```
/**
 * The 15-410 reference kernel.
 *
 * @author Steve Muckle <smuckle@andrew.cmu.edu>
 *
 * Edited by zra for the 2003-2004 season.
 * Edited by mpa for spring 2004.
 * Edited by ajo for spring 2005.
 *
 * Functions for turning keyboard scancodes
 * into chars.
 */
/*@{*/

#include <keyhelp.h>
#include <stdio.h>

#define LSHIFT_KEY_ON      0x001
#define RSHIFT_KEY_ON      0x002
#define LCONTROL_KEY_ON    0x004
#define RCONTROL_KEY_ON    0x008
#define LALT_KEY_ON        0x010
#define RALT_KEY_ON        0x020
#define CAPS_LOCK_ON       0x040
#define NUM_LOCK_ON        0x080
#define EXTENDED_KEY_PRESS 0x100

#define toggle(sc, flag)                 \
            if (keypress == sc) {        \
                if (pressed)             \
                  key_state |= flag;     \
                else key_state &= ~flag; \
                return -1;               \
            }

#define sticky_toggle(sc, flag)  \
        if (keypress == sc) {    \
            if (pressed)         \
              key_state ^= flag; \
            return -1;           \
        }


/**
 *  key_state flags:
 *  Bit 1: Left Shift key.
 *  Bit 2: Right Shift key.
 *  Bit 3: Left Control key.
 *  Bit 4: Right Control key.
 *  Bit 5: Left Alt key.
 *  Bit 6: Right Alt key.
 *  Bit 7: Caps lock on.
 *  Bit 8: Extended key press (0xE0 extension).
 */
static unsigned int key_state = 0;

#define F_ALT     (0 != (LALT_KEY_ON | RALT_KEY_ON) & key_state))
#define F_CONTROL (0 != (LCONTROL_KEY_ON | RCONTROL_KEY_ON) & key_state))
#define F_SHIFT   (0 != ((LSHIFT_KEY_ON | RSHIFT_KEY_ON) & key_state))
#define F_CAPS    (0 != (CAPS_LOCK_ON & key_state))
#define F_NUM     (0 != (NUM_LOCK_ON & key_state))
#define F_UPPER   (F_CAPS != F_SHIFT)
#define TRY_SHIFT(a, A)  (F_SHIFT? A: a)
#define TRY_UPPER(a, A)  (F_UPPER? A: a)

/**
 * This function performs the mapping
 * from simple scancodes to chars.
 *
 * @param scancode a simple scancode.
 *
 * @return the corresponding character.
 */
int
scan_to_ascii(int scancode)
{
    switch (scancode)
    {
        case 1: return 27; /* Escape */
        case 2: return TRY_SHIFT(’1’, ’!’);
        case 3: return TRY_SHIFT(’2’, ’@’);
        case 4: return TRY_SHIFT(’3’, ’#’);
        case 5: return TRY_SHIFT(’4’, ’$’);
        case 6: return TRY_SHIFT(’5’, ’%’);
        case 7: return TRY_SHIFT(’6’, ’^’);
        case 8: return TRY_SHIFT(’7’, ’&’);
        case 9: return TRY_SHIFT(’8’, ’*’);
        case 10: return TRY_SHIFT(’9’, ’(’);
        case 11: return TRY_SHIFT(’0’, ’)’);
        case 12: return TRY_SHIFT(’-’, ’_’);
        case 13: return TRY_SHIFT(’=’, ’+’);
        case 14: return ’\b’; /* Backspace */
        case 15: return ’\t’; /* Tab */
        case 16: return TRY_UPPER(’q’, ’Q’);
        case 17: return TRY_UPPER(’w’, ’W’);
        case 18: return TRY_UPPER(’e’, ’E’);
        case 19: return TRY_UPPER(’r’, ’R’);
        case 20: return TRY_UPPER(’t’, ’T’);
        case 21: return TRY_UPPER(’y’, ’Y’);
        case 22: return TRY_UPPER(’u’, ’U’);
        case 23: return TRY_UPPER(’i’, ’I’);
        case 24: return TRY_UPPER(’o’, ’O’);
        case 25: return TRY_UPPER(’p’, ’P’);
        case 26: return TRY_SHIFT(’[’, ’{’);
        case 27: return TRY_SHIFT(’]’, ’}’);
        case 28: return ’\n’; /* Enter */
        case 29: return ’?’; /* Left control; should never reach here */
        case 30: return TRY_UPPER(’a’, ’A’);
        case 31: return TRY_UPPER(’s’, ’S’);
        case 32: return TRY_UPPER(’d’, ’D’);
        case 33: return TRY_UPPER(’f’, ’F’);
        case 34: return TRY_UPPER(’g’, ’G’);
        case 35: return TRY_UPPER(’h’, ’H’);
        case 36: return TRY_UPPER(’j’, ’J’);
        case 37: return TRY_UPPER(’k’, ’K’);
        case 38: return TRY_UPPER(’l’, ’L’);
        case 39: return TRY_SHIFT(’;’, ’:’);
        case 40: return TRY_SHIFT(’\’’, ’"’);
        case 41: return TRY_SHIFT(’‘’, ’~’);
        case 42: return -1; /* Left shift; should never reach here */
        case 43: return TRY_SHIFT(’\\’, ’|’);
        case 44: return TRY_UPPER(’z’, ’Z’);
        case 45: return TRY_UPPER(’x’, ’X’);
        case 46: return TRY_UPPER(’c’, ’C’);
        case 47: return TRY_UPPER(’v’, ’V’);
        case 48: return TRY_UPPER(’b’, ’B’);
        case 49: return TRY_UPPER(’n’, ’N’);
        case 50: return TRY_UPPER(’m’, ’M’);
        case 51: return TRY_SHIFT(’,’, ’<’);
        case 52: return TRY_SHIFT(’.’, ’>’);
        case 53: return TRY_SHIFT(’/’, ’?’);
