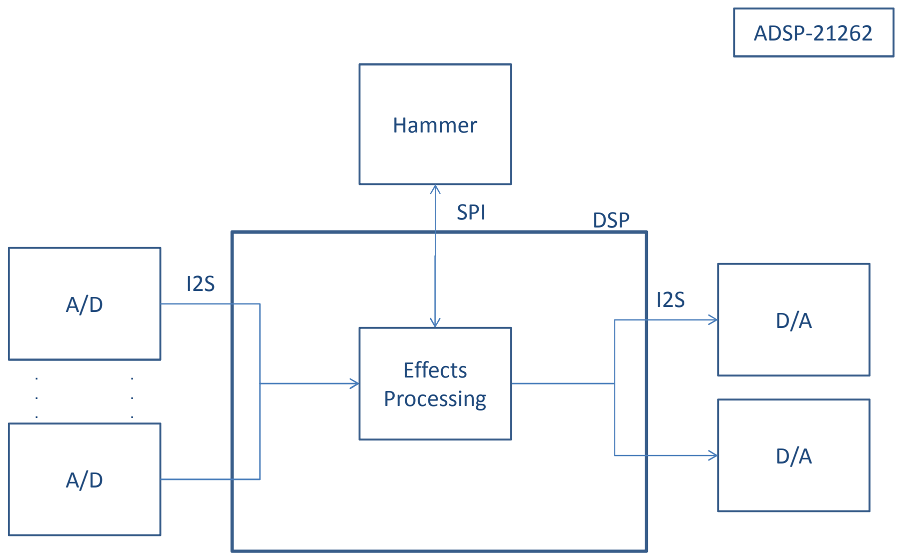



ADSP-21262
Start
Initialize SRAM, DAI, SPI, Setup Interrupts
Wait for Interrupts
Input Ready?
FX Interrupt
Output Ready?
SPI Interrupt
Add to Channel Buffer
Apply Filter
Send to Output Buffer
Send to FX
Process Effects
Send to Output
Send to Output Buffers
Receive CMD
Send Amplitude
Update Filter Transforms
Update Gain Settings

Appendix B: Hierarchical Block Diagram of Code Organization






ADSP-21262
Hammer
SPI
DSP
A/D
I2S
A/D
Effects Processing
I2S
D/A
D/A