Musical Instrument

REPORT

Friday 10:00 lab

Date: 07.12.2012

Team:

A.B.C.

Telecommunications and Computer Science

IFE 7th semester

Subject chosen as elective course

Devices used:

LPCXpresso Board G with LPC1343 Cortex-M3

LPCXpresso Board C with LPC1343 Cortex-M3

No external modules used.

Interfaces used:

GPIO, I2C, SPI, UART

Devices used:

Accelerometer, light sensor, OLED, speaker, XBee, button, rotary encoder, 7-segment display

Contents

1. Project description 3
   1.1. General description 3
   1.2. Playing the instrument 3
   1.3. Instrument modes 4
   1.3.1. Continuous mode – mode '0' 4
   1.3.2. Musical scale mode – mode '1' 4
   1.4. Boards playing in unison (ZigBee) 5
2. Peripherals and interface configuration 6
   2.1. GPIO 6
   2.2. I2C 8
   2.3. SPI 8
   2.4. UART (ZigBee) 9
   2.5. Accelerometer 10
   2.6. Light Sensor 12
   2.7. PWM timer 12
   2.8. Button debounce timer 13
3. Failure Mode and Effect Analysis 14
4. References 15
5. Other information 16

1. Project description

1.1. General description

The project is an embedded musical instrument which introduces a novel and unique way of controlling sound. The instrument is played by hovering your hand over the board and rotating the board to change the sound pitch. An addition is the ability to make another board play in unison with the user, but an octave lower, using wireless communication.

The program offers multiple other features such as displaying the currently played frequency in Hertz on the OLED display, switching the mode of the instrument by pressing a button, displaying the current instrument mode on the 7-segment display. Volume can also be controlled by means of a rotary encoder on both boards. The other board's sound can be enabled or disabled with a button.

1.2. Playing the instrument

Playing the instrument is very simple and intuitive. Hovering your hand above the board limits the light coming to the light sensor on the board. When the value of light intensity is lower than the predefined threshold value – sound is emitted. In normal room light conditions this means hovering the hand a few centimeters above the board. When the hand is removed, sound is stopped. The pitch of the sound will be the highest value from the predefined range when the board is lying flat. It will be lowest when the board is vertically oriented with respect to the floor. Rotating the board changes the pitch of the sound in real-time. The range is 50-1050Hz for continuous mode (mode '0') and about 523.25Hz to 1046.50Hz for musical scale mode (mode '1'). The value of currently played frequency is displayed on the OLED in Hertz. Volume may be increased or decreased in small steps using the rotary encoder (clockwise turn – increase volume). Pressing the wake-up button on the board switches the instrument mode. The current mode is displayed on the 7-segment display by means of a 0 or 1 digit (0 – continuous mode, 1 – musical scale mode).

1.3. Instrument modes

1.3.1. Continuous mode – mode ‘0’

The first mode, mode ‘0’, is continuous frequency mode. In this mode, frequency changes continuously depending on the values read out from the accelerometer. The frequency can be any integer value from the predefined range. This mode was the simple to implement. Possible discrete accelerometer values (from 0G to 1G) are mapped onto our frequency range, to cover the whole range. Only the Z axis (signed byte) is read from the accelerometer. When the board is lying flat, the value is maximum (Earth’s gravitational acceleration: 9.81 m/s^2). This value is 64 in the decimal system, half of the accelerometer positive range (full range is 2G). When the board is rotated at a steady rotational speed along one of the axes, the changes of Z values are linear (gravitational component of Z decreases linearly) so the frequency changes are also linear. It may be noted that when we shake the board upwards, the frequency gets higher and the sound played is higher than the maximum value of our range (as 1G acceleration corresponds to maximum frequency which is 1050Hz). Higher values are allowed, but low frequencies are always rounded to 50Hz. Negative accelerometer readouts are ignored.

1.3.2. Musical scale mode – mode ‘1’

The second mode, mode ‘1’ is a bit more complex than mode ‘0’. It rounds the frequencies to nearest frequencies in a pre-calculated musical scale note table. The musical scale used is twelve-tone equal temperament scale. Every pair of adjacent notes in this scale has an identical frequency ratio. Each octave is divided into 12 semitones. The frequency of a note is exactly twelfth root of two times the frequency of the previous semitone. The program precalculates these values for 13 semitones (one octave + 1) according to this formula, rounds them to integers, and puts then in an integer array. Then when an accelerometer value is read out, it is mapped to an integer frequency from the range (same as in continuous mode), but then it is rounded to the nearest value from this array.