        case 54: return -1; /* Right shift; should never reach here */
        case 55: return ’*’; /* Numpad asterisk */
        case 56: return -1; /* Left alt; should never reach here */
        case 57: return ’ ’; /* Space bar */
        case 58: return -1; /* Caps lock; should never reach here */
        case 59: return CUR_F1;
        case 60: return CUR_F2;
        case 61: return CUR_F3;
        case 62: return CUR_F4;
        case 63: return CUR_F5;
        case 64: return CUR_F6;
        case 65: return CUR_F7;
        case 66: return CUR_F8;
        case 67: return CUR_F9;
        case 68: return CUR_F10;
        case 69: return -1; /* Num lock; should never reach here */
        case 70: return CUR_SCROLL_LOCK;
        case 71: return CUR_HOME;
        case 72: return CUR_UP;
        case 73: return CUR_PGUP;
        case 74: return ’-’; /* Numpad minus */
        case 75: return CUR_LEFT;
        case 76: return CUR_FIVE;
        case 77: return CUR_RIGHT;
        case 78: return ’+’; /* Numpad plus */
        case 79: return CUR_END;
        case 80: return CUR_DOWN;
        case 81: return CUR_PGDN;
        case 82: return CUR_INS;
        case 83: return CUR_DEL;
        case 84: return -1; /* Unassigned */
        case 85: return -1; /* Unassigned */
        case 86: return -1; /* Unassigned */
        case 87: return CUR_F11;
        case 88: return CUR_F12;
        default: return -1; /* Unassigned */
    }

    /* This line is never executed. */
    return -1;
}

/**
 * Processes some special extended scancodes beginning with 224.
 * Note that the cluster machines’ keyboards also have two keys marked
 * "Page Left" and "Page Right," which generate the same scancodes as
 * "Left Alt + Non-Numpad Left Arrow" and "Left Alt + Non-Numpad Right
 * Arrow," respectively.
 *
 * @param keypress the extended scancode.
 * @param 0 if released. non-zero if pressed.
 *
 * @return -1 if we recognized the press.
 *         ’?’ otherwise.
 */
int
extended_scan_to_ascii(int keypress, int pressed)
{
    key_state &= ~EXTENDED_KEY_PRESS;

    toggle(29, RCONTROL_KEY_ON);
    toggle(56, RALT_KEY_ON);

    if (!pressed) return -1;

    switch (keypress)
    {
        case 28: return ’\n’; /* Numpad enter */
        case 42: return CUR_PRINTSCR; /* Print Screen */
        case 53: return ’/’; /* Numpad slash */
        case 71: return CUR_HOME; /* Non-numpad Home */
        case 72: return CUR_UP; /* Non-numpad Up */
        case 73: return CUR_PGUP; /* Non-numpad Page Up */
        case 75: return CUR_LEFT; /* Non-numpad Left */
        case 77: return CUR_RIGHT; /* Non-numpad Right */
        case 79: return CUR_END; /* Non-numpad End */
        case 80: return CUR_DOWN; /* Non-numpad Down */
        case 81: return CUR_PGDN; /* Non-numpad Page Down */
        case 82: return CUR_INS; /* Non-numpad Insert */
        case 83: return CUR_DEL; /* Non-numpad Delete */
        case 91: return CUR_WINDOWS; /* Left "Windows" key */
        case 92: return CUR_WINDOWS; /* Right "Windows" key */
        case 93: return CUR_MENU; /* Windows "Menu" key */
        default: return -1; /* Unassigned */
    }
}

/**
 * Converts keyboard scan codes into ascii chars
 * by calling scan_to_ascii and extended_scan_to_ascii.
 * Keeps track of shift, caps lock etc.
 *
 * Note that the "Pause" key generates the extended scancode 225 29 69
 * 225 157 197 on a single press, and then immediately freezes Simics.
 * We only extract the 225, leaving a bogus "Left Control + Num Lock"
 * waiting on the scancode buffer. This is broken, but harmless, given
 * Simics’ behavior under the circumstances.
 *
 * @param keypress A scancode retrieved from the keyboard
 *
 * @return The corresponding ASCII character
 */
int
process_scancode(int keypress)
{
    int pressed = !(keypress & 0x80);

    if (keypress == 224) {
        /* Extended key code. Set extended marker. */
        key_state |= EXTENDED_KEY_PRESS;
        return -1;
    }

    if (keypress == 225) {
        /* The "Pause" key. */
        return CUR_PAUSE;
    }

    keypress &= 0x7F;

    if (key_state & EXTENDED_KEY_PRESS)
      return extended_scan_to_ascii(keypress, pressed);

    toggle(42, LSHIFT_KEY_ON);
    toggle(54, RSHIFT_KEY_ON);
    toggle(29, LCONTROL_KEY_ON);
    toggle(56, LALT_KEY_ON);
    sticky_toggle(58, CAPS_LOCK_ON);
    sticky_toggle(69, NUM_LOCK_ON);

    /*
       If we’ve gotten this far, it’s just a normal key.
