# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Manual</td>
<td>3</td>
</tr>
<tr>
<td>Hardware Manual</td>
<td>11</td>
</tr>
<tr>
<td>System Overview</td>
<td>11</td>
</tr>
<tr>
<td>Memory Map</td>
<td>15</td>
</tr>
<tr>
<td>Original Schematic</td>
<td>16</td>
</tr>
<tr>
<td>Current Schematic</td>
<td>17</td>
</tr>
<tr>
<td>Current Schematic with revisions labeled</td>
<td>18</td>
</tr>
<tr>
<td>PCB Top Layers</td>
<td>19</td>
</tr>
<tr>
<td>PCB Bottom Layers</td>
<td>22</td>
</tr>
<tr>
<td>Rotary Encoder</td>
<td>25</td>
</tr>
<tr>
<td>Display</td>
<td>28</td>
</tr>
<tr>
<td>ROM</td>
<td>30</td>
</tr>
<tr>
<td>SRAM</td>
<td>32</td>
</tr>
<tr>
<td>IDE</td>
<td>35</td>
</tr>
<tr>
<td>DRAM</td>
<td>37</td>
</tr>
<tr>
<td>Buffers</td>
<td>44</td>
</tr>
<tr>
<td>Reset Circuitry</td>
<td>45</td>
</tr>
<tr>
<td>Appendix: Complete CPLD file</td>
<td>46</td>
</tr>
<tr>
<td>Software Manual</td>
<td>52</td>
</tr>
<tr>
<td>Software System Overview</td>
<td>52</td>
</tr>
<tr>
<td>Boot Code</td>
<td>53</td>
</tr>
<tr>
<td>Rotary Encoder Software</td>
<td>64</td>
</tr>
<tr>
<td>Display Software</td>
<td>80</td>
</tr>
<tr>
<td>IDE Software</td>
<td>103</td>
</tr>
<tr>
<td>Audio Software</td>
<td>116</td>
</tr>
<tr>
<td>Interrupts</td>
<td>131</td>
</tr>
<tr>
<td>Schematic and PCB Revisions</td>
<td>140</td>
</tr>
</tbody>
</table>
User Manual

System Components

The following is a list of components in the system.

1. ADSP-BF533 Blackfin Processor (U11)
2. 1MB (64 kB x 16-Bit) IDTV016SA SRAM (U13)
3. 16MB (4 MB x 32-Bit) MT8D432(X) DRAM (SIMM1)
4. 4 MB (512 kB x 8-Bit) AM29LV040B ROM (U12)
5. ispLSI2032A CPLD (U2)
6. 74LS157 2-to-1 Multiplexer (U4, U5, U6)
7. Rotary Encoder (SW2)
8. IDE Hard Drive (J3) (external)
9. LCD Display (LCD1) (external)
10. MP3 Decoder Board (J4) (external)
11. Speakers (external)

Specifications

• Rotary Encoder used for user interface.
• LCD Display to show system status. The LCD Display includes 11 characters for the track name, 11 characters for the artist name, 8 characters for the status of the system, and 6 digits for the time on the track.
• Audio output using an MP3 decoder board.

Setup

The following steps must be taken in order to ensure that the system has been setup properly:

1. Plug in the MP3 Decoder board into the connector at the bottom of the board.
2. Ensure that the MP3 Decoder board is connected to a power source, and to working speakers.

3. Plug the IDE into the board via the IDE connector. Ensure that the red strip on the cable is oriented towards the top of the board.
4. Plug the display onto the board using the appropriate cable. Ensure that the red strip on the cable is oriented towards the bottom of the board.

5. Plug in power to the IDE using an appropriate power supply.
6. Plug in a power cable on the left side of the board in order to turn on the system. The red pushbutton switch is useful to determine whether the system is on or not. Once pressed, the neighboring LED should light up. Also, the display should look like the one in the picture below.

7. If setup correctly, the display should turn on after a few seconds to display the user interface.
User Interface

Once the system has powered on and all initial setup has been completed, the system begins in the Idle state. The user can navigate through directories and songs by using the <Track Up> or <Track Down> actions and then begin playing the desired track. The user interacts with the system by using the Rotary Switch. The user can rotate through different key codes – each key code corresponds to a different action. The user can then pick a desired action by pressing the switch. The key code is displayed on the top right portion of the display. The display also shows the current status of the system on the top center portion of the display, the current artist (if applicable) on the top left, the current song title or directory on the bottom left, and the song time in milliseconds on the bottom right. An example of the display interface is shown below.

![Display Interface Example]

As the above image shows, the display truncates the track title if it is too long (longer than 11 characters).

The display also indicates the system state. The system state descriptions follow.

System State Descriptions

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Waiting for input.</td>
</tr>
<tr>
<td>Play</td>
<td>A track is playing.</td>
</tr>
<tr>
<td>Repeat Play</td>
<td>A track is playing repeatedly.</td>
</tr>
<tr>
<td>Fast Forward</td>
<td>A track is being fast forwarded.</td>
</tr>
<tr>
<td>Reverse</td>
<td>A track is being reversed.</td>
</tr>
</tbody>
</table>

The display also indicates the current key code. Each key code corresponds to a different action. A mapping of key codes to actions is provided below.
Key Code Mappings

<table>
<thead>
<tr>
<th>Key Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Track Up</td>
</tr>
<tr>
<td>1</td>
<td>Track Down</td>
</tr>
<tr>
<td>2</td>
<td>Play</td>
</tr>
<tr>
<td>3</td>
<td>Repeat Play</td>
</tr>
<tr>
<td>4</td>
<td>Fast Forward</td>
</tr>
<tr>
<td>5</td>
<td>Reverse</td>
</tr>
<tr>
<td>6</td>
<td>Stop</td>
</tr>
</tbody>
</table>

The function of each action follows.

Action Descriptions

<table>
<thead>
<tr>
<th>Action</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track Up</td>
<td>Moves to the next song in the directory. If at the last song, wraps to the first song.</td>
</tr>
<tr>
<td>Track Down</td>
<td>Moves to the previous song in the directory. If at the first song, wraps to the last song.</td>
</tr>
<tr>
<td>Play</td>
<td>If the current selection is a song, plays the song from the current position on the track. Playing stops at the end of the song, or if the Stop action is issued. If the current selection is a playlist, the playlist becomes the active playlist.</td>
</tr>
<tr>
<td>Repeat Play</td>
<td>Repeatedly plays the current song selection.</td>
</tr>
<tr>
<td>Fast Forward</td>
<td>Moves the current position on the track forward, faster than normal play (by at least a factor of two), without playing the track. Fast forwarding stops at the end of the track, or if the Fast Forward or Stop actions are issued.</td>
</tr>
<tr>
<td>Reverse</td>
<td>Moves the current position on the track backward, faster than normal play (by at least a factor of two), without playing the track. Reversing stops at the beginning of the track, or if the Reverse or Stop actions are issued.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops the current action, or resets to the beginning of the current track or</td>
</tr>
</tbody>
</table>
Each action has a different effect depending on the state of the system. The following tables describe the various effects that each action has in a given state.

### Idle State Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track Up</td>
<td>Go to the start of the next directory or track.</td>
</tr>
<tr>
<td>Track Down</td>
<td>Go to the start of the previous directory or track.</td>
</tr>
<tr>
<td>Play</td>
<td>Begin playing the current track or move into the selected directory.</td>
</tr>
<tr>
<td>Repeat Play</td>
<td>Begin playing the current track repeatedly.</td>
</tr>
<tr>
<td>Fast Forward</td>
<td>Fast forward the current track.</td>
</tr>
<tr>
<td>Reverse</td>
<td>Reverse the current track.</td>
</tr>
<tr>
<td>Stop</td>
<td>Reset to the start of the directory or track.</td>
</tr>
</tbody>
</table>

### Play State Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track Up</td>
<td>n/a</td>
</tr>
<tr>
<td>Track Down</td>
<td>n/a</td>
</tr>
<tr>
<td>Play</td>
<td>n/a</td>
</tr>
<tr>
<td>Repeat Play</td>
<td>When the end of the track is reached, the song will start over again.</td>
</tr>
<tr>
<td>Fast Forward</td>
<td>Stop playing the current track and start to fast forward from current position.</td>
</tr>
<tr>
<td>Reverse</td>
<td>Stop playing the current track and start to reverse from current position.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stop playing and reset to start of track.</td>
</tr>
</tbody>
</table>

### Repeat Play State Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track Up</td>
<td>n/a</td>
</tr>
<tr>
<td>Track Down</td>
<td>n/a</td>
</tr>
<tr>
<td>Play</td>
<td>When the end of the track is reached, the song will not start over again.</td>
</tr>
<tr>
<td>Repeat Play</td>
<td>n/a</td>
</tr>
<tr>
<td>Fast Forward</td>
<td>Stop playing the current track and start to fast forward from current position.</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Reverse</td>
<td>Stop playing the current track and start to reverse from current position.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stop playing and reset to start of track.</td>
</tr>
</tbody>
</table>

**Fast Forward State Actions**

<table>
<thead>
<tr>
<th>Action</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track Up</td>
<td>n/a</td>
</tr>
<tr>
<td>Track Down</td>
<td>n/a</td>
</tr>
<tr>
<td>Play</td>
<td>Begin playing from the current position.</td>
</tr>
<tr>
<td>Repeat Play</td>
<td>Begin playing from the current position, and when the end of the track is reached, the song will start over again.</td>
</tr>
<tr>
<td>Fast Forward</td>
<td>Stop fast forwarding.</td>
</tr>
<tr>
<td>Reverse</td>
<td>Start to reverse from current position.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stop fast forwarding.</td>
</tr>
</tbody>
</table>

**Reverse State Actions**

<table>
<thead>
<tr>
<th>Action</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track Up</td>
<td>n/a</td>
</tr>
<tr>
<td>Track Down</td>
<td>n/a</td>
</tr>
<tr>
<td>Play</td>
<td>Begin playing from the current position.</td>
</tr>
<tr>
<td>Repeat Play</td>
<td>Begin playing from the current position, and when the end of the track is reached, the song will start over again.</td>
</tr>
<tr>
<td>Fast Forward</td>
<td>Start to fast forward from current position.</td>
</tr>
<tr>
<td>Reverse</td>
<td>Stop reversing.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stop reversing.</td>
</tr>
</tbody>
</table>

Note that the user can reset the system by plugging out the power connector from the board and plugging out the power to the IDE, waiting a few seconds, and then plugging the cables back in.
Once powered on, the system takes a few seconds to boot up. During this time, code is loaded from the ROM to the Blackfin processor (CPU) and begins to execute. The code sets up the CPU clock rates, sets up the External Bus Interface so that the CPU can read and write to memory mapped I/O (ROM, SRAM, IDE, DRAM). Next, various system components are initialized, including the Display. Once this initialization is finished, the user interface shows up on the display.

The user interacts with the system by using the Rotary Encoder. The current key code corresponding to the current action that the user may take is indicated on the top right on the display. The user can rotate left and right in order to change the current key code, and press the button to have the system take the appropriate action (for more information on the user interface, consult the User Manual). The Display shows either the current directory, or the current song title on the bottom left, and the current artist on the top left. The bottom right portion of the display shows the song length in milliseconds. The top center portion of the display shows the system status.

There are several actions that the user may take. When the user rotates the Rotary Encoder, the CPLD uses logic to generate the appropriate signals for the CPU, which handles the rotation by changing the key code. When the user selects an action by pressing the switch, the CPU handles the press and takes the appropriate action. Some examples of actions follow in order to demonstrate how the system functions as a whole.

The user may wish to change the directory or move to another song in the directory. In this case, data from the IDE is retrieved and stored into DRAM memory. The CPU takes the data from the DRAM, and loads it into its own processor memory for immediate usage. Then, it sends the data to the Display to update the song title, artist, time, and status appropriately. Note that there are several buffers between the CPU and other components. This is to properly handle voltage differences from the signals that are connected to the CPU (3.3V) and signals that are connected to other components (5V).

The user may also wish to play a song. When this option is selected, audio data is loaded from the IDE into the DRAM, and from the DRAM to the internal memory on the CPU. The CPU outputs this data (retrieving data from the DRAM memory bank as needed) through a serial connection to the MP3 Decoder, which then outputs sound through the Speakers.

Note that the SRAM may be used instead of the DRAM by changing a constant in the code. More information on this can be found in the Software Manual. Also note that every read/write access to the DRAM is made possible by the CPLD, which uses
logic in order to control the timing on the signals in order to make sure that the **DRAM** signal timing matches with that of the **CPU** control signals.

The above descriptions of common user operation demonstrate how the various system components interact with one another. A block diagram is provided below in order to give a more graphical overview of the system and to illustrate the signal flow between the blocks. A picture of the actual PCB with color-coded blocks follows.

A block diagram of the entire system.
The next diagram shows the actual PCB, with the blocks outlined.

The actual PCB with labeled blocks. The blocks are the Rotary Encoder (white), the MP3 decoder connector (orange), the Display connector (teal), the CPLD (dark pink), the CPU (blue), the SRAM (green), the DRAM (yellow), the ROM (red), the buffers (brown), and the IDE connector (purple). Note that the actual MP3 decoder, the IDE, and the display are all external devices that connect to the board.
The next diagram shows the board layout.

The Board Layout.
Memory Map

The following diagram illustrates the memory layout of the Blackfin processor on the PCB.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF FFFF</td>
<td>Core MMR Registers (2M Byte)</td>
</tr>
<tr>
<td>0xFFE0 0000</td>
<td>System MMR Registers (2M Byte)</td>
</tr>
<tr>
<td>0xFFC0 0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0xFFB0 1000</td>
<td>Scratchpad SRAM (512K Byte)</td>
</tr>
<tr>
<td>0xFFB0 0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0xFFA1 4000</td>
<td>Instruction SRAM/CACHE (16K Byte)</td>
</tr>
<tr>
<td>0xFFA1 0000</td>
<td>Instruction SRAM (32K Byte)</td>
</tr>
<tr>
<td>0xFFA0 8000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0xFFA0 0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0xFF90 8000</td>
<td>Data Bank B SRAM/CACHE (16K Byte)</td>
</tr>
<tr>
<td>0xFF90 4000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0xFF80 8000</td>
<td>Data Bank A SRAM/CACHE (16K Byte)</td>
</tr>
<tr>
<td>0xFF80 4000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0xEF00 4000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0x2040 0000</td>
<td>DRAM (SIMM1)</td>
</tr>
<tr>
<td>0x2030 0000</td>
<td>IDE (J3)</td>
</tr>
<tr>
<td>0x2020 0000</td>
<td>SRAM (U13)</td>
</tr>
<tr>
<td>0x2010 0000</td>
<td>ROM (U12)</td>
</tr>
<tr>
<td>0x2000 0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0x0800 0000</td>
<td>SDRAM Memory (16M Byte to 128M Byte)</td>
</tr>
</tbody>
</table>
Rotary Encoder

The user interacts with the system through the Rotary Encoder. The rotary encoder allows the user to navigate through the menu and select songs to play.

The rotary encoder has two signals, RE_A and RE_B, which can be used to determine the direction of the rotation. The rotary encoder uses Gray Code to vary RE_A and RE_B appropriately as the user rotates left or right. Out of the four possible combinations of the RE_A and RE_B signals, two of them are détentes. The RE_A and RE_B signals always land on one of these détentes after completing a turn.

The rotary encoder has a third signal that can be used to determine when the user has pressed the rotary switch. The signal, which was connected to PF13 on the CPU, was pulled high with a 10 kΩ pull-up resistor. When the rotary encoder is pressed, the pin shorts to ground. Thus, PF13 is high when the switch is not pressed, and low when the switch is pressed. This is how the software determines when to register a press. Note that switch debouncing is also done in software.

The CPLD does logic to determine the direction that a user turned. It takes the RE_A and RE_B signals as inputs and generates two outputs – the RIGHT signal and the TURN signal. The RIGHT signal runs through a buffer and drives PF11, which is connected to the CPU. This signal is high if the user turned right and low if the user turned left. The TURN signal runs through a buffer and drives PF12, which is also connected to the CPU. The turn signal goes high when the CPLD registers a turn, and is low otherwise. Thus, the turn signal is used to generate interrupts on rising edge so that an interrupt handler can respond appropriately to the user turning the rotary encoder.

The following diagram summarizes the interaction of the Rotary Encoder with other system components.

![Block Diagram for the Rotary Encoder](image-url)
CPLD Logic

It was necessary to write ABEL code to generate the proper RIGHT and TURN signal given the RE_A and RE_B signals from the Rotary Encoder in order to properly determine the types of turns and when they occur. The code is reproduced below.

" Pins

Clock pin 11; "input Clock (Y0)
RE_A pin 17; "input first input from rotary encoder
RE_B pin 18; "input second input from rotary encoder
RIGHT pin 31 ISTYPE 'reg'; "output moved right or left
TURN pin 32 ISTYPE 'reg'; "output turn occurred

LA pin ISTYPE 'reg'; "last value of RE_A
LB pin ISTYPE 'reg'; "last value of RE_B
LDA pin ISTYPE 'reg'; "last detente value of RE_A
LDB pin ISTYPE 'reg'; "last detente value of RE_B
HRIGHT pin ISTYPE 'reg'; "holds value of right for extra clock

EQUATIONS

" output enables

RIGHT.OE = 1;
TURN.OE = 1;
BUFF_RESET.OE = 1;
BUFF_TOGGLE.OE = 1;
RAS.OE = 1;
CAS0.OE = 1;
CAS1.OE = 1;
Select.OE = 1;

" clocks for the outputs

LA.CLK = Clock;
LB.CLK = Clock;
LDA.CLK = Clock;
LDB.CLK = Clock;
RIGHT.CLK = Clock;
TURN.CLK = Clock;
HRIGHT.CLK = Clock;

" Rotary Encoder equations

" update value on clock to store last A and B values
LA := RE_A;
LB := RE_B;

" determine last detente value of A and B
LDA := LDA $ ((!LDA & RE_A) & (!LDB & RE_B) & (RE_A & RE_B));
LDB := LDB $ ((!LDA & RE_A) & (!LDB & RE_B) & (RE_A & RE_B));

" if not turning, hold direction, or else if turning, determine " direction based on current A and B value and previous A and B value
HRIGHT := (HRIGHT & !TURN) # (TURN & ((!RE_A & RE_B) & (LA & LB)) # ((RE_A & !RE_B) & (!LA & !LB)));
if RE_A and RE_B are each different from LDA and LDB, a turn has occurred

TURN := (RE_A $ LDA) & (RE_B $ LDB);

update value of right signal based on last value
RIGHT := HRIGHT;

END

The code essentially just keeps track of the last values of RE_A and RE_B seen, as well as the last détente values of RE_A and RE_B seen in order to determine in which direction the user turned, or if the user even turned at all.
**Display**

The LCD display (LCD1) is used to indicate the state of the system. It displays the current song title, artist, song time, system state, and the current key code that represents the action that the user can currently take by pressing the rotary switch.

The display is connected to the board via a connector and a ribbon cable. To write to the display, the CPU sets PF0-7 appropriately. These signals go through a buffer and drive the display data pins. The CPU can also set PF8 to drive the RS signal going to the display to indicate whether it is issuing a command to the display, or simply sending data. The CPU sets PF9 to drive the R/W signal on the display to indicate a read or a write, and PF10 to drive the E (enable) signal on the display.

When writing to the display, the CPU must drive the control signals on the display appropriately to conform to the display timing requirements. The CPU also has to initialize the display by writing the appropriate start-up commands with the correct wait times between sending each command. Both of these timing requirements are met through software.

The following diagram summarizes how the display interacts with other system components.

![Diagram of display interaction](image)

The CPU sets PF0-10 appropriately to write to the LCD display.

**LCD Display Write Cycle Timing**

The following diagram shows the timing and state of various signals during a write to the LCD display.
See the Software Manual for more details on how every write to the display meets these timing requirements (see the DisplayChar function in particular).
ROM

The ROM (U12) is used to load code onto the Blackfin processor on bootup. It is located in memory at Memory Bank 0 (AMS0) at addresses 0x20000000-0x200FFFFF.

The ROM chip used is the AM29LV040B with the 32-pin PLCC package. It holds 4 MB of memory (512 kB x 8-Bit).

The following diagram shows how the ROM is oriented on the board. Note that it is contained in a socket, so the pins on the bottom side of the board will be different than what is shown here. However, this is an quick reference for the pins that are directly connected to the ROM.

![Diagram showing ROM orientation on the board.](image)

The AMS0 signal is connected to the Chip Select pin (CE), the AOE signal is connected to the Output Enable pin (OE), and the AWE signal is connected to the Write Enable pin (WE). The Vss pin is grounded and the Vcc pin is connected to 3.3V. The Data pins are connected to the data bus, and the address pins are connected to the address bus.