The entire array is scanned and the algorithm tries to find the smallest difference between the given value and values from the array. It then returns the closest value from the array. This way, the instrument plays only notes from the musical scale, the same scale used in the piano and other instruments.

There is one more difference between this mode and the continuous mode. In this mode, I wanted to map the entire range of accelerometer values to one octave, from C5 to C6. Frequency of C5 is 523.25Hz, so an offset is added to make C5 the lowest possible frequency (lowest acceleration value). The fact that the equal temperament scale is a geometric series causes a slightly non-linear distribution of notes in the acceleration range.

1.4. Boards playing in unison (ZigBee)

The last piece of functionality that has been implemented is wireless communication between two boards, making them play in unison, one playing at an octave lower. A simple one-way communication protocol was devised for this purpose. Each board contains an XBee module socket which uses UART to communicate with the processor. The main board is the instrument controlled by the user and the other board lies flat and listens to ZigBee commands coming from the main board. Commands are packed in frames. Each frame has a specific start and end byte. Received frames that do not contain proper start and end bytes are discarded. There are two types of frames: commands and frequency data. A command tells the receiver board to start or stop playing (synchronized with the user blocking and unblocking light from the sensor) and the frequency data frame contains a high byte and low byte of frequency currently played by the main board. This frequency is then divided by 2 (making it exactly an octave lower) and played by the receiver board, provided that the previously received command was to start playing and that the button was set to enable playing as well. The button flag variable and remote enable flag are ANDed, so sound can be disabled either by button or by a remote “stop playing” command.

Each frame is always 4 bytes long. The possible frame byte values are shown below.

```
#define FRAME_START 0x5C

#define COMMAND_PADDING 0xC1

#define START_PLAYING 0xE2

#define STOP_PLAYING 0xA7

#define FRAME_END 0x9A
```

Example frames:

5C C1 E2 9A - start playing

5C C1 A7 9A - stop playing

5C 04 10 9A - send 1040Hz as frequency to the other board (will be divided by two to play an octave lower)

The baud rate for the communication is set as 9600bps. The frequency commands are sent once every few iterations of the main loop, as it turned out they cannot be sent as often as every iteration, as it causes framing errors and RX FIFO overflow errors at the receiver. The result is a lag between the change of frequency at the main board, and at the receiver board, which is less noticeable in the musical scale mode of the instrument.

The range of wireless communication has been tested to be about 3 meters, due to the lack of external antenna on one of the XBee modules.

2. Peripherals and interface configuration

2.1. GPIO

Various general purpose input and output pins are used in the project. The procedure for initializing the GPIO block is as follows.

First enable the AHB clock for the GPIO block:

```
LPC_SYSCON->SYSAHBCLKCTRL |= (1<<6);
```

The project uses GPIO PORT1 interrupts, this line enables them:

```
NVIC_EnableIRQ(EINT1_IRQn);
```

Next is configuration of the pins going to the audio amplifier:

```
GPIOSetDir( PORT3, 0, 1 ); //LM4811-clk
GPIOSetDir( PORT3, 1, 1 ); //LM4811-up/dn
GPIOSetDir( PORT3, 2, 1 ); //LM4811-shutdn
GPIOSetDir( PORT1, 2, 1 ); //PWM output
GPIOSetValue( PORT3, 0, 0 );  //LM4811-clk
GPIOSetValue( PORT3, 1, 0 );  //LM4811-up/dn
GPIOSetValue( PORT3, 2, 0 );  //LM4811-shutdn
LPC_IOCON->JTAG_nTRST_PIO1_2 = 0x13; //Pull-up resistor enabled,
selects function CT32B1_MAT1
```

Pins PIO3_0, PIO3_1 and PIO3_2 enable controlling of the gain of the amplifier. Clk is the clock signal, up/down is 0 or 1 depending on whether we want to increase or decrease volume, and shutdown disables the output of the amplifier. They are all configured as output pins. The last line is to configure the PIO1_2 pin in the IOCON to set it as a PWM output of Timer32B1 MAT1 (and enable pull-up resistor). This will make it change the pin's state every time the Timer reaches the match value. The PIO1_2 pin is connected to the amplifier input, and the amplifier's output is connected to the speaker. There is also low-pass filter circuitry on the path of the audio signal.