       If it’s a release, we don’t care about it.
    */
    if (!pressed) return -1;
    return scan_to_ascii(keypress);
}
```

```
/*@}*/
```

```
#ifndef H_KEYHELP
 #define H_KEYHELP

/* The index of the IDT entry for the keyboard handler */
#define KEY_IDT_ENTRY 0x21

/* The port from which keyboard scancodes are retrieved */
#define KEYBOARD_PORT 0x60

/* The function mapping scancodes to characters */
int process_scancode(int keypress);

/* Macro values returned from process_scancode() */
#define CUR_HOME  129
#define CUR_UP    130
#define CUR_PGUP  131
#define CUR_LEFT  132
#define CUR_FIVE  133
#define CUR_RIGHT 134
#define CUR_PGDN  135
#define CUR_DOWN  136
#define CUR_END   137
#define CUR_INS   138
#define CUR_DEL   139

#define CUR_SCROLL_LOCK  140
#define CUR_WINDOWS      141
#define CUR_MENU         142
#define CUR_PAUSE        143
#define CUR_PRINTSCR     144

#define CUR_F1    145
#define CUR_F2    146
#define CUR_F3    147
#define CUR_F4    148
#define CUR_F5    149
#define CUR_F6    150
#define CUR_F7    151
#define CUR_F8    152
#define CUR_F9    153
#define CUR_F10   154
#define CUR_F11   155
#define CUR_F12   156

#endif
```

```
/*
   This is the main source file for the "Bomb" game. All the game-specific
   routines are here, except for the timer handler, which is in "tick.c".
   That file and this one communicate through the global structure
   |global_game_state| and the global state variable
   |global_tick_behavior|; for those declarations, see "bomb_game.h".
*/

#include <console.h>
#include <ctype.h>
#include "kernkbd.h"
#include "kerntimer.h"
#include <keyhelp.h>
#include <levels.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bomb_game.h"

/* Menu options. */
#define ME_PLAY_GAME  0
#define ME_TURBO      1
#define ME_MAX        2  /* This must be the last item in this list */


volatile struct game_state_t global_game_state;

static void splash_screen(void);
static int main_menu(void);
static int pick_idx(void);
 static unsigned int randint0(unsigned int m);
static void do_level(const char *word);
 static int guess(const char *word, int k, char *guessed);
static void do_turbo_setting(void);


/*
   The |game_run| routine is the core of the game; it controls
   everything game-related.
*/
void
game_run(void)
{
    int rc;

    global_tick_behavior = TB_NOTHING;
    global_game_state.seed = 1;
    global_game_state.ticks_per_fuse = 100;
    hide_cursor();

    /* Produce the splash screen and main menu. */
    splash_screen();

    while (1) {
        rc = main_menu();
        if (rc == ME_PLAY_GAME) {
            /* Pick a new word. */
            int idx = pick_idx();
            const char *word = game_strings[idx];

            /* Play a level with that word. */
            do_level(word);
        }
        else if (rc == ME_TURBO) {
            do_turbo_setting();
        }
```

```
    }
}


/*
   This function keeps track of the past 100 indices selected,
   to ensure that we never pick the same index twice within that
   window. If |num_game_strings < 200|, then our queue has size
   |num_game_strings/2| --- we could make it bigger, but then
   it would be really easy to guess which index would be chosen
   next, after the majority of indices had been chosen once.
*/
static int
pick_idx(void)
{
    static unsigned int chosen[100] = {0};
    static unsigned int timestamp = 100;

    if (num_game_strings <= 100) {
        int i, r;
        int sum = 0;

        timestamp += 1;
        if (timestamp == 0) {
            /*
               This is total paranoia. Who would play the game
               for 2^32 rounds?
            */
            memset(chosen, 0x00, sizeof chosen);
            timestamp = 100;
        }

        for (i=0; i < num_game_strings; ++i)
          sum += (chosen[i] <= timestamp - num_game_strings/2);
        r = randint0(sum);
        for (i=0; i < num_game_strings; ++i) {
            r -= (chosen[i] <= timestamp - num_game_strings/2);
            if (r < 0) return i;
        }
    }
    else {
        return randint0(num_game_strings);
    }

    /* This line is never executed. */
    return 0;
}


/*
   This function returns a random integer $0 <= r < m$.
   We use a linear congruential generator which has been shown
   not to be too bad (not that it matters, in practice).
