Laboratorio de Microcomputadoras - 66.09

Keep Coding And Nobody Explodes

Profesor:			Ing. Guillermo Campiglio									
Cuatrimestre/Año:			2º/2018									
Turno de las clases prácticas			Miércoles									
Jefe de trabajos prácticos:			Ing. Ricardo Arias									
Docente guía:												
Autores			Seguimiento del proyecto									
Nombre	Apellido	Padron										
Ana	Czarnitzki	96812										
Juan	Fresia	96631										
Alejandro	García	96661										

Observaciones:

.....
.....
.....
.....
.....
.....
.....
.....

Fecha de aprobación				Firma J.T.P

Coloquio	
Nota final	
Firma profesor	

1. Introducción

Keep Talking and Nobody Explodes (KTANE por sus siglas) es un videojuego promotor de la interacción social, en el que un jugador debe desactivar una bomba que sólo él puede ver en su pantalla.[1] Para ello, cuenta con la ayuda de los demás jugadores, que representan al grupo de expertos que saben desarmar las bombas. Bajo esta modalidad, la persona *que puede ver* la bomba deberá decirle al grupo de expertos qué ve, y dicho grupo deberá responderle qué acciones tomar para evitar que esta explote.

El objetivo del juego entonces consiste en lograr desactivar la bomba antes de que detone, debiendo los participantes resolver en el proceso una serie de minujuegos en forma de módulos de la propia bomba. En la versión original, los módulos van desde minijuegos similares al *Simon says* hasta laberintos y conjuntos de cables a desconectar.

2. Objetivos propuestos

En este proyecto, nos propusimos crear una versión simplificada y en la vida real de éste juego, en la que los jugadores también deban resolver una serie de módulos para lograr desarmar la bomba. Para esto, utilizamos un microcontrolador de 8 bits, displays de 7 segmentos para la cuenta regresiva de la bomba, y una pantalla táctil para la representación de los juegos. El jugador contará con tres vidas (o *strikes*).

En particular, nos propusimos realizar los siguientes módulos/minijuegos:

- El simón dice: una versión del juego clásico de *simon says*. En este se debe repetir una secuencia de colores apretando los botones de dichos colores. Para esta implementación en particular, los colores a oprimir no son los mismos de la secuencia sino que dependen de la cantidad de veces que el jugador haya perdido en los minijuegos anteriores.
- El juego de la memoria: se compone de una serie de botones con números que aparecen y que el jugador debe ir presionando según determinadas reglas. El juego progresa en etapas, y el jugador debe recordar en cada una qué botones y en qué posiciones presionó en etapas pasadas.
- Los cables: consiste en varios cables conectados al microcontrolador. Dependiendo de ciertos colores y un número en pantalla se deberá desconectar un cable en particular. Es el último minijuego y no cuenta con segundas oportunidades como el resto.
- El código de barras: se tiene un conjunto de barras que indican un subconjunto dentro de un *diagrama de Venn*, según este conjunto cuál de las barras se deberá presionar.

Además se estos módulos, se implementan:

- Cuenta regresiva: marcará el tiempo disponible para finalizar el juego de forma exitosa.
- Sonidos: se agregan sonidos de victoria y derrota, además de indicadores de click.

[1] Página oficial: http://www.keeptalkinggame.com/

- Strikes: marcan la cantidad de vidas perdidas.

En particular, se realizaron las siguientes simplificaciones respecto a la versión original:

- Los juegos se deberán jugar en secuencia en vez de poder elegir el orden a realizarlos.
- Por una cuestión de practicidad los cables se desconectan en vez de cortarse.
- El juego de los cables es el último y es *a todo o nada*.

La reglamentación para esta versión puede encontrarse en el *manual del juego* adjunto en este informe.

3. Objetivos logrados

Luego de la realización de este trabajo logramos implementar los juegos propuestos, además, implementamos un driver para la pantalla desde cero específico para el microcontrolador y la pantalla que utilizamos que resultó más corto y velóz que el original (específicamente, el de la librería de la pantalla).

Por último, gracias a que generamos una función de randomización para hacer las inicializaciones de los juegos, pudimos obtener un juego divertido y jugable.

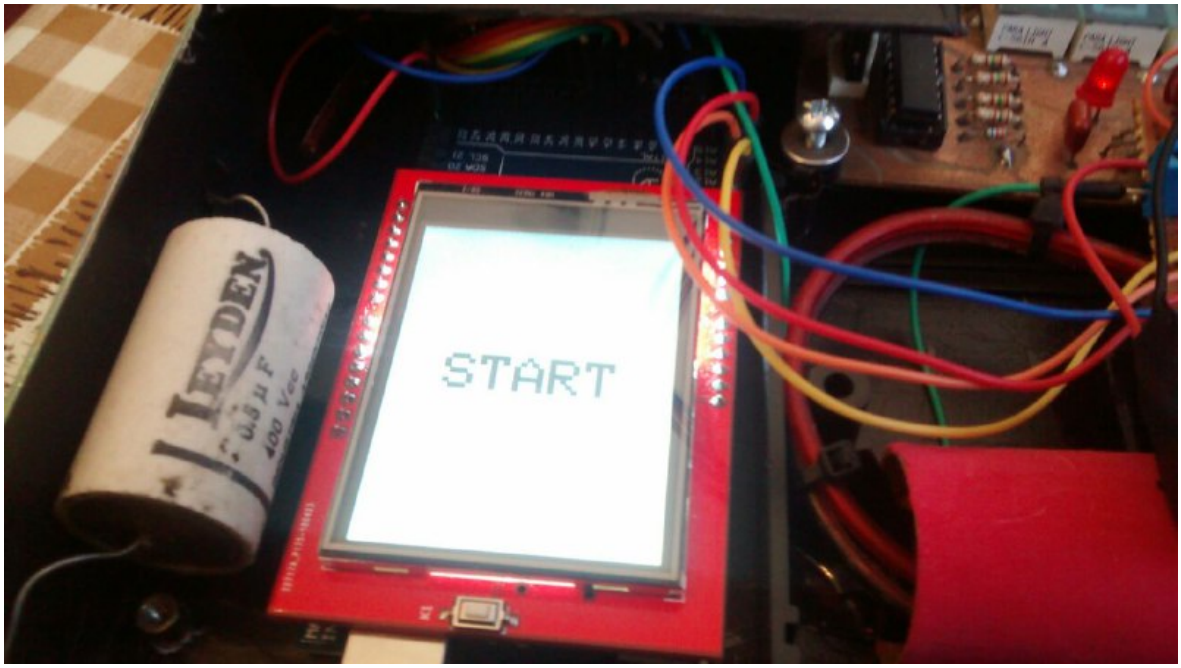

Figura 1: Pantalla mostrando "START"para iniciar el juego.

4. Descripción del Hardware

Como ya se mencionó, el proyecto fue basado en un microcontrolador de 8 bits que maneja la cuenta regresiva de la bomba. Ésta será desactivada al cortar algún cable específico, y para averiguar el mismo se deberán completar minijuegos a realizar sobre una pantalla táctil. El microcontrolador mostrará los minijuegos en esa pantalla, y en caso de que los jugadores no puedan desactivar la bomba, la hará "detonar", terminando con la partida en derrota.

La figura 2 muestra un diagrama en bloques con las conexiones de estos componentes:

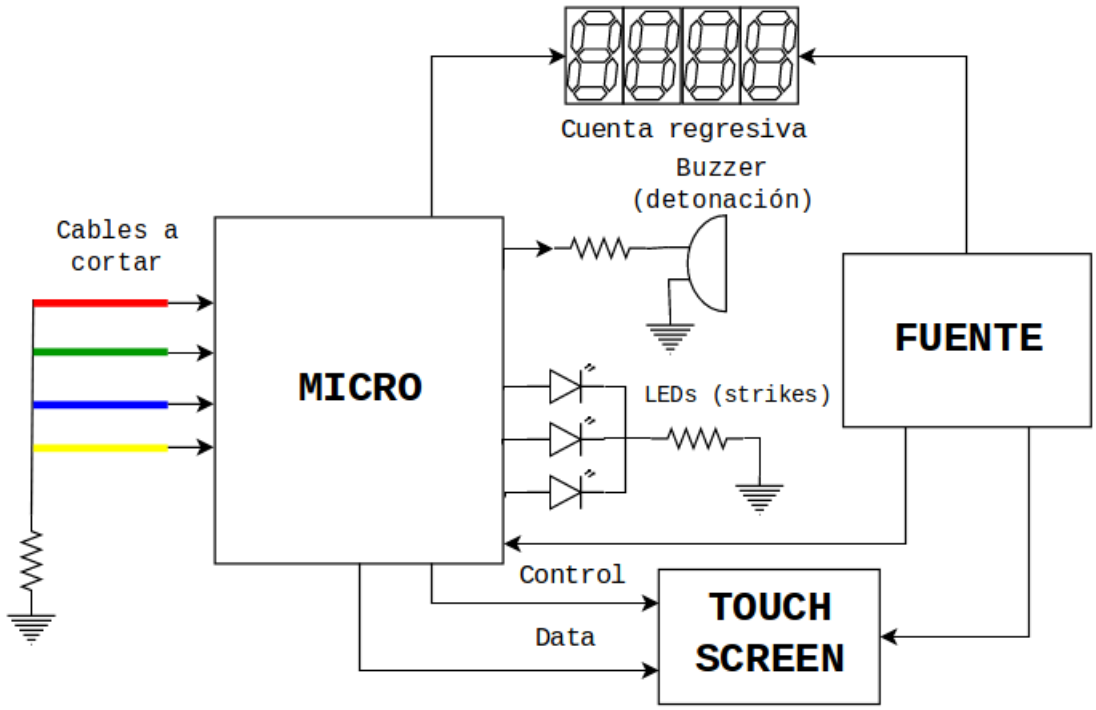

Figura 2: Diagrama en bloques de los componentes del proyecto.

Además, las fig. 3 y 4 muestran los circuitos pertinentes para el proyecto. En la primera de ellas, se observa el esquemático de la placa Arduino Mega (empleada para este proyecto), que incluye la posibilidad de alimentarse por USB o a través del programador. Si bien la placa es utilizada en conjunto para programar el microcontrolador, sólo se utiliza el cristal y la alimentación a través del regulador de tensión en el proyecto en sí.

En la segunda figura, se presenta el esquemático del circuito usado para el proyecto, con las conexiones empleadas para la pantalla táctil y la cuenta regresiva. Ésta en particular consiste en una placa de multiplexado de display de siete segmentos, donde se utilizan tres displays. Además, se cuenta con un buzzer que sonará al perder o ganar y varios LEDs que marcarán la cantidad de vidas que le quedan al jugador.

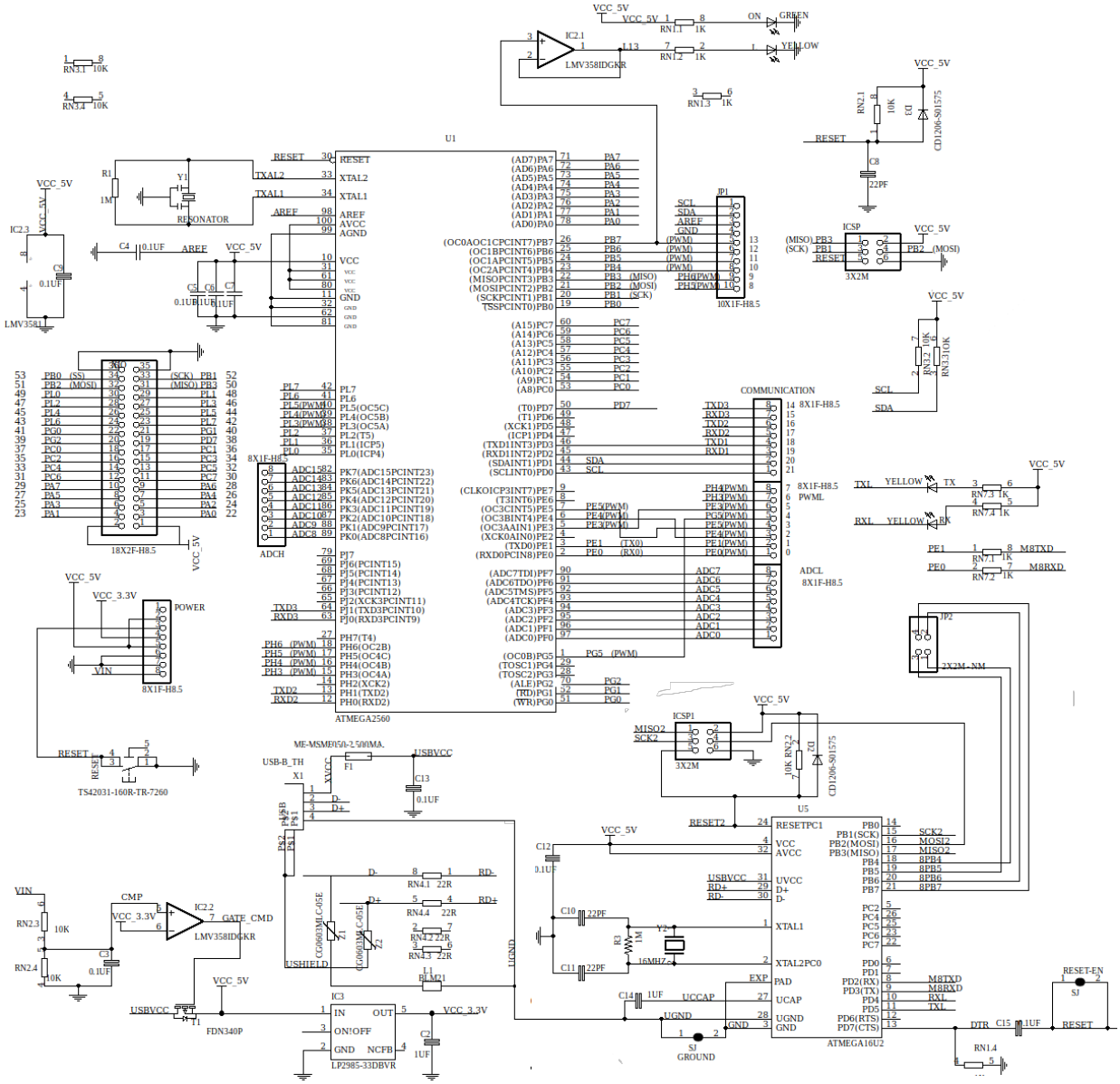

Figura 3: Esquemático del circuito del Arduino Mega.

El hardware específico usado por cada uno de los módulos (minijuegos) es el siguiente:

- **Módulo I: "Los cables".** Emplea los cables de colores conectados al microcontrolador. Se utiliza además la pantalla para dar pistas sobre qué cable desconectar.
- **Módulo II: "Simon says".** Este módulo utiliza la pantalla táctil únicamente, tanto para mostrar los diversos colores como para determinar qué color presionó el usuario.
- **Módulo III: "El código de barras"** Este módulo sólo utiliza la pantalla táctil para mostrar el código de barras y recibir un único toque del usuario.
- **Módulo IV: "El juego de la memoria"** Emplea sólamente la pantalla táctil para mostrar sucesivas rondas de números y determinar en cada una de ellas un botón presionado por el usuario.

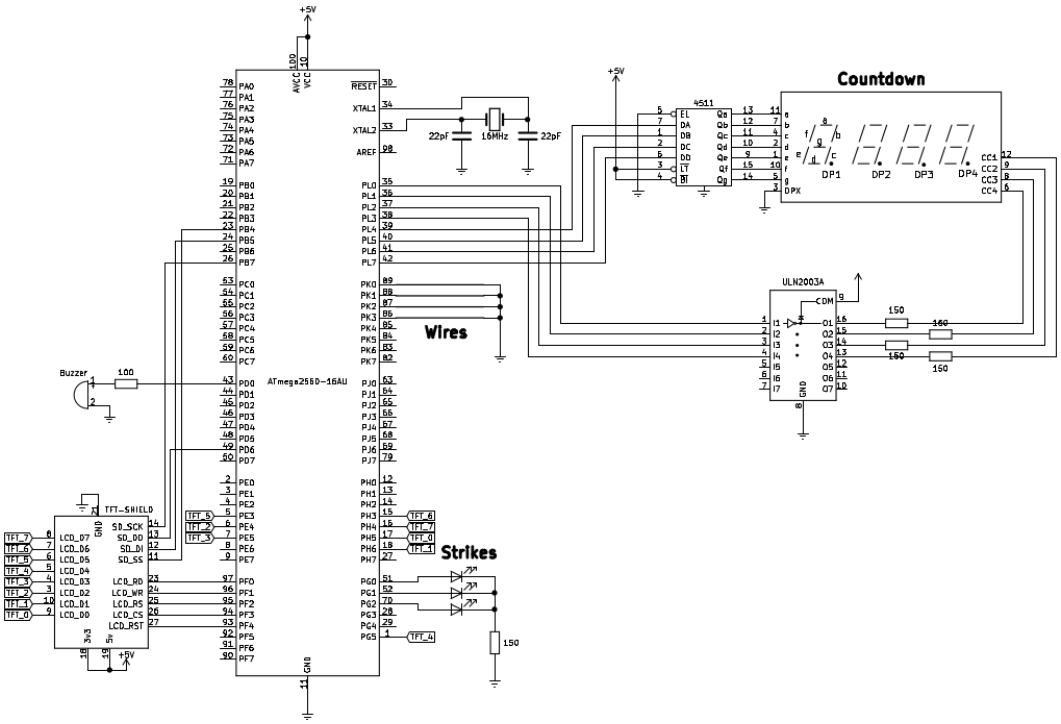

Figura 4: Esquemático del circuito usado para el proyecto.

La siguiente imagen muestra una fotografía de la bomba final funcionando:

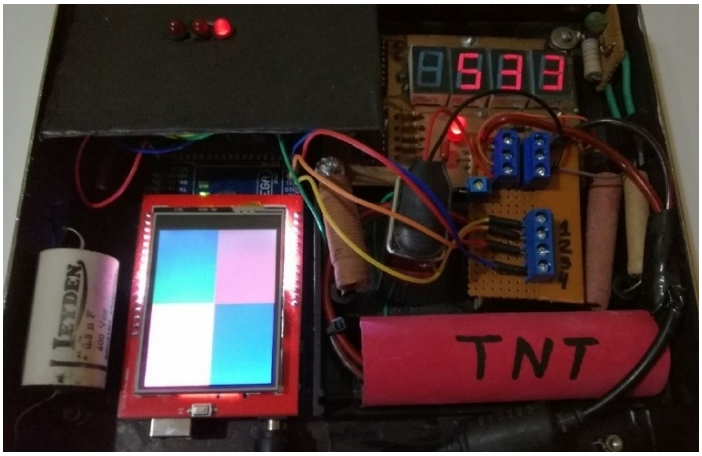

Figura 5: Imagen de la bomba funcionando con el juego del “Simon says”.

5. Descripción del Software

El diagrama de flujo básico para el microcontrolador de este proyecto se puede apreciar en la fig. 6. Allí se esquematiza la interrupción por timer, encargada de decrementar la cuenta regresiva de la bomba cada cierto tiempo, y el programa principal del microcontrolador, el cual se invoca obviamente de forma constante en un loop infinito.

Como se puede ver, luego de configurar correctamente el microcontrolador, se invoca una rutina de randomización, en la que se elegirán los distintos parámetros aleatorios de los juegos y el cable correcto que desactiva la bomba. La rutina de randomización utiliza como *semilla* al timer.

Una vez presionado el botón de inicio, comenzará la cuenta regresiva y los jugadores podrán participar en los distintos minijuegos. Mientras los jugadores no hayan desactivado la bomba ni hayan perdido, el juego continuará normalmente. Luego, ya sea por el caso de victoria o derrota (con la bomba detonando), el juego terminará con algún mensaje en la pantalla, y los jugadores podrán iniciar el proceso de nuevo.

