Digitala Projekt

Chip Quiz

Projektmedlemmar: Olov Nordenstam och Linus Hägerbrand

Grupp 9

050516

Abstract

We have accomplished a project in the course Digital Systems, Project Laboratory. We decided to construct a quiz game for two competitors. The project was carried out in four parts. The first step was to create a requirements specification, followed by designing the circuit. The final two steps were the wiring and programming parts. We succeeded in creating an enjoying gaming experience that fulfilled most of our requirements.

Inledning

Systemet vi valt att konstruera är ett frågesportspel med en display som visar frågor och svarsalternativ. Två spelare tävlar mot varandra och svarar på frågorna genom att trycka ned en knapp som motsvarar det svarsalternativ denne väljer. Varje spelare har tre knappar var.

Kravspecifikation

Nedan följer de krav vi ställde på systemet innan vi började konstruera.

Krav 1. Då en fråga ställs ska texten visas upp på displayen i fem sekunder.

Krav 2. Efter att frågetexten har visats skall tre svarsalternativ presenteras på displayen.

Krav 3. När svarsalternativen visas upp ska det vara möjligt att svara genom att trycka ned motsvarande svarsknapp.

Krav 4. Systemet registrerar den först nedtryckta svarsknappen och ger motsvarande spelare en poäng om det är rätt svar och en minuspoäng om det är fel svar.

Krav 5. Om ingen spelare svarar inom fem sekunder så delas ingen poäng ut och en ny fråga visas upp.

Krav 6. Innan en ny fråga ställs så visas ställningen upp på displayen.

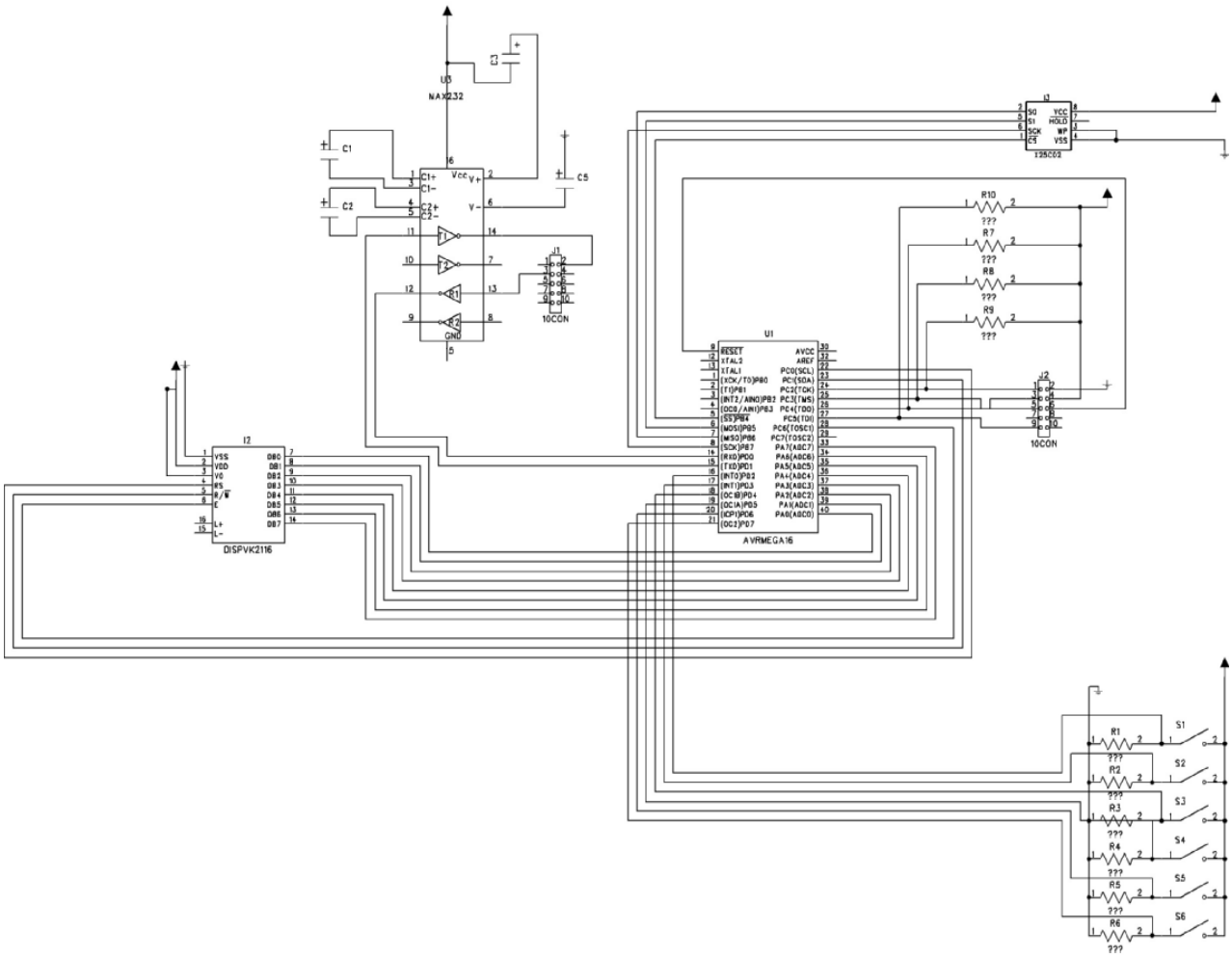
Krav 7. Då en speciell nollställningsknapp trycks ned så startas spelet om.