We also want to configure the GPIO interrupt for the wake-up button:

```
NVIC_EnableIRQ(EINT1_IRQn);
GPIOSetDir(PORT1,4,0); //set button as input
GPIOSetInterrupt(PORT1, 4, 0, 0, 0 ); //set interrupt from wakeup
button, PORT1_4, falling edge
GPIOIntEnable(PORT1,4); //enable PORT1_4 interrupt
```

The interrupt handler for the falling-edge GPIO interrupt:

```
void PIOINT1_IRQHandler(void) //handler for GPIO interrupt (button)
{
      if (GPIOIntStatus(PORT1,4)) //if interrupt caused by the wake-
up button
      {
            if (debounceFlag == 0) {
                  debounceFlag = 1; //disable repeated ISR calls (due
to bouncing) until timer expires
                  LPC_TMR16B0->TCR = 0x02; //reset and stop timer
                  LPC_TMR16B0->TCR = 0x01; //start timer
            }
      }
      GPIOIntClear(PORT1,4); //clear interrupt
}
```

The interrupt is set to falling-edge triggering. The button shorts the line to ground when it is pressed, so a button press should trigger a falling-edge interrupt. The handler checks if the interrupt came from the wakeup button pin, if yes then it sets the debounceFlag to disable repeated ISR calls for some time, due to bouncing. The debounce timer is started. When it matches a set value then it checks the button state. If the button is still pressed, the debounceFlag is switched to 0 again, enabling further interrupts. If not, then it means it was just bouncing that had caused the interrupt. The timer match value is set to correspond to about 80ms.
It has to be noted that for the button to work, the J28 jumper must be removed.

The last device using GPIO pins it the rotary encoder:

```
GPIOSetDir( PORT1, 0, 0 ); //set as outputs
GPIOSetDir( PORT1, 1, 0 );
```

The rotary encoder is read out at each iteration of the main loop. If there is a rotation detected, then the amplifier gain is changed using appropriate pins and simply clocking in bits.

2.2. I2C

The I2C bus is used by two devices: light sensor ISL29003 and accelerometer MMA7455. Each uses a different I2C address.

Proper registers need to be configured:

```
LPC_SYSCON->PRESETCTRL |= (0x1<<1);
LPC_SYSCON->SYSAHBCLKCTRL |= (1<<5);
LPC_IOCON->PIO0_4 &= ~0x3F;   /*  I2C I/O config */
LPC_IOCON->PIO0_4 |= 0x01;          /* I2C SCL */
LPC_IOCON->PIO0_5 &= ~0x3F;
LPC_IOCON->PIO0_5 |= 0x01;          /* I2C SDA */
```

The first line deasserts the reset signal to the I2C block.
The next line enables the AHB clock for I2C.
The last four lines configure the IOCON to enable these two pins to work as SCL and SDA for I2C.
I2C is configured as master by default.
Last step is enable I2C interrupt and I2C itself.

```
NVIC_EnableIRQ(I2C_IRQn);
LPC_I2C->CONSET = I2CONSET_I2EN;
```

2.3. SPI

SPI (SSP) is used by the shift register controlling the 7-segment display (one digit) and it is also used by the OLED display.

SYSCON and IOCON configuration for SPI:

```
LPC_SYSCON->PRESETCTRL |= (0x1<<0);
LPC_SYSCON->SYSAHBCLKCTRL |= (1<<11);
LPC_SYSCON->SSPCLKDIV = 0x02;          /* Divided by 2 */
LPC_IOCON->PIO0_8           &= ~0x07;    /*  SSP I/O config */
LPC_IOCON->PIO0_8           |= 0x01;          /* SSP MISO */
LPC_IOCON->PIO0_9           &= ~0x07;
LPC_IOCON->PIO0_9           |= 0x01;          /* SSP MOSI */
```

The first line deasserts the reset signal to the SPI block. The next line enables the AHB clock for SPI. The third line makes the SPI clock half the PCLK clock. The last four lines set the IOCON to use two pins as MISO and MOSI for SPI.