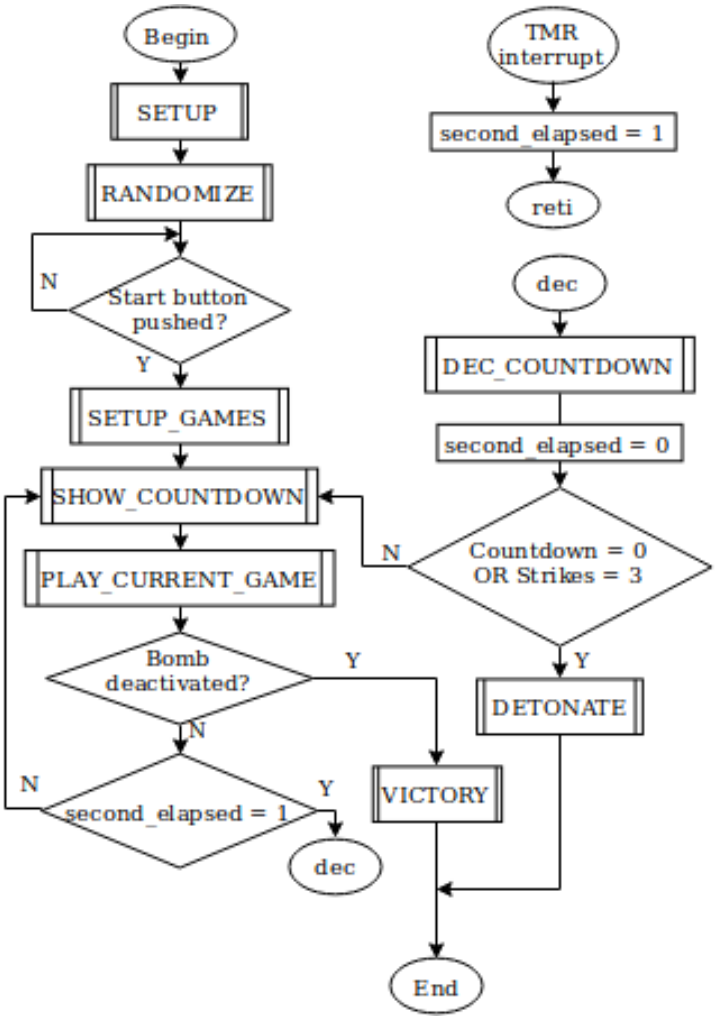

Figura 6: Diagrama de flujo del software desarrollado.

6. Conclusiones y posibles mejoras

Como primera conclusión podemos mencionar la mejora de velocidad del código en *assembly* respecto de la misma funcionalidad hecha en *C++*. Pudimos observar esto al hacer desde cero las librerías de control de la pantalla táctil usada para este proyecto y compararlas con las oficiales, las cuales originalmente estaban escritas en *C++* y de forma genérica, para soportar diversos dispositivos.

De la mano con esto, nos parece importante también recalcar cómo afecta el cableado de los componentes al momento de la velocidad de ejecución del código. En particular, la pantalla no utilizaba pines de todo un mismo puerto, de forma que fue necesario realizar varias máscaras para no pisar los valores de aquellos pines que no se empleaban. Estos cálculos (aplicar las máscaras para seleccionar pines de un puerto y para cargar los valores a la pantalla) agregaban tiempo de procesamiento extra. Dado que esta configuración nos otorgaba otras ventajas, la mantuvimos mejorando la velocidad usando números de 2 bytes simétricos (es decir, si un número lo consideramos un valor de 16 bits, los primeros 8 bits serán igual que los segundos 8 bits), de forma de no necesitar cargar dos valores separados para luego hacer el seteo.

Por último, dado que los tres miembros del equipo utilizamos distribuciones *Linux*, nos encontramos con dificultades respecto a las diferencias entre la bibliografía y las consecuencias de ensamblar en un entorno que no fuera *Windows*. Por ejemplo, las posiciones de la tabla de interrupciones no coincidían dado que las direcciones que figuran en el manual del *interrupt handler* ya estaban dividas por dos (*shifteadas*) representando una dirección de flash que apuntaba a palabras de 16 bits, mientras que nosotros al utilizar la directiva .ORG debimos direccionar al byte (*avr-gcc*, compilador que utilizamos, direcciona a byte).

Por otro lado, como posibles mejoras y próximos pasos podemos proponer:

- Permitir que los jugadores cambien de un juego a otro en cualquier momento en vez de que se deban jugar uno después del otro.
- Agregar juegos que dependan del tiempo que marca el contador.
- Agregar juegos que dependan de si se realizó ya antes otro juego.
- Agregar juegos que no utilicen la pantalla sólamente sino también otros dispositivos que podrían conectarse al microcontrolador, como sensores de movimiento o temperatura.

7. Bibliografía y recursos

- "AVR Microcontroller and Embedded Systems: Using Assembly and C", Muhammad Ali Mazidi, Sarmad Naimi y Sepehr Naimi. Prentice Hall, 2011
- Manual de referencia de avr-gcc. Disponible en: https:www.microchip.comwebdocAVRLibc ReferenceManualoverview_1overview_gcc.html.
- "Adafruit GFX Graphics Library", Phillip Burgess. Disponible en: https:cdn-learn.adafruit .comdownloadspdfadafruit-gfx-graphics-library.pdf
- "AVR341: Four and five-wire Touch Screen Controller", Atmel Application Note. Disponible en: http:www.lysator.liu.se kjell-eembeddeddoc8091.pdf.
- Hojas de datos de los diversos componentes.

8. Anexo I: Código producido

main.S

```
; -------------------------------------------------
;       KEEP CODING AND NOBODY EXPLODES (KCANE)
; -------------------------------------------------
; Project for 66.09.Laboratorio de Microcomputadoras
; Faculty of Engineering, University of Buenos Aires
;
; By Ana Czarnitzki, Alejandro García & Juan Fresia
;
; -------------------------------------------------
;
; -------------------------------------------------
;            GENERAL PROJECT INFORMATION
; -------------------------------------------------
; Microcontroller used: ATmega2560 @16MHz
;
; Other hardware used:
;   - 2.4'' TFT based on the ILI9320
;   - 7 segment displays multiplexing board, based
;   on CD4511 decoder and ULN2003
;   - 3v buzzer and LEDs for game results
;
; Ports used:
;   - TFT control pins on TFT_CONTROL_PORT (PORTF)
;   - TFT data pins distributed on PORTE, PORTG and
;   PORTH
;   - Displays on DISPLAYS_PORT (PORTL)
;   - Wires for the bomb on WIRES_PORT (PORTK)
; -------------------------------------------------
  .MACRO INPUT dest, source
  .IF \source<0x40
  IN \dest, \source
  .ELSE
  LDS \dest, \source
  .ENDIF
  .ENDM

  .MACRO OUTPUT dest, source
  .IF \dest<0x40
  OUT \dest, \source
  .ELSE
  STS \dest, \source
  .ENDIF
  .ENDM

  #include "avr-linux.h"
  #include "interrupts.S"
  #include "displays.S"
  #include "tft-touch.S"
  #include "tft-draw.S"
  #include "games.S"
  #include "buzzer.S"

; -------------------------------------------------
;                   MAIN PROGRAM
; -------------------------------------------------

  CSEG

main:
  ; Stack configuration
  LDI R16, LOW(RAMEND)
  OUTPUT SPL, R16
  LDI R16, HIGH(RAMEND)
  OUTPUT SPH, R16

  CALL SETUP
  SEI

loop:
  CALL SHOW_DISPLAYS
  CALL PLAY_CURRENT_GAME
  CALL CHECK_BOMB_WAS_DEACTIVATED
  CPI R24, 1
  BREQ player_deactivated_bomb

  CALL CHECK_STRIKES_REACHED_MAX
  CPI R24, 1
  BREQ player_exploded

  LDS R18, second_has_passed
  CPI R18, 1
  BREQ decrement_countdown
  RJMP loop

player_deactivated_bomb:
  CALL GAME_VICTORY_SCREEN
  RJMP main

decrement_countdown:
  CALL DEC_DISPLAYS
  CLR R18
  STS second_has_passed, R18
  ; Check if the displays reached zero
  CALL CHECK_DISPLAYS_ARE_ZERO
  CPI R24, 1
  BRNE loop
  ; If the displays reached zero, game over!
player_exploded:
  CALL GAME_DEFEAT_SCREEN
  RJMP main

; -------------------------------------------------
;                AUXILIAR FUNCTIONS
; -------------------------------------------------

; Sets the used PORTs as input/output, enables and
; starts timmer0 overflow interrupt, and gives an
; initial value to used RAM variables.
SETUP:
  CALL SETUP_INTERRUPTS
  CALL SETUP_DISPLAYS
  CALL TFT_START
  CALL SETUP_BUZZER

  CALL GAME_STARTING_SCREEN

  CALL SETUP_GAMES

  ; Clear the second_has_passed flag
  CLR R20
  STS second_has_passed, R20
  RET

  DSEG
; Auxiliar flag to know if a second has passed
second_has_passed:
  BYTE 1
```

interrupts.S

```
  CSEG

  .ORG 0
  JMP main

  ; Timer0 interrupt
  .ORG TIMER0_OVF_ISR
  JMP T0_ISR

  .ORG _VECTORS_SIZE

; Setups the used interrupts and their registers.
; Note this function DOES NOT enable interrupts.
SETUP_INTERRUPTS:
  ; Configure timer0 interrupt to count up to
  ; 1 second with an extra count variable.
  ; Making timer0 count 157 times with a
  ; 1024:1 prescale and CLK of 16MHZ makes
  ; it count 1024*157/16M = 0.01 second. Hence
  ; count must count up to 100 for 1 second.
  CLR R20
  STS count, R20

  LDI R20, (1 << TOIE0)
  OUTPUT TIMSK0, R20

  LDI R20, 99 ; 256 - 99 = 157
  OUTPUT TCNT0, R20
  LDI R20, 0x05 ; Prescaler 1024:1
  OUTPUT TCCR0B, R20
  RET
```

```
; Timer0 interrupt handler that will be called every
; 0.01 seconds. Once the count variable reaches 100
; (i.e. 1 second elapsed), it must DEC_DISPLAYS.
T0_ISR:
  ; First store the called saved registers
  ; since we don't know what important value
  ; they could have right now
  PUSH R18
  PUSH R19
  PUSH R20
  PUSH R21
  PUSH R22
  PUSH R23
  PUSH R24
  PUSH R25
  PUSH R26
  PUSH R27
  PUSH R30
  PUSH R31
  INPUT R18, SREG ; Very important
  PUSH R18
  LDI R20, 99
  OUTPUT TCNT0, R20

  LDS R20, count
  INC R20
  CPI R20, 100
  BREQ clear_count
  RJMP store_count
clear_count:
  ; If here, one second has elapsed
  LDI R24, 1
  STS second_has_passed, R24
  CLR R20
store_count:
  STS count, R20
  ; Restore the saved registers
  POP R18
  OUTPUT SREG, R18
  POP R31
  POP R30
  POP R27
  POP R26
  POP R25
  POP R24
  POP R23
  POP R22
  POP R21
  POP R20
  POP R19
  POP R18
  RETI

  DSEG
; Auxiliar counter for timer0 interrupt
count:
  BYTE 1
```

displays.S

```
  .GLOBAL SETUP_DISPLAYS
  .GLOBAL SHOW_DISPLAYS
  .GLOBAL DEC_DISPLAYS
  .GLOBAL CHECK_DISPLAYS_ARE_ZERO

  .EQU DISPLAYS_PORT, PORTL
  .EQU DISPLAYS_DDR, DDRL
  .EQU DISPLAYS_AMOUNT, 4
  .EQU PLAYING_MINUTES, 6

  CSEG

; Initializes the displays array in RAM memory. Uses
; the PLAYING_MINUTES constant as the initial value
; for a countdown in the displays.
SETUP_DISPLAYS:
  LDI R18, 0xFF
  OUTPUT DISPLAYS_DDR, R18
  LDI ZL, LOW(displays_ram)
  LDI ZH, HIGH(displays_ram)

  LDI R18, DISPLAYS_AMOUNT
  CLR R1
setup_displays_loop:
  ST Z+, R1
  DEC R18
  BRNE setup_displays_loop
  ; Load the PLAYING_MINUTES tens and units
  LDI R18, (PLAYING_MINUTES / 10)
  STS displays_ram + 3, R18
  LDI R18, (PLAYING_MINUTES % 10)
  STS displays_ram + 2, R18
  RET

; Performs a displays multiplexing, showing in the
; displays the values of displays_ram. DISPLAYS_PORT
; is used for the multiplexing, being D0-D3 used as
; the control pins (connected to the ULN) and D4-D7
; as the BCD number (connected to the 4511).
SHOW_DISPLAYS:
  LDI ZL, LOW(displays_ram)
  LDI ZH, HIGH(displays_ram)
  LDI R18, 0x1
next_display:
  LD R19, Z+
  SWAP R19
  ADD R19, R18
  OUTPUT DISPLAYS_PORT, R19

  PUSH R18
  PUSH ZL
  PUSH ZH
  CALL DISPLAY_DELAY
  POP ZH
  POP ZL
  POP R18

  LSL R18
  SBRS R18, DISPLAYS_AMOUNT
  RJMP next_display
  RET

; Decrement the displays by "1 second", considering
; the displays represent a MM:SS format. This
; function should be called every one second by the
; main program to properly make the countdown.
DEC_DISPLAYS:
  LDI ZL, LOW(displays_ram)
  LDI ZH, HIGH(displays_ram)
  ; R18=display number (0 to DISPLAYS_AMOUNT)
  CLR R18
next_dec:
  LD R19, Z
  DEC R19

  CPI R19, 255 ; display = -1?
  BRNE dec_finished

  ; If here, I have to decrement the
  ; next display by 1 unit too
  LDI R19, 9 ; reset this display
  CPI R18, 1
  BRNE store_and_next_dec
  ; Note the second display must be
  ; reset to 5 and not to 9
  LDI R19, 5
store_and_next_dec:
  ST Z+, R19
  INC R18
  CPI R18, DISPLAYS_AMOUNT
  BRNE next_dec
  RET

dec_finished:
  ST Z, R19
  RET

; Checks if all the displays reached the 0 value
; (i.e. if the countdown is over). Returns true or
; false on R24 (false as 0, true as 1).
CHECK_DISPLAYS_ARE_ZERO:
  LDI ZL, LOW(displays_ram)
  LDI ZH, HIGH(displays_ram)
  CLR R24 ; R24 initially false
  ; R18=display number (0 to DISPLAYS_AMOUNT)
  CLR R18;
check_zero_loop:
  LD R19, Z+
  CPI R19, 0
  BRNE displays_not_zero
```

```
  INC R18
  CPI R18, DISPLAYS_AMOUNT
  BRNE check_zero_loop
  ; If here, all displays are zero
  INC R24 ; R24 = true
displays_not_zero:
  RET

; Performs a small delay: the time each display is
; turned on during the multiplexing.
DISPLAY_DELAY:
  LDI R18, 80
loop1:
  LDI R19, 20
loop2:
  LDI R20, 10
loop3:
  NOP
  DEC R20
  BRNE loop3

  DEC R19
  BRNE loop2

  DEC R18
  BRNE loop1
  RET

  DSEG
; Displays array in RAM memory
displays_ram:
  BYTE DISPLAYS_AMOUNT
```