*/
static unsigned int
randint0(unsigned int m)
{
#define M 0x7FFFFFFF
#define A 16807
#define B 0
    global_game_state.seed = A*global_game_state.seed + B;
    global_game_state.seed %= M;
    return global_game_state.seed % m;
}


static void
```

```
do_level(const char *word)
{
    char guessed[100];
    int i;

    global_tick_behavior = TB_NOTHING;
    clear_console();

    /* Draw the blanks for the word. */
    set_cursor(WORD_Y, WORD_X);
    set_term_color(15);
    for (i=0; word[i] != ’\0’; ++i) {
         int k = (word[i] == ’ ’)? ’ ’: ’_’;
         putbyte(k);
         guessed[i] = k;
    }
    guessed[i] = ’\0’;

    /* Draw the initial fuse. */
    global_game_state.fuse = INITIAL_FUSE;
    set_cursor(FUSE_Y, FUSE_X);
    set_term_color(8);
    for (i=0; i < INITIAL_FUSE; ++i)
      putbytes("-", 1);
    set_term_color(14);
    putbytes("*", 1);

    /* Play the game. */
    global_tick_behavior = TB_PLAYING;
    while (global_game_state.fuse) {
        int k = readchar();
        if (k == -1) continue;
        if (isalpha(k)) {
            int rc = guess(word, k, guessed);
            if (rc < 0) {
                global_game_state.fuse -= 1;
                draw_char(FUSE_Y, FUSE_X+global_game_state.fuse, ’*’, 14);
                draw_char(FUSE_Y, FUSE_X+global_game_state.fuse+1, ’ ’, 0);
            }
            else if (rc > 0) {
                global_tick_behavior = TB_NOTHING;
                break;
            }
        }
    }

    /* Did the player win or lose? */
    if (global_game_state.fuse > 0) {
        set_cursor(MESSAGE_Y, MESSAGE_X);
        set_term_color(10);
        printf("You win!");
    }
    else {
        for (i=0; word[i] != ’\0’; ++i) {
            if (guessed[i] == ’_’)
              draw_char(WORD_Y, WORD_X+i, word[i], 12);
        }
        set_cursor(MESSAGE_Y, MESSAGE_X);
        set_term_color(15);
        printf("You lose!");
    }

    global_game_state.fuse = 80;
    global_tick_behavior = TB_PAUSE;
    while (readchar() == -1 && global_game_state.fuse) continue;

    global_tick_behavior = TB_NOTHING;
    return;
}


static int
guess(const char *word, int k, char *guessed)
{
    int i;
    int done = 1;
    int success = 0;
    for (i=0; word[i] != ’\0’; ++i) {
        if (tolower(word[i]) == tolower(k)) {
            guessed[i] = word[i];
            draw_char(WORD_Y, WORD_X+i, word[i], 15);
            success = 1;
        }
        else if (guessed[i] == ’_’)
          done = 0;
    }
    return done? 1: success? 0: -1;
}


/*
   Print a splash screen to start the game.
*/
static void
splash_screen(void)
{
    int i;
    const char splashpic[] =
"                                                                                "
"                                                                                "
"                                                                                "
"                                               \016.
 "
"                                              \007.- \016.’
     "
"\017             888 8888 888  8888           \007  .’ .\016V   ,
         "
"\017             88P 8888 Y88  8888           \007  :\016‘\007:\016>*< ’
                 "
"\017             8P  8888  Y8  8888 88b.   d88 8b,\007 \\\016A\007_ ,
              "
"\017                 8888      8888 8888  8888 oo8b\007._ ‘-..\010o.
         "
"\017                 8888      8888 8888  8888   ,o  \007‘\010.od888o  _____
         "
"\017                 8888      8888 8888   Y88o8P\"   \010.d88888888888888bo.
      "
"                                                  Y888888888888888888b.         "
"\017    8888 88b.                          8888  \010   \"88888888888888888888b
      "
"\017    8888 8888)                         8888  \010  .d8888888888888888888888.
     "
"\017    8888 888P  .d88 88b.  8888 8888 88b.  8888 88b\010<888888888888888888888888
     "
"\017    8888 8888o 8888 8888  8888 8888 8888  8888 8888\010I88888888888888888888888
     "
"\017    8888 88888 8888 8888  8888 8888 8888  8888 8888\010I88888888888888888888888
     "
"\017    8888 888P\" \"Y88 88P\"  8888 8888 8888  8888 88Y\010<888888888888888888888
88P     "
"\010                                                   Y888888888888888888888P
 "
"                                                    \"Y88888888888888888P\"       "
"                                                      \"Y8888888888888P\"         "
```

```c
"                                                         \"\"Y8888P\"\"             "
"                                                                                "
"                                                                                "
"                                                                               "
;

    global_tick_behavior = TB_NOTHING;
    clear_console();
    hide_cursor();
    set_cursor(0, 0);

    /* Draw a pretty picture. */
    for (i=0; splashpic[i]; ++i) {
        if (splashpic[i] < 32)
          set_term_color(splashpic[i]);
        else putbyte(splashpic[i]);
    }

    global_game_state.fuse = 100;
    global_tick_behavior = TB_PAUSE;
    while (global_game_state.fuse && (readchar() == -1))
      continue;

    global_tick_behavior = TB_NOTHING;
    while (readchar() != -1) continue;
    return;
}


/*
   This routine prints the game’s main menu, and get the user’s selection
   from it. It returns an index from 0 to |ME_MAX-1|.

   Notice that there is no "Quit" option, because there’s nowhere to
   quit /to/ --- we can’t exactly return control of the machine to
   the operating system, because we /are/ the operating system!
