Atmel AVR 2004 Design Contest – Entry # A3182

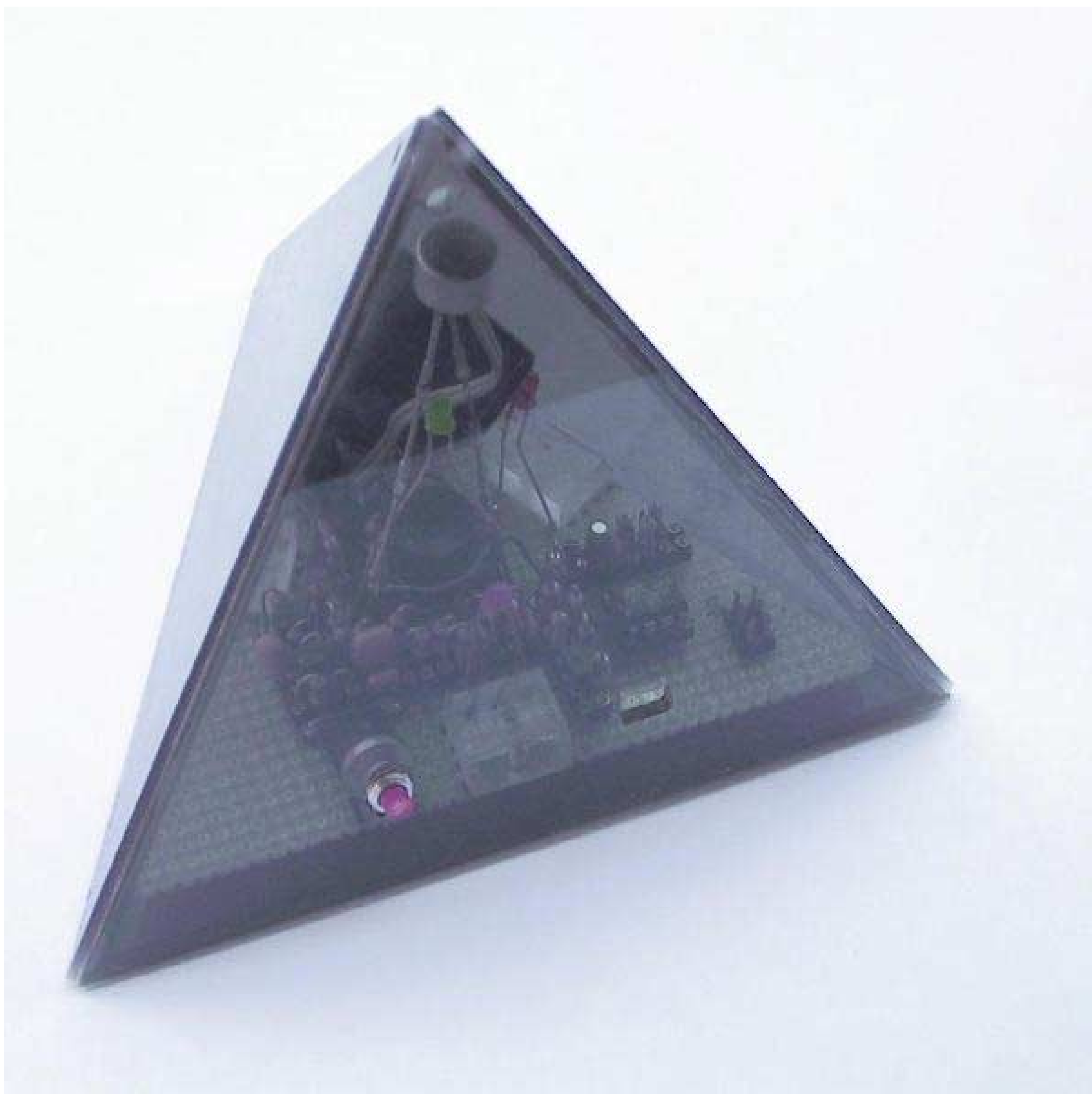
Sing-Along Game

Sing-along is a musical tone memory game for 1 to 3 players. The game plays a random sequence of notes, then it listens for the player(s) to sing or hum the notes back to it. The sequence keeps getting longer each time the player sings it correctly.