tft-draw.S

```
#include "avr-linux.h"
#include "tft-registers.S"
#include "tft-colors.S"
#include "tft-writing.S"
#include "tft-char-ROM.S"
#include "tft-delay.S"

CSEG

TFT_START:
  CALL TFT_INIT_PORTS
  CALL TFT_ANALOG_INIT
  CALL TFT_BEGIN

  LDI R25, 1
  CALL MILI_DELAY
  RET

TFT_RESET:
  CS_IDLE
  WR_IDLE
  RD_IDLE

  TFT_RESET_LOW
  LDI R25, 2    ; Magic recipe
  CALL MILI_DELAY
  TFT_RESET_HIGH

  CS_ACTIVE
  CD_COMMAND
  CLR R25
  CALL WRITE_8
  WR_STROBE
  WR_STROBE
  WR_STROBE
  CS_IDLE
  RET

TFT_INIT_PORTS:
  PUSH R19

  INPUT R19, TFT_CONTROL_DDR
  ORI R19, LCD_CONTROL_PORT_MASK
  OUTPUT TFT_CONTROL_DDR, R19

  CALL SET_WRITE_DIR
  POP R19
  RET

; Defines the valid range for the screen addresses.
; The range is set by a pair of points (x1, y1) and
; (x2, y2) which makes a rectangle. Preconditions:
; x2 > x1 and y2 > y1.
; Each coordinate is a 16 bit number, and all of
    them
; are passed through the following registers:
;     x1 in R25:R24
;     y1 in R23:R22
;     x2 in R21:R20
;     y2 in R19:R18
TFT_SET_ADDR_WINDOW:
  PUSH R16
  PUSH R17
  PUSH R20
  PUSH R21
  PUSH R22
  PUSH R23
  PUSH R25

  CS_ACTIVE
  ; Save y1 value
  MOVW R16, R22 ; R17:R16 = R23:R22
  ; Define column address set
  ; WRITE_REGISTER_32(ILI9341_COLADDRSET, x1, x2)
  MOV R23, R25
  MOV R22, R24
  LDI R25, ILI9341_COLADDRSET
  CALL WRITE_REGISTER_32
  ; Define rows address set
  ; WRITE_REGISTER_32(ILI9341_COLADDRSET, y1, y2)
  MOVW R22, R16 ; R23:R22 = R17:R16
  MOVW R20, R18 ; R21:R20 = R19:R18
  LDI R25, ILI9341_PAGEADDRSET
  CALL WRITE_REGISTER_32

  CS_IDLE

  POP R25
  POP R23
  POP R22
  POP R21
  POP R20
  POP R17
  POP R16
  RET

TFT_BEGIN:
  CALL TFT_RESET
  LDI R25, 200
  CALL MILI_DELAY

  CS_ACTIVE

  LDI R25, ILI9341_SOFTRESET
  LDI R24, 0x0
  CALL WRITE_REGISTER_8

  LDI R25, 50
  CALL MILI_DELAY

  LDI R25, ILI9341_DISPLAYOFF
  LDI R24, 0x0
  CALL WRITE_REGISTER_8

  LDI R25, ILI9341_POWERCONTROL1
  LDI R24, 0x23
  CALL WRITE_REGISTER_8

  LDI R25, ILI9341_POWERCONTROL2
  LDI R24, 0x10
  CALL WRITE_REGISTER_8

  LDI R25, HIGH(ILI9341_VCOMCONTROL1)
  LDI R24, LOW(ILI9341_VCOMCONTROL1)
  LDI R23, 0x2B
  LDI R22, 0x2B
  CALL WRITE_REGISTER_16

  LDI R25, ILI9341_VCOMCONTROL2
  LDI R24, 0xC0
  CALL WRITE_REGISTER_8

  LDI R25, ILI9341_MEMCONTROL
  LDI R24, ILI9341_MADCTL_MY | ILI9341_MADCTL_BGR
  CALL WRITE_REGISTER_8
```

```
  LDI R25, ILI9341_PIXELFORMAT
  LDI R24, 0x55
  CALL WRITE_REGISTER_8

  LDI R25, HIGH(ILI9341_FRAMECONTROL)
  LDI R24, LOW(ILI9341_FRAMECONTROL)
  LDI R23, 0x00
  LDI R22, 0x1B
  CALL WRITE_REGISTER_16

  LDI R25, ILI9341_ENTRYMODE
  LDI R24, 0x07
  CALL WRITE_REGISTER_8

  LDI R25, ILI9341_SLEEPOUT
  LDI R24, 0x00
  CALL WRITE_REGISTER_8

  LDI R25, 150
  CALL MILI_DELAY

  LDI R25, ILI9341_DISPLAYON
  LDI R24, 0x00
  CALL WRITE_REGISTER_8

  RET

; Fills the screen with a given color. The color is
; received on R25:R24.
TFT_FILL_SCREEN:
  PUSH R16
  PUSH R17

  MOVW R16, R24 ; R17:R16 = R25:R24
  CLR R25
  CLR R24
  CLR R23
  CLR R22
  LDI R21, HIGH(TFT_WIDTH - 1)
  LDI R20, LOW(TFT_WIDTH - 1)
  LDI R19, HIGH(TFT_HEIGHT - 1)
  LDI R18, LOW(TFT_HEIGHT - 1)
  CALL TFT_FILL_RECT

  POP R17
  POP R16
  RET

; Draws a filled rectangle in a range of screen
; addresses. The range is set by a point (x, y) on
; the upper-left vertex of the rectangle and a witdh
; and height. Each value is a 16 bit number, and all
; of them are passed through the following registers
   :
;      x in R25:R24
;      y in R23:R22
;  width in R21:R20
; height in R19:R18
;  color in R17:R16
TFT_FILL_RECT_HW:
  PUSH R18
  PUSH R19
  PUSH R20
  PUSH R21

  ADD R20, R24
  ADC R21, R25

  ADD R18, R22
  ADC R19, R23
  CALL TFT_FILL_RECT

  POP R21
  POP R20
  POP R19
  POP R18
  RET

; Draws a filled rectangle in a range of screen
; addresses. The range is set by a pair of points
; (x1, y1) and (x2, y2) which makes a rectangle.
; Precondition x2 > x1 and y2 > y1. Each value is a
; 16 bit number, and all of them are passed through
; the following registers:
;      x1 in R25:R24
;      y1 in R23:R22
;      x2 in R21:R20
;      y2 in R19:R18
;  color in R17:R16
TFT_FILL_RECT:
  PUSH R14
  PUSH R15
  PUSH R18
  PUSH R19
  PUSH R20
  PUSH R21
  PUSH R22
  PUSH R23
  PUSH R24
  PUSH R25
  PUSH R26
  PUSH R27
  PUSH R28
  PUSH R29

  CALL TFT_INIT_PORTS
  ; x2-x1 <--- R29:R28
  ; y2-y1 <--- R27:R26
  MOVW R28, R20
  SUB R28, R24
  SBC R29, R25
  MOVW R26, R18
  SUB R26, R22
  SBC R27, R23

  ; setAddrWindow(x1, y1, x2, y2)
  CALL TFT_SET_ADDR_WINDOW
  ; Loop to print every pixel
  CS_ACTIVE
  CD_COMMAND
  LDI R25, ILI9341_MEMORYWRITE
  CALL WRITE_8
  CD_DATA;

  ; for (int x = (x2-x1); x > 0; x--)
  ;      for (int y = (y2-y1); y > 0; y--)
  ;      WRITE_8(hi(color))
  ;      WRITE_8(lo(color))
  ; x = R29:R28
  ; y = R27:R26
  ADIW R28, 1
  ADIW R26, 1
  MOVW R14, R26 ; R15:R14 = R27:R26

  ; If hi(color) == lo(color), write color once
  ; the rest of the loop will only do WR_STROBE
  CP R16, R17
  BRNE tft_fill_rect_loop_x
  MOV R25, R17
  CALL WRITE_8
  WR_STROBE

tft_fill_rect_loop_x:
  MOVW R26, R14 ; R27:R26 = R15:R14
  CPI R28, 0x0
  BRNE tft_fill_rect_loop_y
  CPI R29, 0x0
  BRNE tft_fill_rect_loop_y
  RJMP tft_fill_rect_exit

tft_fill_rect_loop_y:
  CPI R27, 0x0
  BRNE tft_fill_rect_color_pixel
  CPI R26, 0x0
  BRNE tft_fill_rect_color_pixel
  SBIW R28, 1
  RJMP tft_fill_rect_loop_x

tft_fill_rect_color_pixel:
  ; Skip optimization
  CP R16, R17
  BRNE tft_fill_rect_color_pixel_diff

tft_fill_rect_color_pixel_same:
    WR_STROBE
  WR_STROBE
  SBIW R26, 1
  RJMP tft_fill_rect_loop_y
```

```
tft_fill_rect_color_pixel_diff:
  ; WRITE_8(high(color))
  MOV R25, R17
  CALL WRITE_8
  ; WRITE_8(low(color))
  MOV R25, R16
  CALL WRITE_8
  SBIW R26, 1
  RJMP tft_fill_rect_loop_y

tft_fill_rect_exit:
  POP R29
  POP R28
  POP R27
  POP R26
  POP R25
  POP R24
  POP R23
  POP R22
  POP R21
  POP R20
  POP R19
  POP R18
  POP R15
  POP R14
  RET

; Draws a character c on the screen with upper-left
; position (x, y), colour fg and size s. Each value
; is 16 bits, and are passed through registers:
;       x in R25:R24
;       y in R23:R22
;  char c in R21:R20 (R21 not used)
;      fg in R19:R18
;       s in R17:R16 (R17 not used)
TFT_DRAW_CHAR:
  PUSH R11
  PUSH R12
  PUSH R13
  PUSH R14
  PUSH R15
  PUSH R26
  PUSH R27
  PUSH R30
  PUSH R31
  ; Save size in temporal registers
  CLR R15
  MOV R14, R16
  ; Save color in R17:R16 to prepare for fill rect
  MOVW R16, R18
  ; Save y coordinate
  MOVW R12, R22
  ; Z = tft_char_ROM + (c*5)
  LDI ZH, HIGH(tft_char_ROM)
  LDI ZL, LOW(tft_char_ROM)

  LDI R26, 0x5
  MUL R20, R26
  ADD ZL, R0
  ADC ZH, R1

  CLR R26 ; R26 = i
  ; Set width and height to size
  CLR R21
  MOV R20, R14
  CLR R19
  MOV R18, R14

tft_draw_char_next_line:
  CPI R26, TFT_CHAR_W
  BREQ tft_draw_char_exit

  CLR R27 ; R27 = j
  ; Get char ROM data
  LPM R11, Z+   ; line = font[char*5 + i]
  CPI R26, 0x5
  BRNE tft_draw_char_line_loop
  CLR R11

tft_draw_char_line_loop:
    CPI R27, TFT_CHAR_H
  BREQ tft_draw_char_end_line
  INC R27
  SBRS R11, 0 ; line & 0x1
  RJMP tft_draw_char_line_loopNext

tft_drawChar_drawRect:
  ; fillRect_HW(x+(i*size), y+(j*size), size, size,
      fg | bg);
  CALL TFT_FILL_RECT_HW

tft_draw_char_line_loopNext:
  ; y += size
  ADD R22, R14
  ADC R23, R15
  LSR R11
  RJMP tft_draw_char_line_loop

tft_draw_char_end_line:
    INC R26
  ; x += size
  ADD R24, R14
  ADC R25, R15
  MOVW R22, R12 ; R23:R22 = R13:12
  RJMP tft_draw_char_next_line

tft_draw_char_exit:
  POP R31
  POP R30
  POP R27
  POP R26
  POP R15
  POP R14
  POP R13
  POP R12
  POP R11
  RET
```

tft-touch.S

```
  ; TouchScreen control pins
  ; xp (x+) H5
  ; xm (x-) F2
  ; yp (y+) F3
  ; ym (y-) H6

  .EQU TOUCH_XP_DDR, DDRH
  .EQU TOUCH_XM_DDR, DDRF
  .EQU TOUCH_YP_DDR, DDRF
  .EQU TOUCH_YM_DDR, DDRH

  .EQU TOUCH_XP_PORT, PORTH
  .EQU TOUCH_XM_PORT, PORTF
  .EQU TOUCH_YP_PORT, PORTF
  .EQU TOUCH_YM_PORT, PORTH

  .EQU TOUCH_XP_MASK, 0x20
  .EQU TOUCH_XM_MASK, 0x04
  .EQU TOUCH_YP_MASK, 0x08
  .EQU TOUCH_YM_MASK, 0x40

  .EQU TOUCH_Y_ADC, 0x3
  .EQU TOUCH_X_ADC, 0x2

  .EQU ADC_CHAN_MASK, 0x0F

CSEG

; Configures the ADCs used for reading the touched
; point on screen coordinaates.
TFT_ANALOG_INIT:
  PUSH R16

  LDI R16, (1 << REFS0)
  STS ADMUX, R16
  LDI R16, (1 << ADEN | 1 << ADPS0 | 1 << ADPS1 | 1
      << ADPS2)
  STS ADCSRA, R16

  POP R16
  RET

; ANALOG_READ performs an ADC read from a channel
; (ADC pin) indicated by register R25.
; Result is stored in R25:R24.
ANALOG_READ:
  PUSH R16
  PUSH R17
  ; Trigger ADC read
  LDS R16, ADMUX
```

```
  ANDI R16, ~ADC_CHAN_MASK
  ANDI R25, ADC_CHAN_MASK
  OR R16, R25
  STS ADMUX, R16
  LDI R16, (1 << ADSC)
  LDS R17, ADCSRA
  OR R17, R16
  STS ADCSRA, R17
adc_wait:
  LDS R17, ADCSRA
  SBRC R17, ADSC
  RJMP adc_wait

  LDS R24, ADCL
  LDS R25, ADCH

  POP R17
  POP R16
  RET

; Performs an ADC conversion of the x coordinate
; with ANALOG_READ. Returns the result on R25:R24.
TFT_READ_TOUCH_X:
  PUSH R17
  PUSH R16

  ; yp and ym as input with port = 0
  INPUT R17, TOUCH_YP_DDR
  ANDI R17, ~TOUCH_YP_MASK
  OUTPUT TOUCH_YP_DDR, R17

  LDS R17, TOUCH_YM_DDR
  ANDI R17, ~TOUCH_YM_MASK
  STS TOUCH_YM_DDR, R17

  INPUT R17, TOUCH_YP_PORT
  ANDI R17, ~TOUCH_YP_MASK
  OUTPUT TOUCH_YP_PORT, R17

  LDS R17, TOUCH_YM_PORT
  ANDI R17, ~TOUCH_YM_MASK
  STS TOUCH_YM_PORT, R17

  ; xp as output high
  LDS R17, TOUCH_XP_DDR
  ORI R17, TOUCH_XP_MASK
  STS TOUCH_XP_DDR, R17

  LDS R17, TOUCH_XP_PORT
  ORI R17, TOUCH_XP_MASK
  STS TOUCH_XP_PORT, R17

  ; xm as output low
  INPUT R17, TOUCH_XM_DDR
  ORI R17, TOUCH_XM_MASK
  OUTPUT TOUCH_XM_DDR, R17

  INPUT R17, TOUCH_XM_PORT
  ANDI R17, ~TOUCH_XM_MASK
  OUTPUT TOUCH_XM_PORT, R17

  ; Returns the read value in R25:R24.
  ; Yes, this is right, the touchscreen coords
  ; are inverted with respect to the TFT ones.
  LDI R25, TOUCH_Y_ADC
  CALL ANALOG_READ

  POP R16
  POP R17
  RET

; Performs an ADC conversion of the y coordinate
; with ANALOG_READ. Returns the result on R25:R24.
TFT_READ_TOUCH_Y:
  PUSH R17
  PUSH R16

  ; xp and xm as input with port = 0
  INPUT R17, TOUCH_XP_DDR
  ANDI R17, ~TOUCH_XP_MASK
  OUTPUT TOUCH_XP_DDR, R17

  INPUT R17, TOUCH_XM_DDR
  ANDI R17, ~TOUCH_XM_MASK
  OUTPUT TOUCH_XM_DDR, R17

  INPUT R17, TOUCH_XP_PORT
  ANDI R17, ~TOUCH_XP_MASK
  OUTPUT TOUCH_XP_PORT, R17

  INPUT R17, TOUCH_XM_PORT
  ANDI R17, ~TOUCH_XM_MASK
  OUTPUT TOUCH_XM_PORT, R17

  ; yp as output high
  INPUT R17, TOUCH_YP_DDR
  ORI R17, TOUCH_YP_MASK
  OUTPUT TOUCH_YP_DDR, R17

  INPUT R17, TOUCH_YP_PORT
  ORI R17, TOUCH_YP_MASK
  OUTPUT TOUCH_YP_PORT, R17

  ; ym as output low
  INPUT R17, TOUCH_YM_DDR
  ORI R17, TOUCH_YM_MASK
  OUTPUT TOUCH_YM_DDR, R17

  INPUT R17, TOUCH_YM_PORT
  ANDI R17, ~TOUCH_YM_MASK
  OUTPUT TOUCH_YM_PORT, R17

  ; Returns the read value in R25:R24.
  ; Read ADC from XP (coordintes are inverted).
  LDI R25, TOUCH_X_ADC
  CALL ANALOG_READ

  POP R16
  POP R17
  RET

; Reads both x and y analog pins from touchscreen
; and return its readings in registers:
; z2 (from yp): R25:R24
; z1 (from xm): R23:R22
TFT_READ_TOUCH_Z:
  PUSH R16
  PUSH R17

  ; xp as output low
  INPUT R17, TOUCH_XP_DDR
  ORI R17, TOUCH_XP_MASK
  OUTPUT TOUCH_XP_DDR, R17

  INPUT R17, TOUCH_XP_PORT
  ANDI R17, ~TOUCH_XP_MASK
  OUTPUT TOUCH_XP_PORT, R17

  ; ym as output high
  INPUT R17, TOUCH_YM_DDR
  ORI R17, TOUCH_YM_MASK
  OUTPUT TOUCH_YM_DDR, R17

  INPUT R17, TOUCH_YM_PORT
  ORI R17, TOUCH_YM_MASK
  OUTPUT TOUCH_YM_PORT, R17

  ; xm and yp as input low
  INPUT R17, TOUCH_XM_DDR
  ANDI R17, ~TOUCH_XM_MASK
  OUTPUT TOUCH_XM_DDR, R17

  INPUT R17, TOUCH_XM_PORT
  ANDI R17, ~TOUCH_XM_MASK
  OUTPUT TOUCH_XM_PORT, R17

  INPUT R17, TOUCH_YP_DDR
  ANDI R17, ~TOUCH_YP_MASK
  OUTPUT TOUCH_YP_DDR, R17

  INPUT R17, TOUCH_YP_PORT
  ANDI R17, ~TOUCH_YP_MASK
  OUTPUT TOUCH_YP_PORT, R17

  ; Get both analog readings
  ; z1 = ANALOG_READ xm (R25:R24)
  ; z2 = ANALOG_READ yp (R23:R22)
  LDI R25, TOUCH_X_ADC
  CALL ANALOG_READ
```

```
  LDI R23, HIGH(1024)
  LDI R22, LOW(1024)

  SUB R22, R24
  SBC R23, R25

  LDI R25, TOUCH_Y_ADC
  CALL ANALOG_READ
  LDI R17, HIGH(1024)
  LDI R16, LOW(1024)
  SUB R16, R24
  SBC R17, R25
  MOV R24, R16 ; R25:R24 = R17:R16

  POP R16
  POP R17
  RET

; Performs an scaling of the x coordinate returned
; from ANALOG_READ and maps it to an interval suited
; for the screen. The mapping range is empirical and
; based on tftpaint example. The mapping is:
; [120, 920] ---> [0, 239]
; Which can be done like this:
; x' = (x - 120) * 239/800
; The multiplication is approximated with 5/16.
; The x and x' values are 16 bit numbers passed and
; returned through registers R25:R24.
TFT_SCALE_COORD_X:
  PUSH R16
  PUSH R17
  ; x = x - 120
  LDI R16, LOW(120)
  LDI R17, HIGH(120)
  SUB R24, R16
  SBC R25, R17
  ; x = x * 5
  ; This can never go out of range
  LDI R16, 5
  MUL R24, R16
  MOV R24, R0
  MOV R17, R1
  MUL R25, R16
  MOV R25, R0
  ADD R25, R17
  ; x = x / 16 = x >> 4
  LSR R25
  ROR R24
  LSR R25
  ROR R24
  LSR R25
  ROR R24
  LSR R25
  ROR R24

  POP R17
  POP R16
  RET

; Performs an scaling of the y coordinate returned
; from ANALOG_READ and maps it to an interval suited
; for the screen. The mapping range is empirical and
; based on tftpaint example. The mapping is:
; [120, 940] ---> [0, 319]
; Which can be done like this:
; y' = (y - 120) * 319/820
; The multiplication is approximated with 3/8.
; The y and y' values are 16 bit numbers passed and
; returned through registers R25:R24.
TFT_SCALE_COORD_Y:
  PUSH R16
  PUSH R17
  ; y = y - 120
  LDI R16, LOW(120)
  LDI R17, HIGH(120)
  SUB R24, R16
  SBC R25, R17
  ; y = y * 3
  ; This can never go out of range
  LDI R16, 3
  MUL R24, R16
  MOV R24, R0
  MOV R17, R1
  MUL R25, R16
  MOV R25, R0
  ADD R25, R17
  ; y = y / 8 = y >> 3
  LSR R25
  ROR R24
  LSR R25
  ROR R24
  LSR R25
  ROR R24

  POP R17
  POP R16
  RET

TFT_READ_TOUCH:
  PUSH R18
  PUSH R19
  PUSH R20
  PUSH R21
  PUSH R22
  PUSH R23
  PUSH R24
  PUSH R25

  CALL TFT_READ_TOUCH_X
  MOVW R18, R24 ; R19:R18 = R25:R24

  CALL TFT_READ_TOUCH_Y
  MOVW R20, R24 ; R21:R20 = R25:R24

  CALL TFT_READ_TOUCH_Z
  CPI R24, 0x32
  BRSH tft_read_touch_valid
  CPI R22, 0x32
  BRSH tft_read_touch_valid

  LDI R25, 0x0
  STS TOUCH_Z_VALID, R25
  RJMP tft_read_touch_exit

tft_read_touch_valid:

  LDI R25, 0xFF
  STS TOUCH_Z_VALID, R25

  MOVW R24, R20 ; R25:R24 = R21:R20
  CALL TFT_SCALE_COORD_Y
  STS TOUCH_Y_HIGH, R25
  STS TOUCH_Y_LOW, R24

  MOVW R24, R18 ; R25:R24 = R19:R18
  CALL TFT_SCALE_COORD_X
  STS TOUCH_X_HIGH, R25
  STS TOUCH_X_LOW, R24

tft_read_touch_exit:
  POP R25
  POP R24
  POP R23
  POP R22
  POP R21
  POP R20
  POP R19
  POP R18
  RET

DSEG

TOUCH_X_HIGH: BYTE 1
TOUCH_X_LOW: BYTE 1
TOUCH_Y_HIGH: BYTE 1
TOUCH_Y_LOW: BYTE 1
TOUCH_Z_VALID: BYTE 1
```