Krav 8. Via ett seriellportsgränssnitt kan nya frågor laddas ner från en PC till systemets databas.

Hårdvara

Vårt spel Chip Quiz är baserat på en AVR Mega16-processor och är byggt enligt figur 1. Information till användaren presenteras på en display av typen Optrex DMC40457-LY. För att spara all data som tillhör frågorna använder vi ett externt EEPROM-minne av typen Xicor X25128 med vilket vi kommunicerar via processorns SPI-interface. För att kunna kommunicera med användarna har vi sex knappar vilka ger 0V om knappen är upptryckt och 5V om den är nedtryckt. Nedladdning av programvara och debugging sköts via ett J-Tag-inteface. Dessutom finns en RS232-krets inkopplad för kommunikation med serieporten på en PC. Tyvärr stöds inte den sistnämnda funktionen av mjukvaran

.



Figur 1 – Kopplingsschema över Chip Quiz

Mjukvara

Vår mjukvara består av två program som återfinns i Appendix A. Dels ett som utgör själva spelet och dels ett annat som fyller EEPROM-minnet med frågedata. Det förstnämnda programmet utför spellogiken enligt följande:

1. Hämtar en slumpmässig fråga ur minnet
2. Visar frågan
3. Visar svarsalternativ
4. Väntar på att alla knappar är upptryckta

5. Väntar på att en knapp trycks ner
6. Kontrollerar om nedtryckt knapp motsvarar rätt alternativ
7. Om alternativet var rätt addera en poäng till aktuell spelare annars subtrahera
8. Visa aktuell ställning
9. Om inte alla frågor visats gå till punkt 1
10. Visa vinnare och starta ny spelomgång

På grund av tidsbrist så hann vi inte implementera den koppling mellan kretsen och PC:n som vi hade planerat. Denna skulle bestå av en serieportsförbindelse med vilken man skulle kunna ladda ner frågor i kretsens EEPROM. Istället har vi skrivit ett speciellt program som laddar frågor från programminnet till EEPROM-minnet.