The following lines select the SCK pin that will be used and enables it as SCK.

```
LPC_IOCON->SCKLOC = 0x01;
LPC_IOCON->PIO2_11 = 0x01;/* P2.11 function 1 is SSP CLK
```

The following lines select a GPIO pin as chip select

```
LPC_IOCON->PIO0_2 &= ~0x07;        /* SSP SSEL is a GPIO pin */
/* port0, bit 2 is set to GPIO output and high */
GPIOSetDir( PORT0, 2, 1 );
GPIOSetValue( PORT0, 2, 1 );
```

We need to configure the format of the SPI communication:

```
/* Set DSS data to 8-bit, Frame format SPI, CPOL = 0, CPHA = 0, and
SCR is 15 */
LPC_SSP->CR0 = 0x0707;
```

Clock is low between frames and data is latched on first clock transition. Frames are 8-bit.

Set clock prescaler, master mode.

```
/* SSPCPSR clock prescale register, master mode, minimum divisor is
0x02 */
LPC_SSP->CPSR = 0x2;

/* Enable the SSP Interrupt */
NVIC_EnableIRQ(SSP_IRQn);

/* Device select as master, SSP Enabled */
/* Master mode */
LPC_SSP->CR1 = SSPCR1_SSE;
/* Set SSPINMS registers to enable interrupts */
/* enable all error related interrupts */
LPC_SSP->IMSC = SSPIMSC_RORIM | SSPIMSC_RTIM;
```

2.4. UART (ZigBee)

UART is used to talk to XBee modules. The XBee modules have default configuration and act as a wireless extension of UART. In this state, only point-to-point communication can be achieved. Baud rate is set to default which is 9600bps.

Initial UART configuration:

```
NVIC_DisableIRQ(UART_IRQn);

LPC_IOCON->PIO1_6 &= ~0x07;    /*  UART I/O config */
LPC_IOCON->PIO1_6 |= 0x01;     /* UART RXD */
LPC_IOCON->PIO1_7 &= ~0x07;
LPC_IOCON->PIO1_7 |= 0x01;     /* UART TXD */
/* Enable UART clock */
LPC_SYSCON->SYSAHBCLKCTRL |= (1<<12);
LPC_SYSCON->UARTCLKDIV = 0x1;     /* divided by 1 */

LPC_UART->LCR = 0x83;             /* 8 bits, no Parity, 1 Stop bit */
Fdiv = (((SystemFrequency/LPC_SYSCON-
>SYSAHBCLKDIV)/regVal)/16)/baudrate ;    /*baud rate */

LPC_UART->DLM = Fdiv / 256;
LPC_UART->DLL = Fdiv % 256;
LPC_UART->LCR = 0x03;          /* DLAB = 0 */
LPC_UART->FCR = 0x07;          /* Enable and reset TX and RX FIFO. */
NVIC_EnableIRQ(UART_IRQn);
```

```
LPC_UART->IER = IER_RBR | IER_THRE | IER_RLS; /* Enable UART RX and
TX interrupt */
```

UART is configured the same way on both boards. The default communication format is 8bits, no parity, 1 stop bit and no flow control.
One important remark is that the UART pins from the microcontroller are multiplexed by an external multiplexer chip. By default the UART signals are forwarded to the UART to USB chip, for communication with a computer. To route the signals to the XBee module, we need to remove jumper B from J7.

2.5. Accelerometer

The accelerometer used is a three-axis digital output accelerometer which uses I2C. We only use one axis – axis Z, because it is the easiest option as it always has some acceleration (full of partial component of gravitational acceleration pointing downwards) without the need to shake it. For readouts, we use only register ZOUT8 which contains a signed 2's complement 8-bit value of the acceleration (maximum is 2G). 0x00 is 0G, 0x7F is 2G and 0x80 is -2G. Default configuration is 2G range and MEASURE mode. For our purposes, there is no need for additional configuration of the accelerometer.