games.S

```
  #include "randgen.S"
  #include "simon.S"
  #include "wires.S"
  #include "memory.S"
  #include "barcode.S"

  .GLOBAL PLAY_CURRENT_GAME
  .GLOBAL SETUP_GAMES
  .GLOBAL GET_STRIKES
  .GLOBAL INC_STRIKES
```

```
  .GLOBAL CHECK_STRIKES_REACHED_MAX
  .GLOBAL CHECK_BOMB_WAS_DEACTIVATED

  .EQU GAMES_AMOUNT, 4
  .EQU MAX_STRIKES_AMOUNT, 3
  .EQU STRIKES_LEDS_PORT, PORTG
  .EQU STRIKES_LEDS_DDR, DDRG

  ; Starting screen constants
  .EQU SS_BCK_COLOR, WHITE
  .EQU SS_TEXT_COLOR, BLUE_LIGHT
  .EQU SS_FONT_SIZE, 5
  .EQU SS_CHAR_W, (SS_FONT_SIZE * TFT_CHAR_W)
  .EQU SS_CHAR_H, (SS_FONT_SIZE * TFT_CHAR_H)
  .EQU SS_TEXT_CHARS, 5 ; "START" is the text
  .EQU SS_TEXT_Y, ((TFT_HEIGHT - SS_CHAR_H) / 2)
  .EQU SS_TEXT_X, ((TFT_WIDTH - SS_CHAR_W *
      SS_TEXT_CHARS) / 2)

  ; Defeat screen constants
  .EQU DS_BCK_COLOR, RED_LIGHT
  .EQU DS_TEXT_COLOR, WHITE
  .EQU DS_FONT_SIZE, 6
  .EQU DS_CHAR_W, (DS_FONT_SIZE * TFT_CHAR_W)
  .EQU DS_CHAR_H, (DS_FONT_SIZE * TFT_CHAR_H)
  .EQU DS_TEXT_CHARS, 5 ; "BOOM!" is the text
  .EQU DS_TEXT_Y, ((TFT_HEIGHT - DS_CHAR_H) / 2)
  .EQU DS_TEXT_X, ((TFT_WIDTH - DS_CHAR_W *
      DS_TEXT_CHARS) / 2)

  ; Victory screen constants
  .EQU VS_BCK_COLOR, GREEN_LIGHT
  .EQU VS_TEXT_COLOR, WHITE
  .EQU VS_FONT_SIZE, 4
  .EQU VS_CHAR_W, (VS_FONT_SIZE * TFT_CHAR_W)
  .EQU VS_CHAR_H, (VS_FONT_SIZE * TFT_CHAR_H)
  .EQU VS_TEXT_CHARS, 8 ; "VICTORY!" is the text
  .EQU VS_TEXT_Y, ((TFT_HEIGHT - VS_CHAR_H) / 2)
  .EQU VS_TEXT_X, ((TFT_WIDTH - VS_CHAR_W *
      VS_TEXT_CHARS) / 2)


  CSEG

; Initializes all minigames variables together with
; the strike counter. Should be called after
; setting up the random generator.
SETUP_GAMES:
  CALL SETUP_RANDGEN
  CLR R1
  STS strikes_counter, R1
  STS current_minigame, R1
  ; Setup LEDs for strikes
  INPUT R18, STRIKES_LEDS_DDR
  ORI R18, 0X07
  OUTPUT STRIKES_LEDS_DDR, R18
  INPUT R18, STRIKES_LEDS_PORT
  ANDI R18, 0xF8
  OUTPUT STRIKES_LEDS_PORT, R18
  ; Call setup for all minigames
  CALL WIRES_SETUP
  CALL MEMORY_SETUP
  CALL SIMON_SAYS_SETUP
  CALL BARCODE_SETUP
  RET

; Shows a nice starting screen with the message
; defined on the starting_screen_text bufffer. The
; screen vanishes after the player touches it.
; IMPORTANT: This function is blocking!
GAME_STARTING_SCREEN:
  PUSH R17
  PUSH R16
  PUSH R15
  ; First fill the screen with the right color
  LDI R25, HIGH(SS_BCK_COLOR)
  LDI R24, LOW(SS_BCK_COLOR)
  CALL TFT_FILL_SCREEN
  ; Now loop showing the starting_screen_text
  LDI ZL, LOW(starting_screen_text)
  LDI ZH, HIGH(starting_screen_text)
  LDI R16, SS_TEXT_CHARS
  MOV R15, R16 ; R15 is the iterator
  LDI R25, HIGH(SS_TEXT_X)
  LDI R24, LOW(SS_TEXT_X)
games_starting_screen_text_loop:
  LDI R23, HIGH(SS_TEXT_Y)
  LDI R22, LOW(SS_TEXT_Y)
  CLR R21
  LPM R20, Z+ ; Char in R20
  LDI R19, HIGH(SS_TEXT_COLOR)
  LDI R18, LOW(SS_TEXT_COLOR)
  LDI R17, HIGH(SS_FONT_SIZE)
  LDI R16, LOW(SS_FONT_SIZE)
  PUSH R25
  PUSH R24
  CALL TFT_DRAW_CHAR
  POP R24
  POP R25
  ADIW R24, SS_CHAR_W ; R25:R24 += SS_CHAR_W
  DEC R15
  BRNE games_starting_screen_text_loop
  ; Text was shown, so loop until screen is touched
games_starting_screen_touch_loop:
  CALL TFT_READ_TOUCH
  LDS R18, TOUCH_Z_VALID
  CPI R18, 0
  BREQ games_starting_screen_touch_loop
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_1
  CALL BUZZER_TONE_1
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  POP R17
  POP R16
  POP R15
  RET

; Shows a the defeat screen with the message
; defined on the defeat_screen_text bufffer. The
; screen vanishes after the player touches it.
; IMPORTANT: This function is blocking!
GAME_DEFEAT_SCREEN:
  PUSH R17
  PUSH R16
  PUSH R15
  ; First fill the screen with the right color
  LDI R25, HIGH(DS_BCK_COLOR)
  LDI R24, LOW(DS_BCK_COLOR)
  CALL TFT_FILL_SCREEN
  ; Boom explosion sound (?)
  CALL BUZZER_EXPLOSION_FANFARE
  ; Now loop showing the defeat_screen_text
  LDI ZL, LOW(defeat_screen_text)
  LDI ZH, HIGH(defeat_screen_text)
  LDI R16, DS_TEXT_CHARS
  MOV R15, R16 ; R15 is the iterator
  LDI R25, HIGH(DS_TEXT_X)
  LDI R24, LOW(DS_TEXT_X)
games_defeat_screen_text_loop:
  LDI R23, HIGH(DS_TEXT_Y)
  LDI R22, LOW(DS_TEXT_Y)
  CLR R21
  LPM R20, Z+ ; Char in R20
  LDI R19, HIGH(DS_TEXT_COLOR)
  LDI R18, LOW(DS_TEXT_COLOR)
  LDI R17, HIGH(DS_FONT_SIZE)
  LDI R16, LOW(DS_FONT_SIZE)
  PUSH R25
  PUSH R24
  CALL TFT_DRAW_CHAR
  POP R24
  POP R25
  ADIW R24, DS_CHAR_W ; R25:R24 += DS_CHAR_W
  DEC R15
  BRNE games_defeat_screen_text_loop
  ; Text was shown, so loop until screen is touched
games_defeat_screen_touch_loop:
  CALL TFT_READ_TOUCH
  LDS R18, TOUCH_Z_VALID
  CPI R18, 0
  BREQ games_defeat_screen_touch_loop
  POP R17
  POP R16
  POP R15
  RET

; Shows a the victory screen with the message
; defined on the victory_screen_text bufffer. The
```

```
; screen vanishes after the player touches it.
; IMPORTANT: This function is blocking!
GAME_VICTORY_SCREEN:
  PUSH R17
  PUSH R16
  PUSH R15
  ; First fill the screen with the right color
  LDI R25, HIGH(VS_BCK_COLOR)
  LDI R24, LOW(VS_BCK_COLOR)
  CALL TFT_FILL_SCREEN
  ; Victory sound!
  CALL BUZZER_DEACTIVATION_FANFARE
  ; Now loop showing the victory_screen_text
  LDI ZL, LOW(victory_screen_text)
  LDI ZH, HIGH(victory_screen_text)
  LDI R16, VS_TEXT_CHARS
  MOV R15, R16 ; R15 is the iterator
  LDI R25, HIGH(VS_TEXT_X)
  LDI R24, LOW(VS_TEXT_X)
games_victory_screen_text_loop:
  LDI R23, HIGH(VS_TEXT_Y)
  LDI R22, LOW(VS_TEXT_Y)
  CLR R21
  LPM R20, Z+ ; Char in R20
  LDI R19, HIGH(VS_TEXT_COLOR)
  LDI R18, LOW(VS_TEXT_COLOR)
  LDI R17, HIGH(VS_FONT_SIZE)
  LDI R16, LOW(VS_FONT_SIZE)
  PUSH R25
  PUSH R24
  CALL TFT_DRAW_CHAR
  POP R24
  POP R25
  ADIW R24, VS_CHAR_W ; R25:R24 += VS_CHAR_W
  DEC R15
  BRNE games_victory_screen_text_loop
  ; Text was shown, so loop until screen is touched
games_victory_screen_touch_loop:
  CALL TFT_READ_TOUCH
  LDS R18, TOUCH_Z_VALID
  CPI R18, 0
  BREQ games_victory_screen_touch_loop
  POP R17
  POP R16
  POP R15
  RET

; Increases by 1 the number of strikes of the player
; during this game.
INC_STRIKES:
  LDS R18, strikes_counter
  INC R18
  STS strikes_counter, R18
  ; Turn on LEDs strikes
  INPUT R18, STRIKES_LEDS_PORT
  MOV R19, R18
  ANDI R19, 0X07
  LSL R19
  INC R19
  ANDI R19, 0X07
  OR R18, R19
  OUTPUT STRIKES_LEDS_PORT, R18
  CALL BUZZER_DEFEAT_FANFARE
  RET

; Returns on R24 the number of strikes of this game.
GET_STRIKES:
  LDS R24, strikes_counter
  RET

; Checks whether or not the number of strikes has
; reached the MAX_STRIKES_AMOUNT value (i.e. if the
; game is over or not). Returns true or false (as 1
; or 0) on R24.
CHECK_STRIKES_REACHED_MAX:
  CLR R24
  LDS R18, strikes_counter
  CPI R18, MAX_STRIKES_AMOUNT
  BRLO check_strikes_ret
  INC R24
check_strikes_ret:
  RET

; Checks if the bomb was deactivated by clearing all
; the minigames on it. That happens when the
    variable
; current_minigame reaches MINIGAMES_AMOUNT value.
; The function returns true or false (1 or 0) on R24
    .
CHECK_BOMB_WAS_DEACTIVATED:
  CLR R24
  LDS R18, current_minigame
  CPI R18, GAMES_AMOUNT
  BRLO check_deactivated_ret
  INC R24
check_deactivated_ret:
  RET

; REALLY THE MOST IMPORTANT FUNCTION OF THE PROJECT!
; This function allows the user to play the current
; minigame in a non-blocking manner, and advances to
; the next minigame when it is finished by
    increasing
; the current_minigame variable. In order for this,
; all minigames must implement these two functions:
; * <MINIGAME NAME>_PLAY : allows the user to play
; that minigame in a NON-BLOCKING way. The easiest
; way to achieve this is by simply asking if the
; player has touched the screen and returning if not
    .
; * <MINIGAME NAME>_FINISHED : must store on R24 a
; true or false value, telling if the player has or
; not cleared that game.
PLAY_CURRENT_GAME:
  PUSH R16
  CLR R1
  ; Load in R16 the current minigame number
  LDS R16, current_minigame
  CPI R16, 0
  BREQ minigame_0
  CPI R16, 1
  BREQ minigame_1
  CPI R16, 2
  BREQ minigame_2
  CPI R16, 3
  BREQ minigame_3
  ; SHOULD NOT HAPPEN!
  RET

check_minigame_finished:
  ; R16 still has the current minigame. If R24 is
  CPI R24, 0 ; 1, the current minigame was cleared
  BREQ keep_playing_current_minigame
  ; Current minigame was cleared if here
  INC R16
  STS current_minigame, R16
  CALL BUZZER_VICTORY_FANFARE
keep_playing_current_minigame:
  POP R16
  RET

; ------------- Games sorted by number -------------

minigame_0:
  CALL SIMON_SAYS_PLAY
  CALL SIMON_SAYS_FINISHED
  RJMP check_minigame_finished

minigame_1:
  CALL BARCODE_PLAY
  CALL BARCODE_FINISHED
  RJMP check_minigame_finished

minigame_2:
  CALL MEMORY_PLAY
  CALL MEMORY_FINISHED
  RJMP check_minigame_finished

minigame_3:
  CALL WIRES_PLAY
  CALL WIRES_FINISHED
  RJMP check_minigame_finished

; Text to show on the starting screen
starting_screen_text:
  DB 'S', 'T', 'A', 'R', 'T', 0x00
; Text to show on the defeat screen
defeat_screen_text:
  DB 'B', 'O', 'O', 'M', '!', 0x00
; Text to show on the victory screen
```

```
victory_screen_text:
  DB 'V', 'I', 'C', 'T', 'O', 'R', 'Y', '!'

  DSEG
; Number of strikes during this game
strikes_counter:
  BYTE 1
; Current game counter
current_minigame:
  BYTE 1
```

buzzer.S

```
  .GLOBAL SETUP_BUZZER
  .GLOBAL RANDGEN

  .EQU BUZZER_PORT, PORTD
  .EQU BUZZER_DDR, DDRD
  .EQU BUZZER_PIN_NUMBER, 0

CSEG

; Sets the buzzer pin as output and clears it.
SETUP_BUZZER:
  SBI BUZZER_DDR, BUZZER_PIN_NUMBER
  CBI BUZZER_PORT, BUZZER_PIN_NUMBER
  RET

; Plays a tone on the buzzer for some time. The
; 'on' and 'off' times of the buzzer in microseconds
; are received on R24 and R22 respectively.
BUZZER_PLAY:
  SBI BUZZER_PORT, BUZZER_PIN_NUMBER
  MOV R25, R24
  CALL HALF_MICRO_DELAY
  MOV R25, R24
  CALL HALF_MICRO_DELAY
  CBI BUZZER_PORT, BUZZER_PIN_NUMBER
  MOV R25, R22
  CALL HALF_MICRO_DELAY
  MOV R25, R22
  CALL HALF_MICRO_DELAY
  RET

; Plays a tone on the buzzer that will last for
; (R24 + R22) * R20 * R18 microseconds. In the
; bottom, R24 and R22 will directly map to the 'on'
; and 'off' times of the buzzer, so they should be
; equal for a duty cycle of 50%.
BUZZER_TONE_GENERIC:
  PUSH R13
  PUSH R14
  PUSH R15
  PUSH R16
  PUSH R17
  MOV R17, R24
  MOV R16, R22
  MOV R15, R20
  MOV R14, R18
buzzer_tone_loop_1:
  MOV R13, R15
buzzer_tone_loop_2:
  MOV R24, R17
  MOV R22, R16
  CALL BUZZER_PLAY
  DEC R13
  BRNE buzzer_tone_loop_2
  DEC R14
  BRNE buzzer_tone_loop_1
  POP R17
  POP R16
  POP R15
  POP R14
  POP R13
  RET

; Buzzer tone functions to be called on minigames

BUZZER_TONE_1:
  LDI R24, 220
  LDI R22, 220
  LDI R20, 180
  LDI R18, 1
  CALL BUZZER_TONE_GENERIC
  RET

BUZZER_TONE_2:
  LDI R24, 200
  LDI R22, 200
  LDI R20, 198
  LDI R18, 1
  CALL BUZZER_TONE_GENERIC
  RET

BUZZER_TONE_3:
  LDI R24, 180
  LDI R22, 180
  LDI R20, 110
  LDI R18, 2
  CALL BUZZER_TONE_GENERIC
  RET

BUZZER_TONE_4:
  LDI R24, 160
  LDI R22, 160
  LDI R20, 123
  LDI R18, 2
  CALL BUZZER_TONE_GENERIC
  RET

BUZZER_TONE_5:
  LDI R24, 120
  LDI R22, 120
  LDI R20, 115
  LDI R18, 2
  CALL BUZZER_TONE_GENERIC
  RET

BUZZER_TONE_6:
  LDI R24, 80
  LDI R22, 80
  LDI R20, 165
  LDI R18, 3
  CALL BUZZER_TONE_GENERIC
  RET

; Plays a victory fanfare with the buzzer.
BUZZER_VICTORY_FANFARE:
  CALL BUZZER_TONE_3
  CALL BUZZER_TONE_3
  CALL BUZZER_TONE_4
  CALL BUZZER_TONE_4
  CALL BUZZER_TONE_5
  RET

; Plays a defeat fanfare with the buzzer.
BUZZER_DEFEAT_FANFARE:
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_1
  CALL BUZZER_TONE_1
  RET

BUZZER_DELAY:
  PUSH ZL
  PUSH ZH
  PUSH R16
  PUSH R17
  PUSH R18
  PUSH R19
  PUSH R20
  LDI R16, 15
buzzer_delay_loop:
  CALL SHOW_DISPLAYS
  DEC R16
  BRNE buzzer_delay_loop
  POP R20
  POP R19
  POP R18
  POP R17
  POP R16
  POP ZH
  POP ZL
  RET

BUZZER_EXPLOSION_FANFARE:
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
```

```
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_DEFEAT_FANFARE
  CALL BUZZER_DELAY
  CALL BUZZER_TONE_1
  CALL BUZZER_TONE_1
  CALL BUZZER_TONE_1
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_DELAY
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_2
  CALL BUZZER_TONE_1
  CALL BUZZER_TONE_1
  CALL BUZZER_DEFEAT_FANFARE
  RET

BUZZER_DEACTIVATION_FANFARE:
  CALL BUZZER_TONE_3
  CALL BUZZER_TONE_3
  CALL BUZZER_TONE_3
  CALL BUZZER_TONE_3
  CALL BUZZER_DELAY
  CALL BUZZER_TONE_4
  CALL BUZZER_TONE_4
  CALL BUZZER_TONE_4
  CALL BUZZER_DELAY
  CALL BUZZER_TONE_3
  CALL BUZZER_TONE_3
  CALL BUZZER_DELAY
  CALL BUZZER_TONE_5
  CALL BUZZER_TONE_5
  CALL BUZZER_DELAY
  CALL BUZZER_TONE_4
  CALL BUZZER_TONE_4
  CALL BUZZER_DELAY

  CALL BUZZER_VICTORY_FANFARE
  RET
```

randgen.S

```
  .GLOBAL SETUP_RANDGEN
  .GLOBAL RANDGEN

  .EQU RANDGEN_ITERATIONS, 5
  .EQU RANDGEN_MAGIC, 0xAA

CSEG

; Setups randgen_buffer variable for using RANDGEN.
; Basically, loads the RANDGEN_MAGIC value to the
; randgen_buffer (so it aint start with a 0 value).
SETUP_RANDGEN:
  LDI R18, RANDGEN_MAGIC
  STS randgen_buffer, R18
  RET

; Returns a random N bits number on R24. The value
; of N is received on R24. It uses timer0 register
; value as an entropy source, looping
; RANDGEN_ITERATIONS times performing some XORs. The
; randgen_buffer is used for persisting the random
; values through calls.
RANDGEN:
  ; IF N > 8 or N == 0, panic (return 0)
  MOV R22, R24
  CLR R24
  CPI R22, 8
  BRSH randgen_panic
  CPI R22, 0
  BREQ randgen_panic
  ; Load the randgen_buffer and do randomness
  LDS R18, randgen_buffer
  ; Load timer 0 value and XOR it with the
  ; previous randgen_buffer value
  INPUT R19, TCNT0
  EOR R18, R19

  ; Now do RANDGEN_ITERATIONS XORs
  CLR R19
randgen_loop:
  CLR R20
  CLR R21
  SBRS R18, 7
  INC R20
  SBRS R18, 6
  INC R21
  EOR R21, R20
  LSL R18
  ADD R18, R21

  INC R19
  CPI R19, RANDGEN_ITERATIONS
  BRNE randgen_loop

  STS randgen_buffer, R18
  ; Since we must return a N bits random number
  ; we must do R18 % (2**N). Note this is the
  ; same than taking the least N significative
  ; bits from R18 register.
mask_loop:
  LSL R24
  INC R24
  DEC R22
  BRNE mask_loop
  ; R24 is now a mask of N bits
  AND R24, R18
randgen_panic:
  RET

  DSEG
; Randgen buffer for generating random numbers
randgen_buffer:
  BYTE 1
```