Resultat

Vårt arbete med kretsen har tagit tid eftersom vi är ovana hårdvarukonstruktörer. Vi har dock hunnit få med all tänkt funktionalitet förutom en resetknapp och PC-kopplingen.
Designen och den första kopplingsfasen gick lite trögt till att börja med. När vi senare hade blivit bekanta med komponenternas datablad lossnade det.
Även i kodningsstadiet stötte vi på några mindre svårigheter som vi var tvungna att brottas med. Det kunde röra sig om exempelvis en utelämnad bit i ett register, glapp i kopplingen eller dylikt.
När vi tvingades ta till lösningen att ladda ner frågor via ett speciellt laddningsprogram så råkade vi placera frågedatan i internminnet som då inte räckte till vilket resulterade i konstiga utskrifter. Genom att studera AVR-information på webben så fann vi hur man genom att definiera sin data på ett speciellt vis kunde placera denna i programminnet.
Överlag är vi dock nöjda med vår konstruktion som erbjudit många trevliga tävlingsstunder.

Förbättringsförslag

Den mest självklara förbättringen som skulle kunna göras är att implementera den funktionalitet för att kunna ladda in nya frågor från en PC som vi inte hann med. Vi skulle även vilja förbättra knappkonstruktionen. Knapparna borde vara stabilt monterade och separerade från kopplingsplattan eftersom man i stridens hetta lätt orsakar glapp när man trycker ner knapparna för hårt.
Koden skulle vi vilja förbättra strukturen på eftersom den blev lidande på grund av tidsbrist. Det fulaste är att vi blev tvungna att duplicera kod när vi skapade laddningsprogrammet. Om koden hade varit uppdelad i moduler hade vi sluppit detta problem.

Slutsatser

Kursen har varit väldigt givande. Tidigare har vi inte arbetat hårdvarunära i den omfattningen som vi gjort i denna kurs. Det var just den erfarenheten som vi hoppades på att få när vi valde att läsa Digitala projekt.
De problem som vi har stött på har gett oss insikter om svårigheterna som finns vid konstruktion av denna typ av system.

Att vi själva fick välja vad vi ville konstruera har vi upplevt som mycket positivt. Dels känns det roligare att konstruera någonting som känns angeläget och dels har det förbättrat vår problemlösarförmåga genom att vi på egen hand fick designa lösningen.