**ROM Read Cycle Timing**

The following diagram illustrates the timing and state of various signals for a ROM read cycle. Various delays, setup and hold times are also labeled.
At the beginning of the read access, there is one setup cycle. At the end of this cycle, the write enable goes inactive (to indicate a read). Then, there are 3 read access cycles. At the end of these cycles, the CPU reads valid data from the ROM. Then, there is one hold cycle. Notice that the specified number of cycles was chosen in order to satisfy the setup and hold times required by the CPU to read the ROM successfully, while minimizing the read access time.
The SRAM (U13) could have been used instead of the DRAM (SIMM1) for storing the audio data that was retrieved from the IDE (J3). It is located in memory at Memory Bank 1 (AMS1) at addresses 0x20100000-0x201FFFFFF.

The SRAM chip used is the IDTV016SA with the 44-pin SOJ package. It is a 3.3V CMOS Static RAM and holds 1 MB of memory (64 kB x 16-Bit).

The following diagram shows how the SRAM is oriented on the board. Pins 11 and 33 were connected to 3.3V while pins 12 and 34 were connected to ground. Pin 6 was connected to AMS1, the chip select signal. Pin 17 was connected to AWE, the write enable signal. Pin 41 was connected to AOE, the output enable signal. Pins 40 and 39 were connected to ABE1 and ABE0, the high byte enable and low byte enable signals. The I/O pins were connected to the data bus while the address pins were connected to the address bus.

**SRAM Read Cycle Timing**

The following diagram illustrates the timing and state of various signals for an SRAM read cycle. Various delays, setup and hold times are also labeled.
Initially, there is one setup cycle, followed by a read cycle. At this time, the CPU reads valid data from the SRAM. This is followed by one hold cycle. Note that using the specified cycles satisfies the setup and hold times specified by the CPU, while minimizing the SRAM read access time.

**SRAM Write Cycle Timing**

The following diagram illustrates the timing and state of various signals for the SRAM write cycle. Various delays, setup and hold times are also labeled.

Initially, there is one setup cycle, followed by a write access cycle. At the beginning of the write access cycle, the write enable signal goes active. At the end of the cycle, the CPU writes data to the SRAM. A hold cycle follows. The appropriate signals then
go inactive. Note that using the specified number of cycles satisfies the setup and hold times required by the SRAM while minimizing the SRAM write access time.
**IDE**

The IDE (J3) is the hard drive used to store MP3 songs within playlists and directories. It is the main source of data for the Blackfin MP3 Jukebox. It is located in memory at Memory Bank 2 (AMS2) at addresses 0x20200000-0x202FFFFFF.

There are various registers that must be written to in order to extract data from the IDE. Either CHS or LBA addressing mode may be used to retrieve the data. For more information on extracting data from the IDE, see the Software Manual.

The IDE is connected to the board using a connector and a 40-pin ribbon cable.

**IDE Read Cycle Timing**

The following diagram illustrates the timing and state of various signals for an IDE read cycle. Various delays, setup and hold times are also labeled.

![IDE Read Cycle Timing Diagram](image)

At the start of the read access, there are four setup cycles, after which the read enable signal goes low to indicate the beginning of the read access cycles. There are 15 read access cycles. At the end of these cycles, the CPU reads valid data from the IDE. Next, there are 3 hold cycles. A transition time of 4 cycles has also been marked on the diagram. This is the minimum number of wait states between one read cycle and another read or write cycle. The specified number of cycles satisfies the setup and hold times requested by the IDE and the CPU when data is being read. Note that initially, there were less wait cycles allocated to the IDE read cycle process, but the IDE seemed to operate better with an increased number of wait cycles, so the timing was maximized.
IDE Write Cycle Timing

The following diagram illustrates the timing and state of various signals for an IDE write cycle. Various delays, setup and hold times are also labeled.

At the start of the write access, there are four setup cycles, after which the write enable signal goes low to indicate the beginning of the write access cycles. There are 15 write access cycles. At the end of these cycles, the CPU reads valid data from the IDE. Next, there are 3 hold cycles. A transition time of 4 cycles has also been marked on the diagram. This is the minimum number of wait states between one write cycle and another read or write cycle. The specified number of cycles satisfies the setup and hold times requested by the IDE when data is being written. Note that initially, there were less wait cycles allocated to the IDE write cycle process, but the IDE seemed to operate better with an increased number of wait cycles, so the timing was maximized.
The DRAM (SIMM1) is used to store audio data retrieved from the IDE (J3). It is located in memory on Chip Select 3 (AMS3) at addresses 0x20300000-0x203FFFFF.

The DRAM chip used is the MT8D432(X) with the 72-pin SIMM package. The DRAM module holds 16MB of memory (4 MB x 32 Bit). On the board, there are 3 multiplexers that lead to a socket that contains the DRAM module. The multiplexers are necessary to ensure that the correct address bits are active when accessing either the RAS or CAS lines (the Row Address Select or Column Address select) on the DRAM module. The CPLD handles the logic for interfacing with the DRAM and its corresponding multiplexers.

The following block diagram shows how the DRAM interacts with other components on the board.

The CPLD takes the AMS3, AWE, ABE1, and ABE0 signals as inputs and generates the RAS, CAS0, CAS1, and Select signals appropriately as outputs. The Select signal is an input to the multiplexers, which select either the row or the column address bits from the CPU to pass on to the DRAM module. The RAS, CAS0, and CAS1 signals let the DRAM fetch the memory at a given address using the row and the column address. The CAS0 and CAS1 signals are there to differentiate between low and high byte accesses. Note that the CPLD has to conform to the timing set up for the DRAM (which has been specified in the appropriate register corresponding to Chip Select 3 on the Blackfin processor).
DRAM Read Cycle Timing (w/Hidden Refresh)

The following diagram illustrates the timing and state of various signals for a DRAM read cycle. Various delays, setup and hold times are also labeled. Note that buffer and MUX delays have also been taken into account.

At the start of the read access, there is one setup cycle, during which the Select signal goes high to indicate Row Address access. Next, there are 8 cycles of read access. At the beginning of the read access cycles, the RAS signal goes active. After 3 more cycles, the appropriate CAS signal(s) goes active (the CPLD determines which CAS signal(s) go active). Note that by this time, the Select signal has gone low to indicate Column Address access. After two more cycles the RAS signal goes inactive, and then active again. This is in order to refresh the DRAM module. At the end of the read access cycles, the RAS and appropriate CAS signal(s) go inactive together, and the CPU reads from the DRAM. This is followed by a hold cycle. Note that the CPU
setup and hold times are satisfied. The specified number of wait states was chosen in order to account for all of the delays and to comply with the CPU and DRAM timing requirements while minimizing the DRAM read access time.

**DRAM Write Cycle Timing**

The following diagram illustrates the timing and state of various signals for a DRAM write cycle. Various delays, setup and hold times are also labeled. Note that buffer and MUX delays have also been taken into account.

At the start of the write access, there is one setup cycle, during which the Select signal goes high to indicate Row Address access. Next, there are 6 cycles of write access. At the beginning of the write access cycles, the RAS signal goes active. After 3
more!cycles,!the!appropriate!CAS!signal(s)!go!active!(the!CPLD!determines!which
CAS!signal(s)!go!active).!Note!that!by!this!time,!the!Select!signal!has!gone!low!to
indicate!a!Column!Address!access. At!this!time,!the!CPU!writes!to!the!DRAM. At!the
end!of!the!write!access!cycles,!the!RAS!and!CAS!signals!go!inactive!together. A!hold
cycle!follows. Note!that!the!setup!and!hold!times!specified!by!the!DRAM!module!are
satisfied!for!the!write!cycle. The!specified!number!of!wait!states!was!chosen!in!order!
to!account!for!the!various!signal!delays!and!to!satisfy!the!timing!requirements!set
forth!by!the!DRAM!module!and!the!CPU!while!minimizing!the!write!access!time.

CPLD Logic

In!order!to!satisfy!the!DRAM!read!and!write!access!timing,!it!was!necessary!to!write
ABEL!code!to!set!the!various!DRAM!control!signals!appropriately!with!respect!to!the
CPU!control!signals. The!code!is!reproduced!below. Note!that!a!state!machine!was
used!in!order!to!efficiently!set!the!outputs.

" Pins
Clock  pin 11; "input  Clock (Y0)
AMS3  pin 20; "input  Chip Select 3 (DRAM)
AWE  pin 19; "input  Write Enable
ABE1  pin 21; "input  high byte enable
ABE0  pin 22; "input  low byte enable
Select  pin 37 ISTYPE 'reg, buffer'; "output  whether row or
"column address is active
"currently
RAS  pin 38 ISTYPE 'reg, buffer'; "output  row address select
CAS0  pin 39 ISTYPE 'reg, buffer'; "output  low byte column
"address select
CAS1  pin 40 ISTYPE 'reg, buffer'; "output  high byte column
"address select
ReadWrite  pin ISTYPE 'reg, buffer'; "whether it is a read or write
"cycle
CAS_state  pin ISTYPE 'reg, buffer'; "used for state machine
"purposes
Trans1  pin ISTYPE 'reg, buffer'; "used for state machine
"purposes
Trans2  pin ISTYPE 'reg, buffer'; "used for state machine
"purposes

"the states
StateBits = [ Select, RAS, CAS_state, Trans1, Trans2 ]; " state bits

TheStart  =  [ 0, 0, 0, 0, 0 ]; " the starting state
Idle  =  [ 1, 1, 0, 0, 0 ]; " the idle state
Start  =  [ 1, 1, 1, 0, 1 ]; " start to read or write
RASLow  =  [ 1, 0, 1, 0, 0 ]; " RAS goes low
RASLowWait  =  [ 1, 0, 1, 1, 0 ]; " wait while RAS low
RASLowWait2  =  [ 1, 0, 1, 1, 1 ]; " keep waiting while RAS
" low
SelectColumn  =  [ 0, 0, 1, 1, 0 ]; " select column address
CASLow  =  [ 0, 0, 0, 1, 0 ]; " CAS goes low
CASLowWait  =  [ 0, 0, 0, 1, 1 ]; " wait while CAS low
RefreshStart  =  [ 0, 1, 0, 1, 1 ]; " start refresh by sending


EQUATIONS

" output enables

RAS.OE = 1;
CAS0.OE = 1;
CAS1.OE = 1;
Select.OE = 1;

" clocks for the outputs

StateBits.CLK = Clock;
CAS0.CLK = Clock;
CAS1.CLK = Clock;
ReadWrite.CLK = Clock;

" State Diagram for DRAM

STATE_DIAGRAM StateBits "a Moore state machine

STATE TheStart: " the initial state
        GOTO Idle;

STATE Idle: "idle state waiting for the beginning of a cycle

        CAS0 = 1;
        CAS1 = 1;
        IF (!AMS3) THEN Start " ready to start cycle if chip select is active
        ELSE Idle; " or just wait here

STATE Start: " start to read or write
        CAS0 = CAS0;
        CAS1 = CAS1;
        GOTO RASLow;

STATE RASLow: " RAS goes low

        CAS0 = CAS0;
        CAS1 = CAS1;
        GOTO RASLowWait;

STATE RASLowWait: " wait while RAS low

        CAS0 = CAS0;
        CAS1 = CAS1;
        GOTO RASLowWait2;

STATE RASLowWait2: " wait while RAS low again
CAS0 = CAS0;
CAS1 = CAS1;
GOTO SelectColumn;

STATE SelectColumn: " select column address

CAS0 = ABE0; " set CAS lines according to the high and low byte " enables
CAS1 = ABE1;
ReadWrite = AWE; " information on whether read or write stored
GOTO CASLow;

STATE CASLow: " CAS goes low

CAS0 = CAS0;
CAS1 = CAS1;
ReadWrite = ReadWrite;
GOTO CASLowWait;

STATE CASLowWait: " wait while CAS low

CAS0 = CAS0;
CAS1 = CAS1;
ReadWrite = ReadWrite;
IF (ReadWrite) THEN RefreshStart " If reading, should refresh
ELSE EndWriteState; " If writing, then should not refresh

STATE EndWriteState: " end write

CAS0 = 1;
CAS1 = 1;
GOTO EndState;

STATE RefreshStart: " start refresh

CAS0 = CAS0;
CAS1 = CAS1;
GOTO RefreshEnd;

STATE RefreshEnd: " end refresh

CAS0 = 1;
CAS1 = 1;
GOTO EndState;

STATE EndState: " end of read cycle

CAS0 = CAS0;
CAS1 = CAS1;
GOTO Idle; " ready to wait for next cycle

END

The following state diagram is a visual representation of the ABEL code.
Buffers

There are several buffers that are used in the system. The buffer used is the 74LVT16245 3.3V 16-Bit Bus Transceiver with Tri-State Outputs. The buffers are used to account for voltage differences between components that are 3.3V and components that are 5V.

The buffer direction pins were normally tied either high or low depending on the signals on either side of the buffer and considering which signal should drive the value of the other signal. Most of the connections to the buffer direction pins are straightforward, and most of the output enables on the buffers are tied to be active. However, there are some exceptions.

Buffer U17 is the data buffer. It buffers the data pins that are connected directly to the CPU from the data pins that go to 5V components. The direction pins on the buffer are connected to AWE, the write enable signal. This is because during a write, the signals from the CPU should drive the signals going to the other components, and doing a read, the opposite should be true. The output enable pins on the buffer were connected to the BUFF_TOGGLE signal, which is a CPLD output. The CPLD essentially enables the buffer when AMS0 or AMS1 is not active, since these chip select signals correspond to ROM and SRAM accesses, which are the only components that do not use the buffered data bus. When the data bus is being used by non-buffered components, the buffer should not be enabled (otherwise, bus conflict issues may occur).

The CPLD code for enabling the data buffer is reproduced below.

```plaintext
" Pins
BUFF_TOGGLE   pin 41 ISTYPE 'com'; "output the toggle for the data buffer
AMS0          pin 25;   "input Chip Select 0 (ROM)
AMS1          pin 26;   "input Chip Select 1 (SRAM)

EQUATIONS

" output enables
BUFF_TOGGLE.OE = 1;

" Data buffer equation
BUFF_TOGGLE = !AMS0 # !AMS1; " data buffer should be enabled when ROM/RAM not in use

END
```
Reset Circuitry

Originally, the reset circuitry was designed to use a reset chip and a pushbutton switch. However, due to certain complications (see the revisions section), the system does not include a reset switch. Some CPLD logic is used to drive the CPU reset signal with the JTAG (debugger) reset. The logic is rather simple – the CPLD just sets the CPU reset to take on the value of the JTAG reset signal. The CPLD code is reproduced below.

```plaintext
BUFF_JRESET     pin   30;  "input   JTAG reset
BUFF_RESET      pin   15 ISTYPE 'com'; "output the buffered reset 
                      " signal from CPU

EQUATIONS

" output enables
BUFF_RESET.OE = 1;

" Reset equation

!BUFF_RESET = !BUFF_JRESET;

END
```
**Appendix: Complete CPLD Logic**

```plaintext
MODULE     cpldlogic
TITLE      'CPLD Logic'

" PLD DEVICE  'ISPLSI2032PGA'

" Note: Pin numbers correspond to 44-pin PLCC CPLD

" Description:  The logic for the CPLD that handles DRAM reads and writes, the
" logic for determining turns for the rotary encoder, and other
" logic.

" Revision History:
" 02/07/14   Initial Revision.
" 05/01/14   Removed Switch Reset due to board problems.
" 06/17/14  Updated comments.

" Pins

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFF_JRESET</td>
<td>pin 30; &quot;input JTAG reset&quot;</td>
</tr>
<tr>
<td>AMS0</td>
<td>pin 25; &quot;input Chip Select 0 (ROM)&quot;</td>
</tr>
<tr>
<td>AMS1</td>
<td>pin 26; &quot;input Chip Select 1 (SRAM)&quot;</td>
</tr>
<tr>
<td>BUFF_RESET</td>
<td>pin 15 ISTYPE 'com'; &quot;output the buffered reset signal from CPU&quot;</td>
</tr>
<tr>
<td>BUFF_TOGGLE</td>
<td>pin 41 ISTYPE 'com'; &quot;output the toggle for the data buffer&quot;</td>
</tr>
<tr>
<td>Clock</td>
<td>pin 11; &quot;input Clock (Y0)&quot;</td>
</tr>
<tr>
<td>RE_A</td>
<td>pin 17; &quot;input first input from rotary encoder&quot;</td>
</tr>
<tr>
<td>RE_B</td>
<td>pin 18; &quot;input second input from rotary encoder&quot;</td>
</tr>
<tr>
<td>RIGHT</td>
<td>pin 31 ISTYPE 'reg'; &quot;output moved right or left&quot;</td>
</tr>
<tr>
<td>TURN</td>
<td>pin 32 ISTYPE 'reg'; &quot;output turn occurred&quot;</td>
</tr>
<tr>
<td>AMS3</td>
<td>pin 20; &quot;input Chip Select 3 (DRAM)&quot;</td>
</tr>
<tr>
<td>AWE</td>
<td>pin 19; &quot;input Write Enable&quot;</td>
</tr>
<tr>
<td>ABE1</td>
<td>pin 21; &quot;input high byte enable&quot;</td>
</tr>
<tr>
<td>ABE0</td>
<td>pin 22; &quot;input low byte enable&quot;</td>
</tr>
<tr>
<td>Select</td>
<td>pin 37 ISTYPE 'reg, buffer'; &quot;output signal that decides whether row or column address is active&quot;</td>
</tr>
<tr>
<td>RAS</td>
<td>pin 38 ISTYPE 'reg, buffer'; &quot;output row address select&quot;</td>
</tr>
<tr>
<td>CAS0</td>
<td>pin 39 ISTYPE 'reg, buffer'; &quot;output column address select&quot; (low byte)</td>
</tr>
<tr>
<td>CAS1</td>
<td>pin 40 ISTYPE 'reg, buffer'; &quot;output column address select&quot; (high byte)</td>
</tr>
<tr>
<td>LA</td>
<td>pin ISTYPE 'reg'; &quot;last value of RE_A&quot;</td>
</tr>
<tr>
<td>LB</td>
<td>pin ISTYPE 'reg'; &quot;last value of RE_B&quot;</td>
</tr>
<tr>
<td>LDA</td>
<td>pin ISTYPE 'reg'; &quot;last detente value of RE_A&quot;</td>
</tr>
<tr>
<td>LDB</td>
<td>pin ISTYPE 'reg'; &quot;last detente value of RE_B&quot;</td>
</tr>
<tr>
<td>HRIGHT</td>
<td>pin ISTYPE 'reg'; &quot;holds value of right for extra clock&quot;</td>
</tr>
<tr>
<td>ReadWrite</td>
<td>pin ISTYPE 'reg, buffer'; &quot;whether it is a read or write cycle&quot;</td>
</tr>
</tbody>
</table>
```
CAS_state  pin  ISTYPE 'reg, buffer'; "used for state machine purposes
Trans1  pin  ISTYPE 'reg, buffer'; "used for state machine purposes
Trans2  pin  ISTYPE 'reg, buffer'; "used for state machine purposes

"the states

StateBits = [ Select, RAS, CAS_state, Trans1, Trans2 ]; " state bits

TheStart  = [ 0, 0, 0, 0, 0 ]; " the starting state
Idle  = [ 1, 1, 0, 0, 0 ]; " the idle state
Start  = [ 1, 1, 1, 0, 1 ]; " start to read or write
RASLow  = [ 1, 0, 1, 0, 0 ]; " RAS goes low
RASLowWait  = [ 1, 0, 1, 1, 0 ]; " wait while RAS low
RASLowWait2  = [ 1, 0, 1, 1, 1 ]; " keep waiting while RAS low
SelectColumn  = [ 0, 0, 1, 1, 0 ]; " select column address
CASLow  = [ 0, 0, 0, 1, 0 ]; " CAS goes low
CASLowWait  = [ 0, 0, 0, 1, 1 ]; " wait while CAS low
RefreshStart  = [ 0, 1, 0, 1, 1 ]; " start refresh by sending RAS high
RefreshEnd  = [ 0, 0, 0, 1, 1 ]; " end refresh by sending RAS low
EndWriteState  = [ 0, 0, 1, 0, 1 ]; " almost done with write
EndState  = [ 0, 1, 1, 1, 1 ]; " done with cycle, RAS, CAS go high

EQUATIONS

" output enables

RIGHT.OE = 1;
TURN.OE = 1;
BUFF_RESET.OE = 1;
BUFF_TOGGLE.OE = 1;
RAS.OE = 1;
CAS0.OE = 1;
CAS1.OE = 1;
Select.OE = 1;