*/
static int
main_menu(void)
{
    int selection = 0;

    global_tick_behavior = TB_NOTHING;
    clear_console();

    set_cursor(10, 10);
    set_term_color(15);
    printf("Main Menu");

    while (1) {
        int k;

        set_cursor(12, 10);
        set_term_color(selection==0? 15: 7);
        printf("Play Game");

        set_cursor(14, 10);
        set_term_color(selection==1? 15: 7);
        printf("Turbo Setting");

        while ((k = readchar()) == -1) continue;

        if (k == CUR_UP) selection = (selection+ME_MAX-1) % ME_MAX;
        else if (k == CUR_DOWN) selection = (selection+1) % ME_MAX;
        else if (k == ’\n’) return selection;
    }
    return 0;
```

```c
}


/*
   The |do_turbo_setting| routine allows the user to update the
   turbo control, to make the game’s fuse burn faster or slower.
*/
static void
do_turbo_setting(void)
{
    int saved_tpf;
    int k;

    global_tick_behavior = TB_NOTHING;
    saved_tpf = global_game_state.ticks_per_fuse;
    global_game_state.ticks_per_fuse = 0;
    clear_console();

    hide_cursor();
    set_term_color(14);
    set_cursor(10, 10);
    printf("Please hit any alphabetic key three times, at one-second intervals.\n");
    set_cursor(11, 10);
    printf("Begin any time you like, or hit Escape to cancel.\n");

    /* Wait for the first keypress. */
    while (readchar() != -1) continue;
    while ((k = readchar()) == -1) continue;
    if (k == 27) goto cancel;

    global_tick_behavior = TB_SET_TURBO;
    draw_char(13, 12, ’1’, 2);

    /* Wait for the second keypress. */
    while ((k = readchar()) != -1)
      if (k == 27) goto cancel;
    while ((k = readchar()) == -1) continue;
    if (k == 27) goto cancel;

    draw_char(13, 14, ’2’, 10);

    /* Wait for the third keypress. */
    while ((k = readchar()) != -1)
      if (k == 27) goto cancel;
    while ((k = readchar()) == -1) continue;
    if (k == 27) goto cancel;

    global_tick_behavior = TB_NOTHING;
    draw_char(13, 16, ’3’, 15);

    global_game_state.ticks_per_fuse /= 2;

    if (0) {
        cancel:
        /* The user pressed Escape. Restore the old value and return. */
        global_game_state.ticks_per_fuse = saved_tpf;
        set_term_color(14);
        set_cursor(15, 10);
        printf("Canceled!\n");
        set_cursor(16, 10);
        printf("I’m keeping the old turbo setting, %d ms per game second.\n",
            global_game_state.ticks_per_fuse * 10);
    }
    else if (global_game_state.ticks_per_fuse <= 0) {
        global_game_state.ticks_per_fuse = saved_tpf;
        set_term_color(12);
        set_cursor(15, 10);
        printf("Something went wrong with my reckoning.\n");
        set_cursor(16, 10);
        printf("I’m keeping the old turbo setting, %d ms per game second.\n",
            global_game_state.ticks_per_fuse * 10);
    }
    else {
        set_term_color(14);
        set_cursor(15, 10);
        printf("Thank you.\n");
        set_cursor(16, 10);
        printf("The new turbo setting is %d ms per game second.\n",
            global_game_state.ticks_per_fuse * 10);
    }

    /* Pause for the reader to catch up. */
    set_term_color(14);
    set_cursor(18,10);
    printf("Press any key to continue.");
    while (readchar() == -1) continue;
    while (readchar() != -1) continue;
    return;
}
```

```
#ifndef H_BOMB_GAME
 #define H_BOMB_GAME


/* The initial length of the bomb’s fuse. */
#define INITIAL_FUSE 10

/*
   These positions on the screen are used by the main game routines
   and also by some of the behaviors of |tick| (see below).
*/
#define WORD_X 10
#define WORD_Y 10
#define FUSE_X 10
#define FUSE_Y 12
#define MESSAGE_X 10
#define MESSAGE_Y 13


/*
   The game has several distinct states in which we want to capture
   timer ticks, but only one |tick| routine. Therefore, we define
   several constants representing different states, and whenever we
   want the behavior of |tick| to change, we assign a different state
   to |global_tick_behavior|.
*/
#define TB_PLAYING       10
#define TB_PAUSE         20
#define TB_NOTHING       30
#define TB_SET_TURBO     40

extern unsigned int global_tick_behavior;
extern void tick(unsigned int nticks);

struct game_state_t {
    unsigned int ticks_per_fuse;
    unsigned int fuse;
    unsigned int seed; /* seed for the RNG */
};

/* The |global_game_state| is defined in "bomb_game.c". */
extern volatile struct game_state_t global_game_state;

#endif
```

```
/** @file console.c
 *  @brief A console driver.
 *
 *  @author Harry Q. Bovik (hbovik) <-- change this
 *  @bug None known
 */

#include <string.h>
#include <video_defines.h>
#include <x86/pio.h>
#include <console.h>

#define CONSOLE_TAB_WIDTH 8

static int get_color(int row, int col);

static int logical_cursor_row = 0;
static int logical_cursor_col = 0;
static int logical_cursor_on = 1;
static int logical_term_color = 7;


/** @brief Prints character |ch| at the current location of the cursor.

   If the character is a newline, the cursor moves to the start of the
   next line. If the character is a carriage return, the cursor moves to
   the start of the current line. If the character is a backspace, the
   current character is overwritten with a backspace and the cursor moved
   one column to the left, if possible.

   When the screen scrolls, the areas that "come onto" the screen
   will use the current |logical_term_color|. When a backspace replaces
   the current character with a space, the space will use the existing
   color at that position, rather than the current |logical_term_color|.