Appendix A

Game.c

```
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>

#define NR_QUESTIONS 29
#define QUESTION_TEXT_LENGTH 160
#define ANSWER_TEXT_LENGTH 90
#define MEM_CHUNK_SIZE 10

typedef unsigned char byte;
typedef struct{
      char text[QUESTION_TEXT_LENGTH];
      char answers[ANSWER_TEXT_LENGTH];
      byte correct_answer;
}question_data;

byte p1_score, p2_score;
byte done_questions[NR_QUESTIONS];
char result_str[100];
char tmp_buf[80];
question_data data;

void display_cmd(byte cmd)
{
      byte i, j;
    DDRA = 0xff; //set PORT A for out
      PORTA = cmd; //put the command on the port
      // RS R/W = 0
      DDRC = 0xff;
      cbi(PORTC, 0);
      cbi(PORTC, 1);
      // enable = 1
      sbi(PORTC, 6);
      // enable = 0
      cbi(PORTC, 6);
      // enable = 1
      sbi(PORTC, 7);
      // enable = 0
      cbi(PORTC, 7);
      // wait
      for (i = 0; i < 255; ++i)
            for (j = 0; j < 255; ++j)
                  j = j;
}

void display_init()
{
      // send some commands to initialize and reset the display
      display_cmd(0x38);
      display_cmd(0x0E);
      display_cmd(0x06);
      display_cmd(0x01);
      display_cmd(0x02);
}

void display_char(byte display, byte c)
```

```
{
	byte i, j;
	// translate swedish ascii numbers to the correct characters of
the display
    switch (c) {
		case 'Å':
		case 'å':
			c = 'a';
			break;
		case 'Ä':
		case 'ä':
			c = 0xE1;
			break;
		case 'Ö':
		case 'ö':
			c = 0xEF;
			break;
	}
	DDRA = 0xff; //set PORT A for out
	PORTA = c; // set output character on port A
	// RS = 1
	DDRC = 0xff;
	sbi(PORTC, 0);
	// R/W = 0
	cbi(PORTC, 1);
	// enable = 1
	sbi(PORTC, 6 + display);
	// enable = 0
	cbi(PORTC, 6 + display);
	// RS = 0
	cbi(PORTC, 0);
	// wait
	for (i = 0; i < 255; ++i)
		;
}

void display_str(char *str)
{
	byte i = 0, j, nr, k = 0;
	byte len = strlen(str);
	if (len > 160)
		len = 160;
	k = 0;

	for (i = 0; i < len; ++i, ++k) {
		if (str[i] == '\n') { // translate new line characters
			nr = 40 - k % 40;
			for (j = 0; j < nr; ++j, ++k)
				display_char(k / 80, ' ');
		}
		else
			display_char(k / 80, str[i]);
	}
}

void wait()
{
	byte i, j, k;
	for (i = 0; i < 255; ++i)
		for (j = 0; j < 255; ++j)
			for (k = 0; k < 254; ++k)
```

```
                        ;
}

void short_wait()
{
      byte i, j;
      for (i = 0; i < 255; ++i)
            for (j = 0; j < 255; ++j)
                  j = j;
}

void next_question()
{
      byte button_state;
      byte p1_answer = 0;
      byte p2_answer = 0;
      byte q_nr = 0;
      // find a new question
      q_nr = rand() % NR_QUESTIONS;
      while (done_questions[q_nr]) {
            q_nr = rand() % NR_QUESTIONS;
      }
      done_questions[q_nr] = 1;
      // Read a question from the EEPROM
      mem_read_question(&data, q_nr);
      // show question
      display_init();
      display_str(data.text);
      // timeout
      wait();
      // show the alternatives
      display_init();
      display_str(data.answers);
      // wait for an answer
      DDRD = 0x00;
      button_state = PIND;
      // wait for all buttons to be released
      if ((button_state & 0xFC)) {
            display_init();
            display_str("Släpp upp knapparna!");
            while ((button_state & 0xFC))
                  button_state = PIND;

            display_init();
            display_str(data.answers);
      }
      // wait for a button to be pressed
      while (!(button_state & 0xFC))
            button_state = PIND;
      // determine which button that was pressed
      if (button_state & 0x10) { // p1 button 1
            p1_answer = 1;
      }
      if (button_state & 0x08) { // p1 button 2
            p1_answer = 2;
      }
      if (button_state & 0x04) { // p1 button 2
            p1_answer = 3;
      }
      if (button_state & 0x20) { // p2 button 1
            p2_answer = 1;
```

```
	}
	if (button_state & 0x40) { // p2 button 2