" clocks for the outputs

LA.CLK = Clock;
LB.CLK = Clock;
LDA.CLK = Clock;
LDB.CLK = Clock;
RIGHT.CLK = Clock;
TURN.CLK = Clock;
HRIGHT.CLK = Clock;
StateBits.CLK = Clock;
CAS0.CLK = Clock;
CAS1.CLK = Clock;
ReadWrite.CLK = Clock;

" Rotary Encoder equations

" update value on clock to store last A and B values
LA := RE_A;
LB := RE_B;

" determine last detente value of A and B
LDA := LDA $ ((!LDA & RE_A) & (!LDB & RE_B) & (RE_A & RE_B));
LDB := LDB $ ((!LDA & RE_A) & (!LDB & RE_B) & (RE_A & RE_B));

" if not turning, hold direction, or else if turning, determine direction based
" on current A and B value and previous A and B value
HRIGHT := (HRIGHT & !TURN) # (TURN & ((!RE_A & RE_B) & (LA & LB)) # ((RE_A & !RE_B) & (!LA & !LB)));

" if RE_A and RE_B are each different from LDA and LDB, a turn has occurred
TURN := (RE_A $ LDA) & (RE_B $ LDB);

" update value of right signal based on last value
RIGHT := HRIGHT;

" Reset equation
!BUFF_RESET = !BUFF_JRESET;

" Data buffer equation
BUFF_TOGGLE = !AMS0 # !AMS1; " data buffer should be enabled when ROM/RAM not in use

" State Diagram for DRAM
STATE_DIAGRAM StateBits "a Moore state machine

STATE TheStart: " the initial state
    GOTO Idle;

STATE Idle: "idle state waiting for the beginning of a cycle
    CAS0 = 1;
    CAS1 = 1;
    IF (!AMS3) THEN Start " ready to start cycle if chip select is active
    ELSE Idle; " or just wait here

STATE Start: " start to read or write
    CAS0 = CAS0;
    CAS1 = CAS1;
    GOTO RASLow;

STATE RASLow: " RAS goes low
    CAS0 = CAS0;
    CAS1 = CAS1;
    GOTO RASLowWait;

STATE RASLowWait: " wait while RAS low
    CAS0 = CAS0;
CAS1 = CAS1;
GOTO RASLowWait2;

STATE RASLowWait2: " wait while RAS low again

    CAS0 = CAS0;
    CAS1 = CAS1;
    GOTO SelectColumn;

STATE SelectColumn: " select column address

    CAS0 = ABE0; " set CAS lines according to the high and low byte enables
    CAS1 = ABE1;
    ReadWrite = AWE; " information on whether read or write stored
    GOTO CASLow;

STATE CASLow: " CAS goes low

    CAS0 = CAS0;
    CAS1 = CAS1;
    ReadWrite = ReadWrite;
    GOTO CASLowWait;

STATE CASLowWait: " wait while CAS low

    CAS0 = CAS0;
    CAS1 = CAS1;
    ReadWrite = ReadWrite;
    IF (ReadWrite) THEN RefreshStart " If reading, should refresh
    ELSE EndWriteState; " If writing, then should not refresh

STATE EndWriteState: " end write

    CAS0 = 1;
    CAS1 = 1;
    GOTO EndState;

STATE RefreshStart: " start refresh

    CAS0 = CAS0;
    CAS1 = CAS1;
    GOTO RefreshEnd;

STATE RefreshEnd: " end refresh

    CAS0 = 1;
    CAS1 = 1;
    GOTO EndState;

STATE EndState: " end of read cycle

    CAS0 = CAS0;
    CAS1 = CAS1;
    GOTO Idle; " ready to wait for next cycle
" test vectors for DRAM
TEST_VECTORS
( [ Clock, AMS3, AWE, ABE1, ABE0 ] -> [ ReadWrite, Select, RAS, CAS0, CAS1, Trans1, Trans2 ] )

" The first test is for a read/refresh cycle with low byte enable

[ .C., 1, 1, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " go to idle
[ .C., 1, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " go to idle
[ .C., 1, 1, 0 ] -> [ .X., .X., 1, 1, 1, .X., .X. ]; " go to idle

[ .C., 0, 1, 1, 0 ] -> [ .X., 1, 1, 1, 1, .X., .X. ]; " in start state,
" about to read/write

[ .C., 0, 1, 1, 0 ] -> [ .X., 1, 0, 1, 1, .X., .X. ]; " RAS goes low
[ .C., 0, 1, 1, 0 ] -> [ .X., 1, 0, 1, 1, .X., .X. ]; " wait while RAS low
[ .C., 0, 1, 1, 0 ] -> [ .X., 1, 0, 1, 1, .X., .X. ]; " wait while RAS low
[ .C., 0, 1, 1, 0 ] -> [ .X., 0, 1, 1, .X., .X. ]; " select column
[ .C., 0, 1, 1, 0 ] -> [ 1, 0, 0, 1, .X., .X. ]; " CAS goes low
[ .C., 0, 1, 1, 0 ] -> [ 1, 0, 0, 1, .X., .X. ]; " wait while CAS low
[ .C., 0, 1, 1, 0 ] -> [ .X., 0, 1, 1, .X., .X. ]; " start refresh
[ .C., 0, 1, 1, 0 ] -> [ .X., 0, 1, 1, .X., .X. ]; " end refresh
[ .C., 1, 1, 1, 0 ] -> [ .X., .X., 1, 1, .X., .X. ]; " go back to idle

" The first test is for a read/refresh cycle with both low and high byte enable

[ .C., 1, 1, 0, 0 ] -> [ .X., 1, 1, 1, 1, .X., .X. ]; " in start state,
" about to read/write

[ .C., 0, 1, 0, 0 ] -> [ .X., 1, 1, 1, 1, .X., .X. ]; " RAS goes low
[ .C., 0, 1, 0, 0 ] -> [ .X., 1, 1, 1, 1, .X., .X. ]; " wait while RAS low
[ .C., 0, 1, 0, 0 ] -> [ .X., 1, 1, 1, 1, .X., .X. ]; " wait while RAS low
[ .C., 0, 1, 0, 0 ] -> [ .X., 0, 1, 1, .X., .X. ]; " select column
[ .C., 0, 1, 0, 0 ] -> [ 1, 0, 0, 0, .X., .X. ]; " CAS goes low
[ .C., 0, 1, 0, 0 ] -> [ 1, 0, 0, 0, .X., .X. ]; " wait while CAS low
[ .C., 0, 1, 0, 0 ] -> [ .X., 0, 1, 1, .X., .X. ]; " start refresh
[ .C., 0, 1, 0, 0 ] -> [ .X., 0, 1, 1, .X., .X. ]; " end refresh
[ .C., 1, 1, 0, 0 ] -> [ .X., .X., 1, 1, .X., .X. ]; " go back to idle

" This test is for a write cycle with low byte enable

[ .C., 1, 1, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " go to idle
[ .C., 1, 1, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " go to idle

[ .C., 1, 1, 1, 0 ] -> [ .X., 1, 1, 1, .X., .X. ]; " in start state,
" about to read/write

[ .C., 0, 0, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " RAS goes low
[ .C., 0, 0, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " RAS goes low
[ .C., 0, 0, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " RAS goes low
[ .C., 0, 0, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " RAS goes low
[ .C., 0, 0, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " RAS goes low
[ .C., 0, 0, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " RAS goes low
[ .C., 0, 0, 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ]; " RAS goes low
[ .C., 0, 0, 1, 0 ] -> [ .X., 0, 0, 1, 1, .X., .X. ];  " select column
[ .C., 0, 0, 1, 0 ] -> [ 0, 0, 0, 0, 1, .X., .X. ];  " CAS goes low
[ .C., 0, 0, 1, 0 ] -> [ 0, 0, 0, 0, 1, .X., .X. ];  " wait while CAS low
[ .C., 0, 0, 1, 0 ] -> [ 0, 0, 0, 0, 1, .X., .X. ];  " almost at end of write cycle
[ .C., 0, 0, 1, 0 ] -> [ .X., 0, 1, 1, 1, .X., .X. ];  " at end
[ .C., 1, 1, 1, 0 ] -> [ .X., .X., 1, 1, 1, .X., .X. ];  " go back to idle

" test vectors for data buffer toggle
TEST_VECTORS

( [ AMS0, AMS1 ] -> [ BUFF_TOGGLE ] )

"When AMS0 or AMS1 is active (low), enable on buffer should be inactive (high)
[ 0, 0 ] -> [ 1 ];
[ 0, 1 ] -> [ 1 ];
[ 1, 0 ] -> [ 1 ];

"When AMS0 and AMS1 are inactive (high), enable on buffer should be active (low)
[ 1, 1 ] -> [ 0 ];

" test vectors for reset
TEST_VECTORS

( [ BUFF_JRESET ] -> [ BUFF_RESET ] )

[ 1 ] -> [ 1 ];
[ 0 ] -> [ 0 ];

( [ BUFF_JRESET, BUFF_SWITCH_RESET ] -> [ BUFF_RESET ] )

" Input resets are inactive
" [ 1, 1 ] -> [ 1 ];

" One input reset is active, so JTAG reset should be active
"[ 0, 1 ] -> [ 0 ];
"[ 1, 0 ] -> [ 1 ];

" Both inputs are active, JTAG reset should be active
"[ 0, 0 ] -> [ 0 ];

" test vectors for rotary encoder
TEST_VECTORS

( [ Clock, RE_A, RE_B] -> [LA, LB, LDA, LDB, HRIGHT, RIGHT, TURN] )

[ .C., 1, 0 ] -> [ .X., .X., .X., .X., .X., .X., .X. ];

END resetlogic
Software System Overview

The software runs the entire system. First, the boot code runs. The boot code essentially initializes all parts of the system. This includes setting up the system and core clocks on the CPU, setting up the Asynchronous Memory Banks, initializing the rotary encoder, the display, the IDE, the SPI serial interface, the timer, and enabling interrupts. Then, the main loop is called. The main loop runs the user interface by using the display software. When the user interacts with the rotary encoder, interrupts occur in order to update shared variables appropriately. The main loop constantly polls these shared variables to find out if the system received user input, and if so, which action the user wants to take. Most actions involve using the IDE software to retrieve data from the IDE, and using the audio software to play the song.

The following diagram summarizes the interaction between all files in the system.
**Boot Code**

The software that runs immediately after starting the system performs some important initialization. It first writes to the appropriate registers in the Blackfin processor to have the PLL generate the proper core clock and system clock rates. Then, the boot code sets up the Asynchronous Memory Banks by writing to the appropriate registers. The code enables each bank and also sets up the timing for each bank (the number of setup cycles, read cycles, write cycles, and hold cycles). For more information on the timing of each bank, see the ROM (Bank0), SRAM (Bank1), IDE (Bank2), and DRAM (Bank3) sections of the Hardware Manual.

Afterwards, the boot code writes to the appropriate registers to set up the Programmable Flag pins (PF pins). This includes configuring them as inputs/outputs, enabling PF interrupts on the appropriate PF pins, whether the pins are active high or low, whether the PF pins are edge or level sensitive, and enabling the PF inputs to be read.

The rest of the boot code simply calls the other functions that were mentioned in other parts of the Software Manual. See the code for more details.

The boot code is reproduced below. The relevant files are crt0.s, bfinstart.inc, and bfinregaddr.inc. Note that the bfinregaddr.inc file contains the Blackfin register addresses for the whole system to use.
Initialization file for EE52 Blackfin MP3 project. It sets up the IRQ vector table and sets the processor to supervisor mode. The user must implement project specific initialization such as clock setup and calling initialization functions.

Revision History:

03/24/2013 Josh Fromm Initial revision
06/01/2014 Ajay Mandlekar Updated boot code to include initialization for various interrupts and external bus interface units.

The .ALIGN directive forces the address alignment of an instruction or data item. Here, ensure the first element of the section is aligned.*/
.ALIGN 4;

Set the scope of the '_main' symbol to global. This makes the symbol available for reference in object files that are linked to the current one. Use '.EXTERN' to refer to a global symbol from another file.*/
.extern _main /* need to jump to the mainloop after initializing */.
.extern _KeyInit
.extern _Display_Init
.extern _serial_init
.extern _InitTimer0
.extern _InterruptInit
.extern _InstallTimer0EventHandler
.extern _InstallPFEventHandlers
.extern _InstallSerialEventHandler
.extern _IDE_Init

.global __start /* allow other files to see this function */
.include "src/sys/crt0.inc"
.include "src/sys/boot.inc"
.include "src/sys/bfinstart.inc"
.include "src/sys/bfinregaddr.inc"

.equ IVB, 0xFFE02000
.equ IVG5, 0xFFE02014
.equ UNASSIGNED_VAL, 0x8181
.equ INTERRUPT_BITS, 0x400 // just IVG15
.equ SYSCFG_VALUE, 0x30
.equ STACK_MEM, 0xFFB01000
.equ IPEND, 0xFFE02108
.equiv IMASK, 0xFFE02104

.text;
__start:

SP.L = STACK_MEM;
SP.H = STACK_MEM;
usp = sp;

// Initialise the Event Vector table.
P0.H = 0xFFE0;
P0.L = 0x2000;

// Install __unknown_exception_occurred in EVT so that
// there is defined behaviour.
P0 += 2*4; // Skip Emulation and Reset
P1 = 13;
R1.L = __unknown_exception_occurred;
R1.H = __unknown_exception_occurred;
LSETUP (.ivt,.ivt) LC0 = P1;
.ivt: [P0++] = R1;

// Set IVG15's handler to be the start of the mode-change
// code. Then, before we return from the Reset back to user
// mode, we'll raise IVG15. This will mean we stay in supervisor
// mode, and continue from the mode-change point, but at a
// much lower priority.
P1.H = supervisor_mode;
P1.L = supervisor_mode;
[P0] = P1;

// We're still in supervisor mode at the moment, so the FP
// needs to point to the supervisor stack.
FP = SP;

// Make space for incoming "parameters" for functions
// we call from here:
SP += -12;

// When the processor is turned on, it is in reset mode which is one of
// the highest priority interrupts. While in reset mode, the processor will
// not service any other interrupt types. Instead of reset mode, we'd like
// the processor to be in the lowest priority mode possible. To get to that
// mode, we first raise the IVG15 interrupt flag. However, the processors
// current priority will not allow IVG15 to be triggered.

// Enable IVG
R4 = INTERRUPT_BITS;
R4 <<= 5;
STI R4;
RAISE 15;

// In order to leave reset mode, we perform an RTI which jumps to the
// usermode label. Usermode is a simple infinite loop, however once the RTI
// is processed, the processor will acknowledge IVG15. Since we earlier
// set the handler of IVG15 to be supervisor_mode, the processor will
// immediately handle IVG15 and end up at the supervisor_mode label. From
// this point on, the processor will be in the lowest interrupt priority
// level which is pretty neat.

P0.L = usermode;
P0.H = usermode;
RETI = P0;

usermode:
    NOP;
    RTI;

supervisor_mode:

    // enable all other interrupts, pushing RETI globally enables interrupts
    // for some reason
    [--SP] = RETI;
    FP = SP;
    SP += 12;

my_init:

    // Set up PLL registers to generate core and sys clocks
    P0.L = PLL_CTL;
P0.H = PLL_CTL;
P1.L = PLL_DIV;
P1.H = PLL_DIV;
R0.L = PLL_CTL_VAL;
R1.L = PLL_DIV_VAL;
W[P0] = R0.L;
W[P1] = R1.L;

    // Set up Asynch Memory Banks
    P0.L = EBIU_AMGCTL;
P0.H = EBIU_AMGCTL;
P1.L = EBIU_AMBCTL0;
P1.H = EBIU_AMBCTL0;
P2.L = EBIU_AMBCTL1;
P2.H = EBIU_AMBCTL1;
R0.L = EBIU_AMGCTL_VAL;
R1.L = EBIU_AMBCTL0_VAL;
R1.H = EBIU_AMBCTL0_VAL;
R2.L = EBIU_AMBCTL1_VAL;
R2.H = EBIU_AMBCTL1_VAL;
W[P0] = R0;           // write to global control register
[P1] = R1;            // write to AMS0, AMS1 control register
[P2] = R2;            // write to AMS2, AMS3 control register

    // Set up Programmable Flags (PF pins)
P3.L = FIO_DIR;
P3.H = FIO_DIR;
R3.L = FIO_DIR_VAL;
W[P3] = R3.L;  // configure PFs as input/output by writing to
              // FIO_DIR

R0.L = W[P3];
P4.L = FIO_MASKA_D;
P4.H = FIO_MASKA_D;
R4.L = FIO_MASKA_D_VAL;
W[P4] = R4.L;  // enable PF interrupts on the appropriate PF pin(s)
P5.L = FIO_POLAR;
P5.H = FIO_POLAR;
R5.L = FIO_POLAR_VAL;  // PF interrupts trigger on rising edge
W[P5] = R5.L;
P0.L = FIO_EDGE;
P0.H = FIO_EDGE;
R0.L = FIO_EDGE_VAL;
W[P0] = R0.L;  // PF interrupt pins are edge-sensitive sources
P1.L = FIO_INEN;
P1.H = FIO_INEN;
R1.L = FIO_INEN_VAL;  // enable PF input so that rotary encoder can
                      // trigger interrupts
W[P1] = R1.L;

// initialize keypad, display, IDE, and the SPI serial port
[--SP] = RETS;
CALL _KeyInit;
RETS = [SP++];

[--SP] = RETS;
CALL _Display_Init;
RETS = [SP++];

[--SP] = RETS;
CALL _IDE_Init;
RETS = [SP++];

[--SP] = RETS;
CALL _serial_init;
RETS = [SP++];

// initialize timer 0
[--SP] = RETS;
CALL _InitTimer0;
RETS = [SP++];

// install event handlers in vector table
[--SP] = RETS;
CALL _InstallTimer0EventHandler;
RETS = [SP++];
[--SP] = RETS;
CALL _InstallPFEventHandler;
RETS = [SP++];
[--SP] = RETS;
CALL _InstallSerialEventHandler;
RETS = [SP++];

// enable interrupts
[--SP] = RETS;
CALL _InterruptInit;
RETS = [SP++];

CALL.X _main; /* Jump to Glen Code */
RTS;

/* Delimit the '_main' symbol so the linker knows which code is associated with the symbol. */
.__start.END:

// Default exception handling does pretty much nothing besides sit in a loop
__unknown_exception_occurred:
  NOP;
  JUMP __unknown_exception_occurred;
.__unknown_exception_occurred.end:

HE_handler:
  RTI;
.HE_handler.end:

.data

.end
// bfinstart.inc
// Contains values to write to registers in order to set
// up clocks and chip selects
//
// Revision History:
// 5/27/2014 Ajay Mandlekar Updated PF register values.
//
// PLL Register Values

// value to write to PLL control reg to set VCO/CLK_IN to 9 (VCO = 180MHz)
//
// 0001 0010 0000 0000
// 0--- ---- ---- ---- no added hysteresis to SPORT input pins
// -001 001- ---- ---- set VCO/CLK_IN multiplier to 9
// ---- ---- 0--- ---- do not bypass PLL
// ---- ---- 0--- ---- do not add output delay
// ---- ---- -0-- ---- do not add input delay
// ---- ---- --0- ---- all internal clocks on
// ---- ---- ---0 ---- reserved
// ---- ---- ---- 0--- CCLK on
// ---- ---- ---- -0-- reserved
// ---- ---- ---- --0- enable power to PLL
// ---- ---- ---- ---1 pass CLK_IN to PLL
//
.equ PLL_CTL_VAL, 0x1200

// value to write to PLL divide reg to set division ratio of 10 (VC0 / SCK) and
// 4 (VC0 / CCK) so that core clock is 45 MHz and system clock is 18 MHz
//
// 0000 0000 0010 1010
// 0000 0000 00-- ---- reserved
// ---- ---10 ---- CCLK = VCO/4 (45 MHz)
// ---- ---- 1010 SCLK = VCO/10 (18 MHz)
//
.equ PLL_DIV_VAL, 0x002A

// Asynchronous Memory Controller Reg Values

// value to write to EBIU_AMGCTL reg in order to set
// up Asynch Mem Controller to enable all banks and CLKOUT
//
// 0000 0000 1111 1111
// 0000 0000 00-- ---- reserved
// ---- ---- 0---- ---- core has priority over DMA for external accesses
// ---- ---- 1111 ---- reserved
// ---- ---- 111- all banks (Bank0, Bank1, Bank2, Bank3) enabled
// ---- ---- ----1 enable CLKOUT for asynchronous memory region accesses
.equ EBIU_AMGCTL_VAL, 0x00FF

// value to write to EBIU_AMBCTL0 reg in order to set up
// memory banks 0 (ROM) and 1 (SRAM).
// memory bank 0 - 2 cycle setup, 1 cycle hold, 3 cycle read,
// (ROM) 1 cycle bank transition
// memory bank 1 - 1 cycle setup, 1 cycle hold, 1 cycle read,
// (RAM) 1 cycle write, 1 cycle bank transition