If a player does not sing a note within the allotted time, or sings an incorrect note, then a 'lose' sound will play. If a player correctly sings the entire 16-note sequence, then a 'win' tune will automatically play. In addition, the game has several LEDs to indicate which player's turn is in progress, and to follow the notes as they are played or sung.

The hardware is built around an Atmel ATtiny12, with its analog comparator fed by an analog front end which processes the input from an electret microphone. An automatic gain control loop allows the MCU to detect inputs over a wide range of sound levels. A speaker, voltage regulator, pushbutton switch, and four LEDs round out the design. The MCU has only 5 I/O pins available, so several of them are used for multiple functions. The system is powered by a single 9V battery, which is fed through a transistor so power can be removed from the analog circuitry during sleep mode.

The software was written in assembler, and is relatively straightforward. A set of routines keeps track of the tone sequences for each player, and saves them in a set of registers. At the beginning of each turn, the playback routine plays the entire sequence through the speaker. As the sequence gets longer, the playback speed increases, making the game increasingly more difficult. Then the recognition routine is called, which listens for tones sung into the microphone. It times consecutive zero crossings to determine the waveform period, then repeatedly divides the period in half to bring the value within a specified range. This makes the result independent of the octave that was actually sung, since the frequency of adjacent octaves differ by powers of two.