		p2_answer = 2;
	}
	if (button_state & 0x80) { // p2 button 2
		p2_answer = 3;
	}
	// validate the answer and update the standings
	display_init();
	if(p1_answer) {
		if (p1_answer == data.correct_answer) {
			if (p1_score < 255)
				++p1_score;
		}
		else {
			if (p1_score > 0)
				--p1_score;
		}
		sprintf(tmp_buf, "Rätt svar: %d\n" \
				"Spelare 1 svarade %d\n" \
				"Ställning: Spelare 1: %d   Spelare 2: %d",
				data.correct_answer, p1_answer, p1_score,
p2_score);
		display_str(tmp_buf);
	} else if(p2_answer) {
		if (p2_answer == data.correct_answer) {
			if (p2_score < 255)
				++p2_score;
		}
		else {
			if (p2_score > 0)
				--p2_score;
		}
		sprintf(tmp_buf, "Rätt svar: %d\n" \
				"Spelare 2 svarade %d\n" \
				"Ställning: Spelare 1: %d   Spelare 2: %d",
				data.correct_answer, p2_answer, p1_score,
p2_score);
		display_str(tmp_buf);
	}
	wait();
}

void mem_write(byte val)
{
	// write val to the EEPROM
	SPDR = val;
	while (!(SPSR & (1<<SPIF)))
		;
}

byte mem_read()
{
	// read a byte from the EEPROM
	SPDR = 0;
	while (!(SPSR & (1<<SPIF)))
		;
	return SPDR;
}

void mem_wait_ready(void){
```

```
  do{
    PORTB |= 0x10;  //CS = 1
    PORTB &= ~0x10;  //CS = 0
    mem_write(0x05);     // read the status register
  }while((mem_read() & 0x01));
  PORTB |= 0x10;  //CS = 1
}

void mem_init()
{
      // master init
      DDRB = (1<<DDB5) | (1<<DDB7) | (1<<DDB4);
      SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR0);
}

void mem_read_question(question_data *data, byte nr)
{
      byte i, j, k;
      byte adr1, adr2;
      adr1 = nr;
      adr2 = 0;
      k = 0;
      for (i = 0; i < QUESTION_TEXT_LENGTH / MEM_CHUNK_SIZE; i++) {

            // read question data
            for (j = 0; j < MEM_CHUNK_SIZE; j++) {
                  PORTB &= ~0x10;  //CS = 0
                  mem_write(0x03); // read a batch of bytes
                  // address
                  mem_write(adr1);  // adr
                  mem_write(adr2);  // adr
                  data->text[k++] = mem_read();
                  PORTB |= 0x10;   //CS = 1
                  ++adr2;
            }
      }
      k = 0;
      for (i = 0; i < ANSWER_TEXT_LENGTH / MEM_CHUNK_SIZE; i++) {
            // read answer data
            for (j = 0; j < MEM_CHUNK_SIZE; j++) {
                  PORTB &= ~0x10;  //CS = 0
                  mem_write(0x03); // read a batch of bytes
                  // address
                  mem_write(adr1);  // adr
                  mem_write(adr2);  // adr
                  data->answers[k++] = mem_read();
                  PORTB |= 0x10;   //CS = 1
                  ++adr2;
            }
      }
      PORTB &= ~0x10;  //CS = 0
      mem_write(0x03); // read a batch of bytes
      // address
      mem_write(adr1);  // adr
      mem_write(adr2);  // adr
      data->correct_answer = mem_read();
      PORTB |= 0x10;   //CS = 1
}

byte wait_for_game_to_start()
{
```

```
	byte rand_seed = 0, button_state;
	display_init();
	display_str("Tryck på valfri knapp för att starta spelet!");

	DDRD = 0x00;
	button_state = PIND;
	// get a random seed for an unpredictible order of questions
	while (!(button_state & 0xFC)) {
		button_state = PIND;
		++rand_seed;
	}
	return rand_seed;
}

void main(void)
{
	byte rand_seed_val;
	byte i;
	display_init();
	mem_init();
	rand_seed_val = wait_for_game_to_start();
	srand(rand_seed_val);
	while(1)
	{
		p1_score = p2_score = 0;
		// reset completed questions array
		for (i = 0; i < NR_QUESTIONS; i++)
			done_questions[i] = 0;
		// play a round
		for (i = 0; i < NR_QUESTIONS; i++)
			next_question();
		// determine who won
		if (p1_score > p2_score) {
			display_init();
			display_str("Yippi! Gratulerar! Spelare 1 har
vunnit!");
		}
		else if (p2_score > p1_score) {
			display_init();
			display_str("Yippi! Gratulerar! Spelare 2 har
vunnit!");
		}
		else {
			display_init();
			display_str("Oj det blev oavgjort!");
		}
		wait();
	}
}
```