   A tab is printed as a sequence of up to 8 space characters using the
   current |logical_term_color|. The cursor is advanced to the next tab
   stop; there is a tab stop every 8 columns (or as defined by
   |CONSOLE_TAB_WIDTH|).

   @return |ch|
*/
int
putbyte(char ch)
{
    putbytes(&ch, 1);
    return ch;
}


/** @brief Prints |len| characters as per |putbyte|.

   This function prints |len| characters from the array |s|, which need
   not be null-terminated. The output follows the same form as the output
   of |putbyte|, and updates the cursor and scrolls the screen as needed.

   The entire array is processed before any output is undertaken, so if
   |putbytes| is used to dump more than one page of output to the screen,
   the first N-1 pages will never be written to video memory and never
   appear on the screen. If this behavior is undesirable, use |putbyte|
   inside a loop instead.

   @param[in] s  An array of length |len|
   @param len  The number of characters to process
   @return  Nothing
*/
void
putbytes(const char *s, int len)
{
    int last_row = logical_cursor_row;
    int col = logical_cursor_col;
    int rows_to_ignore = 0;
    int i;

    if (len <= 0) return;

    for (i=0; i < len; ++i) {
        switch (s[i]) {
            case ’\n’: last_row += 1; col = 0; break;
            case ’\r’: col = 0; break;
            case ’\b’: col -= (col > 0); break;
            case ’\t’: {
                /* Eight-column tabs. */
                int endcol = col + CONSOLE_TAB_WIDTH;
                endcol -= (endcol % CONSOLE_TAB_WIDTH);
                last_row += endcol / CONSOLE_WIDTH;
                col = endcol % CONSOLE_WIDTH;
                break;
            }
            default: {
                col += 1;
                if (col >= CONSOLE_WIDTH) {
                    col = 0;
                    last_row += 1;
                }
            }
        }
    }

    /*
       Now, if |last_row| is at least |CONSOLE_HEIGHT|, we need to
       start pushing text off the top of the screen. Grab the bottom
       $N-k$ rows of the video buffer and copy them up; then fill the
       bottom with blank space. Finally, copy the new content into
       that blank space.
    */
    if (last_row >= CONSOLE_HEIGHT) {
        int first_displayed_row;
        int rows_to_keep, rows_to_trash;

        first_displayed_row = last_row - CONSOLE_HEIGHT;
        if (first_displayed_row < CONSOLE_HEIGHT) {
            rows_to_keep = logical_cursor_row - first_displayed_row;
            rows_to_ignore = 0;
        }
        else {
            rows_to_keep = 0;
            rows_to_ignore = first_displayed_row - logical_cursor_row;
        }
        rows_to_trash = CONSOLE_HEIGHT - rows_to_keep;

        /* Scroll the bottom half of the screen up to the top. */
        memcpy((char*)CONSOLE_MEM_BASE,
               (char*)CONSOLE_MEM_BASE + 2*CONSOLE_WIDTH*rows_to_trash,
               2*CONSOLE_WIDTH*rows_to_keep);

        /* Blank the bottom half of the screen. */
        for (i=CONSOLE_WIDTH*rows_to_keep;
             i < CONSOLE_WIDTH*CONSOLE_HEIGHT; ++i)
        {
            ((char*)CONSOLE_MEM_BASE)[2*i] = ’ ’;
            ((char*)CONSOLE_MEM_BASE)[2*i+1] = logical_term_color;
        }
        /*
           This line makes the logical cursor row go negative. We rely
           on the error-checking in |draw_char| not to actually print
           out anything until the logical cursor row becomes non-negative.
        */
        logical_cursor_row -= last_row - (CONSOLE_HEIGHT-1);
    }

    for (i=0; i < len; ++i) {
        switch (s[i]) {
            case ’\n’: {
                logical_cursor_row += 1;
                logical_cursor_col = 0;
                break;
            }
            case ’\r’: logical_cursor_col = 0; break;
            case ’\b’: {
                int backspace_color;
                logical_cursor_col -= (logical_cursor_col > 0);
                backspace_color =
                    get_color(logical_cursor_row, logical_cursor_col);
                draw_char(logical_cursor_row, logical_cursor_col,
                    ’ ’, backspace_color);
                break;
            }
            case ’\t’: {
                /* Eight-column tabs. */
                int endcol = logical_cursor_col + CONSOLE_TAB_WIDTH;
                int j;
                endcol -= (endcol % CONSOLE_TAB_WIDTH);
                for (j=logical_cursor_col; j < endcol; ++j) {
                    draw_char(logical_cursor_row + j/CONSOLE_WIDTH,
                        j%CONSOLE_WIDTH, ’ ’, logical_term_color);
                }
                logical_cursor_row += endcol / CONSOLE_WIDTH;
                logical_cursor_col = endcol % CONSOLE_WIDTH;