// 0001 0001 0101 0100 0000 0011 0110 0100
// 0001 ---- ---- ---- ---- ---- ---- ---- 1 write access cycle (bank 1)
// ---- 0001 ---- ---- ---- ---- ---- ---- 1 read access cycle (bank 1)
// ---- ---- 01-- ---- ---- ---- ---- ---- 1 hold cycle (bank 1)
// ---- ---- --01 ---- ---- ---- ---- ---- 1 setup cycle (bank 1)
// ---- ---- ---- 01-- ---- ---- ---- ---- ---- transaction completes if ARDY low
// ---- ---- ---- ---- 0000 ---- ---- ---- ---- ignore ARDY for bank 1 accesses
// ---- ---- ---- ---- 0011 ---- ---- ---- ---- 3 read access cycles (bank 0)
// ---- ---- ---- ---- ---- 01-- ---- ---- ---- 1 hold cycle (bank 0)
// ---- ---- ---- ---- ---- --10 ---- ---- ---- 2 setup cycles (bank 0)
// ---- ---- ---- ---- ---- ---- ---- ---- 1 cycle bank transition (bank 0)
// ---- ---- ---- ---- ---- ---- ---- ---- --0- transaction completes if ARDY low
// ---- ---- ---- ---- ---- ---- ---- ---- ----0 ignore ARDY for bank 0 accesses
// .equ EBIU_AMBCTL0_VAL, 0x11540364

// value to write to EBIU_AMBCTL1 reg in order to set up
// memory banks 2 and 3.
// memory bank 2 - 4 cycle setup, 3 cycle hold, 15 cycle read,
// (IDE) 15 cycle write, 4 cycle bank transition
// memory bank 3 - 2 cycle setup, 1 cycle hold, 8 cycle read,
// (DRAM) 6 cycle write, 1 cycle bank transition

// 0110 1000 0100 1111 1111 1100 0000
// 0110 ---- ---- ---- ---- ---- ---- ---- 6 write access cycles (bank 3)
// ---- 1000 ---- ---- ---- ---- ---- ---- 8 read access cycles (bank 3)
// ---- ---- 01-- ---- ---- ---- ---- ---- 1 hold cycle (bank 3)
// ---- ---- --10 ---- ---- ---- ---- ---- 2 setup cycles (bank 3)
// ---- ---- ---- 01-- ---- ---- ---- ---- ---- 1 cycle bank transition (bank 3)
// ---- ---- ---- --0- ---- ---- ---- ---- ---- transaction completes if ARDY low
// ---- ---- ---- ---- ---- ---- ---- ---- ----0 ignore ARDY for bank 3 accesses
// ---- ---- ---- ---- ---- 1111 ---- ---- ---- ---- 15 write access cycles (bank 2)
// ---- ---- ---- ---- ---- ---- ---- ---- ----15 read access cycles (bank 2)
// ---- ---- ---- ---- ---- ---- ---- ---- ----11-- 3 hold cycles (bank 2)
// ---- ---- ---- ---- ---- ---- ---- ---- ----00 ---- 4 setup cycles (bank 2)
// ---- ---- ---- ---- ---- ---- ---- ---- ----00-- 4 cycle bank transition (bank 2)
// ---- ---- ---- ---- ---- ---- ---- ---- ----0- transaction completes if ARDY low
// ---- ---- ---- ---- ---- ---- ---- ---- ----0 ignore ARDY for bank 2 accesses
// .equ EBIU_AMBCTL1_VAL, 0x6864FFC0
// PF Pin Reg Values

// Value to write to FIO_DIR reg in order to configure PF pin input/output
// directions. PF0-10 are configured to be outputs while the rest are inputs.
// 0000 0111 1111 1111
// 0000 0--- ---- ---- PF11-15 are inputs (from rotary encoder and MP3 decoder)
// ---- -111 1111 1111 PF0-10 are outputs (to the display)
// .equ FIO_DIR_VAL, 0x07FF

// Value to write to FIO_MASKA_D reg in order to enable PF Interrupt A for
// PF12, used to generate interrupts when the user rotates the rotary
// encoder. Note that PF14, the demand pin, is not enabled as an interrupt
// here. This will be done in the audio functions.
// .equ FIO_MASKA_D_VAL, 0x1000

// Value to write to FIO_POLAR reg in order to configure the PF pin inputs
// to rising-edge, or active high. This is necessary to generate the proper
// interrupts on rising edge for PF12, the turn signal, and PF14, the demand
// pin for the MP3 Decoder.
// 0000 0000 0000 0000 all PF pin inputs are rising-edge or active high
// .equ FIO_POLAR_VAL, 0x0000

// Value to write to FIO_EDGE reg in order to configure PF12 as an edge
// sensitive source and PF14 as an level sensitive source
// 0001 0000 0000 0000
// -0-- ---- ---- ---- PF14 is a level sensitive source
// ----1 ---- ---- ---- PF12 is an edge sensitive source
// .equ FIO_EDGE_VAL, 0x1000

// Value to write to FIO_INEN reg in order to enable PF inputs
// 0111 1000 0000 0000
// -1-- ---- ---- ---- enable PF14 (demand pin) as input
// --1- ---- ---- ---- enable PF13 (rotary switch signal) as input
// ----1 ---- ---- ---- enable PF12 (turn signal) as input
// ---- 1--- ---- ---- ---- enable PF11 (right signal) as input
// .equ FIO_INEN_VAL, 0x7800
Contains the register addresses for the Blackfin processor.

Revision History:
4/17/2014 Ajay Mandlekar Initial Revision.
6/1/2014 Ajay Mandlekar Added addresses. Note that file was periodically updated each week with new addresses until this date.

Clock/Regulator Control (0xFFC00000 - 0xFFC000FF)

.equ PLL_CTL, 0xFFC00000 // PLL Control register (16-bit)
.equ PLL_DIV, 0xFFC00004 // PLL Divide Register (16-bit)
.equ VR_CTL, 0xFFC00008 // Voltage Regulator Control Register (16-bit)
.equ PLL_STAT, 0xFFC0000C // PLL Status register (16-bit)
.equ PLL_LOCKCNT, 0xFFC00010 // PLL Lock Count register (16-bit)
.equ CHIPID, 0xFFC00014 // Chip ID Register

Asynchronous Memory Controller

.equ EBIU_AMGCTL, 0xFFC000A00 // Asynchronous Memory Global Control Register
.equ EBIU_AMBCTL0, 0xFFC000A04 // Asynchronous Memory Bank Control Register 0
.equ EBIU_AMBCTL1, 0xFFC000A08 // Asynchronous Memory Bank Control Register 1

SDRAM Controller

.equ EBIU_SDGCTL, 0xFFC000A10 // SDRAM Global Control Register
.equ EBIU_SDBCTL, 0xFFC000A14 // SDRAM Bank Control Register
.equ EBIU_SDRRC, 0xFFC000A18 // SDRAM Refresh Rate Control Register
.equ EBIU_SDSTAT, 0xFFC000A1C // SDRAM Status Register

General Purpose IO (0xFFC00700 - 0xFFC007FF) (PF Pins)

.equ FIO_FLAG_D, 0xFFC000700 // Flag Data Register
.equ FIO_FLAG_C, 0xFFC000704 // Peripheral Interrupt Flag Register (clear)
.equ FIO_FLAG_S, 0xFFC000708 // Peripheral Interrupt Flag Register (set)
.equ FIO_FLAG_T, 0xFFC00070C // Flag Mask to directly toggle state of pins
.equ FIO_MASKA_D, 0xFFC000710 // Flag Mask Interrupt A Register (set directly)
.equ FIO_MASKA_C, 0xFFC000714 // Flag Mask Interrupt A Register (clear)
.equ FIO_MASKA_S, 0xFFC000718 // Flag Mask Interrupt A Register (set)
.equ FIO_MASKA_T, 0xFFC00071C // Flag Mask Interrupt A Register (toggle)
.equ FIO_MASKB_D, 0xFFC000720 // Flag Mask Interrupt B Register (set directly)
.equ FIO_MASKB_C, 0xFFC000724 // Flag Mask Interrupt B Register (clear)
.equ FIO_MASKB_S, 0xFFC000728 // Flag Mask Interrupt B Register (set)
.equ FIO_MASKB_T, 0xFFC00072C // Flag Mask Interrupt B Register (toggle)
.equ FIO_DIR, 0xFFC000730 // Peripheral Flag Direction Register
.equ FIO_POLAR, 0xFFC000734 // Flag Source Polarity Register
.equ FIO_EDGE, 0xFFC000738 // Flag Source Sensitivity Register
.equ FIO_BOTH, 0xFFC00073C // Flag Set on BOTH Edges Register
.equ FIO_INEN, 0xFFC000740 // Flag Input Enable Register
// System Interrupt Controller (0xFFFFC00100 - 0xFFFFC001FF)
equ SIC_IMASK, 0xFFFFC0010C // System Interrupt Mask Register
.equ IMASK, 0xFFFFE02104 // Core Interrupt Mask Register
.equ SIC_IAR2, 0xFFFFC00118 // Interrupt Assignment Register 2

// Interrupt Vector Memory Locations
.equ IVG8, 0xFFFFE02020 // Interrupt 8 address
.equ IVG10, 0xFFFFE02028 // Interrupt 10 address
.equ IVG11, 0xFFFFE0202C // Interrupt 11 address
.equ IVG12, 0xFFFFE02030 // Interrupt 12 address
.equ IVG14, 0xFFFFE02038 // Interrupt 14 address
.equ IVG15, 0xFFFFE0203C // Interrupt 15 address

// TIMER 0, 1, 2 Registers (0xFFFFC00600 - 0xFFFFC006FF)
equ TIMER0_CONFIG, 0xFFFFC00600 // Timer 0 Configuration Register
.equ TIMER0_COUNTER, 0xFFFFC00604 // Timer 0 Counter Register
.equ TIMER0_PERIOD, 0xFFFFC00608 // Timer 0 Period Register
.equ TIMER0_WIDTH, 0xFFFFC0060C // Timer 0 Width Register

.equ TIMER1_CONFIG, 0xFFFFC00610 // Timer 1 Configuration Register
.equ TIMER1_COUNTER, 0xFFFFC00614 // Timer 1 Counter Register
.equ TIMER1_PERIOD, 0xFFFFC00618 // Timer 1 Period Register
.equ TIMER1_WIDTH, 0xFFFFC0061C // Timer 1 Width Register

.equ TIMER_ENABLE, 0xFFFFC00640 // Timer Enable Register

.equ TIMER_STATUS, 0xFFFFC00648 // Timer Status Register

// SPI registers
.equ SPI_CTL, 0xFFFFC00500 // SPI Control Register
.equ SPI_STAT, 0xFFFFC00508 // SPI Status Register
.equ SPI_TDBR, 0xFFFFC0050C // SPI Transmit Data Buffer Register
.equ SPI_BAUD, 0xFFFFC00514 // SPI Baud rate Register
**Rotary Encoder Software**

The software for the Rotary Encoder is responsible for responding to a user rotation and a user press.

The Rotate_Handler function is responsible for handling an interrupt corresponding to a rotation. The interrupt is triggered on the TURN signal (PF12) on rising-edge (for more information on how this is done, see the Hardware Reference). It simply checks the RIGHT signal on PF11 to find out whether the user rotated right or left, and either increments or decrements the current key code in the current_key shared variable. It also wraps the current_key around if necessary.

The Press_Handler function is a timer interrupt handler. Timer0 is set to count to a certain period. At the end of the count, it generates an interrupt, and the Press_Handler function is called. The Press_Handler is responsible for debouncing and registering a user press. It does so by keeping a debounce counter, stored in the bounce_count shared variable. On every call, it checks PF13 to determine whether the user is pressing the rotary switch. If so, it decrements the count. Once the count hits zero, a press is registered by setting the key_flag shared variable. At this time, the debounce counter is reset to implement auto-repeat on the key press.

The Rotary Encoder software also includes a getkey and key_available function. These are accessor functions used by the main loop to determine whether the user pressed the rotary encoder, and if so, which key code was selected.

Note that initialization of the shared variables happens in the KeyInit function. Also note that the timer that the Press_Handler function runs on is initialized in the InitTimer0 function. Both of these functions are called in the boot code.

Another function included in this file is the elapsed_time function. This function returns the time in milliseconds since the last call to the elapsed_time function. It does so by using the elapsed shared variable, which is incremented in every call to the Press_Handler (since it runs on a timer interrupt).

The following block diagram shows how all of the Rotary Encoder functions interact with the shared variables.
The Rotary Encoder software has been reproduced below. The relevant files are the keypad.s, keypad.inc, timerinit.s, and timers.inc files.
// This file contains the rotary encoder routines for the Blackfin MP3 project.
//
// Table of Contents:
//  Rotate_Handler - handles the interrupt corresponding to a rotation.
//  Press_Handler - handles the interrupt corresponding to a button press.
//  KeyInit - Initializes shared variables used by all of the routines.
//  getkey - Returns the keycode corresponding to the most recent valid key. Note that the keycode values are defined in interfac.h.
//  key_available - Returns whether a key press has been registered.
//  elapsed_time - Returns the time elapsed since the last call to the elapsed_time function.
//
// allow other files to see these functions
.global _Rotate_Handler
.global _Press_Handler
.global _KeyInit
.global _getkey
.global _key_available
.global _elapsed_time

.include "src/sys/keypad.inc"
.include "src/sys/bfinregaddr.inc"
.include "src/sys/display.inc"

.extern _Display_Num

.section .text
// Rotate_Handler
//
// Description: This function is an interrupt handler that handles
// interrupts corresponding to a rotation from the rotary encoder.
//
// Operation: The function simply updates the current_key shared variable in order to update which key the user currently might be selecting. The key will not be selected until the Press_Handler is called after the user presses the button. The keycode is either incremented or decremented, and rolls back around to the last or first keycode if necessary.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: R0 - temporary location to store direction bit and modify current_key shared variable.
// P0 - used to address registers
// P1 - used to store address of current_key
//
// Shared Variables: current_key - key code corresponding to the current key that the user can select by pressing the button on the rotary encoder.
//
// Global Variables: None.
//
// Input: Rotary Encoder
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 3, 2014
//
_rotate_Handler:
    [--SP] = RETI;
    [--SP] = (R7:0, P5:0);      // push all registers
R7.L = LOOP_CONST;  // wait loop to improve functionality
R7.H = LOOP_CONST;
LC0 = R7;
LOOP myloop1 LC0;
LOOP_BEGIN myloop1;
NOP;
LOOP_END myloop1;

P0.L = FIO_FLAG_C;
P0.H = FIO_FLAG_C;
R1 = TURN_BIT;
W[P0] = R1.L;  // clear the interrupt
SSYNC;  // ensure that peripheral pin releases
// interrupt request before the RTI

P0.L = FIO_FLAG_D;
P0.H = FIO_FLAG_D;
P1.L = current_key;  // use P1 to store current_key address
P1.H = current_key;
R0 = W[P0] (z);  // read in PF pin values
CC = BITTST (R0, 11);  // CC has direction bit
if CC JUMP MovedRight;  // if direction bit set, user rotated right
//JUMP MovedLeft;  // else user rotated left

MovedLeft:  // user moved left, decrement the keycode
R0 = W[P1] (z);  // read in current_key
CC = R0 == 0;  // check if need to wrap around
if !CC JUMP MovedLeftContinue;  // no need to wrap around
R0.L = NUM_KEYS;  // wrap back around to number of key_codes,
// before the decrement

MovedLeftContinue:  // do the decrement
R0 += -1;
JUMP Rotate_Handler_Done;

MovedRight:  // user moved right, increment the keycode
R0 = W[P1] (z);  // read in current_key
R0 += 1;
R7 = NUM_KEYS;
CC = R0 == R7;  // check if need to wrap around
if !CC JUMP Rotate_Handler_Done;  // no need to wrap around
R0.L = 0;  // wrap back to first keycode
//JUMP Rotate_Handler_Done;

Rotate_Handler_Done:
W[P1] = R0.L;  // write correct modified keycode
R1 = R0;
R0 = DDRAM_KEY_OFFSET;

[--SP] = RETS;
CALL _Display_Num;  // display current keycode
RETS = [SP++];
(R7:0, P5:0) = [SP++] // pop all registers
RETI = [SP++];
RTI;

// Press_Handler

// Description: This function is an interrupt handler that handles
interrupts corresponding to a press from the
rotary encoder. It debounces the key press, and then
records the key press. Note that the function also increments
a counter that is used by the elapsed_time function.

// Operation: The function uses the bounce_count shared variable in
order to debounce a key press. If the key is currently
still pressed and the debounce counter has not hit zero
yet, the function simply decrements the debounce counter.
When the debounce counter hits zero, the function
registers a key press by updating the key_flag shared
variable in order to indicate that a key has been selected.
The key code that corresponds to the key will already be
stored in the current_key shared variable by the
Rotate_Handler. If the key is not currently pressed, the
function will reset the debounce counter. At the end of the
function, the elapsed shared variable is incremented.

// Arguments: None.

// Return Value: None.

// Local Variables: R0 - used to store the bit which signifies a key press
R1 - stores debounce count
P1 - stores address of bounce_count

// Shared Variables: key_flag - whether a key has been pressed or not.
bounce_count - the debounce counter.
elapsed - the amount of time elapsed in milliseconds since
the last call to the elapsed_time function

// Global Variables: None.

// Input: Rotary Encoder.
// Output: None.

// Error Handling: None.

// Limitations: None.

// Algorithms: None.

// Data Structures: None.

// Registers Changed: None.
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 3, 2014
//

_Press_Handler:
    [--SP] = RETI;
    [--SP] = (R7:0, P5:0); // push all registers
    P0.L = FIO_FLAG_D;
    P0.H = FIO_FLAG_D;
    P1.L = bounce_count; // P1 used to store address of bounce_count
    P1.H = bounce_count;
    R0.L = W[P0]; // read in PF pin values
    CC = BITTST (R0, 13); // CC is set if button is not pressed
    if !CC JUMP DecrementCount; // button is pressed, continue debouncing
    //JUMP ResetDebounceCount; // else button is not pressed, reset count

ResetDebounceCount:
    R0 = MAX_DEBOUNCE_COUNT;
    W[P1] = R0.L; // write to bounce_count
    JUMP Press_Handler.Done; // nothing else to be done

DecrementCount: // debounce the key press
    R1 = W[P1] (z);
    R1 += -1; // decrement the counter
    W[P1] = R1.L;
    CC = R1 == 0; // check if key is debounced
    if !CC JUMP Press_Handler.Done; // done if key is not debounced yet
    //JUMP GotAPress; // else key has been debounced, log a key
    // press

GotAPress: // debounced the key, now set key_flag
    P0.L = key_flag;
    P0.H = key_flag;
    R0 = 1;
    W[P0] = R0.L; // set key_flag
    R0 = REPEAT_RATE;
    W[P1] = R0.L; // and reset the debounce count to the
    // auto-repeat rate in order to implement
    // auto-repeat
    //JUMP Press_Handler.Done;

Press_Handler.Done:
    P0.L = elapsed; // increment elapsed time
    P0.H = elapsed;
    R0 = [P0];
    R0 += 1;
    [P0] = R0;
    P0.L = TIMER_STATUS; // indicate that the interrupt has been
    P0.H = TIMER_STATUS; // serviced
R0 = CLEAR_TIMER0_INT;
W[P0] = R0;
(R7:0, P5:0) = [SP++]; // pop all registers
RETI = [SP++];
RTI;

// KeyInit
//
// Description: This function initializes all of the shared variables used by the Rotary Encoder routines.
//
// Operation: The function simply writes the correct values to the current_key and key_flag shared variables to start at
// the first key and indicate that there is no key press yet. It also sets the bounce_count shared variable to be the
// DEBOUNCE_COUNT value.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: P0 - write to shared variables
// R0 - values to write to shared variables
//
// Shared Variables: current_key - key code corresponding to the current key that the user can select by pressing the button on the rotary encoder.
// key_flag - whether a key has been pressed or not.
// bounce_count - the debounce counter.
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 4 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 3, 2014
//
_KeyInit:

[--SP] = P0;  // push P0, R0
[--SP] = R0;
R0 = 0;
P0.L = current_key;
P0.H = current_key;
W[P0] = R0.L;  // start at first key
P0.L = key_flag;
P0.H = key_flag;
W[P0] = R0.L;  // no key press yet
P0.L = bounce_count;
P0.H = bounce_count;
R0 = MAX_DEBOUNCE_COUNT
W[P0] = R0.L;  // initialize debounce count
R0 = [SP++];
P0 = [SP++];  // pop P0, R0
RTS;

// getkey

// Description: This function returns the last key that the user
// selected. Note that this function assumes that the
// key was pressed and debounced successfully.