Loader.c

```
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>
#include <avr/pgmspace.h>

#define QUESTION_TEXT_LENGTH 160
#define ANSWER_TEXT_LENGTH 90
#define MEM_CHUNK_SIZE 10
```

```
#define NR_QUESTIONS 29
typedef unsigned char byte;
byte question_nr = 0;

typedef struct{
      char text[QUESTION_TEXT_LENGTH];
      char answers[ANSWER_TEXT_LENGTH];
      byte correct_answer;
}question_data;

// We are using a method to keep the question data in program memory
// found on the following site
// http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_rom_array

const char question1[] PROGMEM = "Vad heter enligt grekisk myt
underjordens härskare?";
const char answer1[] PROGMEM = "1. Hades\n2. Hermes\n3. Hugo";
//const byte correct_answer1 = 1;
const char question2[] PROGMEM = "Vad kallas de processer genom vilka
företag och andra organisationer inhämtar och uttolkar information om
konkurrenter, teknisk utveckling, politiska risker etc.?";
const char answer2[] PROGMEM = "1. Benchmarking\n2. Priskontroll\n3.
Omvärldsbevakning";
//const byte correct_answer2 = 3;

/*
 * The rest of the questions are left out in the report version of
the code to save space
 */

const correct_answers[] = {1, 3, 1, 3, 2, 3, 2, 2, 1, 1, 3, 1, 1, 2,
2, 3, 3, 1, 3, 1, 2, 1, 1, 2, 2, 2, 1, 2, 3};
PGM_P prog_questions[NR_QUESTIONS] PROGMEM = {
    question1,
    question2,
      // More questions...
};

PGM_P prog_answers[NR_QUESTIONS] PROGMEM = {
    answer1,
    answer2,
      // More answers...
};

void display_cmd(byte cmd)
{
      byte i, j;
    DDRA = 0xff; //set PORT A for out
      PORTA = cmd; //put the command on the port
      // RS R/W = 0
      DDRC = 0xff;
      cbi(PORTC, 0);
      cbi(PORTC, 1);
      // enable = 1
      sbi(PORTC, 6);
      // enable = 0
      cbi(PORTC, 6);
      // enable = 1
      sbi(PORTC, 7);
      // enable = 0
      cbi(PORTC, 7);
```

```
	// wait
	for (i = 0; i < 255; ++i)
		for (j = 0; j < 255; ++j)
			j = j;
}

void display_init()
{
	// send some commands to initialize and reset the display
	display_cmd(0x38);
	display_cmd(0x0E);
	display_cmd(0x06);
	display_cmd(0x01);
	display_cmd(0x02);
}

void display_char(byte display, byte c)
{
	byte i, j;
	// translate swedish ascii numbers to the correct characters of
the display
    switch (c) {
		case 'Å':
		case 'å':
			c = 'a';
			break;
		case 'Ä':
		case 'ä':
			c = 0xE1;
			break;
		case 'Ö':
		case 'ö':
			c = 0xEF;
			break;
	}
	DDRA = 0xff; //set PORT A for out
	PORTA = c; // set output character on port A
	// RS = 1
	DDRC = 0xff;
	sbi(PORTC, 0);
	// R/W = 0
	cbi(PORTC, 1);
	// enable = 1
	sbi(PORTC, 6 + display);
	// enable = 0
	cbi(PORTC, 6 + display);
	// RS = 0
	cbi(PORTC, 0);
	// wait
	for (i = 0; i < 255; ++i)
		;
}

void display_str(char *str)
{
	byte i = 0, j, nr, k = 0;
	byte len = strlen(str);
	if (len > 160)
		len = 160;
	k = 0;
```

```
	for (i = 0; i < len; ++i, ++k) {
		if (str[i] == '\n') { // translate new line characters
			nr = 40 - k % 40;
			for (j = 0; j < nr; ++j, ++k)
				display_char(k / 80, ' ');
		}
		else
			display_char(k / 80, str[i]);
	}
}

void mem_write(byte val)
{
	// write val to the EEPROM
	SPDR = val;
	while (!(SPSR & (1<<SPIF)))
		;
}

byte mem_read()
{
	// read a byte from the EEPROM
	SPDR = 0;
	while (!(SPSR & (1<<SPIF)))
		;
	return SPDR;
}

void mem_wait_ready(void){
  do{
    PORTB |= 0x10;  //CS = 1
    PORTB &= ~0x10;  //CS = 0