simon.S

```
  .GLOBAL SIMON_SAYS_SETUP
  .GLOBAL SIMON_SAYS_PLAY
  .GLOBAL SIMON_SAYS_FINISHED
  ; 2**SIMON_COLORS_BITS = SIMON_COLORS
  .EQU SIMON_COLORS, 4
  .EQU SIMON_COLORS_BITS, 2
  .EQU SIMON_SAYS_ROUNDS, 5

  ; Screen constants for Simon says
  .EQU SIMON_RECTS_WIDTH, (TFT_WIDTH / 2)
  .EQU SIMON_RECTS_HEIGHT, (TFT_HEIGHT / 2)

CSEG

; Configures the Simon says minigame by choosing the
; first random color and storing it. Once the Simon
; says minigame is started, the color will be shown.
SIMON_SAYS_SETUP:
  CLR R1
  STS simon_chosen_colors_amount, R1
  STS simon_player_last_color, R1
  STS simon_game_cleared, R1
  CALL SIMON_SAYS_PICK_NEXT_COLOR
  RET

; Picks the next correct color the player should
; guess, and stores it in the correct address of the
; simon_correct_colors buffer. It also sets the flag
; for showing the colors on the next iteration and
; increases the chosen colors variable.
SIMON_SAYS_PICK_NEXT_COLOR:
  ; If simon_chosen_colors_amount has already
  ; reached SIMON_SAYS_ROUNDS, then the game
  ; has been cleared
  LDS R18, simon_chosen_colors_amount
  CPI R18, SIMON_SAYS_ROUNDS
  BRNE simon_continue_choosing_colors
  LDI R18, 1
  STS simon_game_cleared, R18
  RET

simon_continue_choosing_colors:
  LDI R24, SIMON_COLORS_BITS
  CALL RANDGEN
  ; R24 has now a random color to store
  CLR R1
  LDS R18, simon_chosen_colors_amount
  LDI ZL, LOW(simon_correct_colors)
  LDI ZH, HIGH(simon_correct_colors)
```

```
  ADD ZL, R18
  ADC ZH, R1
  ; The storing address for R24 is pointed by Z
  ST Z, R24
  ; Increase the chosen colors amount
  INC R18
  STS simon_chosen_colors_amount, R18
  STS simon_player_last_color, R1
  ; Set the boolean for showing the colors
  LDI R18, 1
  STS simon_show_colors, R18
  RET

; Blink in white a corner of the Simon says.
; Receives the (x,y) coordinates of the upper-left
; button point on R25:R24 and R23:R22.
SIMON_SAYS_BLINK_BUTTON:
  PUSH R17
  PUSH R16
  LDI R21, HIGH(SIMON_RECTS_WIDTH)
  LDI R20, LOW(SIMON_RECTS_WIDTH)
  LDI R19, HIGH(SIMON_RECTS_HEIGHT)
  LDI R18, LOW(SIMON_RECTS_HEIGHT)
  LDI R17, HIGH(WHITE)
  LDI R16, LOW(WHITE)
  CALL TFT_FILL_RECT_HW
  CALL SIMON_SAYS_DELAY
  CALL SIMON_SAYS_SHOW_SCREEN
  POP R16
  POP R17
  RET

; Shows SIMON_COLORS coloured rectangles on screen.
; They should be arranged in a matrix according to
; SIMON_SCREEN_ROWS and SIMON_SCREEN_COLS.
SIMON_SAYS_SHOW_SCREEN:
  PUSH R16
  PUSH R17
  CLR R24
  CALL SIMON_SAYS_MAP_BUTTON_TO_COORDS
  LDI R21, HIGH(SIMON_RECTS_WIDTH)
  LDI R20, LOW(SIMON_RECTS_WIDTH)
  LDI R19, HIGH(SIMON_RECTS_HEIGHT)
  LDI R18, LOW(SIMON_RECTS_HEIGHT)
  LDI R17, HIGH(GREEN_DARK)
  LDI R16, LOW(GREEN_DARK)
  CALL TFT_FILL_RECT_HW
  LDI R24, 1
  CALL SIMON_SAYS_MAP_BUTTON_TO_COORDS
  LDI R21, HIGH(SIMON_RECTS_WIDTH)
  LDI R20, LOW(SIMON_RECTS_WIDTH)
  LDI R19, HIGH(SIMON_RECTS_HEIGHT)
  LDI R18, LOW(SIMON_RECTS_HEIGHT)
  LDI R17, HIGH(RED_LIGHT)
  LDI R16, LOW(RED_LIGHT)
  CALL TFT_FILL_RECT_HW
  LDI R24, 2
  CALL SIMON_SAYS_MAP_BUTTON_TO_COORDS
  LDI R21, HIGH(SIMON_RECTS_WIDTH)
  LDI R20, LOW(SIMON_RECTS_WIDTH)
  LDI R19, HIGH(SIMON_RECTS_HEIGHT)
  LDI R18, LOW(SIMON_RECTS_HEIGHT)
  LDI R17, HIGH(MAGENTA)
  LDI R16, LOW(MAGENTA)
  CALL TFT_FILL_RECT_HW
  LDI R24, 3
  CALL SIMON_SAYS_MAP_BUTTON_TO_COORDS
  LDI R21, HIGH(SIMON_RECTS_WIDTH)
  LDI R20, LOW(SIMON_RECTS_WIDTH)
  LDI R19, HIGH(SIMON_RECTS_HEIGHT)
  LDI R18, LOW(SIMON_RECTS_HEIGHT)
  LDI R17, HIGH(BLUE_DARK)
  LDI R16, LOW(BLUE_DARK)
  CALL TFT_FILL_RECT_HW
  POP R17
  POP R16
  RET

; Returns on R25:R24 the (x,y) coordinates of the
; upper-left point of a button. The button number is
; received on R24.
SIMON_SAYS_MAP_BUTTON_TO_COORDS:
  CPI R24, 0
  BRNE simon_test_one
  CLR R25
  CLR R24
  CLR R23
  CLR R22
  RET
simon_test_one:
  CPI R24, 1
  BRNE simon_test_two
  LDI R25, HIGH(SIMON_RECTS_WIDTH)
  LDI R24, LOW(SIMON_RECTS_WIDTH)
  CLR R23
  CLR R22
  RET
simon_test_two:
  CPI R24, 2
  BRNE simon_test_three
  CLR R25
  CLR R24
  LDI R23, HIGH(SIMON_RECTS_HEIGHT)
  LDI R22, LOW(SIMON_RECTS_HEIGHT)
  RET
simon_test_three:
  LDI R25, HIGH(SIMON_RECTS_WIDTH)
  LDI R24, LOW(SIMON_RECTS_WIDTH)
  LDI R23, HIGH(SIMON_RECTS_HEIGHT)
  LDI R22, LOW(SIMON_RECTS_HEIGHT)
  RET

; Delay used by the Simon says minigame after
; blinking a color.
SIMON_SAYS_DELAY:
  PUSH ZL
  PUSH ZH
  PUSH R16
  PUSH R17
  PUSH R18
  PUSH R19
  PUSH R20
  LDI R16, 25
simon_says_delay_loop:
  CALL SHOW_DISPLAYS
  DEC R16
  BRNE simon_says_delay_loop
  POP R20
  POP R19
  POP R18
  POP R17
  POP R16
  POP ZH
  POP ZL
  RET

; Shows the correct colors chosen so far in sequence
; by highlighting them on the screen.
SIMON_SAYS_SHOW_RIGHT_COLORS:
  PUSH R16
  CALL SIMON_SAYS_SHOW_SCREEN
  CALL SIMON_SAYS_DELAY
  LDS R16, simon_chosen_colors_amount
  LDI ZL, LOW(simon_correct_colors)
  LDI ZH, HIGH(simon_correct_colors)
simon_show_colors_loop:
  LD R24, Z+
  CALL SIMON_SAYS_PLAY_TONE
  CALL SIMON_SAYS_MAP_BUTTON_TO_COORDS
  CALL SIMON_SAYS_BLINK_BUTTON
  CALL SIMON_SAYS_DELAY
  DEC R16
  BRNE simon_show_colors_loop
  ; Clear the show colors flag
  CLR R1
  STS simon_show_colors, R1
  POP R16
  RET

; Retrieves the button number of a touched button
; on screen by reading the TFT touched x,y values.
; Note: this function assumes the player has indeed
; touched the screen (i.e. you shoudl validate the
; z coordinate before calling this function).
  Returns
; the pressed button number on R24.
SIMON_SAYS_MAP_TOUCH_TO_BUTTON:
  CLR R24
  LDS R19, TOUCH_Y_HIGH
  LDS R18, TOUCH_Y_LOW
```

```
  ROR R19
  ROR R18
  CPI R18, (TFT_HEIGHT / 4)
  BRLO simon_touch_test_x
  LDI R24, 2
simon_touch_test_x:
  LDS R19, TOUCH_X_HIGH
  LDS R18, TOUCH_X_LOW
  ROR R19
  ROR R18
  CPI R18, (TFT_WIDTH / 4)
  BRLO simon_touch_ret
  INC R24
simon_touch_ret:
  RET

; Performs the magic conversion of blinked to
; touch button of the game by using the
; simon_touch_to_blink_table. The initial and final
; color values are received/returned on R24.
SIMON_SAYS_MAP_COLOR_BLINK_TO_TOUCH:
  MOV R18, R24
  CALL GET_STRIKES
  LSL R24
  LSL R24
  ADD R24, R18
  LDI ZL, LOW(simon_touch_to_blink_table)
  LDI ZH, HIGH(simon_touch_to_blink_table)
  ADD ZL, R24
  ADC ZH, R1
  LPM R24, Z
  RET

; Pretty function that plays a different buzzer tone
; according to the button touched. The button is
; received on R24. The function also returns the
; same button number on R24.
SIMON_SAYS_PLAY_TONE:
  PUSH R24
  CPI R24, 0
  BREQ simon_tone_0
  CPI R24, 1
  BREQ simon_tone_1
  CPI R24, 2
  BREQ simon_tone_2
  CPI R24, 3
  BREQ simon_tone_3
simon_tone_0:
  CALL BUZZER_TONE_1
  POP R24
  RET
simon_tone_1:
  CALL BUZZER_TONE_2
  POP R24
  RET
simon_tone_2:
  CALL BUZZER_TONE_3
  POP R24
  RET
simon_tone_3:
  CALL BUZZER_TONE_5
  POP R24
  RET

; Plays the simon says minigame in a non-blocking
; way by checking if the player has touched a button
; or not. Every time a player touches a button, the
; function determinates whether or not the button
    was
; the correct one, and makes progress on the
    minigame
; by calling SIMON_SAYS_PICK_NEXT_COLOR if necessary
    .
SIMON_SAYS_PLAY:
  ; Check if the right colors should be shown
  LDS R18, simon_show_colors
  CPI R18, 1
  BREQ show_colors_and_exit
  ; Now check if the user pressed a color
  CALL TFT_READ_TOUCH
  LDS R18, TOUCH_Z_VALID
  CPI R18, 0
  BREQ simon_play_no_button_touched
  ; If here, the touchscreen was pressed
  CALL SIMON_SAYS_MAP_TOUCH_TO_BUTTON
  PUSH R24
  CALL SIMON_SAYS_PLAY_TONE
  CALL SIMON_SAYS_MAP_BUTTON_TO_COORDS
  CALL SIMON_SAYS_BLINK_BUTTON
  POP R24
  CALL SIMON_SAYS_MAP_COLOR_BLINK_TO_TOUCH
  CLR R1
  LDS R19, simon_player_last_color
  LDI ZL, LOW(simon_correct_colors)
  LDI ZH, HIGH(simon_correct_colors)
  ADD ZL, R19
  ADC ZH, R1
  LD R18, Z
  CP R18, R24
  BREQ simon_play_good_color_touched
  ; If here, player has touched a bad color
  CALL INC_STRIKES
  CALL SIMON_SAYS_SETUP
  RET
simon_play_good_color_touched:
  ; If here, player has touched a good color
  LDS R18, simon_chosen_colors_amount
  LDS R19, simon_player_last_color
  INC R19
  CP R18, R19
  BREQ simon_play_pick_another_color
  STS simon_player_last_color, R19
  CALL SIMON_SAYS_DELAY
  RET

simon_play_pick_another_color:
  CALL SIMON_SAYS_PICK_NEXT_COLOR
  RET

show_colors_and_exit:
  CALL SIMON_SAYS_SHOW_RIGHT_COLORS
simon_play_no_button_touched:
  RET

; Returns true (as 1) on R24 if the simon says
; minigame was cleared by the player. If it hasn't
; already, returns false (as 0).
SIMON_SAYS_FINISHED:
  LDS R24, simon_game_cleared
  RET

; Tables used for the magic mapping of this game
simon_touch_to_blink_table:
  ; Cases when strikes are 0
  DB 2, 3, 0, 1
  ; Cases when strikes are 1
  DB 3, 2, 1, 0
  ; Cases when strikes are 2
  DB 1, 3, 0, 2

  DSEG
  ; Buffer for storing the correct colors
simon_correct_colors:
  BYTE SIMON_SAYS_ROUNDS
  ; Number of colors shown so far
simon_chosen_colors_amount:
  BYTE 1
  ; Last number of color pressed by player
simon_player_last_color:
  BYTE 1
  ; Boolean to determinate whether or not the
  ; correct colors should be shown on the
  ; next minigame iteration
simon_show_colors:
  BYTE 1
  ; Boolean to set when minigame finishes
simon_game_cleared:
  BYTE 1
```

barcode.S

```
  .GLOBAL BARCODE_SETUP
  .GLOBAL BARCODE_PLAY
  .GLOBAL BARCODE_FINISHED

  .EQU BARCODE_BUTTONS, 4

  ; 2**BARCODE_COMBS is the amount of possible
  ; combinations for this minigame
  .EQU BARCODE_COMBS, 4
```

```
  ; Entry size in the barcode_combinations table
  .EQU BARCODE_COMB_SIZE, 2

  ; Memory screen constants
  .EQU BARCODE_BORDER, 30
  .EQU BARCODE_BARS_AMOUNT, 10
  .EQU BARCODE_BAR_H, ((TFT_HEIGHT - 3 *
      BARCODE_BORDER) / BARCODE_BARS_AMOUNT)

CSEG

; Randomly picks one of the possible entries on the
; barcode_combinations table and stores that data on
; the barcode_bars_buffer variable.
BARCODE_SETUP:
  LDI R24, BARCODE_COMBS
  CALL RANDGEN
  ; R24 now has a random number between 0 and
  ; 2**BARCODE_COMBS-1, so we use it to select
  ; one entry in the barcode_combinations table
  LDI ZL, LOW(barcode_combinations)
  LDI ZH, HIGH(barcode_combinations)
  LDI R18, BARCODE_COMB_SIZE
  MUL R24, R18
  ADD ZL, R0
  ADC ZH, R1
  CLR R1
  ; Now that Z is pointing to some entry on
  ; the table, load that entry's values into
  ; the variables we will be using
  LDI XL, LOW(barcode_bars_buffer)
  LDI XH, HIGH(barcode_bars_buffer)
  LDI R19, BARCODE_COMB_SIZE
barcode_setup_loop:
  LPM R18, Z+
  ST X+, R18
  DEC R19
  BRNE barcode_setup_loop
  ; Tell the game to show the screen once
  LDI R18, 1
  STS barcode_need_show_screen, R18
  ; Finally, clear the game cleared flag
  STS barcode_game_cleared, R1
  RET

; Plays the barcode minigame in a non-blocking way
    by
; just asking if the screen was touched. Only when a
; the screen is touched, performs a check to see if
; it was on the correct place or not.
BARCODE_PLAY:
  ; Check if the game needs to show the screen
  LDS R18, barcode_need_show_screen
  CPI R18, 0
  BREQ barcode_check_button_touched
  CALL BARCODE_SHOW_SCREEN
  CLR R1
  STS barcode_need_show_screen, R1
barcode_check_button_touched:
  CALL TFT_READ_TOUCH
  LDS R18, TOUCH_Z_VALID
  CPI R18, 0
  BREQ barcode_no_button_touched
  ; If here, the touchscreen was pressed
  CALL BARCODE_MAP_TOUCH_TO_BUTTON
  ; Check if player touched the proper button
  ; Point at the current stage correct button
  LDI ZL, LOW(barcode_bars_buffer +
      BARCODE_COMB_SIZE - 1)
  LDI ZH, HIGH(barcode_bars_buffer +
      BARCODE_COMB_SIZE - 1)
  LD R18, Z
  ANDI R18, 0X0F; R18 is the correct button
  CP R18, R24
  BREQ barcode_play_good_button_touched
  ; If here, player has touched a bad button
  CALL INC_STRIKES
  CALL BARCODE_SETUP
  RET
barcode_play_good_button_touched:
  LDI R18, 1
  STS barcode_game_cleared, R18
barcode_no_button_touched:
  RET

; Returns true (as 1) on R24 if the barcode minigame
; was cleared by the player. If it hasn't already,
; returns false (as 0).
BARCODE_FINISHED:
  LDS R24, barcode_game_cleared
  RET

; Retrieves the button number of a touched button
; on screen by reading the TFT touched x,y values.
; Note: this function assumes the player has indeed
; touched the screen (i.e. you should validate the
; z coordinate before calling this function).
    Returns
; the button number on R24. Stolen from Simon says.
BARCODE_MAP_TOUCH_TO_BUTTON:
  CLR R24
  LDS R19, TOUCH_Y_HIGH
  LDS R18, TOUCH_Y_LOW
  ROR R19
  ROR R18
  CPI R18, (TFT_HEIGHT / 4)
  BRLO barcode_touch_test_x
  LDI R24, 2
barcode_touch_test_x:
  LDS R19, TOUCH_X_HIGH
  LDS R18, TOUCH_X_LOW
  ROR R19
  ROR R18
  CPI R18, (TFT_WIDTH / 4)
  BRLO barcode_touch_ret
  INC R24
barcode_touch_ret:
  RET

; Shows the barcode on screen
BARCODE_SHOW_SCREEN:
  PUSH R17
  PUSH R16
  PUSH R15
  ; First fill screen with white
  LDI R25, HIGH(WHITE)
  LDI R24, LOW(WHITE)
  CALL TFT_FILL_SCREEN
  ; Draw two small squares at the bottom
  CLR R25
  CLR R24
  LDI R23, HIGH(TFT_HEIGHT - BARCODE_BORDER)
  LDI R22, LOW(TFT_HEIGHT - BARCODE_BORDER)
  LDI R21, HIGH(BARCODE_BORDER)
  LDI R20, LOW(BARCODE_BORDER)
  LDI R19, HIGH(BARCODE_BORDER)
  LDI R18, LOW(BARCODE_BORDER)
  LDI R17, HIGH(BLACK)
  LDI R16, HIGH(BLACK)
  CALL TFT_FILL_RECT_HW
  LDI R25, HIGH(TFT_WIDTH - BARCODE_BORDER)
  LDI R24, LOW(TFT_WIDTH - BARCODE_BORDER)
  LDI R23, HIGH(TFT_HEIGHT - BARCODE_BORDER)
  LDI R22, LOW(TFT_HEIGHT - BARCODE_BORDER)
  LDI R21, HIGH(BARCODE_BORDER)
  LDI R20, LOW(BARCODE_BORDER)
  LDI R19, HIGH(BARCODE_BORDER)
  LDI R18, LOW(BARCODE_BORDER)
  LDI R17, HIGH(BLACK)
  LDI R16, HIGH(BLACK)
  CALL TFT_FILL_RECT_HW
  ; Point at the barcode_bars_buffer and prepare
  ; to loop for BARCODE_BARS_AMOUNT times
  LDI ZL, LOW(barcode_bars_buffer)
  LDI ZH, HIGH(barcode_bars_buffer)
  LDI R16, BARCODE_BARS_AMOUNT
  LDI R22, LOW(BARCODE_BORDER)
  LDI R23, HIGH(BARCODE_BORDER)
barcode_show_screen_byte_loop:
  CLR R17; R17 is a bit counter
  LD R15, Z+
barcode_show_screen_bit_loop:
  CPI R17, 8
  BREQ barcode_show_screen_byte_loop
  ; If the 7th bit of R15 is 1, show a black bar
  SBRC R15, 7
  RJMP bardcode_show_screen_bar
bardcode_show_screen_next_bit:
  LDI R25, HIGH(BARCODE_BAR_H)
  LDI R24, LOW(BARCODE_BAR_H)
```

```
  ADD R22, R24
  ADC R23, R25
  LSL R15
  INC R17
  DEC R16
  BRNE barcode_show_screen_bit_loop
  POP R15
  POP R16
  POP R17
  RET

bardcode_show_screen_bar:
  ; Show a black horizontal bar
  CLR R25
  CLR R24
  LDI R21, HIGH(TFT_WIDTH)
  LDI R20, LOW(TFT_WIDTH)
  LDI R19, HIGH(BARCODE_BAR_H)
  LDI R18, LOW(BARCODE_BAR_H)
  PUSH R17
  PUSH R16
  PUSH R22
  PUSH R23
  LDI R17, HIGH(BLACK)
  LDI R16, HIGH(BLACK)
  CALL TFT_FILL_RECT_HW
  POP R23
  POP R22
  POP R16
  POP R17
  RJMP bardcode_show_screen_next_bit

; The following is a list of the barcode
    combinations
; (i.e. every possible scenario that could happen).
; The format for listing them is:
; a stream of bits (1 for black bar, 0 for white one
  )
; plus 4 bits for the right button to be touched.
barcode_combinations:
  DB 0x5A, 0x80
  DB 0xAD, 0X41
  DB 0xD5, 0X40
  DB 0xB5, 0X43
  DB 0xA6, 0xC0
  DB 0xDA, 0x82
  DB 0xAA, 0X80
  DB 0xB6, 0XC2
  DB 0x5B, 0x40
  DB 0xD5, 0X03
  DB 0xAB, 0X42
  DB 0xAA, 0XC3
  DB 0xD6, 0x82
  DB 0xDA, 0XC2
  DB 0xDB, 0X42
  DB 0x6A, 0x83

  DSEG
  BYTE 1
  ; Buffer for allocating the bars data
barcode_bars_buffer:
  BYTE BARCODE_COMB_SIZE
  ; Boolean to set when minigame finishes
barcode_game_cleared:
  BYTE 1
  ; Boolean to set when need to show screen
barcode_need_show_screen:
  BYTE 1
```