// Operation: The function simply returns the current_key shared
// variable.

// Arguments: None.

// Return Value: Current_key shared variable in R0.

// Local Variables: P0 - shared var addresses

// Shared Variables: current_key - key code corresponding to the current
// key that the user can select by
// pressing the button on the rotary
// encoder.
// key_flag - whether a key has been pressed or not.

// Global Variables: None.

// Input: None.
// Output: None.

// Error Handling: None.

// Limitations: None.

// Algorithms: None.

// Data Structures: None.

// Registers Changed: R0.
// Stack Depth: 0 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 3, 2014
//

_getkey:
    [--SP] = P0;       // push P0
    P0.L = key_flag;
    P0.H = key_flag;
    R0 = 0;
    W[P0] = R0.L;     // reset key_flag as the key has been retrieved
    P1.L = current_key;
    P1.H = current_key;
    R0 = W[P1] (z);   // and retrieve the keycode
    P0 = [SP++];      // pop P0
    RTS;

// key_available
//
// Description: This function returns whether a key press has been
// registered or not.
//
// Operation: The function simply returns the key_flag shared variable
// in order to indicate whether a key press has been
// registered or not.
//
// Arguments: None.
//
// Return Value: Key_flag shared variable in R0.
//
// Local Variables: None.
//
// Shared Variables: key_flag - whether a key has been pressed or not.
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: R0.
//
// Stack Depth: 0 words.
_key_available:
    [--SP] = P0;          // push P0
    P0.L = key_flag;
    P0.H = key_flag
    R0 = W[P0] (z);      // retrieve key_flag
    P0 = [SP++];         // pop P0
    RTS;

elapsed_time

// Description: This function returns the elapsed time since the last call to
// this function in milliseconds.
//
// Operation: The function simply returns the elapsed shared variable and
// resets it to 0. Note that the elapsed variable is update after
// every timer interrupt in the Press_Handler.
//
// Arguments: None.
//
// Return Value: R0 - elapsed time since last call in milliseconds
//
// Local Variables: None.
//
// Shared Variables: elapsed - time in milliseconds since last call to this
// function
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 0 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: June 1, 2014
//
_elapsed_time:
    [--SP] = (R7:1, P5:0); // push all registers, except R0
    P0.L = elapsed;
    P0.H = elapsed;
    R0 = [P0]; // get count
    R1 = 0;
    [P0] = R1; // reset count
    (R7:1, P5:0) = [SP++]; // pop all registers, except R0
    RTS;

// Declare shared variables here
.section .data

    current_key: // key code corresponding to the current key that the user
                  // can select by pressing the button on the rotary encoder.
        .int 0

    key_flag: // whether a key has been pressed or not
        .int 0

    bounce_count: // the debounce count
        .int 20

    elapsed: // elapsed time since last elapsed time call
        .int 0
// keypad.inc
//
// This file contains the constants used by the rotary encoder
// subroutines.
//
// Revision History:
// 5/03/2014    Ajay Mandlekar    Initial Revision.
// 5/27/2014    Ajay Mandlekar    Added a loop constant to account for
//                                possible switch bouncing issues.
//
.equ NUM_KEYS,    7 // the number of keys that the rotary encoder
                   // rotates through
.equ DIRECTION_BIT, 11 // the PF pin corresponding to the direction
                       // that the encoder turned
.equ PRESS_BIT, 13 // the PF pin that signifies whether the
                   // encoder switch is pressed or not
.equ MAX_DEBOUNCE_COUNT, 200 // number of timer interrupts it takes
                              // to debounce a key
.equ REPEAT_RATE, 500 // auto-repeat rate for key press, in timer interrupts
.equ TURN_BIT, 0x1000 // value to isolate turn bit in PF register (PF 12)
.equ CLEAR_TIMER0_INT, 0x0001 // value to write to TIMER_STATUS reg to indicate
                             // that the interrupt has been serviced
.equ LOOP_CONST, 0x1000000 // delay time to wait for at the start of
                           // handling every rotary interrupt. This is
                           // necessary to ensure that multiple interrupts
                           // do not occur (the rotary encoder may be
                           // bouncing after every rotation)
This file contains the subroutines necessary to set up the timers.

Table of Contents:
- InitTimer0 - Initializes Timer 0.

.global _InitTimer0

.include "src/sys/timers.inc"
.include "src/sys/bfinregaddr.inc"

.section .text;

// InitTimer0
// Description: This function initializes Timer 0.
// Operation: This function writes the appropriate values to the timer registers in order to initialize timer 0.
// Arguments: None.
// Return Value: None.
// Local Variables: P0, P1 - used to store address of registers
// R0, R1 - values to write to registers
// Shared Variables: None.
// Global Variables: None.
// Input: None.
// Output: None.
// Error Handling: None.
// Limitations: None.
// Algorithms: None.
// Data Structures: None.
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 3, 2014
//

_InitTimer0:
  [--SP] = (R7:0, P5:0); // push regs
  P0.L = TIMER0_CONFIG;
  P0.H = TIMER0_CONFIG;
  R0.L = TIMER0_CONFIG_VAL;
  W[P0] = R0.L;           // write to TIMER0_CONFIG reg to configure Timer 0
  P1.L = TIMER0_PERIOD;
  P1.H = TIMER0_PERIOD;
  R1.H = TIMER0_PERIOD_VAL_HIGH;
  R1.L = TIMER0_PERIOD_VAL_LOW;
  [P1] = R1;              // configure the period of Timer 0
  P2.L = TIMER_ENABLE;
  P2.H = TIMER_ENABLE;
  R2.L = TIMER_ENABLE_VAL;
  W[P2] = R2.L;           // write to TIMER_ENABLE reg to enable Timer 0
  (R7:0, P5:0) = [SP++];   // restore regs
  RTS;
// Contains definitions related to the timers for the Blackfin MP3 Project.

// Revision History:
// 5/03/2014    Ajay Mandlekar    Initial Revision.

// value to write to Timer Enable Register to enable Timer 0
.equ TIMER_ENABLE_VAL, 0x0001

// value to write to Timer 0 Config Register to setup Timer 0
// 0000 0000 0001 1001
// 00-- ---- ---- ---- no error type
// --00 00-- ---- ---- reserved
// ---- --0- ---- ---- timer counter stops during emulation
// ---- ---0 ---- ---- effective state of PULSE_HI is programmed state
// ---- ---- 0--- ---- use system clock for counter
// ---- ---- -0-- ---- enable pad in PWM_OUT mode
// ---- ---- --01 ---- sample TMR0 pin
// ---- ---- ---- 1---- interrupt request enable
// ---- ---- ---- 1--- count down to end of period
// ---- ---- ---- -0-- negative action pulse
// ---- ---- ---- --01 PWM_OUT mode
// .equ TIMERO_CONFIG_VAL, 0x0019

// Value to write to Timer 0 Period Register to setup Timer 0 to count to
// 20,000 system clocks, which is 1 millisecond. Thus the interrupt rate is
// once every millisecond.
.equ TIMERO_PERIOD_VAL_HIGH, 0x0000
.equ TIMERO_PERIOD_VAL_LOW, 0x4E20
Display Software

The LCD Display shows the user interface. It is first initialized by a call to the Display_Init function in the boot code. This function uses wait loops and the Display_Char function to send the appropriate commands to the display to initialize it.

The Display Software has several helper functions. The Display_Char function handles the LCD write timing through wait loops in order to write commands and characters to the display. The Display_String function loops through a string character by character, calling Display_Char on each one, to display a string. The Display_Num function writes a number to the display. It does so by first calling the Dec2String function, which converts a signed number to a string representing its decimal value. The string is stored in the num_str shared variable. Then, the Display_String function is called to display the string.

The Dec2String function uses the Divide function to do divide operations. Note that it was necessary to write a divide function as the Blackfin processor does not have a built-in divide operation.

The main display functions that are accessed by the main loop are the display_artist, display_title, display_time, and display_status functions. The display_artist and display_title just call Display_String to display the artist and track title at the appropriate locations on the display. The display_time function just uses the Display_Num function to display the time in milliseconds. The display_status function uses the passed status value to index into a table of status strings. This table is located in the status_table shared variable. It then displays the appropriate status string with a call to Display_String.

The following block diagram shows how the display functions interact with each other and the shared variables.
The display code is reproduced below. The relevant files are `display.s`, `display.inc`, and `divide.s`. 
// This file contains the LCD Display routines for the Blackfin
// MP3 project.

// Table of Contents:
// Dec2String - converts signed number to a string representing its
decimal value
// Display_Init - initializes shared variables used by display routines
// Display_Num - writes a number to the display
// Display_String - writes a string to the display
// Display_Char - writes a character to the display by writing to the
corresponding PF pins. This function also handles
the necessary timing requirements to write to the
display.
// display_artist - writes an artist string to the display
// display_title - writes a title string to the display
// display_time - writes a time value to the display
// display_status - writes a status string to the display

// allow other files to see these functions
.global _Dec2String
.global _Display_Init
.global _Display_Num
.global _Display_String
.global _Display_Char
.global _display_artist
.global _display_title
.global _display_time
.global _display_status

.include "src/sys/bfinregaddr.inc"
.include "src/sys/display.inc"

.extern _divide

.section .text;
// Dec2String
//
// Description: This function takes two arguments - a 16-bit unsigned value to
// convert to decimal and store as a string, and an address. The
// function stores the null terminated decimal representation of the
// value in ASCII as a string. The string is stored at the address
// specified by the second argument. The number is passed in AX by
// value. The address is passed in SI by value (thus the string will
// be written to memory starting at DS:SI). The string size is
// always 7 bytes (including the null termination character). It
// contains the ASCII representation of the decimal value of the
// argument, followed by a null character.
//
// Operation: The function finds the largest power of 10 possible and
// loops dividing the number by the power of 10, the quotient is a
digit and the remainder is used in the next iteration of the loop.
// When the digit is obtained, it is converted to its ASCII
// representation value and stored in memory at the correct location.
// Each loop iteration divides the power of 10 by 10 until it is 0.
// When it is 0, the number has been converted to its null terminated
// representation in ASCII and stored at the correct location. Then,
a null character is added to the end of the string.
//
// Arguments: R1.L - the 16-bit unsigned value to convert
//           P0 - the address at which to store the string
//
// Return Value: None.
//
// Local Variables: R1 - current argument to convert (this is updated in the
//                   loop through divisions)
//               R2 - current maximum power of 10 used to compute next
digit
//
// Shared Variables: none
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 10, 2014

_DEC2STRING:

[--SP] = (R7:0, P5:0); // push all registers
R1.H = 0; // clear high word to ensure that R1 contains only
          // the argument and nothing more
R2 = MAX_PWR_10; // start with max pwr of 10 to find first digit
// JUMP Dec2StringLoop; // now start looping

Dec2StringLoop:
R4 = 0;
CC = R2 <= R4; // check if the power of 10 is still positive
if CC JUMP Dec2StringDone; // if not, loop is done
// JUMP Dec2StringLoopBody;

Dec2StringLoopBody:

[--SP] = RETS;
CALL _Divide; // digit (R0) = arg / pwr10
RETS = [SP++];
R1 = R7; // update argument by setting it equal to remainder
R4 = DIGIT_TO_ASCII;
R6 = R6 + R4; // digit in R6 from division, convert the digit to
              // ascii
B[P0] = R6; // and write it to memory
P0 += 1; // move ptr over for next digit
R3 = R1; // preserve current arg
R1 = R2;
R2 = 10;
[--SP] = RETS;
CALL _Divide; // new pwr10 = old pwr10 / 10
RETS = [SP++];
R2 = R6; // quotient in R6, store updated pwr10 back into R2
R1 = R3; // and restore arg in R1
JUMP Dec2StringLoop; // continue looping

Dec2StringDone:
R4 = ASCII_NULL;
B[P0] = R4; // null-terminate the string
(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// Display_Init
//
// Description: This function initializes the LCD display by writing the
//               appropriate values to the display.
//
// Operation: This function just goes through the initialization process for
//             the display. It first waits for the appropriate amount of time,
//             then writes the first function command to the display. It then
//             waits again, and then writes the second function command to the
//             display. It waits again, then sends the third function command
//             to the display. It then writes the fourth function command
to the display (this one has additional information such as
whether the cursor should blink or not). It then sends a command
to turn the display off, another one to clear the display, and
a third command to set the enable mode. After another wait loop,
the function sends a command to turn the display on. The function
then returns. Note that the Display_Char function is used to
send the actual commands to the display.

// Arguments: None.
// Return Value: None.

// Local Variables: R0 - frequently used as a loop counter
// Shared Variables: None.
// Global Variables: None.
// Input: None.
// Output: Display.
// Error Handling: None.
// Limitations: None.
// Algorithms: None.
// Data Structures: None.
// Registers Changed: None.
// Stack Depth: 28 words.
// Author: Ajay Mandlekar
// Last Modified: May 10, 2014

_Display_Init:
    [--SP] = (R7:0, P5:0);  // push all registers

DisplayInitPowerOnWait:  // loop to wait for appropriate amount of time after
display is powered on
    R0 = 0;  // use R0 as a loop counter

DisplayInitPowerOnWaitLoop:
    R0 += 1;  // increment counter
    R7.L = POWER_ON_WAIT;
    R7.H = POWER_ON_WAIT;
    CC = R0 < R7;
    if CC JUMP DisplayInitPowerOnWaitLoop;  // if not past wait time, loop
    // JUMP DisplayInitFuncCommand1;  // else move on to next step

DisplayInitFuncCommand1:  // write first func command to display
R0 = FUNC_COMMAND;
R1 = 0; // write func command to display, indicating that
       // R/S should be set to 0, since this is a command
[--SP] = RETS;
CALL _Display_Char; // send the func command
RETS = [SP++];

DisplayInitWait1: // wait after writing first command
R0 = 0; // use R0 for counter

DisplayInitWait1Loop:
    R0 += 1; // increment counter
    R7.L = WAIT_TIME_1;
    R7.H = WAIT_TIME_1;
    CC = R0 < R7;
    if CC JUMP DisplayInitWait1Loop; // if not past wait time, keep looping
    // JUMP DisplayInitFuncCommand2; // else move on to next step

DisplayInitFuncCommand2: // write second func command to display
R0 = FUNC_COMMAND;
R1 = 0;
[--SP] = RETS;
CALL _Display_Char; // send the func command
RETS = [SP++];

DisplayInitWait2: // wait after writing second command
R0 = 0; // use R0 for counter

DisplayInitWait2Loop:
    R0 += 1; // increment counter
    R7.L = WAIT_TIME_2;
    R7.H = WAIT_TIME_2;
    CC = R0 < R7;
    if CC JUMP DisplayInitWait2Loop; // if not past wait time, keep looping
    // JUMP DisplayInitFuncCommand3; // else move on to next step

DisplayInitFuncCommand3: // write third func command to display
R0 = FUNC_COMMAND_LINES;
R1 = 0;
[--SP] = RETS;
CALL _Display_Char; // send the func command
RETS = [SP++];

DisplayInitFuncCommand4: // write fourth func command to display
    // (sets number of lines and font)
R0 = FUNC_COMMAND_LINES;
R1 = 0;
[--SP] = RETS;
CALL _Display_Char; // send the func command
RETS = [SP++];

DisplayInitTurnOff: // turn display off (next in initialization proc)
R0 = DISPLAY_OFF;
R1 = 0;
[--SP] = RETS;
CALL _Display_Char; // send the display_off command
RETS = [SP++];

DisplayInitClear:    // clear the display
R0 = DISPLAY_CLEAR;
R1 = 0;
[--SP] = RETS;
CALL _Display_Char; // send the display_clear command
RETS = [SP++];

DisplayInitEntryModeSet:    // set entry mode
R0 = ENTRY_MODE_SET;
R1 = 0;
[--SP] = RETS;
CALL _Display_Char; // send the entry_mode_set command
RETS = [SP++];

DisplayInitOnWait:     // wait before turning on display
R0 = 0;    // use R0 as a counter

DisplayInitOnWaitLoop:
R0 += 1;    // increment counter
R7.L = WAIT_TIME_2;
R7.H = WAIT_TIME_2;
CC = R0 < R7;
if CC JUMP DisplayInitOnWaitLoop;   // if not past wait time, keep looping
// JUMP DisplayInitOn;    // else turn on display

DisplayInitOn:    // turn display on
R0 = DISPLAY_ON;
R1 = 0;
[--SP] = RETS;
CALL _Display_Char; // send the display_on command
RETS = [SP++];

DisplayInitDone:

(R7:0, P5:0) = [SP++];    // pop all registers
RTS;

// Display_Num
//
// Description: This function writes a number to the display by writing to
// the corresponding PF pins. The number is assumed to be a
// 16-bit unsigned decimal value.
//
// Operation: This function simply calls Dec2String to convert the passed
// decimal number to a string, then it calls Display_String in
// order to display the string.
//
// Arguments: R1 - the 16-bit unsigned decimal value.
R0 - DDRAM offset - where to display the number on the display

Return Value: None.

Local Variables: None.

Shared Variables: num_str - string buffer used in this function

Global Variables: None.

Input: None.

Output: Display.

Error Handling: None.

Limitations: None.

Algorithms: None.

Data Structures: None.

Registers Changed: None.

Stack Depth: 2 words.

Author: Ajay Mandlekar

Last Modified: May 10, 2014

_Display_Num:

[--SP] = P0;
P0.L = num_str;
P0.H = num_str;  // Dec2String takes P0 as arg, where to store str

[--SP] = RETS;  // Note that R1 is already setup as arg to
  // Dec2String
CALL _Dec2String;
RETS = [SP++];

R1 = NUM_LEN;  // pass max num of chars as arg to Display_String

[--SP] = RETS;  // Note that R0 is already set up as arg to
  // Display_String
CALL _Display_String;  // Displays string stored at P0, which is num_str
RETS = [SP++];

P0 = [SP++];
RTS;

_Display_String

// Description: This function writes a string to the display at a
// particular location. It always writes the same number of
// characters - the amount specified in R1 - 1. If the string
// is less than this size in length, the remaining characters
// written to the display are spaces.
//
// Operation: This function loops through the string, calling Display_Char on
// every individual character in the string. It always writes the
// same number of characters - the amount specified in R1 - 1.
//
// Arguments: P0 - address of string to display
//   R0 - the DDRAM offset needed to display the string at the
//    desired location
//   R1 - maximum number of characters allowed in the string
//     (including the ASCII_NULL)
//
// Return Value: None.
//
// Local Variables: R0 - the current string character
//   R3 - counts number of characters displayed
//   R7 - ASCII_NULL
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 10, 2014
//

_Display_String:
   [--SP] = (R7:0, P5:0); // push all registers
   R3 = 1;
   R2 = R1 - R3; // num of chars to write in R2, and exclude
                   // the ascii_null
   R1 = 0; // prepare for call to Display_Char, RS = 0
   [--SP] = RETS;
   CALL _Display_Char; // set the DDRAM offset by calling Display_Char
   RETS = [SP++];
R7 = ASCII_NULL;
R3 = 0; // use R3 as a counter

Display_String_Loop:
R0 = B[P0] (z); // read the character
CC = R0 == R7; // check if at end of string
if CC JUMP Display_String_End; // reached end of string
CC = R3 == R2; // check if already displayed max number of
// characters
if CC JUMP Display_String_Return; // displayed max num of characters
R1 = 1; // indicate that R/S should be 1 during the
// write by passing value to Display_Char
[--SP] = RETS;
CALL _Display_Char; // display the character
RETS = [SP++];
P0 += 1; // move over to next character
R3 += 1; // increment char count
JUMP Display_String_Loop; // and continue looping through the string

Display_String_End: // reached end of string, buffer out remaining
// space on display with spaces
CC = R3 == R2; // check if displayed max number of characters
if CC JUMP Display_String_Return; // if so, can return
R0 = ASCII_SPACE;
R1 = 1;

[--SP] = RETS;
CALL _Display_Char; // display the character
RETS = [SP++];
R3 += 1; // increment the counter
JUMP Display_String_End;

Display_String_Return:
(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// Display_Char
//
// Description: This function writes a character to the display by writing to
// the corresponding PF pins. This function also handles
// the necessary timing requirements by looping for an
// appropriate amount of time. This function can also change the
// display settings if given the appropriate values. In this case,
// the first argument should be the command to send to the display
// and the second argument should set RS appropriately.
//
// Operation: The function first checks the value of R1 and sets the R/S signal
// appropriately. It then writes the passed character in R0 to
// PF pins 0-7 to make the data available for the display. It then
// waits for an appropriate amount of time, then sets the enable
// signal on the display to be active. It uses another wait loop to
// hold enable active for an appropriate amount of time, and then
// sets enable to be inactive. Afterwards, the function uses another
// wait loop to satisfy the write cycle timing requirements of the
// display.