    mem_write(0x05);     // read the status register
  }while((mem_read() & 0x01));
  PORTB |= 0x10;  //CS = 1
}

void mem_init()
{
	// master init
	DDRB = (1<<DDB5) | (1<<DDB7) | (1<<DDB4);
	SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR0);
}

void mem_write_question(question_data *data, byte nr)
{
	byte i, j, k;
	byte adr1, adr2;
	adr1 = nr;
	adr2 = 0;
	k = 0;
	for (i = 0; i < QUESTION_TEXT_LENGTH / MEM_CHUNK_SIZE; i++) {
		// write question data
		for (j = 0; j < MEM_CHUNK_SIZE; j++) {
			PORTB &= ~0x10;  //CS = 0
			mem_write(0x06); // write enable, WREN
			PORTB |= 0x10;   //CS = 1
			PORTB &= ~0x10;  //CS = 0
			mem_write(0x02); // write a batch of bytes, WRITE
			// write the address
			mem_write(adr1);  // adr
```

```
                mem_write(adr2);  // adr
                mem_write(data->text[k++]);
                PORTB |= 0x10;    //CS = 1
                mem_wait_ready();
                ++adr2;
            }
        }
        k = 0;
        for (i = 0; i < ANSWER_TEXT_LENGTH / MEM_CHUNK_SIZE; i++) {
            // write answers data
            for (j = 0; j < MEM_CHUNK_SIZE; j++) {
                PORTB &= ~0x10;  //CS = 0
                mem_write(0x06); // write enable, WREN
                PORTB |= 0x10;   //CS = 1
                PORTB &= ~0x10;  //CS = 0
                mem_write(0x02); // write a batch of bytes, WRITE
                // write the address
                mem_write(adr1);  // adr
                mem_write(adr2);  // adr
                mem_write(data->answers[k++]);
                PORTB |= 0x10;   //CS = 1
                mem_wait_ready();
                ++adr2;
            }
        }
        PORTB &= ~0x10;  //CS = 0
        mem_write(0x06); // write enable, WREN
        PORTB |= 0x10;   //CS = 1
        PORTB &= ~0x10;  //CS = 0
        mem_write(0x02); // write a batch of bytes, WRITE
        // write the address of the correct answer for the question
        mem_write(adr1);  // adr
        mem_write(adr2);  // adr
        mem_write(data->correct_answer);
        PORTB |= 0x10;   //CS = 1
        mem_wait_ready();
}

// a helper function to load questions into the EEPROM
void load_question(char *text, char *answers, byte correct_answer,
byte q_nr)
{
        question_data data;
        strncpy(data.text, text, QUESTION_TEXT_LENGTH - 1);
        data.text[QUESTION_TEXT_LENGTH - 1] = 0;
        strncpy(data.answers, answers, ANSWER_TEXT_LENGTH - 1);
        data.answers[ANSWER_TEXT_LENGTH - 1] = 0;
        data.correct_answer = correct_answer;
        mem_write_question(&data, q_nr);
}

void main(void)
{
        char q_data[QUESTION_TEXT_LENGTH];
        char a_data[ANSWER_TEXT_LENGTH];
        byte i;
      PGM_P p;
        byte str[35];
        question_nr = 0;
        display_init();
        mem_init();
```

```
      display_init();
      display_str("Writing to memory.");
      // fetch the questions from the program memory and save them in
the EEPROM
      for (i = 0; i < NR_QUESTIONS; i++) {
            memcpy_P(&p, &prog_questions[i], sizeof(PGM_P));
            strcpy_P(q_data, p);
            memcpy_P(&p, &prog_answers[i], sizeof(PGM_P));
            strcpy_P(a_data, p);
            load_question(q_data, a_data, correct_answers[i], i);
      }
      sprintf(str, "Memory loaded with %d questions.", i);
      display_init();
      display_str(str);
      while(1) {
      }
    return;
}
```