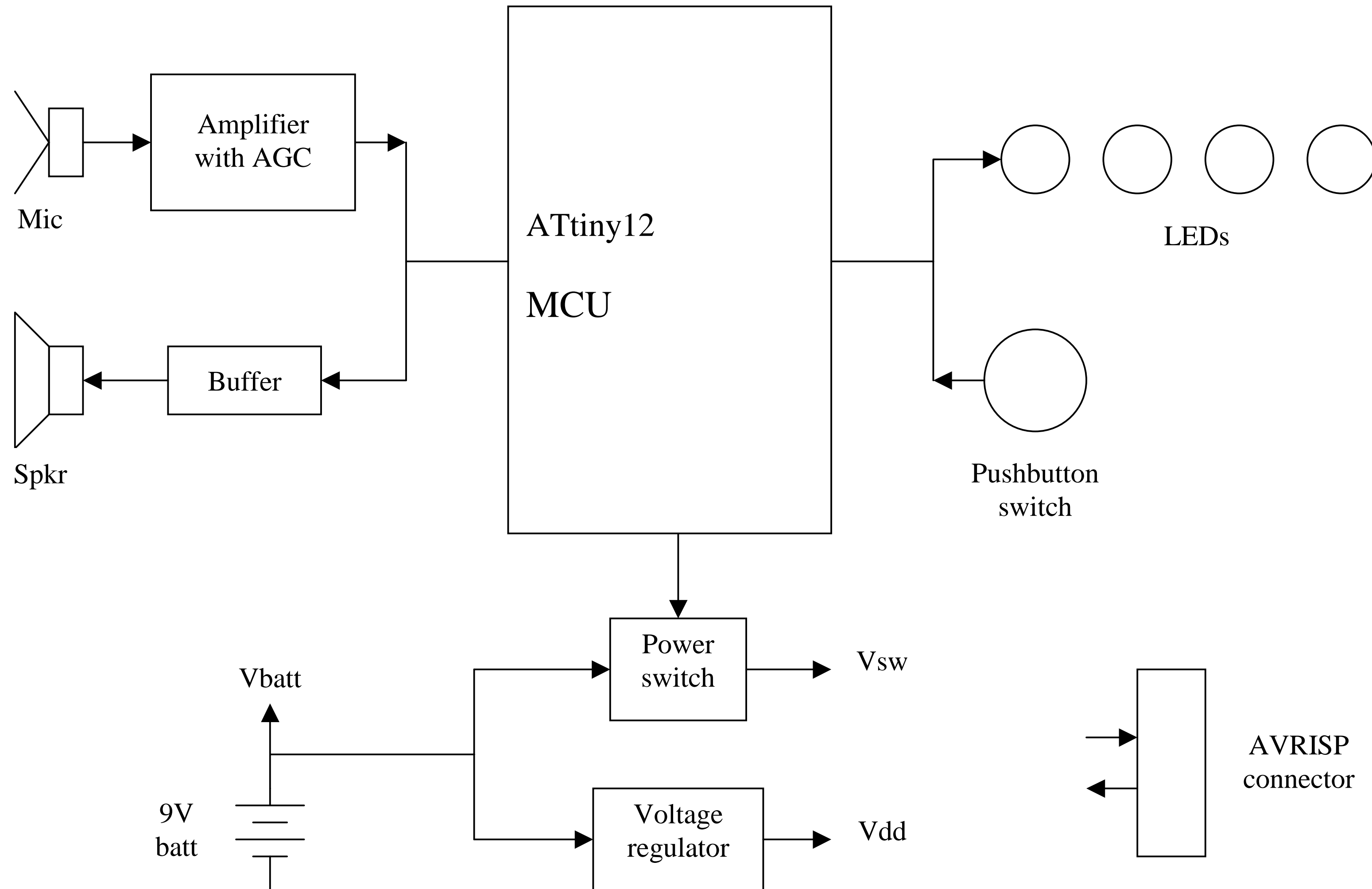
```
GetCycl:
;
;  GetCycl waits for a signal from the microphone input.
;
;     In:   none
;     Out:  A = valid tone value
;           Carry Clear Timed out, no signal
;           Carry Set   Signal detected:
;             Zero Clear        Signal valid
;             Zero Set    Signal invalid
;
;     Uses: A,B,Z, stack: 1 level
;

      ldi   Count,DEBMAX*1000/1707  ;Init max debounce time
      ldi   DebCnt,DEBCYCL    ;Init debounce counter
      clr   DebVal            ;Clear debounced value

      out   TCCR0,DebVal      ;Stop timer

      out   TCNT0,DebVal      ;Init timer value

      ldi   Acc,1<<CS01
      out   TCCR0,Acc   ;Start timer, clock = CK/8

      ldi   Acc,1<<TOV0
      out   TIFR,Acc    ;Clear and
      out   TIMSK,Acc   ; enable timer overflow interrupt

GetCLp:
;
;  Wait for a positive edge on the microphone input, or a timeout
;  of the timer high byte count (CK/8/256/DEBMAX cycles = 1707/DEBMAX usec)
;
;  At each input edge, the timer is read (after stopping
;  it to assure consistency between it and the overflow count.)
;  Then it is restarted with a count value of 1 (= 8 CK cycles),
;  to account for the 8 cycles that it was stopped while reading it.
;
      wdr               ;Reset watchdog timer

      clr   AccZ        ;Init time high byte (timer overflow count)
      ldi   Acc,DEBMAX*1000/1707    ;Init high byte timeout value for int
routine
      sec               ;Clear timeout flag

      sei               ;Enable global interrupts
;
;  The timer overflow interrupt routine doesn't save or restore the
;  status flags, but sets them when it increments the timer high byte
counter.
;  That means we can test them here to check for a timeout condition (when
;  AccZ counts up to Acc and the C flag is cleared in the interrupt
routine),
;  but can't use any status flags for anything else while ints are enabled.
;
      sbi   ACSR,ACI    ;Clear comparator int flag
```