// Arguments: R0 - the character to write to the display
// R1 - the value of R/S to send to the display. A non-zero value
// sets R/S high while a value of zero sets R/S low. Note that
// the R/S value indicates whether the write to the display is
// data, or an actual command to the display.

// Return Value: None.

// Local Variables: P0 - address of FIO_FLAG_S register, used to set PF pins
// P1 - address of FIO_FLAG_C register, used to clear PF pins

// Shared Variables: none

// Global Variables: None.

// Input: None.
// Output: Display.

// Error Handling: None.

// Limitations: None.

// Algorithms: None.

// Data Structures: None.

// Registers Changed: None.

// Stack Depth: 28 words.

// Author: Ajay Mandlekar

// Last Modified: May 10, 2014

_Display_Char:
    [--SP] = (R7:0, P5:0); // push all registers
    R0 = R0.B (z);       // make sure R0 only contains the character
    P0.L = FIO_FLAG_S;    // PF set reg
    P0.H = FIO_FLAG_S;
    P1.L = FIO_FLAG_C;    // PF clear reg
    P1.H = FIO_FLAG_C;
    R2 = RS_PIN;         // value to isolate R/S pin
    CC = R1 == 0;        // find out whether to set or clear R/S
    if CC JUMP DisplayCharClearRS;
    JUMP DisplayCharSetRS;

DisplayCharSetRS:
    W[P0] = R2.L;        // set R/S
    JUMP DisplayCharDataValid;
DisplayCharClearRS:

W[P1] = R2.L; // clear R/S
// JUMP DisplayCharDataValid;

DisplayCharDataValid: // make data valid by writing char to PF pins
R3 = RW_PIN; // isolate R/W pin
W[P1] = R3.L; // clear R/W pin to indicate a write to the display
R4 = CLEAR_DATA;
W[P1] = R4.L; // clear the PF data pins
W[P0] = R0.L; // then set the appropriate ones to write the char
// JUMP DisplayCharWaitEnable;

DisplayCharWaitEnable: // wait appropriate amount of time before setting enable
R1 = 0; // use R1 for a counter

DisplayCharWaitEnableLoop:
R1 += 1; // increment counter
R2 = ENABLE_WAIT;
CC = R1 < R2;
if CC JUMP DisplayCharWaitEnableLoop; // if not past wait time, keep looping
// JUMP DisplayCharSetEnable; // else can set enable

DisplayCharSetEnable: // set the enable signal on the display
R3 = ENABLE_PIN; // isolate E pin
W[P0] = R3.L; // set the enable
// JUMP DisplayCharHoldEnable;

DisplayCharHoldEnable: // hold the enable high
R1 = 0; // use R1 for a counter

DisplayCharHoldEnableLoop:
R1 += 1;
R2 = ENABLE_HOLD;
CC = R1 < R2;
if CC JUMP DisplayCharHoldEnableLoop; // if not past wait time, keep looping
// JUMP DisplayCharClearEnable; // else can clear enable

DisplayCharClearEnable: // clear the enable signal on the display
R3 = ENABLE_PIN; // isolate E pin
W[P1] = R3.L; // clear the enable
// JUMP DisplayCharFinishCycle;

DisplayCharFinishCycle: // finish the write cycle
R1 = 0; // use R1 for a counter

DisplayCharFinishCycleLoop:
R1 += 1;
R2 = CYCLE_HOLD;
CC = R1 < R2;
if CC JUMP DisplayCharFinishCycleLoop; // if not past wait time, loop
// JUMP DisplayCharRet; // else, done

DisplayCharRet: // done, ready to return
(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// display_artist

// Description: This function takes a track artist string as an argument. It displays the artist.

// Operation: This function just sets up the display to display the artist at the right location on the display and then calls the Display_String function to do so.

// Arguments: R0 - address of string to display

// Return Value: None.

// Local Variables: None.

// Shared Variables: None.

// Global Variables: None.

// Input: None.

// Output: Display.

// Error Handling: None.

// Limitations: None.

// Algorithms: None.

// Data Structures: None.

// Registers Changed: None.

// Stack Depth: 28 words.

// Author: Ajay Mandlekar

// Last Modified: May 10, 2014

// display_artist:
[--SP] = (R7:0, P5:0); // push all registers

P0 = R0;
R0 = DDRAM_ARTIST_OFFSET; // set argument to Display_String to display string at right location
R1 = ARTIST_LEN; // set maximum number of characters to max length of artist
[--SP] = RETS;
CALL _Display_String; // display the artist string
RTS = [SP++];
(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// display_title
//
// Description: This function takes a title string as an argument. It displays the title.
//
// Operation: This function just sets up the display to display the title at the right location on the display and then calls the Display_String function to do so.
//
// Arguments: R0 - address of string to display
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: None.
// Output: Display.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 10, 2014
//

/display_title:
[--SP] = (R7:0, P5:0); // push all registers

P0 = R0;
R0 = DDRAM_TITLE_OFFSET; // set argument to Display_String to display string at right location
R1 = TITLE_LEN; // set maximum number of characters to max length of title
[--SP] = RETS;
CALL _Display_String; // display the title string
RET = [SP++];

(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// display_time

// Description: This function takes a time value as an argument. It displays the time.
// Operation: This function just sets up the display to display the time at the right location on the display and then calls the Display_Num function to do so.
// Arguments: R0 - time value to display
// Return Value: None.
// Local Variables: None.
// Shared Variables: None.
// Global Variables: None.
// Input: None.
// Output: Display.
// Error Handling: None.
// Limitations: None.
// Algorithms: None.
// Data Structures: None.
// Registers Changed: None.
// Stack Depth: 28 words.
// Author: Ajay Mandlekar
// Last Modified: May 10, 2014

_display_time:
    [--SP] = (R7:0, P5:0); // push all registers

R1 = R0;
R0 = DDRAM_TIME_OFFSET; // set argument to Display_Num to display string at right location

[--SP] = RETS;
CALL _Display_Num; // display the time
RET = [SP++];

(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// display_status
//
// Description: This function takes a status value as an argument. It
displays the status.
//
// Operation: This function just sets up the display to display the status
at the right location on the display. It then uses a table
lookup using the passed value in R0 to find the status string
to display, and then uses the Display_String function to do
so.
//
// Arguments: R0 - status value to display
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: status_table - a table filled with status strings
//
// Global Variables: None.
//
// Input: None.
// Output: Display.
//
// Error Handling: None.
//
// Limitations: No bounds checking on the value of R0. This function could
potentially access invalid memory if passed an invalid value
in R0.
//
// Algorithms: None.
//
// Data Structures: The status table.
//
// Registers Changed: None.
//
// Stack Depth:
//
// Author: Ajay Mandlekar
//
// Last Modified: May 10, 2014
//

_display_status:
    [--SP] = (R7:0, P5:0); // push all registers

    R1 = R0;
    R0 = DDRAM_STATUS_OFFSET; // set argument to Display_Num to
// display string at right location
R1 = R1 << 3;   // multiply by length of a status table string
              // to get real index
P1 = R1;
P0.L = status_table;
P0.H = status_table;
P0 = P0 + P1;  // get status string to display
R1 = STATUS_LEN; // set max number of chars to be status length

[--SP] = RETS;
CALL _Display_String; // display the status
RETS = [SP++];

(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// Declare shared variables here

.section .data

num_str: // number string, used by DisplayNum function
    .space NUM_LEN

status_table: // table used to display appropriate statuses - the status
              // value is an index into this table
    .string "Play   
    .string "Fastfwd"
    .string "Reverse"
    .string "Idle   
    .string "Illegal"
// This file contains the constants used by the display subroutines.

// Revision History:
// 5/10/2014   Ajay Mandlekar   Initial Revision.
// 5/27/2014   Ajay Mandlekar   Updated certain constants to support updates to display.s file.

.equ MAX_PWR_10, 10000 // maximum power of 10 to divide by for the conversion

.equ ASCII_NULL, 0 // string termination character (null)

.equ ASCII_SPACE, 0x20 // ascii code for a space

.equ DIGIT_TO_ASCII, '0' // value to add to a digit value to get its ascii representation

.equ MAX_CONV_STRING_SIZE, 7 // the maximum size of a string generated from using Dec2String

.equ ARTIST_LEN, 11 // length of artist string in bytes
.equ TITLE_LEN, 11 // length of title string in bytes
.equ STATUS_LEN, 8 // length of status string in bytes
.equ NUM_LEN, 6 // max length of a number converted to a string

.equ STAT_MUL, 3 // amount to shift status number by to index into the status table to get the right status string

// LCD Display Initialization constants

.equ POWER_ON_WAIT, 0x01A02728 // time to wait for in terms of core clocks after display power on (150 ms)

.equ WAIT_TIME_1, 0x0071C118 // time to wait after first set-up display write

.equ WAIT_TIME_2, 0x0002C6F0 // time to wait after second set-up display write and to wait right before turning display on

.equ FUNC_COMMAND, 0x0030 // PF bits to set for func commands (bits 4, 5)

.equ FUNC_COMMAND_LINES, 0x003C // PF bits to set for func command 4 (sets lines and font style) set bits 3 and 2 for 10-dot font and 2 line display

.equ DISPLAY_OFF, 0x0008 // PF bits to clear to turn off display

.equ DISPLAY_CLEAR, 0x0001 // PF bits to set to clear display
.equ ENTRY_MODE_SET, 0x0006
// PF bits to set to set entry mode (bits 2, 1)
// (PF 1 = 1 -> incrementing address makes cursor go right)
// (PF 0 = 0 -> no scrolling)

.equ DISPLAY_ON, 0x000F
// PF bits to set to turn display on (bits 0, 1, 2, 3)
// (PF 1 = 1 -> enable cursor)
// (PF 0 = 1 -> enable blinking)

.equ DDRAM_NEXT_LINE, 0xC0
// The command that tells the display to write to the next line

// LCD Writing Constants (used by DisplayChar function)
// write this value to PF clear reg to clear the display data bits (PF0 - 7)
.equ CLEAR_DATA, 0x00FF

// write this value to PF set or clear regs to set or clear R/S (PF 8)
.equ RS_PIN, 0x0100

// write this value to PF set or clear regs to set or clear R/W (PF 9)
.equ RW_PIN, 0x0200

// write this value to PF set or clear regs to set or clear E on display (PF 10)
.equ ENABLE_PIN, 0x0400

// LCD Timing Constants (for a write)
.equ ENABLE_WAIT, 100
// time in core clocks to wait until setting the enable on the display. This
// was calculated using timing specifications on the LCD display

.equ ENABLE_HOLD, 200
// time in core clocks to keep the enable set on the display.

.equ CYCLE_HOLD, 400
// time in core clocks to wait until the write ends (this is to ensure that
// the write cycle timing is satisfied.

// LCD DDRAM Offsets (determines where items are displayed)
.equ DDRAM_TITLE_OFFSET, 0xC0
// title will be displayed at begining of 1st line on the display

.equ DDRAM_STATUS_OFFSET, 0x8B
// status will be displayed near end of 1st line on the display

.equ DDRAM_KEY_OFFSET, 0x93
// current key will be displayed near end of 1st line on the display
.equ DDRAM_ARTIST_OFFSET, 0x80
// artist will be displayed at start of 2nd line on the display

.equ DDRAM_TIME_OFFSET, 0xD3
// time will be displayed near end of 2nd line on the display
// This file contains the divide routine for the Blackfin MP3 project.

// Table of Contents:
// Divide(a, b) - Takes two unsigned numbers and returns their quotient and remainder.

// allow other files to see these functions
.global _Divide

.section .text;

// Divide

// Description: This function takes two unsigned numbers and computes their quotient and remainder. The quotient and remainder will each be 32 bits.

// Operation: This function simply uses addition and subtraction to implement division. It keeps subtracting the divisor from the dividend to compute the quotient and remainder.

// Arguments: R1 - the dividend
// R2 - the divisor

// Return Value: R6 - quotient
// R7 - remainder

// Local Variables: R6 - the temporary quotient
// R7 - the temporary remainder

// Shared Variables: none

// Global Variables: None.

// Input: None.
// Output: None.

// Error Handling: None.
// Limitations: None.
// Algorithms: None.
// Data Structures: None.
// Registers Changed: None.
// Stack Depth: 0 words.
// Author: Ajay Mandlekar
// Last Modified: June 01, 2014

_Divide:
    R6 = 0;  // quotient = 0
    R7 = R1; // remainder = dividend

DivideComputeQuotient:
    CC = R7 < R2 (IU);  // check if remainder < divisor
    if CC JUMP DivideDone;  // if so, done with divide
    R7 = R7 - R2;  // remainder = remainder - divisor
    R6 += 1;  // increment the quotient
    JUMP DivideComputeQuotient;  // and keep looping

DivideDone:  // done, now return
    RTS;
**IDE Software**

The IDE software is responsible for reading data from the IDE hard drive. The main IDE functions are the IDE_Init function and the get_blocks function. The IDE_Init function is called in the boot code and simply initializes the IDE so that it is prepared to supply the audio data. The get_blocks function reads a specified number of blocks located at a specified LBA address from the IDE and moves the data into a specified location in memory. It uses LBA addressing mode.

There are also three helper functions used by the main functions. The IDE_Busy function blocks until the IDE is not busy. The IDE_CmdRdy function blocks until the IDE is ready to accept a command. The IDE_DataRdy function blocks until the IDE has data available.

Note that there are also three shared variables that are used for bookkeeping purposes.

The following block diagram describes how the functions interact with one another.

![Block Diagram](image)

The code is reproduced below. The relevant files are ide.s and ide.inc.
IDE file for EE52 Blackfin MP3 project.

Revision History:

5/26/2014    Ajay Mandlekar  Initial revision
6/15/2014    Ajay Mandlekar  Changed get_blocks to use LBA addressing instead of CHS.

This file contains the ide routines for the Blackfin MP3 project.

Table of Contents:
IDE_Init - initializes the IDE
IDE_Busy - blocks until IDE is not busy
IDE_CmdRdy - blocks until IDE is ready for a command
IDE_DataRdy - blocks until IDE data is ready
get_blocks - gets memory blocks from the IDE

allow other files to see these functions
.global _IDE_Init
.global _IDE_Busy
.global _IDE_CmdRdy
.global _IDE_DataRdy
.global _get_blocks

.include "src/sys/bfinregaddr.inc"
.include "src/sys/ide.inc"

.section .text;

IDE_Init

Description: This function initializes the IDE by sending commands to the IDE and reading from it.

Operation: This function first writes 0 to the dev bit in the head register to make sure that device 0 is selected. Then, it sends the identity command to the IDE, and then performs several consecutive reads to extract information from the IDE. It uses the IDE_Busy, IDE_CmdRdy, and IDE_DataRdy functions as needed. It stores the extracted information in the appropriate shared variables. Note that these shared variables are not used in the get_blocks function as it uses LBA addressing instead of CHS addressing.

Arguments: None.
// Return Value: None.
// Local Variables: None.
// Shared Variables: num_cylinders - the number of cylinders on the IDE
// hpc - the heads per cylinder on the IDE
// spt - the sectors per track on the IDE
// Global Variables: None.
// Input: IDE.
// Output: None.
// Error Handling: None.
// Limitations: None.
// Algorithms: None.
// Data Structures: None.
// Registers Changed: None.
// Stack Depth: 28 words.
// Author: Ajay Mandlekar
// Last Modified: May 26, 2014

_IDE_Init:
    [--SP] = (R7:0, P5:0); // push all registers

_IDE_Init_Write_Zero_To_Dev:
    [--SP] = RETS;
    CALL _IDE_Busy; // wait until IDE is not busy
    RETS = [SP++];

    P0.L = IDE_DEV_HEAD_REG;
    P0.H = IDE_DEV_HEAD_REG;
    R0 = B[P0] (z);
    BITCLR(R0, 4); // write 0 to dev to indicate device 0
    B[P0] = R0;

_IDE_Init_Identity_Command:
    [--SP] = RETS;
    CALL _IDE_Busy; // wait until IDE is not busy
    RETS = [SP++];
    [--SP] = RETS;
    CALL _IDE_CmdRdy; // wait until IDE is ready for a command
    RETS = [SP++];

    P0.L = IDE_COMMAND_REG; // write identify command to command register
P0.H = IDE_COMMAND_REG;
R0.H = 0;
R0.L = IDE_IDENTIFY_COMMAND;
B[P0] = R0;

IDE_Read_Identity: // do 7 consecutive reads to extract IDE info
[--SP] = RETS;
CALL _IDE_Busy; // wait until IDE is not busy
RETS = [SP++];
[--SP] = RETS;
CALL _IDE_DataRdy; // wait until IDE has data available
RETS = [SP++];

P0.L = IDE_DATA_REG;
P0.H = IDE_DATA_REG;
R0 = W[P0] (z); // R0 has general IDE info

P1.L = num_cylinders;
P1.H = num_cylinders;
R0 = W[P0] (z); // R0 has num of cylinders
W[P1] = R0; // write to num_cylinders shared variable

R0 = W[P0] (z); // R0 has unused value

P1.L = hpc;
P1.H = hpc;
R0 = W[P0] (z); // R0 has heads per cylinder
W[P1] = R0; // write to heads per count shared variable

R0 = W[P0] (z); // R0 has unused value
R0 = W[P0] (z); // R0 has unused value

P1.L = spt;
P1.H = spt;
R0 = W[P0] (z); // R0 has sectors per track
W[P1] = R0; // write to sectors per track shared variable

IDE_Init_Loop: // read the rest of the block
R7 = IDE_INIT_WORDS_LEFT;
LC0 = R7;
LOOP myloop2 LC0;
LOOP_BEGIN myloop2;
R0 = W[P0] (z); // read rest of block
LOOP_END myloop2;

IDE_Init_Done:
(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// IDE_Busy
//
// Description: This function blocks until the IDE is not busy.
// Operation: This function simply keeps reading the IDE Status register,
//   checking the busy flag each time. When the busy flag is not set,
//   the function returns.

// Arguments: None.

// Return Value: None.

// Local Variables: None.

// Shared Variables: None.

// Global Variables: None.

// Input: IDE.
// Output: None.

// Error Handling: None.

// Limitations: None.

// Algorithms: None.

// Data Structures: None.

// Registers Changed: None.

// Stack Depth: 28 words.

// Author: Ajay Mandlekar

// Last Modified: May 26, 2014

_IDE_busy:
   [--SP] = (R7:0, P5:0); // push all registers

IDE_busy_loop:
   P0.L = IDE_STATUS_REG;
   P0.H = IDE_STATUS_REG;
   R0 = B[P0] (z); // read status register
   CC = BITTST(R0, 7); // and check for busy flag
   if CC JUMP IDE_busy_loop; // if busy, keep looping
   // JUMP IDE_busy_ret; // else, not busy so can return

IDE_busy_ret:
   (R7:0, P5:0) = [SP++]; // pop all registers
   RTS;

// IDE_CmdRdy

// Description: This function blocks until the IDE can accept a command.
// Operation: This function simply keeps reading the IDE Status register,  
//   checking the Device Ready bit (DRDY) until it is set, at which  
//   point it will return.  
//  
// Arguments: None.  
//  
// Return Value: None.  
//  
// Local Variables: None.  
//  
// Shared Variables: None.  
//  
// Global Variables: None.  
//  
// Input: IDE.  
// Output: None.  
//  
// Error Handling: None.  
//  
// Limitations: None.  
//  
// Algorithms: None.  
//  
// Data Structures: None.  
//  
// Registers Changed: None.  
//  
// Stack Depth: 28 words.  
//  
// Author: Ajay Mandlekar  
//  
// Last Modified: May 26, 2014  
//

IDEO_CmdRdy:  
    [--SP] = (R7:0, P5:0);  // push all registers

IDEO_CmdRdy_Loop:  
P0.L = IDE_STATUS_REG;  
P0.H = IDE_STATUS_REG;  
R0 = B[P0] (z);  // read status register  
CC = BITTST(R0, 6);  // and check for DRDY  
if !CC JUMP IDE_CmdRdy_Loop;  // if device is not ready, keep looping  
// JUMP IDE_CmdRdy_Ret;  // else, device is ready, can return

IDEO_CmdRdy_Ret:  
    (R7:0, P5:0) = [SP++];  // pop all registers  
RTS;  

// IDE_CmdRdy
//  
// IDE_DataRdy
// Description: This function blocks until the IDE is ready to transfer data.
//
// Operation: This function simply keeps reading the IDE Status register,
//    checking the Device Request bit (DRQ) until it is set, at which
//    point it will return.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: IDE.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 26, 2014
//

 IDE_DataRdy:
    [--SP] = (R7:0, P5:0); // push all registers

IDE_DataRdy_Loop:
    P0.L = IDE_STATUS_REG;
    P0.H = IDE_STATUS_REG;
    R0 = B[P0] (z); // read status register
    CC = BITTST(R0, 3); // and check for DRQ
    if !CC JUMP IDE_DataRdy_Loop; // if device is not ready, keep looping
    // JUMP IDE_DataRdy_Ret; // else, data is ready, can return

IDE_DataRdy_Ret:
    (R7:0, P5:0) = [SP++] ; // pop all registers
    RTS;

// get_blocks
// Description: This function is passed an LBA block address, the number of IDE blocks to retrieve, and a destination address. This function retrieves the specified blocks of memory from the IDE and stores them at the specified destination address.