memory.S

```
  .GLOBAL MEMORY_SETUP
  .GLOBAL MEMORY_PLAY
  .GLOBAL MEMORY_FINISHED

  .EQU MEMORY_STAGES, 5
  .EQU MEMORY_BUTTONS, 4

  ; 2**MEMORY_COMBS is the amount of possible
  ; combinations for this minigame
  .EQU MEMORY_COMBS, 4
  ; Entry size in the memory_combinations table
  .EQU MEMORY_COMB_SIZE, ((MEMORY_BUTTONS + 2) *
      MEMORY_STAGES)
  ; Memory screen constants
  .EQU MEMORY_BUTTON_BORDER, 2
  .EQU MEMORY_GIANT_NUMBER_SIZE, 10
  .EQU MEMORY_GIANT_NUMBER_H, (TFT_CHAR_H *
      MEMORY_GIANT_NUMBER_SIZE)
  .EQU MEMORY_GIANT_NUMBER_W, (TFT_CHAR_W *
      MEMORY_GIANT_NUMBER_SIZE)
  .EQU MEMORY_GIANT_NUMBER_X, ((TFT_WIDTH -
      MEMORY_GIANT_NUMBER_W) / 2 - 2 *
      MEMORY_BUTTON_BORDER)
  .EQU MEMORY_GIANT_NUMBER_Y, ((TFT_HEIGHT -
      MEMORY_GIANT_NUMBER_H) / 2)
  .EQU MEMORY_BUTTON_FONT_SIZE, 6
  .EQU MEMORY_BUTTON_NUMBER_X, ((TFT_WIDTH / 2 -
      MEMORY_BUTTON_FONT_SIZE * TFT_CHAR_W) / 2)
  .EQU MEMORY_BUTTON_NUMBER_Y, ((TFT_HEIGHT / 2 -
      MEMORY_BUTTON_FONT_SIZE * TFT_CHAR_H) / 2)
  .EQU MEMORY_STAGE_BAR_H, (MEMORY_GIANT_NUMBER_H /
      (MEMORY_STAGES + 1))
  .EQU MEMORY_STAGE_BAR_W, MEMORY_BUTTON_BORDER
  .EQU MEMORY_STAGE_BAR_X, (MEMORY_GIANT_NUMBER_X +
      MEMORY_GIANT_NUMBER_W + 5)
  .EQU MEMORY_STAGE_BAR_Y, ((MEMORY_GIANT_NUMBER_Y +
       MEMORY_GIANT_NUMBER_H -
      MEMORY_GIANT_NUMBER_SIZE))

CSEG

; Randomly picks one of the possible entries on the
; memory_combinations table and stores that data on
; the memory_stage_data buffer.
MEMORY_SETUP:
  LDI R24, MEMORY_COMBS
  CALL RANDGEN
  ; R24 now has a random number between 0 and
  ; 2**MEMORY_COMBS-1, so we use it to select
  ; one entry in the memory_combinations table
  LDI ZL, LOW(memory_combinations)
  LDI ZH, HIGH(memory_combinations)
  LDI R18, MEMORY_COMB_SIZE
  MUL R24, R18
  ADD ZL, R0
  ADC ZH, R1
  CLR R1
  ; Now that Z is pointing to some entry on
  ; the table, load that entry's values into
  ; the variables we will be using
  LDI XL, LOW(memory_stage_data)
  LDI XH, HIGH(memory_stage_data)
  LDI R19, MEMORY_COMB_SIZE
memory_setup_loop:
  LPM R18, Z+
  ST X+, R18
  DEC R19
  BRNE memory_setup_loop
  ; Tell the game to show the screen once
  LDI R18, 1
  STS memory_need_show_screen, R18
  ; Finally, clear the variables
  STS memory_current_stage, R1
  STS memory_game_cleared, R1
  RET

; Plays the memory minigame in a non-blocking way by
; just asking if the screen was touched. Only when a
; a button is touched, performs a check to see
; if it was the correct button or not. The game then
; proceeds for MEMORY_STAGES iterations, resetting
; completely if a wrong button is touched.
MEMORY_PLAY:
  ; Check if the game needs to show the screen
  LDS R18, memory_need_show_screen
  CPI R18, 0
  BREQ memory_check_button_touched
  CALL MEMORY_SHOW_SCREEN
  CLR R1
  STS memory_need_show_screen, R1
memory_check_button_touched:
  CALL TFT_READ_TOUCH
  LDS R18, TOUCH_Z_VALID
  CPI R18, 0
  BREQ memory_no_button_touched
  ; If here, the touchscreen was pressed
  CALL MEMORY_MAP_TOUCH_TO_BUTTON
  CALL MEMORY_PLAY_TONE
```

```
  ; Check if player touched the proper button
  ; Point at the current stage correct button
  LDS R18, memory_current_stage
  LDI ZL, LOW(memory_stage_data)
  LDI ZH, HIGH(memory_stage_data)
  LDI R19, (MEMORY_BUTTONS + 2)
  MUL R19, R18
  ADD ZL, R0
  ADC ZH, R1
  CLR R1
  LDI R18, (MEMORY_BUTTONS + 1)
  ADD ZL, R18
  ADC ZH, R1
  LD R18, Z ; R18 is the correct button
  CP R18, R24
  BREQ memory_play_good_button_touched
  ; If here, player has touched a bad button
  CALL INC_STRIKES
  CALL MEMORY_SETUP
  RET
memory_play_good_button_touched:
  ; If here, player has touched a good button
  LDS R18, memory_current_stage
  INC R18
  CPI R18, MEMORY_STAGES
  BREQ memory_player_cleared_game
  STS memory_current_stage, R18
  LDI R18, 1
  STS memory_need_show_screen, R18
  RET

memory_player_cleared_game:
  LDI R18, 1
  STS memory_game_cleared, R18
memory_no_button_touched:
  RET

; Returns true (as 1) on R24 if the memory minigame
; was cleared by the player. If it hasn't already,
; returns false (as 0).
MEMORY_FINISHED:
  LDS R24, memory_game_cleared
  RET

; Pretty function that plays a different buzzer tone
; according to the button touched. The button is
; received on R24. The function also returns the
; same button number on R24. Stolen from Simon says.
MEMORY_PLAY_TONE:
  PUSH R24
  CPI R24, 0
  BREQ memory_tone_0
  CPI R24, 1
  BREQ memory_tone_1
  CPI R24, 2
  BREQ memory_tone_2
  CPI R24, 3
  BREQ memory_tone_3
memory_tone_0:
  CALL BUZZER_TONE_1
  POP R24
  RET
memory_tone_1:
  CALL BUZZER_TONE_2
  POP R24
  RET
memory_tone_2:
  CALL BUZZER_TONE_3
  POP R24
  RET
memory_tone_3:
  CALL BUZZER_TONE_5
  POP R24
  RET

; Retrieves the button number of a touched button
; on screen by reading the TFT touched x,y values.
; Note: this function assumes the player has indeed
; touched the screen (i.e. you shoudl validate the
; z coordinate before calling this function).
    Returns
; the button number on R24. Stolen from Simon says.
MEMORY_MAP_TOUCH_TO_BUTTON:
  CLR R24
  LDS R19, TOUCH_Y_HIGH
  LDS R18, TOUCH_Y_LOW
  ROR R19
  ROR R18
  CPI R18, (TFT_HEIGHT / 4)
  BRLO memory_touch_test_x
  LDI R24, 2
memory_touch_test_x:
  LDS R19, TOUCH_X_HIGH
  LDS R18, TOUCH_X_LOW
  ROR R19
  ROR R18
  CPI R18, (TFT_WIDTH / 4)
  BRLO memory_touch_ret
  INC R24
memory_touch_ret:
  RET

; Delay used by the memory minigame after showing
; the giant number to memoize it.
MEMORY_DELAY:
  PUSH ZL
  PUSH ZH
  PUSH R16
  PUSH R17
  PUSH R18
  PUSH R19
  PUSH R20
  LDI R16, 40
memory_delay_loop:
  CALL SHOW_DISPLAYS
  DEC R16
  BRNE memory_delay_loop
  POP R20
  POP R19
  POP R18
  POP R17
  POP R16
  POP ZH
  POP ZL
  RET

; Shows the current stage giant number and buttons
; labels on the screen in a pretty format.
MEMORY_SHOW_SCREEN:
  PUSH R17
  PUSH R16
  PUSH R15
  ; Start by filling the screen with black
  LDI R25, HIGH(BLACK)
  LDI R24, LOW(BLACK)
  CALL TFT_FILL_SCREEN
  ; Now point at the current stage data
  LDS R18, memory_current_stage
  LDI ZL, LOW(memory_stage_data)
  LDI ZH, HIGH(memory_stage_data)
  LDI R19, (MEMORY_BUTTONS + 2)
  MUL R19, R18
  ADD ZL, R0
  ADC ZH, R1
  CLR R1
  ; Load the first giant number and show it
  LDI R25, HIGH(MEMORY_GIANT_NUMBER_X)
  LDI R24, LOW(MEMORY_GIANT_NUMBER_X)
  LDI R23, HIGH(MEMORY_GIANT_NUMBER_Y)
  LDI R22, LOW(MEMORY_GIANT_NUMBER_Y)
  CLR R21
  LD R20, Z+
  LDI R26, 48
  ADD R20, R26 ; number to ASCII
  CLR R21
  LDI R19, HIGH(WHITE)
  LDI R18, LOW(WHITE)
  CLR R17
  LDI R16, MEMORY_GIANT_NUMBER_SIZE
  PUSH ZL
  PUSH ZH
  CALL TFT_DRAW_CHAR
  ; Draw the current stage progress bar
  LDS R15, memory_current_stage
  INC R15
  LDI R23, HIGH(MEMORY_STAGE_BAR_Y)
  LDI R22, LOW(MEMORY_STAGE_BAR_Y)
memory_show_stage_progress_bar_loop:
  LDI R25, HIGH(MEMORY_STAGE_BAR_X)
  LDI R24, LOW(MEMORY_STAGE_BAR_X)
```

```
LDI R21, HIGH(MEMORY_STAGE_BAR_W)
LDI R20, LOW(MEMORY_STAGE_BAR_W)
LDI R19, HIGH(MEMORY_STAGE_BAR_H)
LDI R18, LOW(MEMORY_STAGE_BAR_H)
LDI R17, HIGH(GREEN_LIGHT)
LDI R16, LOW(GREEN_LIGHT)
PUSH R23
PUSH R22
CALL TFT_FILL_RECT_HW
POP R22
POP R23
LDI R19, HIGH(MEMORY_STAGE_BAR_H + 4)
LDI R18, LOW(MEMORY_STAGE_BAR_H + 4)
SUB R22, R18
SBC R23, R19
DEC R15
BRNE memory_show_stage_progress_bar_loop
CALL MEMORY_DELAY
; Clear the screen and draw the button borders
LDI R25, HIGH(BLACK)
LDI R24, LOW(BLACK)
CALL TFT_FILL_SCREEN
CLR R25 ; Horizontal line
CLR R24
LDI R23, HIGH(TFT_HEIGHT / 2)
LDI R22, LOW(TFT_HEIGHT / 2)
LDI R21, HIGH(TFT_WIDTH)
LDI R20, LOW(TFT_WIDTH)
LDI R19, HIGH(MEMORY_BUTTON_BORDER)
LDI R18, LOW(MEMORY_BUTTON_BORDER)
LDI R17, HIGH(WHITE)
LDI R16, LOW(WHITE)
CALL TFT_FILL_RECT_HW
CLR R23 ; Vertical line
CLR R22
LDI R25, HIGH(TFT_WIDTH / 2)
LDI R24, LOW(TFT_WIDTH / 2)
LDI R19, HIGH(TFT_HEIGHT)
LDI R18, LOW(TFT_HEIGHT)
LDI R21, HIGH(MEMORY_BUTTON_BORDER)
LDI R20, LOW(MEMORY_BUTTON_BORDER)
LDI R17, HIGH(WHITE)
LDI R16, LOW(WHITE)
CALL TFT_FILL_RECT_HW
; Now draw the four button labels
POP ZH
POP ZL
LDI R25, HIGH(MEMORY_BUTTON_NUMBER_X)
LDI R24, LOW(MEMORY_BUTTON_NUMBER_X)
LDI R23, HIGH(MEMORY_BUTTON_NUMBER_Y)
LDI R22, LOW(MEMORY_BUTTON_NUMBER_Y)
CLR R21
LD R20, Z+
LDI R26, 48
ADD R20, R26 ; number to ASCII
CLR R21
LDI R19, HIGH(WHITE)
LDI R18, LOW(WHITE)
CLR R17
LDI R16, MEMORY_BUTTON_FONT_SIZE
PUSH ZL
PUSH ZH
CALL TFT_DRAW_CHAR
POP ZH
POP ZL
LDI R25, HIGH(MEMORY_BUTTON_NUMBER_X + TFT_WIDTH / 2)
LDI R24, LOW(MEMORY_BUTTON_NUMBER_X + TFT_WIDTH / 2)
LDI R23, HIGH(MEMORY_BUTTON_NUMBER_Y)
LDI R22, LOW(MEMORY_BUTTON_NUMBER_Y)
CLR R21
LD R20, Z+
LDI R26, 48
ADD R20, R26 ; number to ASCII
CLR R21
LDI R19, HIGH(WHITE)
LDI R18, LOW(WHITE)
CLR R17
LDI R16, MEMORY_BUTTON_FONT_SIZE
PUSH ZL
PUSH ZH
CALL TFT_DRAW_CHAR
POP ZH
POP ZL
LDI R25, HIGH(MEMORY_BUTTON_NUMBER_X)
LDI R24, LOW(MEMORY_BUTTON_NUMBER_X)
LDI R23, HIGH(MEMORY_BUTTON_NUMBER_Y + TFT_HEIGHT / 2)
LDI R22, LOW(MEMORY_BUTTON_NUMBER_Y + TFT_HEIGHT / 2)
CLR R21
LD R20, Z+
LDI R26, 48
ADD R20, R26 ; number to ASCII
CLR R21
LDI R19, HIGH(WHITE)
LDI R18, LOW(WHITE)
CLR R17
LDI R16, MEMORY_BUTTON_FONT_SIZE
PUSH ZL
PUSH ZH
CALL TFT_DRAW_CHAR
POP ZH
POP ZL
LDI R25, HIGH(MEMORY_BUTTON_NUMBER_X + TFT_WIDTH / 2)
LDI R24, LOW(MEMORY_BUTTON_NUMBER_X + TFT_WIDTH / 2)
LDI R23, HIGH(MEMORY_BUTTON_NUMBER_Y + TFT_HEIGHT / 2)
LDI R22, LOW(MEMORY_BUTTON_NUMBER_Y + TFT_HEIGHT / 2)
CLR R21
LD R20, Z+
LDI R26, 48
ADD R20, R26 ; number to ASCII
CLR R21
LDI R19, HIGH(WHITE)
LDI R18, LOW(WHITE)
CLR R17
LDI R16, MEMORY_BUTTON_FONT_SIZE
CALL TFT_DRAW_CHAR
POP R15
POP R16
POP R17
RET

; The following is a list of the memory combinations
; (i.e. every possible scenario that could happen).
; The format for listing them is:
; <stage 1>, <stage 2>, ... , <stage MEMORY_STAGES>
; Where each stage is made of MEMORY_BUTTONS+2 bytes:
; - The giant initial number on screen
; - The MEMORY_BUTTONS labels for the buttons
; - The right button (from 0 to MEMORY_BUTTONS - 1)
memory_combinations:
  DB 4, 1, 3, 4, 2, 3, 2, 2, 3, 4, 1, 3, 3, 1, 3, 4, 2, 2, 2, 3, 2, 4, 1, 0, 1, 2, 3, 1, 4, 0
  DB 1, 2, 3, 4, 1, 1, 2, 2, 3, 4, 1, 1, 2, 3, 1, 4, 2, 0, 3, 1, 2, 4, 3, 1, 1, 4, 3, 2, 1, 1
  DB 2, 1, 2, 4, 3, 1, 3, 1, 3, 4, 2, 0, 1, 4, 2, 3, 1, 3, 3, 1, 3, 4, 2, 0, 2, 4, 3, 2, 1, 3
  DB 3, 1, 2, 3, 4, 2, 1, 1, 4, 2, 3, 1, 2, 2, 1, 3, 4, 2, 1, 4, 3, 1, 2, 2, 4, 3, 2, 4, 1, 0
  DB 4, 1, 4, 2, 3, 3, 1, 2, 4, 1, 3, 1, 2, 1, 3, 4, 2, 1, 4, 1, 2, 3, 4, 1, 2, 3, 2, 4, 1, 2
  DB 2, 3, 4, 1, 2, 1, 2, 1, 3, 2, 4, 1, 3, 4, 3, 2, 1, 2, 4, 1, 3, 4, 2, 1, 3, 3, 2, 1, 4, 0
  DB 1, 3, 2, 4, 1, 1, 3, 1, 2, 4, 3, 0, 1, 4, 2, 3, 1, 3, 4, 2, 1, 4, 3, 0, 2, 3, 4, 1, 2, 2
  DB 3, 4, 1, 2, 3, 2, 2, 1, 2, 3, 4, 2, 3, 2, 4, 3, 1, 2, 2, 1, 2, 3, 4, 0, 3, 4, 2, 1, 3, 2
  DB 1, 1, 4, 3, 2, 1, 4, 2, 3, 1, 4, 1, 3, 1, 2, 4, 3, 2, 2, 2, 3, 4, 1, 0, 3, 4, 1, 2, 3, 2
  DB 2, 3, 1, 4, 2, 1, 4, 4, 3, 2, 1, 1, 2, 2, 3, 1, 4, 2, 2, 1, 4, 3, 2, 0, 4, 2, 4, 3, 1, 3
  DB 4, 2, 1, 3, 4, 3, 3, 4, 1, 2, 3, 0, 1, 2, 3, 4, 1, 2, 3, 2, 3, 4, 1, 0, 3, 2, 4, 3, 1, 0
  DB 3, 2, 4, 1, 2, 2, 4, 3, 2, 1, 4, 2, 4, 2, 3, 4, 1, 2, 4, 1, 4, 3, 2, 2, 1, 1, 4, 2, 3, 0
  DB 1, 3, 2, 1, 4, 1, 1, 2, 3, 4, 1, 2, 4, 3, 4, 1, 2, 1, 1, 2, 1, 3, 4, 1, 4, 3, 2, 1, 4, 3
  DB 2, 3, 4, 1, 2, 1, 1, 3, 4, 1, 2, 1, 4, 2, 3, 4, 1, 2, 1, 2, 4, 3, 1, 1, 1, 2, 3, 4, 1, 2
  DB 3, 4, 3, 2, 1, 2, 3, 4, 1, 2, 3, 0, 1, 1, 4, 3, 2, 1, 3, 1, 2, 4, 3, 0, 2, 1, 4, 3, 2, 1
  DB 4, 2, 4, 3, 1, 3, 4, 2, 1, 3, 4, 3, 4, 4, 2, 3, 1, 0, 1, 3, 4, 2, 1, 3, 4, 1, 3, 4, 2, 2
```

```
  DSEG
  ; Current minigame stage (0 to MEMORY_STAGES - 1)
memory_current_stage:
  BYTE 1
  ; Buffer for allocating all of the stage data
memory_stage_data:
  BYTE MEMORY_COMB_SIZE
  ; Boolean to set when minigame finishes
memory_game_cleared:
  BYTE 1
  ; Boolean to set when need to show screen
memory_need_show_screen:
  BYTE 1
```