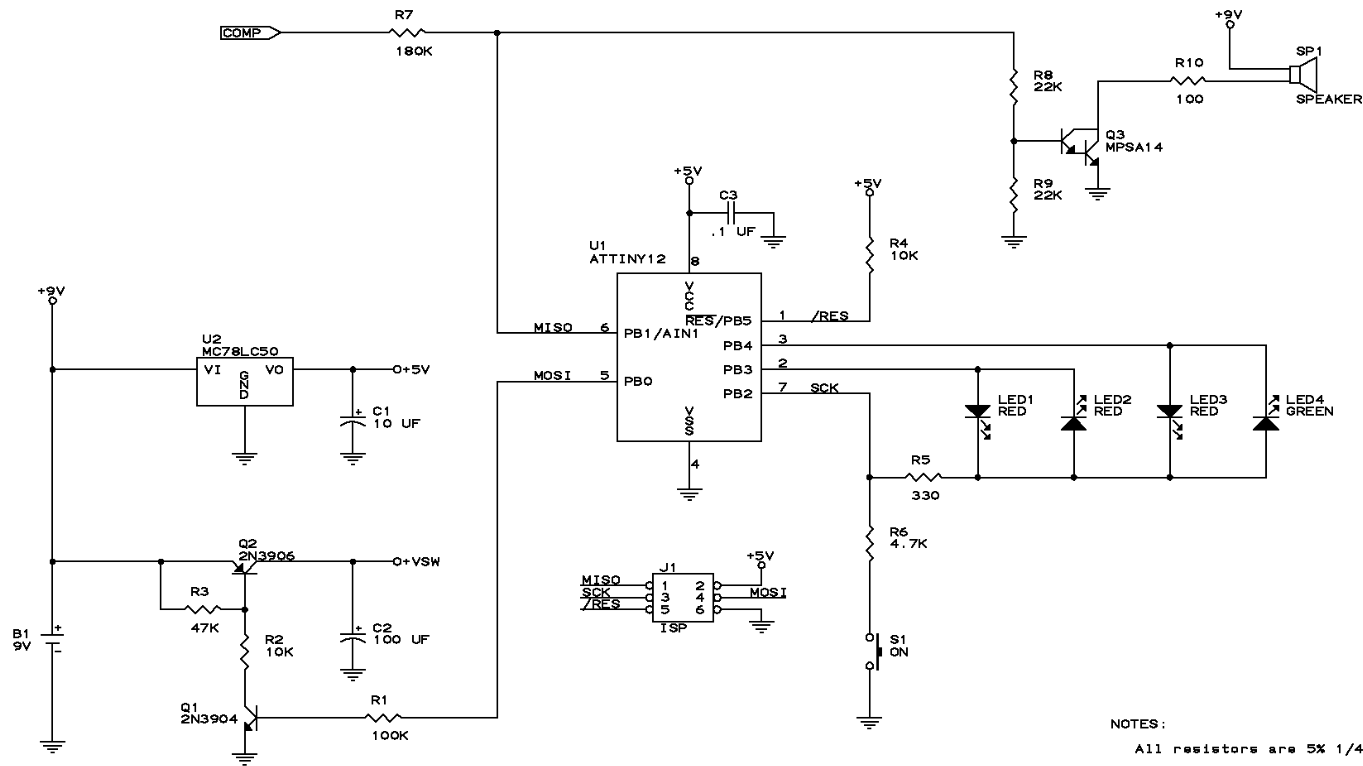
```
MicLp:      sbis  ACSR,ACI    ;Exit loop if comparator pos. edge detected
      brcs  MicLp       ;Loop until timeout (= C cleared by int. routine)

      ldi   AccB,0
      out   TCCR0,AccB  ;Stop timer

      nop               ;1~ ;Give any pending interrupt time to occur
      cli               ;1~ ;Disable global interrupts

      brcc  MicTout           ;1~ ;Timeout occurred earlier

      in    AccB,TCNT0  ;1~ ;Get timer value

      ldi   Acc,1       ;1~
      out   TCNT0,Acc   ;1~ ;Re-init timer value

      ldi   Acc,1<<CS01 ;1~
      out   TCCR0,Acc   ;1~ ;Restart timer
;
;  We leave interrupts disabled for now, which means a timer overflow
;  won't be seen until we re-enable them later. This isn't a problem,
;  since the first overflow won't occur for another 255*8 CK cycles,
;  which is much longer than the time to execute the following code
;  and loop back around to GetCLp (above), where ints are enabled again.
;


;
;  The timer value in AccZ:AccB is now normalized to be less than 160,
;  which is slightly higher than the highest valid value.
;
;  Since we don't care which octave was detected,
;  and octaves differ by powers of 2, we simply right shift
;  the 2-byte time value until it's in the correct range.
;
      mov   Acc,AccZ    ;Get time high byte
      rjmp  MicSh2
MicSh1:
      lsr   Acc         ;Shift high byte
      ror   AccB        ; into low byte
MicSh2:
      tst   Acc
      brne  MicSh1
      cpi   AccB,160    ;Value >= 160?
      brsh  MicSh1            ;yes, shift again

;
;  Now compare the time to the ranges for each valid tone.
;
      ldi   Acc,T_HI
      cpi   AccB,96-6
      brlo  MInv
      cpi   AccB,96+6+1 ;Valid high tone?
      brlo  MCheck            ;yes

      ldi   Acc,T_MID
```