// Operation: This function starts by writing to the DEV_HEAD register in order to indicate that the IDE is device 0 and to indicate that LBA addressing is being used. It also clears the old head value at this time. The function then writes bits 0-7 of the passed LBA address into the SEC_NUM register, bits 8-15 into the CYLINDER_LOW register, bits 16-23 into the CYLINDER_HIGH register, and ors-in bits 24-27 into the DEV_HEAD register. The function does this by bit shifting the passed LBA address in R0. Then, the function iterates over the number of blocks to read. In each iteration, it indicates a single block read by writing to the SEC_COUNT register. It then sends the read command to the IDE. It then uses another loop to read an entire block, one word at a time. It moves each word to the correct place in memory, specified by argument R2. After reading the entire block, the function reads the SEC_COUNT register. If the block read was successful, the value read should be 0. It adds the value read to a running count. At the very end of the function, the number of successful blocks read is determined by subtracting this running count from the total number of requested blocks. This value is returned.

// Arguments: R0 - LBA block address
// R1 - number of blocks to transfer (a block is IDE_BLOCK_SIZE in length)
// R2 - destination address

// Return Value: R0 - number of blocks transferred

// Local Variables: R0 - bit-shifted repeatedly to store LBA address into appropriate registers.
// R3 - stores running count of unsuccessful block reads in the outer read loop
// R4 - used as a loop counter to count the number of words read in the inner read loop.
// R5 - stores IDE_BLOCK_SIZE in inner read loop.
// R6 - stores number of blocks read in the outer read loop
// P0 - used to store the current location in the destination buffer in the inner read loop that iterates over words read from the IDE.
// P1 - used to store address of the IDE Data register in order to reach each word from the IDE in the inner read loop

// Shared Variables: None.

// Global Variables: None.
// Input: IDE.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: June 15, 2014
//
_get_blocks:
    [--SP] = (R7:1, P5:0); // push all registers

get_blocks_init: // indicate LBA mode and device 0

    P0.L = IDE_DEV_HEAD_REG;
    P0.H = IDE_DEV_HEAD_REG;

    [--SP] = RETS;
    CALL _IDE_Busy; // wait until IDE is not busy
    RETS = [SP++];

    R7 = B[P0] (z);
    R6.H = 0;
    R6.L = IDE_HEAD_CLEAR;
    R7 = R7 & R6; // clear old head value and indicate dev 0
    R5.H = 0;
    R5.L = 0x40;
    R7 = R7 | R5; // set LBA mode

    [--SP] = RETS;
    CALL _IDE_Busy; // wait until IDE is not busy
    RETS = [SP++];

    B[P0] = R7;

get_blocks_write_LBA: // write the LBA address to appropriate regs

    P0.L = IDE_SEC_NUM_REG;
    P0.H = IDE_SEC_NUM_REG;

    [--SP] = RETS;
    CALL _IDE_Busy; // wait until IDE is not busy
    RETS = [SP++];
B[P0] = R0;  // write LBA bits 0-7
R0 >>= 8;
P0.L = IDE_CYLINDER_LOW_REG;
P0.H = IDE_CYLINDER_LOW_REG;

[--SP] = RETS;
CALL _IDE_Busy;  // wait until IDE is not busy
RETS = [SP++];

B[P0] = R0;  // write LBA bits 8-15
R0 >>= 8;
P0.L = IDE_CYLINDER_HIGH_REG;
P0.H = IDE_CYLINDER_HIGH_REG;

[--SP] = RETS;
CALL _IDE_Busy;  // wait until IDE is not busy
RETS = [SP++];

B[P0] = R0;  // write LBA bits 16-23
R0 >>= 8;
P0.L = IDE_DEV_HEAD_REG;
P0.H = IDE_DEV_HEAD_REG;

[--SP] = RETS;
CALL _IDE_Busy;  // wait until IDE is not busy
RETS = [SP++];

R7 = B[P0] (z);
R7 = R7 | R0;  // or-in LBA bits 24-27

[--SP] = RETS;
CALL _IDE_Busy;  // wait until IDE is not busy
RETS = [SP++];

B[P0] = R7;
R6 = 0;  // use R6 to keep track of number of blocks read
R3 = 0;  // and R3 to keep track of unsuccessful blocks

get_blocks_read:  // Set up to read data from IDE

P2.L = IDE_SEC_COUNT_REG;
P2.H = IDE_SEC_COUNT_REG;

R7 = 1;  // read one block at a time

[--SP] = RETS;
CALL _IDE_Busy;  // wait until IDE is not busy
RETS = [SP++];

B[P2] = R7;  // write number of blocks to read
P2.L = IDE_COMMAND_REG;  // write read command to command register
P2.H = IDE_COMMAND_REG;
R0.H = 0;
R0.L = IDE_READ_COMMAND;

[--SP] = RETS;
CALL _IDE_Busy;       // wait until IDE is not busy
RETS = [SP++];

[--SP] = RETS;
CALL _IDE_CmdRdy;     // wait until IDE is ready for a command
RETS = [SP++];

B[P2] = R0;

// Set up for read loop
P0 = R2;             // P0 has destination address
R4.H = 0;
R4.L = IDE_BLOCK_SIZE_MUL;
R5 = IDE_BLOCK_SIZE;

R4 = 0;             // use R4 as a loop counter to count number of
                    // words that have been transferred.
P1.L = IDE_DATA_REG;
P1.H = IDE_DATA_REG; // use P1 to read from data register

[--SP] = RETS;
CALL _IDE_Busy;     // wait until IDE is not busy
RETS = [SP++];
[--SP] = RETS;
CALL _IDE_DataRdy;  // wait until IDE has data available
RETS = [SP++];

get_blocks_read_loop:     // loop to transfer IDE data into memory
CC = R4 == R5;            // check if transferred all of the words yet
if CC JUMP get_blocks_end; // if so, done

R0 = W[P1] (z);          // read a word from the IDE
W[P0] = R0.L;            // and write it to memory
R4 += 1;                 // update counter
P0 += 2;                 // and move over in memory
JUMP get_blocks_read_loop;

get_blocks_read_blocks_loopend:
P1.L = IDE_SEC_COUNT_REG;
P1.H = IDE_SEC_COUNT_REG;
R4 = B[P1] (z);          // number of unsuccessful blocks
R3 = R3 + R4;            // add to running count
R6 += 1;
CC = R6 == R1;          // did we read the specified number of blocks?
if !CC JUMP get_blocks_read; // if not, read another block
get_blocks_end:
    R0 = R1 - R3;    // number of blocks transferred
    (R7:1, P5:0) = [SP++];  // pop all registers except R0, the return val
    RTS;

// Declare shared variables here
.section .data

// Note that the following shared variables are not used due to the get_blocks
// function using LBA addressing instead of CHS. However, these parameters
// are useful to keep around for debugging purposes.

num_cylinders:  // number of cylinders on IDE
    .int 0
    .ALIGN 4

hpc:            // heads per cylinder on IDE
    .int 0
    .ALIGN 4

spt:            // sectors per track on IDE
    .int 0
    .ALIGN 4
// ide.inc

// This file contains the constants used by the ide subroutines.

// Revision History:

// IDE Register Addresses
.equ IDE_DATA_REG, 0x20200000
.equ IDE_SEC_COUNT_REG, 0x20200004
.equ IDE_SEC_NUM_REG, 0x20200006
.equ IDE_CYLINDER_LOW_REG, 0x20200008
.equ IDE_CYLINDER_HIGH_REG, 0x2020000A
.equ IDE_DEV_HEAD_REG, 0x2020000C
.equ IDE_STATUS_REG, 0x2020000E // status reg when reading
.equ IDE_COMMAND_REG, 0x2020000E // command reg when writing

// IDE masks
.equ IDE_HEAD_CLEAR, 0xE0  // used to clear the head value
                        // also ensures that dev bit is cleared

// IDE Commands
.equ IDE_IDENTIFY_COMMAND, 0xEC  // to be used in initialization to get
      // IDE to identify itself
.equ IDE_READ_COMMAND, 0x21    // IDE command to start transferring
      // blocks of data

// IDE Block Size
.equ IDE_BLOCK_SIZE, 256
.equ IDE_BLOCK_SIZE_MUL, 8    // amount to shift left by to multiple
      // by the block size
.equ IDE_INIT_WORDS_LEFT, 249 // the number of words left to read
      // after the initial reads in IDE_Init
**Audio Software**

The audio software is responsible for transmitting bytes of audio data to the MP3 Decoder so that it can play audio. The software uses an active buffer that stores the currently playing audio data and a backup buffer that stores the next section of audio data. This is to ensure that there is always audio data available when the MP3 Decoder requests it.

When the play button is pressed, the audio_play function is called by the main loop. This function loads the active buffer with audio data and enables SPI interrupts. An SPI interrupt is triggered when the SPI transmit buffer is empty. The SPI interrupt handler is the serial_handler function. This function reads the demand pin (PF14). If it is high (this means that the MP3 Decoder is requesting audio data), the function transmits a byte of data by using the active buffer. It also checks whether the active buffer needs new data. If so, it replaces the active buffer with the data in the backup buffer and then indicates that another backup buffer is needed by setting the need_buff_flag shared variable. This shared variable is checked in the update function, which is called regularly by the main loop. The update function loads the backup buffer with audio data if the need_buff_flag shared variable is set. If the demand pin is low when the serial_handler is called, the serial_handler function disables SPI interrupts and enables demand pin interrupts, which are active-high level-triggered interrupts on the demand pin.

The next time that the demand pin goes high, an interrupt occurs and the audio_handler function is called. This function transmits a byte of audio data in the same way that the serial_handler function does. It also replaces the active buffer with the backup buffer if necessary. It then disables demand pin interrupts and enables SPI interrupts. The way that the serial_handler and audio_handler interact is necessary to ensure that the system can consistently play audio data. The audio functions should always be sending audio data over SPI when the demand pin is high. This is what the serial_handler function does. However, when the demand pin goes low, it shouldn’t transmit audio data. The reason that SPI interrupts must actually be disabled is that the SPI interrupt handler will not get called again until a transmit occurs. This is why demand pin interrupts are enabled when the demand pin is low – the audio_handler is essentially responsible for waking up the serial_handler so that it can resume sending audio data over SPI.

When the user wishes to stop the music, the audio_halt function is called. This function simply turns off SPI interrupts.

Note that the SPI serial interface is initialized in the serial_init function, which is called in the boot code.

The following block diagram describes the interaction of the audio functions with the shared variables in the file.
The audio code has been reproduced below. The relevant files are the audio.s file and the audio.inc file.
Audio file for EE52 Blackfin MP3 project.

Revision History:

5/10/2014    Ajay Mandlekar  Initial revision.
6/06/2014    Ajay Mandlekar  Added a serial_handler to handle SPI interrupts.
6/16/2014    Ajay Mandlekar  Updated comments.

This file contains the audio routines for the Blackfin MP3 project.

Table of Contents:

audio_play - starts to play the data in the passed buffer of the length
audio_halt - immediately halts the playing of audio
update - checks if the backup buffer needs data. If so, backup buffer is loaded with the passed data.
serial_init - Initializes the SPI interface.
serial_handler - interrupt handler for the demand pin on the audio output interface. Uses SPI serial protocol to transmit a byte of data.

allow other files to see these functions
.global _audio_play
.global _audio_halt
.global _update
.global _serial_init
.global _serial_handler
.global _audio_handler

.include "src/sys/bfinregaddr.inc"
.include "src/sys/audio.inc"

.section .text;

audio_play

Description: This function takes a buffer that is filled with audio data and the length of the buffer as arguments. It sets up the system to start playing audio data from the buffer by writing to appropriate shared variables and enabling the proper interrupt.

Operation: This function simply sets the active_buffer shared variable to point to the passed buffer, initializes the offset shared variable to 0, and then puts the passed size argument into the active_size shared variable. The function also sets a flag in
// in order to indicate that a new backup buffer is needed. Then, the
// function enables the SPI interrupt by writing to the IMASK
// register.
//
// Arguments: R0 - pointer to an audio buffer
// R1 - the size of the buffer
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: active_buffer - pointer to the buffer that contains the
// audio data that is currently being played
// active_offset - the current location within the
// active_buffer, from which audio data is being played
// active_size - the maximum byte index of the active buffer
// need_buff_flag - flag which indicates that the
// backup_buffer should be loaded with new data
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words
//
// Author: Ajay Mandlekar
//
// Last Modified: May 29, 2014
//
_audio_play:
   [--SP] = (R7:0, P5:0); // push all registers

   // Initialize shared variables
   P0.L = active_buffer;
P0.H = active_buffer;
   [P0] = R0; // active_buffer points to the passed audio buffer
   P0.L = active_size;
P0.H = active_size;
   R1 += -1; // decrement to convert to maximum word index
R1 = R1 << 1; // convert to maximum byte index of buffer
[P0] = R1; // active_size has max byte index of passed buffer
P0.L = active_offset;
P0.H = active_offset;
R7 = 0;
[P0] = R7; // offset = 0, since initially at beginning of buff
P0.L = need_buff_flag;
P0.H = need_buff_flag;
R7 = 1;
[P0] = R7; // indicate that backup buffer is needed

// Turn on the proper interrupt
P0.L = IMASK;
P0.H = IMASK;
R7 = [P0];
R6.L = SPI_INT_ENABLE;
R6.H = SPI_INT_ENABLE;
R7 = R7 | R6;
[P0] = R7; // enable SPI interrupts
(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// audio_halt
//
// Description: This function immediately stops playing the audio by turning
//   off the proper interrupt.
//
// Operation: This function simply turns off the SPI interrupt by
//   writing to the IMASK register.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: None.
//Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 29, 2014
//
__audio_halt:
    [--SP] = (R7:0, P5:0); // push all registers

    // Turn off the proper interrupt

    P0.L = IMASK;
P0.H = IMASK;
R7 = [P0];
R6.L = SPI_INT_DISABLE;
R6.H = SPI_INT_DISABLE;
R7 = R7 & R6;
[P0] = R7;                                // disable SPI interrupts

(R7:0, P5:0) = [SP++]; // pop all registers
RTS;

// update
//
// Description: This function is passed a buffer filled with audio data and
// the size of the buffer. This function checks if the backup buffer needs data. If so, the backup buffer pointer is set
to point to the passed buffer, and the function returns 1. Otherwise, the function returns 0.
//
// Operation: This function checks the need_buff_flag shared variable to
// check whether a new backup buffer is needed. If so, it sets
// the backup buffer to point to the passed buffer, sets the
// backup size equal to the passed size, and returns 1. Else,
// it returns 0.
//
// Arguments: R0 - an audio data buffer
//            R1 - the size of the buffer, in words
//
// Return Value: R0 - 1 if the passed data was loaded into the backup buffer,
//               0 otherwise.
//
// Local Variables: None.
//
// Shared Variables: need_buff_flag - flag which indicates that the
// backup_buffer should be loaded with new data
// backup_buffer - pointer to the buffer that contains the
// second buffer of audio data (the next set of audio data)
backup_size - the maximum byte index of the backup buffer

Global Variables: None.

Input: None.
Output: None.

Error Handling: None.
Limitations: None.
Algorithms: None.
Data Structures: None.
Registers Changed: None.
Stack Depth: 26 words.

Author: Ajay Mandlekar
Last Modified: May 29, 2014

_update:
[--SP] = (R7:1, P5:0); // push all registers
P0.L = need_buff_flag;
P0.H = need_buff_flag;
R7 = [P0]; // read the flag
R6 = 0;
CC = R7 == R6; // check the flag
if CC JUMP update_no_load; // flag not set, no load needed
// JUMP update_load; // else, flag set, need load

update_load:
R1 = 0;
[P0] = R1; // clear need_buff_flag
P0.L = backup_buffer;
P0.H = backup_buffer;
[P0] = R0; // backup_buffer points to passed buffer
P0.L = backup_size;
P0.H = backup_size;
R1 += -1; // convert to maximum word index
R1 = R1 << 1; // get number of bytes (max byte index)
[P0] = R1; // set backup_size to equal max byte index
R0 = 1; // indicate that loading occurred
JUMP update_done;

update_no_load:
R0 = 0; // indicate that no load took place
// JUMP update_done;

update_done:
(R7:1, P5:0) = [SP++]; // pop all registers
// serial_init
//
// Description: This function initializes the SPI serial interface.
//
// Operation: This function writes the appropriate baud rate (defined in
//           audio.inc) to the SPI_BAUD register, and the appropriate value to
//           the SPI control register in order to initialize the SPI port.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 29, 2014
//

_serial_init:
  [-SP] = (R7:0, P5:0); // push all registers
  P1.L = SPI_BAUD;
P1.H = SPI_BAUD;
R0.L = SPI_BAUD_VAL;
W[P1] = R0.L; // write baud rate
P0.L = SPI_CTL;
P0.H = SPI_CTL;
R0.L = SPI_CTL_VAL;
W[P0] = R0.L;
(R7:0, P5:0) = [SP++]; // pop all registers
RTS;
// audio_handler
//
// Description: This function is the interrupt handler for the demand pin on the MP3 Decoder board. It uses SPI serial protocol to transmit a byte of audio data from the active buffer, and then replaces the active buffer with the backup buffer if necessary. It also disables demand pin interrupts and enables SPI interrupts.
//
// Operation: This function first transmits a byte of audio data from the active buffer using the active_buffer and active_offset shared variables to get the correct byte of audio data to transmit. It then updates the active_offset and checks whether the active buffer must be replaced. If so, it replaces the active buffer with the backup buffer and indicates that the backup buffer must be replaced by setting the need_buff_flag. At the end of the function, the handler turns off demand pin interrupts and enables SPI interrupts.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: active_buffer - pointer to the buffer that contains the audio data that is currently being played
// active_offset - the current location within the active_buffer, from which audio data is being played
// need_buff_flag - flag which indicates that the backup_buffer should be loaded with new data
// backup_buffer - pointer to the buffer that contains the second buffer of audio data (the next set of audio data)
// backup_size - the maximum byte index of the backup buffer
//
// Global Variables: None.
//
// Input: None.
// Output: SPI (to the MP3 Decoder board)
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 29, 2014
//
_audio_handler:
    [--SP] = RETI;
    [--SP] = (R7:0, P5:0); // push all registers

audio_handler_transmit: // transmit a byte of audio data
    P0.L = SPI_TDBR;
    P0.H = SPI_TDBR;
    P1.L = active_buffer;
    P1.H = active_buffer;
    P2.L = active_offset;
    P2.H = active_offset;
    R0 = [P1]; // get start of buffer
    R1 = [P2]; // and offset
    R0 = R0 + R1;
    P1 = R0; // P1 points to next byte of audio data to transfer
    R0 = B[P1] (z); // R0 has the byte to transfer
    W[P0] = R0.L; // write the byte to the transfer buffer
    R1 += 1; // increment the offset
    [P2] = R1; // and write back updated value

audio_handler_compare: // find out if buffer needs replacement
    P3.L = active_size;
    P3.H = active_size;
    R3 = [P3];
    CC = R1 < R3; // compare offset and size to check if need to use backup buffer
    if CC JUMP audio_handler_done; // don't need to use it
    // JUMP audio_handler_replace; // active buffer has been used, need to replace

audio_handler_replace: // replace the active buffer
    P3.L = backup_buffer;
    P3.H = backup_buffer;
    R3 = [P3];
    P4.L = active_buffer;
    P4.H = active_buffer;
    [P4] = R3; // loads backup buffer to be active
    R4 = 0;
    [P2] = R4; // active_offset = 0
    P2.L = backup_size;
    P2.H = backup_size;
    R4 = [P2];
    P2.L = active_size;
    P2.H = active_size;
    [P2] = R4; // active_size = backup_size
    P5.L = need_buff_flag;
P5.H = need_buff_flag;
R5 = 1;
[P5] = R5; // set the need_buff_flag to indicate
// that backup buffer must be replaced

audio_handler_done:

P0.L = FIO_MASKB_C;
P0.H = FIO_MASKB_C;
R7.L = AUDIO_INT_DISABLE;
W[P0] = R7.L; // disable the demand pin interrupt

P0.L = IMASK;
P0.H = IMASK;
R6.L = SPI_INT_ENABLE;
R6.H = SPI_INT_ENABLE;
R7 = [P0];
R7 = R7 | R6;
[P0] = R7; // enable SPI interrupts

(R7:0, P5:0) = [SP++]; // pop all registers
RETI = [SP++];
RTI;

// serial_handler
//
// Description: This function is the SPI interrupt handler. It checks the
demand pin (PF14), and transmits a byte of audio data if
demand is high. It then replaces the active buffer with the
backup buffer if necessary. If demand is low, the function
disables SPI interrupts and enables demand pin interrupts.
//
// Operation: This function first reads the demand pin. If it is high, the
function transmits a byte of audio data from the
active buffer using the active_buffer and active_offset shared
variables to get the correct byte of audio data to transmit.
It then updates the active_offset and checks whether the
active buffer must be replaced. If so, it replaces the
active buffer with the backup buffer and indicates that the
backup buffer must be replaced by setting the need_buff_flag.
If the demand pin is low, the function turns off SPI interrupts
by writing to the IMASK register and turns on demand pin
interrupts by writing to the FIO_MASKB register.
//
// Arguments: None.
//
// Return Value: None.
/
// Local Variables: R1 - active_offset
// P2 - active_offset (memory location)
//
// Shared Variables: active_buffer - pointer to the buffer that contains the
// audio data that is currently being played
// active_offset - the current location within the
active_buffer, from which audio data is being played

need_buff_flag - flag which indicates that the backup_buffer should be loaded with new data

backup_buffer - pointer to the buffer that contains the second buffer of audio data (the next set of audio data)

backup_size - the maximum byte index of the backup buffer

// Global Variables: None.