wires.S

```
  .GLOBAL WIRES_SETUP
  .GLOBAL WIRES_PLAY
  .GLOBAL WIRES_FINISHED

  .EQU WIRES_PORT, PORTK
  .EQU WIRES_DDR, DDRK
  .EQU WIRES_PIN, PINK

  .EQU WIRES_AMOUNT, 4

  ; 2**WIRES_COMBS is the amount of possible
  ; combinations for this minigame
  .EQU WIRES_COMBS, 4
  .EQU WIRES_COMB_SIZE, 10 ; Each entry in the
  ; wires_combinations table has this size

  ; Wires screen constants
  .EQU SCREEN_RECTS_AMOUNT, 4
  .EQU WIRES_FONT_SIZE, 10
  .EQU WIRES_RECTS_WIDTH, (TFT_WIDTH /
      SCREEN_RECTS_AMOUNT)
  .EQU WIRES_RECTS_HEIGHT, 60
  .EQU WIRES_NUMBER_H, (TFT_CHAR_H * WIRES_FONT_SIZE
      )
  .EQU WIRES_NUMBER_W, (TFT_CHAR_W * WIRES_FONT_SIZE
      )
  .EQU WIRES_NUMBER_X, ((TFT_WIDTH - WIRES_NUMBER_W)
       / 2)
  .EQU WIRES_NUMBER_Y, (WIRES_RECTS_HEIGHT + (
      TFT_HEIGHT - WIRES_RECTS_HEIGHT -
      WIRES_NUMBER_H)/ 2)
  .EQU WIRES_RECTS_BORDER, 2

CSEG

; Configures the WIRES_PORT as input and enables its
; pull ups. It also setups the variables for the
; minigame by calling RANDGEN.
WIRES_SETUP:
  ; First just configure the port
  CLR R18
  OUTPUT WIRES_DDR, R18
  LDI R18, 0xFF
  OUTPUT WIRES_PORT, R18
  ; We must now select some random variables
  ; values using the wires_combinations table
  LDI R24, WIRES_COMBS
  CALL RANDGEN
  ; R24 now has a random number between 0 and
  ; 2**WIRES_COMBS-1, so we use it to select
  ; one entry in the wires_combinations table
  LDI ZL, LOW(wires_combinations)
  LDI ZH, HIGH(wires_combinations)
  LDI R18, WIRES_COMB_SIZE
  MUL R24, R18
  ADD ZL, R0
  ADC ZH, R1
  CLR R1
  ; Now that Z is pointing to some entry on
  ; the table, load that entry's values into
  ; the variables we will be using
  LPM R18, Z+
  STS correct_wire, R18
  LPM R18, Z+
  STS wires_screen_number, R18
  LDI XL, LOW(wires_screen_rects)
  LDI XH, HIGH(wires_screen_rects)
  CLR R24
wires_copy_rects:
  LPM R18, Z+
  ST X+, R18
  INC R24
  CPI R24, (SCREEN_RECTS_AMOUNT * 2)
  BRNE wires_copy_rects
  ; Tell the game to show the screen once
  LDI R18, 1
  STS wires_need_show_screen, R18
  ; Finally, clear the wires_game_cleared var
  STS wires_game_cleared, R1
  RET

; Plays the wires minigame in a non-blocking way by
; just asking if a wire has been cut. Only when one
; has, performs a check to determinate if it was the
; correct wire or not. Detonates the bomb or clears
; the minigame accordingly on the result.
WIRES_PLAY:
  ; Check if the game needs to show the screen
  LDS R18, wires_need_show_screen
  CPI R18, 0
  BREQ wires_check_wires_present
  CALL WIRES_SHOW_SCREEN
  CLR R1
  STS wires_need_show_screen, R1
wires_check_wires_present:
  ; Check if all wires are still present
  ; (i.e. their value in  WIRES_PORT is 0)
  INPUT R18, WIRES_PIN
  LDI R19, ((1 << WIRES_AMOUNT) - 1)
  ; R19 has a mask for taking the wires bits
  AND R18, R19
  CPI R18, 0
  BREQ no_wire_cut
  ; If here, at least one wire has been cut.
  ; Load now in R20 the correct wire
  LDS R20, correct_wire
  ; Since the correct wire bit should be in 1
  ; and the others in 0, make R19 like that
  LDI R19, 0x01
wires_small_loop:
  CPI R20, 0
  BREQ wires_comp_wires
  LSL R19
  DEC R20
  RJMP wires_small_loop
wires_comp_wires:
  ; R19 has now a 1 in the correct_wire-th bit
  ; So if R18 != R19 then a wrong wire was cut
  CP R18, R19
  BREQ correct_wire_cut
  ; If here, a wrong wire has been cut! The
  ; game should end and bomb should explode!
  LDI R18, MAX_STRIKES_AMOUNT
  ; Call INC_STRIKES MAX_STRIKES_AMOUNT times
  ; to force the game to be over
wires_strikes_loop:
  PUSH R18
  CALL INC_STRIKES
  POP R18
  DEC R18
  BRNE wires_strikes_loop
  RET
correct_wire_cut:
  ; If here, the right wire has been cut! It's
  ; a happy day and everybody was saved!
  LDI R18, 1 ; Make wires_game_cleared = true
  STS wires_game_cleared, R18
no_wire_cut:
  RET

; Returns true (as 1) on R24 if the wires minigame
; was cleared by the player. If it hasn't already,
; returns false (as 0).
WIRES_FINISHED:
  LDS R24, wires_game_cleared
  RET

; Shows the number and rectangles on screen. Since
; the wires minigame doesn't use the touchability of
; the screen, this function only draws.
WIRES_SHOW_SCREEN:
  PUSH R17
  PUSH R16
```

```
  PUSH R15
  ; Start by filling the screen with black
  LDI R25, HIGH(BLACK)
  LDI R24, LOW(BLACK)
  CALL TFT_FILL_SCREEN
  ; Draw the wires_number on its position
  LDI R25, HIGH(WIRES_NUMBER_X)
  LDI R24, LOW(WIRES_NUMBER_X)
  LDI R23, HIGH(WIRES_NUMBER_Y)
  LDI R22, LOW(WIRES_NUMBER_Y)
  CLR R21
  LDS R20, wires_screen_number
  LDI R26, 48
  ADD R20, R26 ; number to ASCII
  CLR R21
  LDI R19, HIGH(WHITE)
  LDI R18, LOW(WHITE)
  CLR R17
  LDI R16, WIRES_FONT_SIZE
  CALL TFT_DRAW_CHAR
  ; Iteratively draw the wires rects
  CLR R15 ; R15 is the index i for iteration
  LDI ZL, LOW(wires_screen_rects)
  LDI ZH, HIGH(wires_screen_rects)
wires_show_rects:
  ; Load this rect color and fill the args
  CLR R18
  CLR R25
  CLR R24
  LDI R23, HIGH(WIRES_RECTS_WIDTH)
  LDI R22, LOW(WIRES_RECTS_WIDTH)
  ; Do rect.x = WIRES_RECT_WIDTH * i
wires_rectx_loop:
  CP R18, R15
  BREQ wires_rectx_loop_end
  ADD R24, R22
  ADC R25, R23
  INC R18
  RJMP wires_rectx_loop
wires_rectx_loop_end:
  CLR R23
  CLR R22
  LDI R21, HIGH(WIRES_RECTS_WIDTH -
      WIRES_RECTS_BORDER)
  LDI R20, LOW(WIRES_RECTS_WIDTH -
      WIRES_RECTS_BORDER)
  LDI R19, HIGH(WIRES_RECTS_HEIGHT -
      WIRES_RECTS_BORDER)
  LDI R18, LOW(WIRES_RECTS_HEIGHT -
      WIRES_RECTS_BORDER)
  LD R17, Z+
  LD R16, Z+
  CALL TFT_FILL_RECT_HW
  INC R15
  MOV R22, R15
  CPI R22, SCREEN_RECTS_AMOUNT
  BRNE wires_show_rects

  POP R15
  POP R16
  POP R17
  RET

; The following is a list of the wires combinations
; (i.e. every possible scenario that could happen).
; The format for listing them is:
; DB: <wire>, <screen number>, <screen rects..>
; Note there should be 2**WIRES_COMBS here.
wires_combinations:
  ; Cases when wires_screen_number = 1
  DB 1, 1, HIGH(WHITE), LOW(WHITE), HIGH(BLUE_DARK),
       LOW(BLUE_DARK), HIGH(WHITE), LOW(WHITE), HIGH
      (GREEN_DARK), LOW(GREEN_DARK)
  DB WIRES_AMOUNT-1, 1, HIGH(RED_LIGHT), LOW(
      RED_LIGHT), HIGH(RED_LIGHT), LOW(RED_LIGHT),
      HIGH(GREEN_DARK), LOW(GREEN_DARK), HIGH(WHITE)
      , LOW(WHITE)
  DB WIRES_AMOUNT-1, 1, HIGH(BLUE_DARK), LOW(
      BLUE_DARK), HIGH(RED_LIGHT), LOW(RED_LIGHT),
      HIGH(BLUE_DARK), LOW(BLUE_DARK), HIGH(
      GREEN_DARK), LOW(GREEN_DARK)
  DB 1, 1, HIGH(RED_LIGHT), LOW(RED_LIGHT), HIGH(
      BLUE_DARK), LOW(BLUE_DARK), HIGH(RED_LIGHT),
      LOW(RED_LIGHT), HIGH(GREEN_DARK), LOW(
      GREEN_DARK)
  ; Cases when wires_screen_number = 2
  DB WIRES_AMOUNT-1, 2, HIGH(GREEN_DARK), LOW(
      GREEN_DARK), HIGH(RED_LIGHT), LOW(RED_LIGHT),
      HIGH(BLUE_DARK), LOW(BLUE_DARK), HIGH(
      RED_LIGHT), LOW(RED_LIGHT)
  DB 0, 2, HIGH(WHITE), LOW(WHITE), HIGH(WHITE), LOW
      (WHITE), HIGH(BLUE_DARK), LOW(BLUE_DARK), HIGH
      (GREEN_DARK), LOW(GREEN_DARK)
  DB 0, 2, HIGH(WHITE), LOW(WHITE), HIGH(BLUE_DARK),
       LOW(BLUE_DARK), HIGH(RED_LIGHT), LOW(
      RED_LIGHT), HIGH(GREEN_DARK), LOW(GREEN_DARK)
  DB 1, 2, HIGH(GREEN_DARK), LOW(GREEN_DARK), HIGH(
      WHITE), LOW(WHITE), HIGH(WHITE), LOW(WHITE),
      HIGH(RED_LIGHT), LOW(RED_LIGHT)
  ; Cases when wires_screen_number = 3
  DB 3, 3, HIGH(WHITE), LOW(WHITE), HIGH(BLUE_DARK),
       LOW(BLUE_DARK), HIGH(GREEN_DARK), LOW(
      GREEN_DARK), HIGH(BLUE_DARK), LOW(BLUE_DARK)
  DB 0, 3, HIGH(GREEN_DARK), LOW(GREEN_DARK), HIGH(
      BLUE_DARK), LOW(BLUE_DARK), HIGH(GREEN_DARK),
      LOW(GREEN_DARK), HIGH(RED_LIGHT), LOW(
      RED_LIGHT)
  DB 2, 3, HIGH(WHITE), LOW(WHITE), HIGH(WHITE), LOW
      (WHITE), HIGH(GREEN_DARK), LOW(GREEN_DARK),
      HIGH(RED_LIGHT), LOW(RED_LIGHT)
  DB 0, 3, HIGH(WHITE), LOW(WHITE), HIGH(GREEN_DARK)
      , LOW(GREEN_DARK), HIGH(BLUE_DARK), LOW(
      BLUE_DARK), HIGH(RED_LIGHT), LOW(RED_LIGHT)
  ; Cases when wires_screen_number = 4
  DB 2, 4, HIGH(BLUE_DARK), LOW(BLUE_DARK), HIGH(
      RED_LIGHT), LOW(RED_LIGHT), HIGH(WHITE), LOW(
      WHITE), HIGH(RED_LIGHT), LOW(RED_LIGHT)
  DB 3, 4, HIGH(BLUE_DARK), LOW(BLUE_DARK), HIGH(
      GREEN_DARK), LOW(GREEN_DARK), HIGH(WHITE), LOW
      (WHITE), HIGH(RED_LIGHT), LOW(RED_LIGHT)
  DB 1, 4, HIGH(WHITE), LOW(WHITE), HIGH(BLUE_DARK),
       LOW(BLUE_DARK), HIGH(GREEN_DARK), LOW(
      GREEN_DARK), HIGH(GREEN_DARK), LOW(GREEN_DARK)
  DB WIRES_AMOUNT-1, 4, HIGH(WHITE), LOW(WHITE),
      HIGH(RED_LIGHT), LOW(RED_LIGHT), HIGH(WHITE),
      LOW(WHITE), HIGH(GREEN_DARK), LOW(GREEN_DARK)

  DSEG
  ; Correct wire number (should be in the
  ; range from 0 to WIRES_AMOUNT - 1)
correct_wire:
  BYTE 1
  ; Number to be shown on the screen
wires_screen_number:
  BYTE 1
  ; Rectangles colors to appear on screen
wires_screen_rects:
  BYTE (WIRES_COMB_SIZE - 2)
  ; Boolean to set when minigame finishes
wires_game_cleared:
  BYTE 1
  ; Boolean to set when need to show screen
wires_need_show_screen:
  BYTE 1
```

tft-registers.S

```
; ------------------------------------------------
;       KEEP CODING AND NOBODY EXPLODES (KCANE)
; ------------------------------------------------
; Project for 66.09.Laboratorio de Microcomputadoras
; Faculty of Engineering, University of Buenos Aires
;
; By Ana Czarnitzki, Alejandro García & Juan Fresia
;
; ------------------------------------------------
;
; ------------------------------------------------
;              TFT GENERAL DEFINITIONS
; ------------------------------------------------

  .EQU TFT_WIDTH, 240
  .EQU TFT_HEIGHT, 320

  .EQU ILI9341_SOFTRESET, 0x01
  .EQU ILI9341_SLEEPIN, 0x10
  .EQU ILI9341_SLEEPOUT, 0x11
  .EQU ILI9341_NORMALDISP, 0x13
  .EQU ILI9341_INVERTOFF, 0x20
  .EQU ILI9341_INVERTON, 0x21
```

```
  .EQU ILI9341_GAMMASET, 0x26
  .EQU ILI9341_DISPLAYOFF, 0x28
  .EQU ILI9341_DISPLAYON, 0x29
  .EQU ILI9341_COLADDRSET, 0x2A
  .EQU ILI9341_PAGEADDRSET, 0x2B
  .EQU ILI9341_MEMORYWRITE, 0x2C
  .EQU ILI9341_PIXELFORMAT, 0x3A
  .EQU ILI9341_FRAMECONTROL, 0xB1
  .EQU ILI9341_DISPLAYFUNC, 0xB6
  .EQU ILI9341_ENTRYMODE, 0xB7
  .EQU ILI9341_POWERCONTROL1, 0xC0
  .EQU ILI9341_POWERCONTROL2, 0xC1
  .EQU ILI9341_VCOMCONTROL1, 0xC5
  .EQU ILI9341_VCOMCONTROL2, 0xC7
  .EQU ILI9341_MEMCONTROL, 0x36
  .EQU ILI9341_MADCTL, 0x36

  .EQU ILI9341_MADCTL_MY, 0x80
  .EQU ILI9341_MADCTL_MX, 0x40
  .EQU ILI9341_MADCTL_MV, 0x20
  .EQU ILI9341_MADCTL_ML, 0x10
  .EQU ILI9341_MADCTL_RGB, 0x00
  .EQU ILI9341_MADCTL_BGR, 0x08
  .EQU ILI9341_MADCTL_MH, 0x04
```

tft-colors.S

```
  .EQU MAGENTA, 0xF8F8
  .EQU ORANGE, 0xE4E4
  .EQU BLACK, 0x0000
  .EQU WHITE, 0xFFFF
  .EQU BLUE_LIGHT, 0x1D1D
  .EQU BLUE_DARK, 0x1818
  .EQU RED_LIGHT, 0xE0E0
  .EQU RED_DARK, 0xC0C0
  .EQU GREEN_LIGHT, 0x4747
  .EQU GREEN_DARK, 0x6666
```

tft-delay.S

```
; Delays for (R25 + 1) * 0.5 microseconds. Since R25
; is an 8 bit register, the maximum possible delay
    is
; 128 microseconds.
HALF_MICRO_DELAY:
  CLR R18                     ; 1 cycle
  NOP                         ; 1 cycle
half_micro_delay_loop:          ; 8 cycle loop = 500
    ns
  CP R18, R25                 ; 1 cycle
  BREQ half_micro_delay_end   ; 1 cycle (non-taken)
  NOP
  NOP
  NOP                         ; 3 nop cycles
  INC R18                     ; 1 cycle
  RJMP half_micro_delay_loop  ; 2 cycles
half_micro_delay_end:
  RET                         ; 5 cycles

; Delays for R25 miliseconds. Since R25 is an 8 bit
; register, the maximum possible delay is 256 ms.
MILI_DELAY:
  PUSH R16
  PUSH R17
  CLR R16
  MOV R24, R25
mili_delay_loop:
  CP R16, R24
  BREQ mili_delay_end
  INC R16
  LDI R25, 200
  LDI R17, 10
mili_delay_loop2:
  CPI R17, 0
  BREQ mili_delay_loop
  DEC R17
  PUSH R16
  CALL HALF_MICRO_DELAY ; 100 us
  POP R16
  RJMP mili_delay_loop2
mili_delay_end:
  POP R17
  POP R16
  RET
```

tft-writing.S

```
  #include "avr-linux.h"

  ; TFT control pins
  ; LCD_RD    TFT_CONTROL_PORT.0
  ; LCD_WR    TFT_CONTROL_PORT.1
  ; LCD_CD    TFT_CONTROL_PORT.2
  ; LCD_CS    TFT_CONTROL_PORT.3
  ; LCD_RESET TFT_CONTROL_PORT.4

  .EQU TFT_CONTROL_PORT, PORTF
  .EQU TFT_CONTROL_DDR, DDRF


  .EQU LCD_CONTROL_PORT_MASK, 0x1F

  ; All signals are active low, so use SBI to idle
  ; and CBI to active LCD_RD.
  .MACRO RD_ACTIVE
    CBI TFT_CONTROL_PORT, 0
  .ENDM

  .MACRO RD_IDLE
    SBI TFT_CONTROL_PORT, 0
  .ENDM

  ; LCD_WR
  .MACRO WR_ACTIVE
    CBI TFT_CONTROL_PORT, 1
  .ENDM

  .MACRO WR_IDLE
    SBI TFT_CONTROL_PORT, 1
  .ENDM

  ; LCD_CD
  .MACRO CD_COMMAND
    CBI TFT_CONTROL_PORT, 2
  .ENDM

  .MACRO CD_DATA
    SBI TFT_CONTROL_PORT, 2
  .ENDM

  ; LCD_CS
  .MACRO CS_ACTIVE
    CBI TFT_CONTROL_PORT, 3
  .ENDM

  .MACRO CS_IDLE
    SBI TFT_CONTROL_PORT, 3
  .ENDM

  ; LCD_RESET
  .MACRO TFT_RESET_LOW
    CBI TFT_CONTROL_PORT, 4
  .ENDM

  .MACRO TFT_RESET_HIGH
    SBI TFT_CONTROL_PORT, 4
  .ENDM

  .MACRO WR_STROBE
    WR_ACTIVE
    WR_IDLE
  .ENDM

  .EQU PORTH_MASK, 0x78
  .EQU PORTG_MASK, 0x20
  .EQU PORTE_MASK, 0x38

  .EQU DATA_MASK_76, 0xC0 ; to H4, H3
  .EQU DATA_MASK_5,  0x20 ; to E3
  .EQU DATA_MASK_4,  0x10 ; to G5
  .EQU DATA_MASK_32, 0x0C ; to E4, E4
  .EQU DATA_MASK_10, 0x03 ; to H6, H5

CSEG

  ; WRITE_8 writs register R25 into screen data.
  ; Rewrite this to re-map screen conections.
```

```
; For Arduino Mega (2560) the mapping is:
; ScreenData: D7 D6 D5 D4 D3 D2 D1 D0
; atmega2560: H4 H3 E3 G5 E5 E4 H6 H5
WRITE_8:
  PUSH R16
  PUSH R17
  ; Deal with PORTH
  LDS R16, PORTH
  ANDI R16, ~PORTH_MASK

  ; H6, H5
  MOV R17, R25
  ANDI R17, DATA_MASK_10
  LSL R17 ; H1 --> H6
  LSL R17
  LSL R17
  LSL R17
  LSL R17
  OR R16, R17

  ; H4, H3
  MOV R17, R25
  ANDI R17, DATA_MASK_76
  LSR R17 ; H7 --> H4
  LSR R17
  LSR R17
  OR R16, R17
  STS PORTH, R16

  ; Deal with PORTG (can use IN)
  INPUT R16, PORTG
  ANDI R16, ~PORTG_MASK

  ; G5
  MOV R17, R25
  ANDI R17, DATA_MASK_4
  LSL R17 ; G4 --> G5
  OR R16, R17
  OUTPUT PORTG, R16

  ; Deal with PORTE (can use IN)
  INPUT R16, PORTE
  ANDI R16, ~PORTE_MASK
  ; E5, E4
  MOV R17, R25
  ANDI R17, DATA_MASK_32
  LSL R17 ; E3 --> E5
  LSL R17
  OR R16, R17

  MOV R17, R25
  ANDI R17, DATA_MASK_5
  LSR R17 ; E5 --> E3
  LSR R17
  OR R16, R17

  OUTPUT PORTE, R16

  WR_STROBE

  POP R17
  POP R16
  RET

; Sets screen data pins as output.
SET_WRITE_DIR:
  PUSH R17
  INPUT R17, DDRH
  ORI R17, PORTH_MASK
  OUTPUT DDRH, R17

  INPUT R17, DDRE
  ORI R17, PORTE_MASK
  OUTPUT DDRE, R17

  INPUT R17, DDRG
  ORI R17, PORTG_MASK
  OUTPUT DDRG, R17
  POP R17
  RET

; Mid-level routines to talk to the screen

; WRITE_REGISTER_8 takes two 8 bit parameters:
;     address a in R25
;        data d in R24
; It stores d in address a of the TFT screen.
WRITE_REGISTER_8:
  CD_COMMAND
  CALL WRITE_8
  CD_DATA
  MOV R25, R24
  CALL WRITE_8
  RET

; WRITE_REGISTER_16 takes two 16 bit parameters:
;     address a in R25:R24
;        data d in R23:R22
; It stores d in address a of the TFT screen.
; It writes bytes in hi-lo order.
WRITE_REGISTER_16:
  CD_COMMAND
  CALL WRITE_8
  MOV R25, R24
  CALL WRITE_8

  CD_DATA
  MOV R25, R23
  CALL WRITE_8
  MOV R25, R22
  CALL WRITE_8
  RET

; WRITE_REGISTER_32 takes a one byte addres and a 32
    bit data:
;     address a in R25
;        data d in R23:R22:R21:R20
; It stores d in address a of the TFT screen.
; It writes bytes in hi-lo order.
WRITE_REGISTER_32:
  PUSH R25
  CS_ACTIVE

  CD_COMMAND
  CALL WRITE_8

  CD_DATA
  MOV R25, R23
  CALL WRITE_8
  MOV R25, R22
  CALL WRITE_8
  MOV R25, R21
  CALL WRITE_8
  MOV R25, R20
  CALL WRITE_8

  CS_IDLE
  POP R25
  RET
```