The accelerometer value is read every iteration of the main loop. The readouts from the accelerometer oscillate very rapidly, despite the built-in low-pass filter. When the board was lying still or held still, the sound would still vary its pitch very fast. To correct this, I have implemented a low-pass filter for accelerometer readouts. To design this filter, I used an online coefficient calculator (link here: http://www-users.cs.york.ac.uk/~fisher/mkfilter/trad.html). The website requires the user to input type of filter (Butterworth/Bessel/Chebyshev), order of the filter, cut-off frequency and sample rate. It then returns ready C code for the filter with coefficient values and gain value. These values are all floating-point values, but still the calculations do not decrease the operation speed of the program noticeably, despite lack of FPU on the processor. I chose a third order Butterworth filter. The cut-off frequency required some tweaking. I settled for 3Hz, but it is only 3Hz with respect to 150Hz sample rate that I have put in the form. This sample rate was a wild guess, in reality I think the cut-off frequency may be around 8-10Hz. To determine the sample rate, the time of execution of one iteration of the main loop would have to be measured, but it couldn't be a good measurement because not every iteration takes the same number of cycles and also any further evolution of the main loop would result in the change of the sample rate. I decided not to measure it because the cut-off frequency was already determined by trial and error and the 3Hz value is sufficient.

The algorithm for converting the accelerometer value to a frequency is as follows.

We need two multipliers for two instrument modes, and a reference zFlat value, which is the value read out in the beginning of the program when the board is assumed to be lying flat with the highest z value possible.

```
acc_read(&x, &y, &z); //initial accelerometer readout
zFlat = z; //zFlat is z when board is lying flat parallel to ground
continuousMultiplier = MAX_FREQUENCY_CONTINUOUS/zFlat; //multiplier
to get full sound range between 0-1000hz
scaleMultiplier = MAX_FREQUENCY_SCALE/zFlat; //multiplier for scale
mode (with offset)
```

Before the main loop, we read z, assign it to zFlat (treating it as highest possible z value) and calculate the multipliers. Z axis accelerometer readouts in the main loop will be multiplied by a multiplier for the proper mode and this will get us the frequency to be played. We assign MAX_FREQUENCY_CONTINUOUS/zFlat to the continuous multiplier. In effect we calculate the z/zFlat ratio and multiply it by the maximum frequency that we want(in this case 1050Hz).
For the musical scale mode it is a bit different, we multiply the ratio by MAX_FREQUENCY_SCALE which is MAX_FREQUENCY_CONTINUOUS-OFFSET. We need to subtract that offset because we will be adding it later to the readouts, to get 510Hz (offset) for acceleration = 0, resulting in a range from 510Hz to 1050Hz.

In the main loop, we read the z value, and if it is greater than 0 then we check the instrument mode. For continuous mode we take z*continuousMultiplier as an input frequency to the low-pass filter, and then take the return value of the filter as the frequency to be played. For musical scale mode, we take z*scaleMultiplier, add OFFSET to it, pass it to the filter, take the return value and round it to the closest musical scale frequency. Lastly we save it into the global variable holding the frequency to be played.

```
acc_read(&x, &y, &z); //read accelerometer
if (z>0) //we consider only positive values of z
{
      if (playingMode == 0)
            frequency = (uint32_t)(filterAcc(
                        (float)z*continuousMultiplier)
                        );
                  //frequency will usually be between 0 and 1000Hz
      else
            frequency = roundToMusicalScale(
                        (uint32_t)filterAcc(
                              OFFSET + (float)z*scaleMultiplier
                              )
                        );
                  //frequency is clipped to musical scale
```

2.6. Light Sensor

The light sensor on the board has high sensitivity, selectable gain and range and a 16-bit ADC. It uses I2C for communication. Before the while loop, the command and control registers of the sensor are set via I2C. In the command register we enable the ADC, and we set the ADC work mode to “Difference between diodes (I1 - I2) to signed 15-bit data”. The sensor contains two photodiodes. Diode1 is sensitive to both visible and infrared light, while diode2 is mostly sensitive to infrared light. By taking the difference we should eliminate most of the infrared light influence. Lastly we set 8bit resolution of the ADC because we do not need a high resolution, just enough to detect a major drop or rise in light intensity. In the control register, we set the range to 0 to 1000 lux.