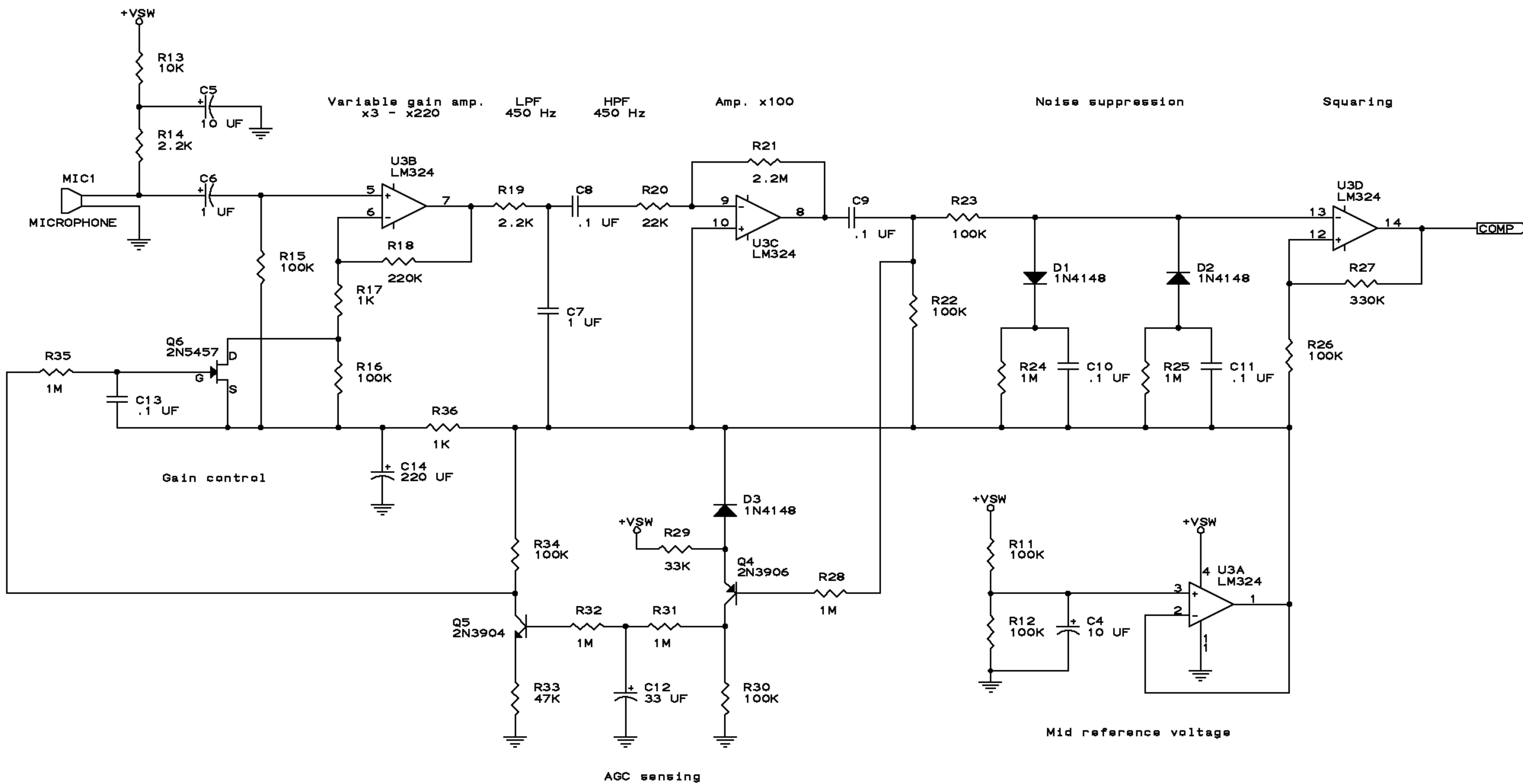
```
      cpi   AccB,113-7
      brlo  MInv
      cpi   AccB,113+7+1      ;Valid mid tone?
      brlo  MCheck            ;yes

      ldi   Acc,T_LO
      cpi   AccB,143-9
      brlo  MInv
      cpi   AccB,143+9+1      ;Valid low tone?
      brlo  MCheck            ;yes

;
;  Update the debounce variables
;
MInv: clr   DebVal            ;Clear debounced value
      rjmp  MCont

MCheck:     cp    Acc,DebVal  ;Same value as last cycle?
      breq  MChk1       ;yes

      mov   DebVal,Acc  ;Save new value
      ldi   DebCnt,DEBCYCL    ;Init debounce counter

MChk1:      dec   DebCnt            ;Debounced for enough cycles?
      breq  MicOk       ;yes

MCont:      tst   AccZ        ;Timer high byte = 0?
      brne  MCont1            ;no
      inc   AccZ        ;yes, set to 1
MCont1:     sub   Count,AccZ  ;Decr wait counter by timer/256
      brcs  MicBad            ;Waited too long

      rjmp  GetCLp            ;Go back for another sample


MicTout:
      clc               ;Return with carry clear
      rjmp  MicRet

MicBad:
      sez               ;Return with zero set and carry set
      sec
      rjmp  MicRet

MicOk:
      clz               ;Return with zero clear and carry set
      sec

MicRet:
      ldi   AccB,0
      out   TCCR0,AccB  ;Stop timer
      out   TIMSK,AccB  ;Disable timer interrupt

      rjmp  GetCRet           ;Jump back to calling routine
```