                break;
            }
            default: {
                draw_char(logical_cursor_row, logical_cursor_col,
                    s[i], logical_term_color);
                logical_cursor_col += 1;
                if (logical_cursor_col >= CONSOLE_WIDTH) {
                    logical_cursor_col = 0;
                    logical_cursor_row += 1;
                }
            }
        }
    }

    set_cursor(logical_cursor_row, logical_cursor_col);
    return;
}


int
set_term_color(int color)
{
    if (color < 0 || color > 0xFF) return 0;
    logical_term_color = color;
    return 0;
}


void
get_term_color(int *color)
{
```

```
    *color = logical_term_color;
    return;
}


int
set_cursor(int row, int col)
{
    if (row < 0 || row >= CONSOLE_HEIGHT) return 0;
    if (col < 0 || col >= CONSOLE_WIDTH) return 0;
    logical_cursor_row = row;
    logical_cursor_col = col;
    if (logical_cursor_on)
    {
        /* Update the hardware cursor position. */
        int idx = row*CONSOLE_WIDTH + col;

        outb(CRTC_IDX_REG, 14);
        outb(CRTC_DATA_REG, 0xFF & (idx >> 8));
        outb(CRTC_IDX_REG, 15);
        outb(CRTC_DATA_REG, 0xFF & idx);
    }
    return 0;
}


void
get_cursor(int *row, int *col)
{
    *row = logical_cursor_row;
    *col = logical_cursor_col;
}


void
hide_cursor()
{
    if (!logical_cursor_on) return;
    logical_cursor_on = 0;

    /* Hide the cursor by setting it to an off-screen position. */
    {
        int idx = CONSOLE_WIDTH*CONSOLE_HEIGHT;
        outb(CRTC_IDX_REG, 14);
        outb(CRTC_DATA_REG, 0xFF & (idx >> 8));
        outb(CRTC_IDX_REG, 15);
        outb(CRTC_DATA_REG, 0xFF & idx);
    }
}


void
show_cursor()
{
    if (logical_cursor_on) return;
    logical_cursor_on = 1;
    set_cursor(logical_cursor_row, logical_cursor_col);
    return;
}


void
clear_console()
{
    register int i;
    for (i=0; i < CONSOLE_WIDTH * CONSOLE_HEIGHT; ++i) {
        ((char*)CONSOLE_MEM_BASE)[2*i] = ’ ’;
        ((char*)CONSOLE_MEM_BASE)[2*i + 1] = logical_term_color;
    }
    set_cursor(0, 0);
}


void
draw_char(int row, int col, int ch, int color)
{
    register unsigned idx = 2*(row*CONSOLE_WIDTH + col);
    if (row < 0 || row >= CONSOLE_HEIGHT) return;
    if (col < 0 || col >= CONSOLE_WIDTH) return;
    if (ch < 0 || ch > 255) return;
    if (color < 0 || color > 255) return;
    ((char *)CONSOLE_MEM_BASE)[idx] = ch;
    ((char *)CONSOLE_MEM_BASE)[idx+1] = color;
}


char
get_char(int row, int col)
{
    register unsigned idx = 2*(row*CONSOLE_WIDTH + col);
    if (row < 0 || row >= CONSOLE_HEIGHT) return 0;
    if (col < 0 || col >= CONSOLE_WIDTH) return 0;
    return ((char *)CONSOLE_MEM_BASE)[idx];
}


int
get_color(int row, int col)
{
    register unsigned idx = 2*(row*CONSOLE_WIDTH + col);
    if (row < 0 || row >= CONSOLE_HEIGHT) return 0;
    if (col < 0 || col >= CONSOLE_WIDTH) return 0;
    return ((char *)CONSOLE_MEM_BASE)[idx+1];
}
```

```
/** @file kernel.c
 *  @brief A kernel with timer, keyboard, console support
 *
 *  This file contains the kernel’s
 *  main() function.
 *
 *  It sets up the drivers, and starts the game.
 *
 *  @author Michael Berman (mberman)
 *  @bug No known bugs.
 */

/* -- Includes -- */

#include <410_reqs.h>

/* libc includes. */
#include <stdio.h>        /* for lprintf_kern() */

/* multiboot header file */
#include <multiboot.h>    /* for boot_info */

/* memory includes. */
#include <lmm.public.h>   /* for lmm_remove_free() */

/* x86 specific includes */
#include <x86/seg.h>      /* for install_user_segs() */
#include <x86/pic.h>      /* for pic_init() */
#include <x86/base_irq.h> /* for base_irq_master/slave */

/*
 * state for kernel memory allocation.
 */
extern lmm_t malloc_lmm;

/*
 * Info about system gathered by the boot loader
 */
extern struct multiboot_info boot_info;

/* The game function itself. */
extern void game_run(void);

/** @brief Kernel entrypoint.
 *
 *  This is the entrypoint for the kernel.  It simply sets up the
 *  drivers and passes control off to game_run().
 *
 * @return Does not return
 */
int kernel_main()
{
    /*
     * Tell the kernel memory allocator which memory it can’t use.
     * It already knows not to touch kernel image.
     */

    /* Everything above 16M */
    lmm_remove_free( &malloc_lmm, (void*)USER_MEM_START, -8 - USER_MEM_START );

    /* Everything below 1M  */
    lmm_remove_free( &malloc_lmm, (void*)0, 0x100000 );


    /* Install handlers for timer and keyboard interrupts. */
    if (handler_install() < 0) {
        /* Is this a valid way to error out? Todo fixme bug hack. */
        return 0;
    }

    /*
       Initialize the PIC so that IRQs and
       exception handlers don’t overlap in the IDT.