In the main loop, we first read register 4 of the sensor (low byte of light level readout) and then check if this light level is below or above some threshold. The threshold is set to 0x12 for hand hovering and this value was discovered by trial and error. Too high threshold creates false positives, while too low a threshold makes you block the light coming to the sensor completely.

I decided to implement a low-pass filter for the light level readouts, just like in case of the accelerometer, but with a cut-off frequency of 10Hz (with respect to 150Hz sample rate). This helped with the oscillations and false positives that happened before.

When the light level is below the threshold and previous readout was above the threshold, it means that a hand or some object is obstructing the light and we need to enable sound output (precisely: start the timer for PWM), send a ZigBee command to the other board to start playing, and change the isPlaying flag. When the light level is above the threshold, we do the opposite.

2.7. PWM timer

A 32-bit timer of the microcontroller is used for PWM output to generate square wave sound. The duty cycle is 50% and needs not be changed. Two match values are used, MR1 and MR3. Match channel 1 is the duty cycle value, and match channel 3 is the PWM cycle value (period of the square wave). Every match on channel 3 generates an interrupt. The register configuration is below:

```
void initPWMTimer() //timer for square wave generation
{
	LPC_SYSCON->SYSAHBCLKCTRL |= (1<<10); //enable clock for Timer1
	LPC_TMR32B1->PR  = 0; //prescaler 0
	LPC_TMR32B1->IR  = 0xff; //reset interrupts
	LPC_TMR32B1->MR1 = 0;    //initial duty cycle value
	LPC_TMR32B1->MR3 = 0; 	//initial pwm cycle value
	LPC_TMR32B1->TCR = 0x2;  //reset timer
	//LPC_TMR32B1->MCR = (1<<10); //reset on match MR3 (pwm cycle)
	LPC_TMR32B1->MCR = (1<<9); //interrupt on match MR3 (pwm cycle)
	LPC_TMR32B1->CTCR = 0x0; // timer mode
	LPC_TMR32B1->PWMC = 0x2; //PWM enable for MAT1
	//USE MATCH CHANNEL 3 TO SET PWM CYCLE
	NVIC_EnableIRQ(TIMER_32_1_IRQn); //enable interrupt
	LPC_TMR32B1->TCR = 0x0; //release timer reset
	}
```

The timer is started and stopped according to the light level. In the ISR of the timer we have the following sequence of operations. First of all we check if it is the interrupt from MR3 (end of PWM cycle). If it is so, then we check if the most recently set frequency (global variable) is higher than 50. If so, then we use sprintf to convert the value of this frequency to a string. This string will later be passed in the main loop to the OLED printing function, which was moved there because it is not time critical so it should not be in a frequently triggered interrupt.

Most importantly, though, we set the new duty cycle and PWM cycle frequency of the timer by calling setPWM(frequency) function. For converting the frequency value to number of ticks, 72MHz clock is assumed. The duty cycle tick value is always half of the PWM cycle value. After setPWM(frequency) we restart the timer and leave the ISR. In effect, we have a PWM output which can change its frequency after each cycle, assuming that the frequency variable has been changed in the main loop since the last PWM cycle.

2.8. Button debounce timer

A 16-bit timer is used for the purpose of debouncing the button. It makes a small delay before re-enabling the button interrupt (more precisely: re-enabling the possibility to toggle the instrument mode). The initialization is below:

```
void initDebounceTimer() //timer for button debouncing delay
{
	LPC_SYSCON->SYSAHBCLKCTRL |= (1<<7); //enable clock for timer
	LPC_TMR16B0->PR  = 2048; //prescaler
	LPC_TMR16B0->IR  = 0xff; //reset interrupts
	LPC_TMR16B0->MR0 = 2812;
	//initial value around 80ms (2812*2048 ticks)
	LPC_TMR16B0->TCR = 0x2;  //reset timer
	LPC_TMR16B0->MCR = (1<<0); //interrupt on match MR0
	LPC_TMR16B0->CTCR = 0x0; //timer mode
	NVIC_EnableIRQ(TIMER_16_0_IRQn); //enable timer interrupt
	LPC_TMR16B0->TCR = 0x0; //release timer reset
}
```

The prescaler times the MR0 value gives us enough ticks to last around 80ms, this value was chosen experimentally. 80ms provides a delay much longer than the bouncing duration of the button.