tft-char-ROM.S

```
  .EQU TFT_CHAR_H, 8
  .EQU TFT_CHAR_W, 6

tft_char_ROM:
  DB 0x00, 0x00, 0x00, 0x00, 0x00
  DB 0x3E, 0x5B, 0x4F, 0x5B, 0x3E
  DB 0x3E, 0x6B, 0x4F, 0x6B, 0x3E
  DB 0x1C, 0x3E, 0x7C, 0x3E, 0x1C
  DB 0x18, 0x3C, 0x7E, 0x3C, 0x18
  DB 0x1C, 0x57, 0x7D, 0x57, 0x1C
  DB 0x1C, 0x5E, 0x7F, 0x5E, 0x1C
  DB 0x00, 0x18, 0x3C, 0x18, 0x00
  DB 0xFF, 0xE7, 0xC3, 0xE7, 0xFF
  DB 0x00, 0x18, 0x24, 0x18, 0x00
  DB 0xFF, 0xE7, 0xDB, 0xE7, 0xFF
  DB 0x30, 0x48, 0x3A, 0x06, 0x0E
  DB 0x26, 0x29, 0x79, 0x29, 0x26
  DB 0x40, 0x7F, 0x05, 0x05, 0x07
  DB 0x40, 0x7F, 0x05, 0x25, 0x3F
  DB 0x5A, 0x3C, 0xE7, 0x3C, 0x5A
  DB 0x7F, 0x3E, 0x1C, 0x1C, 0x08
  DB 0x08, 0x1C, 0x1C, 0x3E, 0x7F
  DB 0x14, 0x22, 0x7F, 0x22, 0x14
  DB 0x5F, 0x5F, 0x00, 0x5F, 0x5F
  DB 0x06, 0x09, 0x7F, 0x01, 0x7F
  DB 0x00, 0x66, 0x89, 0x95, 0x6A
  DB 0x60, 0x60, 0x60, 0x60, 0x60
  DB 0x94, 0xA2, 0xFF, 0xA2, 0x94
```

```
DB 0x08, 0x04, 0x7E, 0x04, 0x08
DB 0x10, 0x20, 0x7E, 0x20, 0x10
DB 0x08, 0x08, 0x2A, 0x1C, 0x08
DB 0x08, 0x1C, 0x2A, 0x08, 0x08
DB 0x1E, 0x10, 0x10, 0x10, 0x10
DB 0x0C, 0x1E, 0x0C, 0x1E, 0x0C
DB 0x30, 0x38, 0x3E, 0x38, 0x30
DB 0x06, 0x0E, 0x3E, 0x0E, 0x06
DB 0x00, 0x00, 0x00, 0x00, 0x00
DB 0x00, 0x00, 0x5F, 0x00, 0x00
DB 0x00, 0x07, 0x00, 0x07, 0x00
DB 0x14, 0x7F, 0x14, 0x7F, 0x14
DB 0x24, 0x2A, 0x7F, 0x2A, 0x12
DB 0x23, 0x13, 0x08, 0x64, 0x62
DB 0x36, 0x49, 0x56, 0x20, 0x50
DB 0x00, 0x08, 0x07, 0x03, 0x00
DB 0x00, 0x1C, 0x22, 0x41, 0x00
DB 0x00, 0x41, 0x22, 0x1C, 0x00
DB 0x2A, 0x1C, 0x7F, 0x1C, 0x2A
DB 0x08, 0x08, 0x3E, 0x08, 0x08
DB 0x00, 0x80, 0x70, 0x30, 0x00
DB 0x08, 0x08, 0x08, 0x08, 0x08
DB 0x00, 0x00, 0x60, 0x60, 0x00
DB 0x20, 0x10, 0x08, 0x04, 0x02
DB 0x3E, 0x51, 0x49, 0x45, 0x3E
DB 0x00, 0x42, 0x7F, 0x40, 0x00
DB 0x72, 0x49, 0x49, 0x49, 0x46
DB 0x21, 0x41, 0x49, 0x4D, 0x33
DB 0x18, 0x14, 0x12, 0x7F, 0x10
DB 0x27, 0x45, 0x45, 0x45, 0x39
DB 0x3C, 0x4A, 0x49, 0x49, 0x31
DB 0x41, 0x21, 0x11, 0x09, 0x07
DB 0x36, 0x49, 0x49, 0x49, 0x36
DB 0x46, 0x49, 0x49, 0x29, 0x1E
DB 0x00, 0x00, 0x14, 0x00, 0x00
DB 0x00, 0x40, 0x34, 0x00, 0x00
DB 0x00, 0x08, 0x14, 0x22, 0x41
DB 0x14, 0x14, 0x14, 0x14, 0x14
DB 0x00, 0x41, 0x22, 0x14, 0x08
DB 0x02, 0x01, 0x59, 0x09, 0x06
DB 0x3E, 0x41, 0x5D, 0x59, 0x4E
DB 0x7C, 0x12, 0x11, 0x12, 0x7C
DB 0x7F, 0x49, 0x49, 0x49, 0x36
DB 0x3E, 0x41, 0x41, 0x41, 0x22
DB 0x7F, 0x41, 0x41, 0x41, 0x3E
DB 0x7F, 0x49, 0x49, 0x49, 0x41
DB 0x7F, 0x09, 0x09, 0x09, 0x01
DB 0x3E, 0x41, 0x41, 0x51, 0x73
DB 0x7F, 0x08, 0x08, 0x08, 0x7F
DB 0x00, 0x41, 0x7F, 0x41, 0x00
DB 0x20, 0x40, 0x41, 0x3F, 0x01
DB 0x7F, 0x08, 0x14, 0x22, 0x41
DB 0x7F, 0x40, 0x40, 0x40, 0x40
DB 0x7F, 0x02, 0x1C, 0x02, 0x7F
DB 0x7F, 0x04, 0x08, 0x10, 0x7F
DB 0x3E, 0x41, 0x41, 0x41, 0x3E
DB 0x7F, 0x09, 0x09, 0x09, 0x06
DB 0x3E, 0x41, 0x51, 0x21, 0x5E
DB 0x7F, 0x09, 0x19, 0x29, 0x46
DB 0x26, 0x49, 0x49, 0x49, 0x32
DB 0x03, 0x01, 0x7F, 0x01, 0x03
DB 0x3F, 0x40, 0x40, 0x40, 0x3F
DB 0x1F, 0x20, 0x40, 0x20, 0x1F
DB 0x3F, 0x40, 0x38, 0x40, 0x3F
DB 0x63, 0x14, 0x08, 0x14, 0x63
DB 0x03, 0x04, 0x78, 0x04, 0x03
DB 0x61, 0x59, 0x49, 0x4D, 0x43
DB 0x00, 0x7F, 0x41, 0x41, 0x41
DB 0x02, 0x04, 0x08, 0x10, 0x20
DB 0x00, 0x41, 0x41, 0x41, 0x7F
DB 0x04, 0x02, 0x01, 0x02, 0x04
DB 0x40, 0x40, 0x40, 0x40, 0x40
DB 0x00, 0x03, 0x07, 0x08, 0x00
DB 0x20, 0x54, 0x54, 0x78, 0x40
DB 0x7F, 0x28, 0x44, 0x44, 0x38
DB 0x38, 0x44, 0x44, 0x44, 0x28
DB 0x38, 0x44, 0x44, 0x28, 0x7F
DB 0x38, 0x54, 0x54, 0x54, 0x18
DB 0x00, 0x08, 0x7E, 0x09, 0x02
DB 0x18, 0xA4, 0xA4, 0x9C, 0x78
DB 0x7F, 0x08, 0x04, 0x04, 0x78
DB 0x00, 0x44, 0x7D, 0x40, 0x00
DB 0x20, 0x40, 0x40, 0x3D, 0x00
DB 0x7F, 0x10, 0x28, 0x44, 0x00
DB 0x00, 0x41, 0x7F, 0x40, 0x00
DB 0x7C, 0x04, 0x78, 0x04, 0x78
DB 0x7C, 0x08, 0x04, 0x04, 0x78
DB 0x38, 0x44, 0x44, 0x44, 0x38
DB 0xFC, 0x18, 0x24, 0x24, 0x18
DB 0x18, 0x24, 0x24, 0x18, 0xFC
DB 0x7C, 0x08, 0x04, 0x04, 0x08
DB 0x48, 0x54, 0x54, 0x54, 0x24
DB 0x04, 0x04, 0x3F, 0x44, 0x24
DB 0x3C, 0x40, 0x40, 0x20, 0x7C
DB 0x1C, 0x20, 0x40, 0x20, 0x1C
DB 0x3C, 0x40, 0x30, 0x40, 0x3C
DB 0x44, 0x28, 0x10, 0x28, 0x44
DB 0x4C, 0x90, 0x90, 0x90, 0x7C
DB 0x44, 0x64, 0x54, 0x4C, 0x44
DB 0x00, 0x08, 0x36, 0x41, 0x00
DB 0x00, 0x00, 0x77, 0x00, 0x00
DB 0x00, 0x41, 0x36, 0x08, 0x00
DB 0x02, 0x01, 0x02, 0x04, 0x02
DB 0x3C, 0x26, 0x23, 0x26, 0x3C
DB 0x1E, 0xA1, 0xA1, 0x61, 0x12
DB 0x3A, 0x40, 0x40, 0x20, 0x7A
DB 0x38, 0x54, 0x54, 0x55, 0x59
DB 0x21, 0x55, 0x55, 0x79, 0x41
DB 0x22, 0x54, 0x54, 0x78, 0x42 ; a-umlaut
DB 0x21, 0x55, 0x54, 0x78, 0x40
DB 0x20, 0x54, 0x55, 0x79, 0x40
DB 0x0C, 0x1E, 0x52, 0x72, 0x12
DB 0x39, 0x55, 0x55, 0x55, 0x59
DB 0x39, 0x54, 0x54, 0x54, 0x59
DB 0x39, 0x55, 0x54, 0x54, 0x58
DB 0x00, 0x00, 0x45, 0x7C, 0x41
DB 0x00, 0x02, 0x45, 0x7D, 0x42
DB 0x00, 0x01, 0x45, 0x7C, 0x40
DB 0x7D, 0x12, 0x11, 0x12, 0x7D ; A-umlaut
DB 0xF0, 0x28, 0x25, 0x28, 0xF0
DB 0x7C, 0x54, 0x55, 0x45, 0x00
DB 0x20, 0x54, 0x54, 0x7C, 0x54
DB 0x7C, 0x0A, 0x09, 0x7F, 0x49
DB 0x32, 0x49, 0x49, 0x49, 0x32
DB 0x3A, 0x44, 0x44, 0x44, 0x3A ; o-umlaut
DB 0x32, 0x4A, 0x48, 0x48, 0x30
DB 0x3A, 0x41, 0x41, 0x21, 0x7A
DB 0x3A, 0x42, 0x40, 0x20, 0x78
DB 0x00, 0x9D, 0xA0, 0xA0, 0x7D
DB 0x3D, 0x42, 0x42, 0x42, 0x3D ; O-umlaut
DB 0x3D, 0x40, 0x40, 0x40, 0x3D
DB 0x3C, 0x24, 0xFF, 0x24, 0x24
DB 0x48, 0x7E, 0x49, 0x43, 0x66
DB 0x2B, 0x2F, 0xFC, 0x2F, 0x2B
DB 0xFF, 0x09, 0x29, 0xF6, 0x20
DB 0xC0, 0x88, 0x7E, 0x09, 0x03
DB 0x20, 0x54, 0x54, 0x79, 0x41
DB 0x00, 0x00, 0x44, 0x7D, 0x41
DB 0x30, 0x48, 0x48, 0x4A, 0x32
DB 0x38, 0x40, 0x40, 0x22, 0x7A
DB 0x00, 0x7A, 0x0A, 0x0A, 0x72
DB 0x7D, 0x0D, 0x19, 0x31, 0x7D
DB 0x26, 0x29, 0x29, 0x2F, 0x28
DB 0x26, 0x29, 0x29, 0x29, 0x26
DB 0x30, 0x48, 0x4D, 0x40, 0x20
DB 0x38, 0x08, 0x08, 0x08, 0x08
DB 0x08, 0x08, 0x08, 0x08, 0x38
DB 0x2F, 0x10, 0xC8, 0xAC, 0xBA
DB 0x2F, 0x10, 0x28, 0x34, 0xFA
DB 0x00, 0x00, 0x7B, 0x00, 0x00
DB 0x08, 0x14, 0x2A, 0x14, 0x22
DB 0x22, 0x14, 0x2A, 0x14, 0x08
DB 0x55, 0x00, 0x55, 0x00, 0x55 ; #176 (25% block)
     missing in old code
DB 0xAA, 0x55, 0xAA, 0x55, 0xAA ; 50% block
DB 0xFF, 0x55, 0xFF, 0x55, 0xFF ; 75% block
DB 0x00, 0x00, 0x00, 0xFF, 0x00
DB 0x10, 0x10, 0x10, 0xFF, 0x00
DB 0x14, 0x14, 0x14, 0xFF, 0x00
DB 0x10, 0x10, 0xFF, 0x00, 0xFF
DB 0x10, 0x10, 0xF0, 0x10, 0xF0
DB 0x14, 0x14, 0x14, 0xFC, 0x00
DB 0x14, 0x14, 0xF7, 0x00, 0xFF
DB 0x00, 0x00, 0xFF, 0x00, 0xFF
DB 0x14, 0x14, 0xF4, 0x04, 0xFC
DB 0x14, 0x14, 0x17, 0x10, 0x1F
DB 0x10, 0x10, 0x1F, 0x10, 0x1F
DB 0x14, 0x14, 0x14, 0x1F, 0x00
DB 0x10, 0x10, 0x10, 0xF0, 0x00
DB 0x00, 0x00, 0x00, 0x1F, 0x10
```

```
DB 0x10, 0x10, 0x10, 0x1F, 0x10
DB 0x10, 0x10, 0x10, 0xF0, 0x10
DB 0x00, 0x00, 0x00, 0xFF, 0x10
DB 0x10, 0x10, 0x10, 0x10, 0x10
DB 0x10, 0x10, 0x10, 0xFF, 0x10
DB 0x00, 0x00, 0x00, 0xFF, 0x14
DB 0x00, 0x00, 0xFF, 0x00, 0xFF
DB 0x00, 0x00, 0x1F, 0x10, 0x17
DB 0x00, 0x00, 0xFC, 0x04, 0xF4
DB 0x14, 0x14, 0x17, 0x10, 0x17
DB 0x14, 0x14, 0xF4, 0x04, 0xF4
DB 0x00, 0x00, 0xFF, 0x00, 0xF7
DB 0x14, 0x14, 0x14, 0x14, 0x14
DB 0x14, 0x14, 0xF7, 0x00, 0xF7
DB 0x14, 0x14, 0x14, 0x17, 0x14
DB 0x10, 0x10, 0x1F, 0x10, 0x1F
DB 0x14, 0x14, 0x14, 0xF4, 0x14
DB 0x10, 0x10, 0xF0, 0x10, 0xF0
DB 0x00, 0x00, 0x1F, 0x10, 0x1F
DB 0x00, 0x00, 0x00, 0x1F, 0x14
DB 0x00, 0x00, 0x00, 0xFC, 0x14
DB 0x00, 0x00, 0xF0, 0x10, 0xF0
DB 0x10, 0x10, 0xFF, 0x10, 0xFF
DB 0x14, 0x14, 0x14, 0xFF, 0x14
DB 0x10, 0x10, 0x10, 0x1F, 0x00
DB 0x00, 0x00, 0x00, 0xF0, 0x10
DB 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
DB 0xF0, 0xF0, 0xF0, 0xF0, 0xF0
DB 0xFF, 0xFF, 0xFF, 0x00, 0x00
DB 0x00, 0x00, 0x00, 0xFF, 0xFF
DB 0x0F, 0x0F, 0x0F, 0x0F, 0x0F
DB 0x38, 0x44, 0x44, 0x38, 0x44
DB 0xFC, 0x4A, 0x4A, 0x4A, 0x34 ; sharp-s or beta
DB 0x7E, 0x02, 0x02, 0x06, 0x06
DB 0x02, 0x7E, 0x02, 0x7E, 0x02
DB 0x63, 0x55, 0x49, 0x41, 0x63
DB 0x38, 0x44, 0x44, 0x3C, 0x04
DB 0x40, 0x7E, 0x20, 0x1E, 0x20
DB 0x06, 0x02, 0x7E, 0x02, 0x02
DB 0x99, 0xA5, 0xE7, 0xA5, 0x99
DB 0x1C, 0x2A, 0x49, 0x2A, 0x1C
DB 0x4C, 0x72, 0x01, 0x72, 0x4C
DB 0x30, 0x4A, 0x4D, 0x4D, 0x30
DB 0x30, 0x48, 0x78, 0x48, 0x30
DB 0xBC, 0x62, 0x5A, 0x46, 0x3D
DB 0x3E, 0x49, 0x49, 0x49, 0x00
DB 0x7E, 0x01, 0x01, 0x01, 0x7E
DB 0x2A, 0x2A, 0x2A, 0x2A, 0x2A
DB 0x44, 0x44, 0x5F, 0x44, 0x44
DB 0x40, 0x51, 0x4A, 0x44, 0x40
DB 0x40, 0x44, 0x4A, 0x51, 0x40
DB 0x00, 0x00, 0xFF, 0x01, 0x03
DB 0xE0, 0x80, 0xFF, 0x00, 0x00
DB 0x08, 0x08, 0x6B, 0x6B, 0x08
DB 0x36, 0x12, 0x36, 0x24, 0x36
DB 0x06, 0x0F, 0x09, 0x0F, 0x06
DB 0x00, 0x00, 0x18, 0x18, 0x00
DB 0x00, 0x00, 0x10, 0x10, 0x00
DB 0x30, 0x40, 0xFF, 0x01, 0x01
DB 0x00, 0x1F, 0x01, 0x01, 0x1E
DB 0x00, 0x19, 0x1D, 0x17, 0x12
DB 0x00, 0x3C, 0x3C, 0x3C, 0x3C
DB 0x00, 0x00, 0x00, 0x00, 0x00 ; #255 NBSP
```