// Input: None.
// Output: SPI (to the MP3 Decoder board)

// Error Handling: None.
// Limitations: None.
// Algorithms: None.
// Data Structures: None.
// Registers Changed: None.
// Stack Depth: 28 words.
// Author: Ajay Mandlekar
// Last Modified: May 29, 2014

_serial_handler:

[--SP] = RETI;
[--SP] = (R7:0, P5:0);  // push all registers

R7 = 100;  // wait loop added for improved functionality
LC0 = R7;
LOOP myloop2 LC0;
LOOP_BEGIN myloop2;
NOP;
LOOP_END myloop2;

P0.L = FIO_FLAG_D;
P0.H = FIO_FLAG_D;
R0 = W[P0];  // read demand pin
CC = BITTST(R0,14);
if CC JUMP serial_handler_transmit;  // if set, can transmit a byte
//JUMP serial_handler_enable  // else, enable demand pin interrupts

serial_handler_enable:

P0.L = IMASK;
P0.H = IMASK;
R6.L = SPI_INT_DISABLE;
R6.H = SPI_INT_DISABLE;
R7 = [P0];
R7 = R7 & R6;
[P0] = R7; // disable SPI interrupts

P0.L = FIO_MASKB_S;
P0.H = FIO_MASKB_S;
R7.L = AUDIO_INT_ENABLE;
W[P0] = R7.L; // enable the demand pin interrupt
JUMP serial_handler_done

serial_handler_transmit: // transmit next byte of audio data
P0.L = SPI_TDBR;
P0.H = SPI_TDBR;
P1.L = active_buffer;
P1.H = active_buffer;
P2.L = active_offset;
P2.H = active_offset;
R0 = [P1]; // get start of buffer
R1 = [P2]; // and offset
R0 = R0 + R1;
P1 = R0;
// P1 points to next byte of audio data to transfer
R0 = B[P1] (z); // R0 has the byte to transfer
W[P0] = R0.L; // write the byte to the transfer buffer
R1 += 1; // increment the offset
[P2] = R1; // and write back updated value

serial_handler_compare: // find out if buffer needs replacement
P3.L = active_size;
P3.H = active_size;
R3 = [P3];
CC = R1 < R3; // compare offset and size to check if // need to use backup buffer
if CC JUMP serial_handler_done; // don't need to use it // active buffer has been used, need // to replace

serial_handler_replace: // replace the active buffer
P3.L = backup_buffer;
P3.H = backup_buffer;
P4.L = active_buffer;
P4.H = active_buffer;
[P4] = R3; // loads backup buffer to be active
R4 = 0;
[P2] = R4; // active_offset = 0
P2.L = backup_size;
P2.H = backup_size;
R4 = [P2];
P2.L = active_size;
P2.H = active_size;
[P2] = R4; // active_size = backup_size
P5.L = need_buff_flag;
P5.H = need_buff_flag;
R5 = 1;
[P5] = R5;  // set the need_buff_flag to indicate
             // that backup buffer must be replaced

serial_handler_done:

    (R7:0, P5:0) = [SP++];  // pop all registers
    RETI = [SP++];
    RTI;

// Declare shared variables here
.section .data
.align 4;

active_buffer: // pointer to the buffer that contains the audio data that is
               // currently being played
    .int 0

active_offset: // the current location within the active_buffer, from which
               // audio data is being played
    .int 0

active_size:  // the maximum byte index of the active buffer. This essentially
              // stores the byte offset of the last element in the buffer.
    .int 0

backup_buffer: // pointer to the buffer that contains the second buffer of
                // audio data (the next set of audio data)
    .int 0

backup_size:  // the maximum byte index of the backup buffer. This essentially
              // stores the byte offset of the last element in the buffer.
    .int 0

need_buff_flag: // flag which indicates that the backup_buffer should be loaded
                 // with new data
    .int 0


// audio.inc
//
// This file contains the constants used by the audio
// subroutines.
//
// Revision History:
// 5/10/2014   Ajay Mandlekar       Initial Revision.
// 6/16/2014   Ajay Mandlekar       Improved documentation.
//

value to write to FIO_MASKB_S reg in order to enable PF14 on PF interrupt
// B, which interrupts whenever audio data is needed
.equ AUDIO_INT_ENABLE, 0x4000

value to write to FIO_MASKB_C reg in order to disable PF14 on PF interrupt
// B, which interrupts whenever audio data is needed
.equ AUDIO_INT_DISABLE, 0x4000

value to isolate PF14, the demand pin
.equ DEMAND_PIN, 0x4000

value to write to SPI control register to initialize serial port
//
// 0101 0100 0000 0001
// 0--- ---- ---- ---- reserved
// -1-- ---- ---- ---- SPI enabled
// --0- ---- ---- ---- normal WOM
// ---1 ---- ---- ---- master
// ---- 0--- ---- ---- ---- active high SCK
// ---- -1-- ---- ---- SCK toggles from beginning of first
data bit and slave select pins controlled by software
// ---- --0- ---- ---- MSB sent first
// ---- ---0 ---- ---- 8 bit transfers
// ---- ---- 00-- ---- reserved
// ---- ---- --0- ---- MISO disabled
// ---- ---- ---- 0---- Slave Select disabled
// ---- ---- ---- 0--- when SPI_RDBR is full, discard incoming data
// ---- ---- ---- -0-- when SPI_TDBR is empty, send last word
// ---- ---- ---- --01 Start transfer with write of SPI_TDBR, interrupt when
//            SPI_TDBR is empty
//
.equ SPI_CTL_VAL, 0x5401

// Need baud rate to be 11 for a 0.9 MHz bit transfer. SCLK is 20MHz (measured
// on oscilloscope).
// SCK Frequency = SCLK / (2 * SPI_BAUD_VAL) = 0.9 MHz
.equ SPI_BAUD_VAL, 11

// Value to or-in to IMASK reg in order to enable SPI interrupts (IVG10)
.equ SPI_INT_ENABLE, 0x00000400

// Value to and-in to IMASK reg in order to disable SPI interrupts (IVG10)
.equ SPI_INT_DISABLE, 0xFFFFFBFF
Interrupts

There are several interrupts that are utilized in the system. The interrupts types that were used include the Timer0 interrupt, the PF Int A interrupt, the PF Int B interrupt, and the SPI interrupt. The interrupts all serve different purposes. The Timer0 interrupt is used to handle rotary switch presses. The PF Int A interrupt is used to handle rotary switch turns. The PF Int B interrupt is used to service demand pin interrupts to transmit audio data. Finally, the SPI interrupt is also used to transmit audio data.

Each interrupt must be assigned to an IVG that resides in the event vector table, and the address of the interrupt handler must be written to the IVG corresponding to the interrupt. The default IVG settings were used for the Timer0 interrupt (IVG11), the PF Int A interrupt (IVG12), and the SPI interrupt (IVG10), while IVG14 was assigned to the PF Int B interrupt. The reason that a different IVG is used for PF Int B is because by default, the PF Int A and B interrupts share an IVG. For two different handlers to be installed for the interrupts, two different IVGs must be used.

There are several InstallHandler functions that write the address of the interrupt handler into the corresponding IVGs. These include the InstallTimer0EventHandler function, the InstallPFEventHandlers function, and the InstallSerialEventHandler function. These functions are all called in the boot code.

The InterruptInit function is responsible for actually enabling the system interrupts. For an interrupt to be enabled, it must be mapped to a corresponding IVG, the IVG must contain the address of an interrupt handler, the interrupt type must be enabled, and the proper IVG must be enabled. By the time the InterruptInit function is called, the first two requirements should have been met. The InterruptInit function takes care of the last two requirements. Note that IVG10, which corresponds to the SPI interrupt, is not enabled here. This is because the SPI interrupt should only be enabled when the user wants to play music. For more information on how the SPI and demand pin interrupts work together to transmit audio data, see the Audio Software section of the Software Manual.

Note that there is some additional setup that applies to PF pin interrupts. See the Boot Code section of the Software Manual for more information on this.

The following block diagram show how the interrupt functions work.
The code has been reproduced below. The relevant files are the interrupts.s and the interrupts.inc files.
// This file contains the subroutines necessary to set up the proper
// interrupts and their handlers for the Blackfin MP3 project.
//
// Table of Contents:
// InterruptInit - writes appropriate values to registers to enable
// the proper interrupts
// InstallTimer0EventHandler - installs the Timer 0 event handler
// InstallPFEventHandlers - installs the PF interrupt event handlers
// InstallSerialEventHandler - installs the SPI serial interrupt handler
//
.extern _Rotate_Handler
.extern _Press_Handler
.extern _serial_handler
.extern _audio_handler

// allow other files to see these functions
.global _InterruptInit
.global _InstallTimer0EventHandler
.global _InstallPFEventHandlers
.global _InstallSerialEventHandler

.include "src/sys/interrupts.inc"
.include "src/sys/bfinregaddr.inc"

.section .text;

// InterruptInit
//
// Description: This function writes to appropriate values to registers to
// enable the proper interrupts.
//
// Operation: This function simply writes appropriate values to the IMASK
// and SIC_IMASK registers in order to enable interrupts and
// the corresponding interrupt vectors. It also writes to SIC_IAR2
// register to assign a new IVG to the PF_B interrupt, as the
default IVG is already being used by the PF_A interrupt.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 3, 2014
//

InterruptInit:
    [ --SP ] = (R7:0, P5:0); // push regs

    // Use default IVGs for Timer0, PF_A, and SPI but assign new IVG to PF_B interrupt.
    P0.L = SIC_IAR2;
    P0.H = SIC_IAR2;
    R0 = [P0];
    R1.L = SIC_IAR2_CLR_PFB;
    R1.H = SIC_IAR2_CLR_PFB;
    R0 = R0 & R1; // clear PF_B IVG setting
    R1.L = SIC_IAR2_SET_PFB;
    R1.H = SIC_IAR2_SET_PFB;
    R0 = R0 | R1; // or-in new IVG setting
    [P0] = R0; // assign new IVG to PF_B interrupt

    // Enable the proper interrupts types.
    P1.L = SIC_IMASK;
    P1.H = SIC_IMASK;
    R1.L = SIC_IMASK_VAL;
    R1.H = SIC_IMASK_VAL;
    [P1] = R1;
// Enable correct interrupt vectors.
P0.L = IMASK;
P0.H = IMASK;
R0.L = IMASK_VAL;
R0.H = IMASK_VAL;
[P0] = R0;

(R7:0, P5:0) = [SP++]; // restore regs
RTS;

// InstallTimer0EventHandler
//
// Description: This function moves the address of the Timer 0 Event Handler into the interrupt vector corresponding to a Timer 0 interrupt.
//
// Operation: This function simply writes the address of the Timer 0 Event Handler into the memory location of IVG11. The Timer 0 Event Handler is the Press_Handler function. It handles debouncing and registering rotary encoder switch presses.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: May 3, 2014
//
_InstallTimer0EventHandler:
  [--SP] = (R7:0, P5:0); // push regs
P0.L = _Press_Handler;
P0.H = _Press_Handler;
P1.L = IVG11;
P1.H = IVG11;
[P1] = P0; // writes address of event handler into interrupt vector table
(R7:0, P5:0) = [SP++]; // restore regs
RTS;

// InstallPFEventHandlers

// Description: This function moves the address of the PF Event Handlers
// into the interrupt vectors corresponding to a PF interrupt.
//
// Operation: This function simply writes the address of the Rotate_Handler
// function into IVG12. It handles the rotation of the rotary
// encoder. Then it writes the address of the audio_handler
// function into IVG14. This handles an audio interrupt, which is
// triggered by the demand pin through the mp3 decoder.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: June 7, 2014
//

__ InstallPFEventHandlers:
    [--SP] = (R7:0, P5:0); // push regs
    P0.L = _Rotate_Handler;
P0.H = _Rotate_Handler;
P1.L = IVG12;
P1.H = IVG12;
[P1] = P0; // writes address of event handler into interrupt vector table
P0.L = _audio_handler;
P0.H = _audio_handler;
P1.L = IVG14;
P1.H = IVG14;
[P1] = P0; // writes address of event handler into interrupt vector table
(R7:0, P5:0) = [SP++]; // restore regs
RTS;

// InstallSerialEventHandler
//
// Description: This function moves the address of the SPI serial interrupt
// handler into the interrupt vector corresponding to an SPI
// interrupt.
//
// Operation: This function simply writes the address of the serial_handler
// function into IVG10. This handles a serial interrupt, which is
// triggered by the serial transmit buffer becoming empty.
//
// Arguments: None.
//
// Return Value: None.
//
// Local Variables: None.
//
// Shared Variables: None.
//
// Global Variables: None.
//
// Input: None.
// Output: None.
//
// Error Handling: None.
//
// Limitations: None.
//
// Algorithms: None.
//
// Data Structures: None.
//
// Registers Changed: None.
//
// Stack Depth: 28 words.
//
// Author: Ajay Mandlekar
//
// Last Modified: June 7, 2014
//

_INSTALLSerialEventHandler:
    [--SP] = (R7:0, P5:0); // push regs
P0.L = _serial_handler;
P0.H = _serial_handler;
P1.L = IVG10;
P1.H = IVG10;
[P1] = P0; // writes address of event handler into interrupt vector table
(R7:0, P5:0) = [SP++]; // restore regs
RTS;
// This file contains constants related to setting up the interrupts for
// the Blackfin MP3 project.

// Revision History:
// 5/03/2014   Ajay Mandlekar   Initial Revision.
// 6/07/2014   Ajay Mandlekar   Modified constants to account for audio
//                       and serial interrupts.

// write to SIC_IMASK register to enable the proper interrupts
// 0000 0000 0001 1001 0010
// ---- ---- ---1 ---- ---- enable PF Int B
// ---- ---- 1--- ---- ---- enable PF Int A
// ---- ---- ---- 1--- ---- enable Timer0 Int
// ---- ---- ---- ---- --1- enable SPI Int
.equ SIC_IMASK_VAL,       0x00192000

// value to write to IMASK register to enable the proper IVGs
// 0000 0000 0000 0000 0101 1000 0001 1111
// 0000 0000 0000 0000 ---- ---- ---- reserved
// ---- ---- ---- -1-- ---- ---- ---- enable IVG14 (for PF Int B)
// ---- ---- ---- ---- -1--- ---- ---- enable IVG12 (for PF Int A)
// ---- ---- ---- ---- ---- 1--- ---- ---- enable IVG11 (for Timer0)
// ---- ---- ---- ---- ---- ---- ---1 1111 reserved

// Note that IVG12 and IVG11 are assigned by default to PFIntA and Timer0
// interrupts and that IVG14 will be assigned to PF Int B in the code.
.equ IMASK_VAL,         0x0000581F

// value to or-in to SIC_IAR2 reg in order to assign IVG14 to PF Int B
.equ SIC_IAR2_SET_PFB, 0x00070000

// value to and-in to SIC_IAR2 reg in order to clear the previous IVG
// setting for PF_B
.equ SIC_IAR2_CLR_PFB, 0xFFFF0000
Schematic and PCB Revisions

It is important to note that the board cannot actually output audio, and only makes sputtering noises. Also, it seems as though both the SRAM and ROM chips are fried and of no use.

The following table describes the revisions that were made from the original design of the schematic and PCB to the final design.

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>04/14/14</td>
<td>All pull-up and pull-down resistor values were changed from 4.7 kΩ to 10 kΩ due to greater availability of 10 kΩ surface mount resistors.</td>
</tr>
<tr>
<td>2</td>
<td>04/21/14</td>
<td>Resistors R35 and R51 were not put on the board. This is because they were originally intended to function as jumpers for the BMODE0 and BMODE1 pins on the CPU to easily change the boot mode of the CPU.</td>
</tr>
<tr>
<td>3</td>
<td>04/28/14</td>
<td>The reset chip was not put on the board due to footprint and soldering issues (the wrong footprint was ordered, and in trying to fix the problem through soldering, a few pads were put out of commission). Thus, the system does not have the capability to reset.</td>
</tr>
<tr>
<td>4</td>
<td>05/05/14</td>
<td>Buffers U7 and U8 were removed due to bus conflict issues. The buffer directions were not correct, and this caused the CPU to heat up. The bypass capacitors on the power lines routed to the buffers were left on the board, as were two pull-up resistors (R36, R37) on the direction pins of buffer U7. The resistors were harmless as the buffer had been removed. However, R38, R39, R40, R41, R42, R43 were all removed. These had been removed in the debugging process while trying to find the buffers that were responsible for buffer conflicts.</td>
</tr>
<tr>
<td>5</td>
<td>05/05/14</td>
<td>Resistor R2 was removed and the BNMI pin was connected directly to ground. This change was made because BNMI had been pulled high, but the datasheet specified that it needed to be pulled low. Pulling it high caused non-maskable interrupts to trigger consistently.</td>
</tr>
<tr>
<td>6</td>
<td>05/12/14</td>
<td>One of the direction pins of buffer U14 was pulled high. It was previously connected to PF9, the read/write signal of the display, but the polarity was wrong, so the trace going to that buffer pin was cut, and the buffer pin was connected directly to 3.3V. This requires that the CPU will only write to the display and never read from it.</td>
</tr>
<tr>
<td>7</td>
<td>05/19/14</td>
<td>The BAMS2 signal was shorted to the AMS2 signal using a wire. This was due to the removal of buffer U7 – the AMS2</td>
</tr>
</tbody>
</table>
The BAMS3 signal was shorted to the AMS3 signal using a wire. This was due to the removal of buffer U7 – the AMS3 signal still needed to reach the DRAM so that it would have a valid Chip Select pin.

Resistors R18, R21, and R22 were removed, and the pads were solder bridged together. Earlier, it was believed to be a good idea to put a 10 kΩ resistor between the RAS and CAS outputs of the CPLD. However, DRAM reads and writes became successful after the resistors were removed so that the RAS and CAS outputs (RAS, CAS0, and CAS1 signals) were routed directly to the corresponding pins on the DRAM.

It should be noted that although the DAC, and most of the components that involve the DAC were put on the board, they were not used. It is the mini-din MP3 connector that was used to output audio by connecting to an external MP3 decoder board.

The following pictures illustrate the changes that were made on the actual PCB.

Revision #2: R35 and R51 were not put on the board.
Revision #3: The reset chip was not put on the board. The reason for this is painfully obvious in the above picture.

Revision #4: Buffers U7 and U8 were removed due to bus conflict issues.
Revision #5: Resistor R2 was removed because BNMI should not be pulled high.

Revision #5: A wire was used to ground the BNMI signal.
Revision #6: One of the direction pins on the U14 buffer (the one that corresponded to PF0-7, the display data pins) was connected directly to 3.3V using a wire.

Revision #7: The BAMS2 signal was connected to the AMS2 signal using a wire.
Revision #8: The BAMS3 signal was connected to the AMS3 signal using a wire.

Revision #9: The pads of resistor R18 were solder bridged together.
Revision #9: The pads of resistor R21 were solder bridged together.

Revision #9: The pads of resistor R22 were solder bridged together.

**Additional Notes**

1. The system does not really play audio. It makes a sputtering noise at best.
2. The SRAM is not functional. It used to be, but it isn’t anymore.
3. The ROM is not functional. It used to be, but it isn’t anymore.