In the GPIO interrupt for the button, we start the debounce timer and when it matches the MR0 value then we check the status of the button pin. If it is still pressed (still 0) then we allow to toggle instrument mode and also change the value on the 7-segment display. Otherwise, we just reset the timer and exit the ISR.

3. Failure Mode and Effect Analysis

The realized project depends on some devices for proper operation. Some pieces of functionality are an optional addition to the project and their failure would not render the project unusable. However, there are some scenarios in which the project would simply cease to work and would become useless. In the table below there are a few items and functions which will be considered in failure analysis. In the left column there is the item name, and in the right column the gravity of this item's failure.

Item/function	**Severity**
Microcontroller	Critical
Power supply	Critical
Speaker	Critical
Accelerometer	Critical
Light Sensor	Critical
Button	Medium
Rotary encoder	Negligible
OLED display	Negligible
7-segment display	Negligible
XBee	Medium

The project heavily depends on proper operation of the microcontroller, power supply and devices such as speaker, accelerometer and light sensor. The speaker (along with the amplifier) and the light sensor are critical because otherwise no sound can be played. The accelerometer is necessary as well to set the frequency of the played sound. If readouts from the light sensor or accelerometer become corrupt, the program would show undefined behavior. The power supply must also be present because there is no backup power and no battery installed. As for other functionalities, devices such as XBee and the button are not critical but contribute greatly to the project. The functionality would be severely weakened without those two elements. Lastly, there are devices such as rotary encoder, OLED display, and 7-segment display which provide just an additional non-critical functionality such as displaying frequency, mode and changing the volume.

As for detecting system failure, a problem with the power supply may become apparent when observing LED diodes, when they blink when they should not, or do not light up at all, this might indicate a problem with the power supply or cable. Power-related faults may also arise when a short in the system is caused, for example from power rail to ground. This may happen either by putting a jumper in the incorrect place, or dropping a piece of metal on the board.

A microcontroller failure would probably be easy to detect, as most likely it would be a complete failure of the chip, not just a small part of the chip.

As for the speaker, it could probably be very easily replaced in case of failure. It would be harder to replace the amplifier but with proper equipment it could be done.

As for the accelerometer and light sensor, their failure would probably be very apparent in this project, as they are critical components of the system. It would be difficult to resolder the chips, but not impossible.

A failure concerning the XBee module would also be apparent because it would probably cause cease of the communication between boards. Replacing the module is easy as there is a standardized socket on the board.

The 7-segment display, button and rotary encoder can probably be very easily replaced, but their functioning is not critical to the operation of the system.

If the OLED display failed, cracked, or started to display random pixels or spots, it would be expensive to buy a new one and replace it, but it contributes so little to the project that I do not think it would be necessary.

4. References

http://www-users.cs.york.ac.uk/~fisher/mkfilter/trad.html

http://www.phy.mtu.edu/~suits/notefreqs.html

http://www.intersil.com/content/dam/Intersil/documents/fn74/fn7464.pdf

http://gadgetgangster.com/scripts/displayasset.php?id=431

http://www.zsk.p.lodz.pl/~morawski/SCR&ES/Guides&Schematics/LPCXpresso_Base_Board_revA.pdf

http://www.zsk.p.lodz.pl/~morawski/SCR&ES/Guides&Schematics/LPCXpresso_BaseBoard_rev_B_Users_Guide_Rev_A.pdf

http://www.nxp.com/documents/user_manual/UM10375.pdf

5. Other information

The project uses two boards, and two programs. I did not explain the operation of the second program (for the receiver board) because its only features are waiting for ZigBee commands, disassembling received frames, changing the volume and displaying frequency on OLED display just like the main program. The button enables disabling the sound from the receiver board.

I could not enclose proof of the operation of the program as it is a sound project.

Below is the schedule for MTC and Final evaluation:

FINAL:
1. Playing square wave sound by blocking light from the light sensor
2. Sound frequency is dependent on accelerometer reading (rotating the board changes pitch)
3. OLED displays currently played frequency in Hz
4. Zigbee communication between another board which will play sound simultaneously, but an octave lower
5. Button switches mode between continous pitch and musical scale

MTC:
light sensor, OLED, accelerometer+sound