Block Diagram

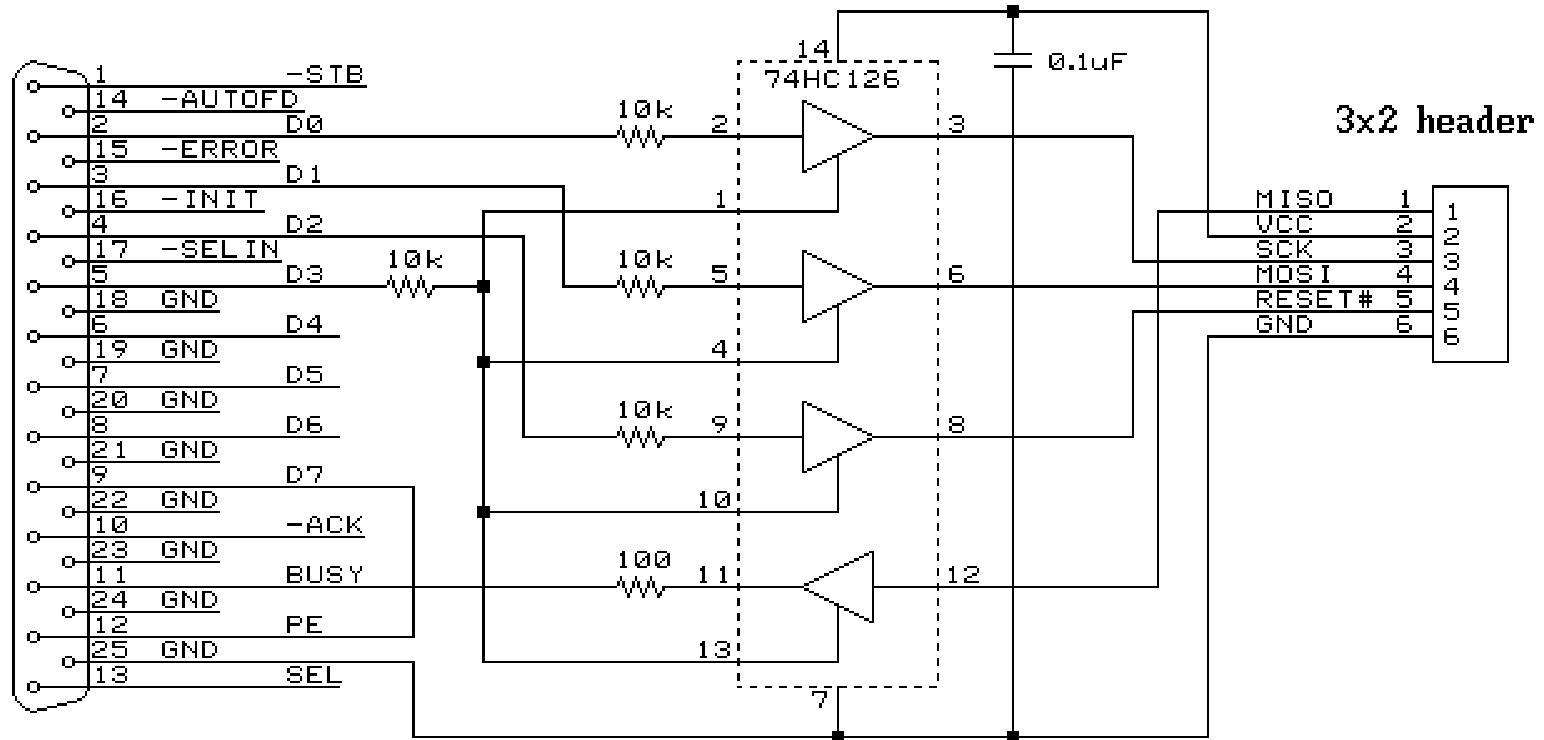
Atmel AVR 2004 Design Contest – Entry # A3182




COMP
R7
180K
R8
22K
R9
22K
Q3
MPSA14
+9V
R10
100
SP1
SPEAKER
+5V
C3
.1 UF
+5V
R4
10K
U1
ATTINY12
8
VCC
RES/PB5
1
/RES
MISO
6
PB1/AIN1
PB4
3
MOSI
5
PB0
PB3
2
PB2
7
SCK
VSS
4
LED1
RED
LED2
RED
LED3
RED
LED4
GREEN
R5
330
R6
4.7K
S1
ON
+9V
U2
MC78LC50
VI
GND
VO
+5V
C1
10 UF
Q2
2N3906
+VSW
R3
47K
R2
10K
C2
100 UF
B1
9V
Q1
2N3904
R1
100K
J1
MISO
SCK
/RES
1
3
5
2
4
6
MOSI
+5V
ISP
NOTES:
All resistors are 5% 1/4W unless specified
All capacitors are 20% 10V unless specified
Atmel AVR 2004 Design Contest
Title
Sing-Along Game
Size
B
Document Number
Entry # A3182
REV
1
Date: June 27, 2004
Sheet 1 of 2



Atmel AVR 2004 Design Contest			
Title	Sing-Along Game		
Size	Document Number		REV
B	Entry # A3182		1
Date:	June 27, 2004	Sheet 2 of 2	



Programming Adapter