    */
    pic_init(BASE_IRQ_MASTER_BASE, BASE_IRQ_SLAVE_BASE);

    /* Now run the game, whatever it is. */
    game_run();

    /* This line should never be executed. */
    return 0;
}
```

```
/** @file handler_install.c
 *
 *  @brief Handler installation function
 *
 *  Edit this file to allow your kernel to initialize and install handlers
 *
 *  Declared in 410_reqs.h
 *
 *  @author Arthur O’Dwyer (ajo)
 *  @bug None known, but insufficiently tested
 **/

#include <410_reqs.h>
#include <interrupts.h>
#include <keyhelp.h>
#include <timer_defines.h>
#include <x86/seg.h>
#include <x86/proc_reg.h>
#include <kerndebug.h>
#include "kerntimer.h"
#include "kernkbd.h"
#include "wrappers.h"

static void install_one_handler(unsigned idx, void (*handlerfp)());


/** @brief Install handlers and set the initial timer rate.
 * @return always zero
 */
int
handler_install(void)
{
    disable_interrupts();
    set_turbo_rate(TIMER_RATE/200);  /* 200 interrupts per second */
    install_one_handler(TIMER_IDT_ENTRY, asm_timer_wrapper);
#if 0
    asm_set_A20();
#endif
    install_one_handler(KEY_IDT_ENTRY, asm_kbd_wrapper);
    enable_interrupts();
    return 0;
}


/** @brief Install a single handler in the IDT entry given by |idx|.
 *
 * Note that |handlerfp| must be a real interrupt handler; it needs to
 * save the registers on entry, restore them on exit, and finish with
 * an IRET instruction. Otherwise, bad things will happen.
 *
 * @param idx  The IDT entry to update
 * @param handerfp  The address of the interrupt handler
 */
static void
install_one_handler(unsigned idx, void (*handlerfp)())
{
    unsigned char *idt_base_address = sidt();
    const unsigned int present = 1;
    const unsigned int DPL = 0;  /* Privilege level */
    const unsigned int D = 1;  /* 1=32-bit gate */
    unsigned int offset = (unsigned)handlerfp;
    unsigned int segsel = KERNEL_CS_SEGSEL;
    unsigned char *gate = &idt_base_address[8*idx];
    unsigned int gate_0;
    unsigned int gate_1;

    gate_0 = segsel << 16
           | (offset & 0xFFFF);
    gate_1 = (offset & ~0xFFFF)
           | (present << 15)
           | (DPL << 13)
           | (D << 11)
           | 0x600;

    gate[0] = gate_0 >> 0;
    gate[1] = gate_0 >> 8;
    gate[2] = gate_0 >> 16;
    gate[3] = gate_0 >> 24;
    gate[4] = gate_1 >> 0;
    gate[5] = gate_1 >> 8;
    gate[6] = gate_1 >> 16;
    gate[7] = gate_1 >> 24;
    return;
}
```

```c
/** @file console.h
 *  @brief Function prototypes for the console driver.
 *
 *  This contains the prototypes and global variables for the console
 *  driver
 *
 *  @author Michael Berman (mberman)
 *  @bug No known bugs.
 */

#ifndef _CONSOLE_H
#define _CONSOLE_H

#include <video_defines.h>

/** @brief Prints character ch at the current location
 *         of the cursor.
 *
 *  If the character is a newline ('\n'), the cursor is
 *  be moved to the beginning of the next line (scrolling if necessary).  If
 *  the character is a carriage return ('\r'), the cursor
 *  is immediately reset to the beginning of the current
 *  line, causing any future output to overwrite any existing
 *  output on the line.  If backsapce ('\b') is encountered,
 *  the previous character is erased.  See the main console.c description
 *  for more backspace behavior.
 *
 *  @param ch the character to print
 *  @return The input character
 */
int putbyte( char ch );

/** @brief Prints the string s, starting at the current
 *         location of the cursor.
 *
 *  If the string is longer than the current line, the
 *  string fills up the current line and then
 *  continues on the next line. If the string exceeds
 *  available space on the entire console, the screen
 *  scrolls up one line, and then the string
 *  continues on the new line.  If '\n', '\r', and '\b' are
 *  encountered within the string, they are handled
 *  as per putbyte. If len is not a positive integer or s
 *  is null, the function has no effect.
 *
 *  @param s The string to be printed.
 *  @param len The length of the string s.
 *  @return Void.
 */
void putbytes(const char* s, int len);

/** @brief Changes the foreground and background color
 *         of future characters printed on the console.
 *
 *  If the color code is invalid, the function has no effect.
 *
 *  @param color The new color code.
 *  @return 0 on success or integer error code less than 0 if
 *          color code is invalid.
 */
int set_term_color(int color);

/** @brief Writes the current foreground and background
 *         color of characters printed on the console
 *         into the argument color.
 *  @param color The address to which the current color
 *         information will be written.
 *  @return Void.
 */
void get_term_color(int* color);

/** @brief Sets the position of the cursor to the
 *         position (row, col).
 *
 *  Subsequent calls to putbytes should cause the console
 *  output to begin at the new position. If the cursor is
 *  currently hidden, a call to set_cursor() does not show
 *  the cursor.
 *
 *  @param row The new row for the cursor.
 *  @param col The new column for the cursor.
 *  @return 0 on success or integer error code less than 0 if
 *          cursor location is invalid.
 */
int set_cursor(int row, int col);

/** @brief Writes the current position of the cursor
 *         into the arguments row and col.
 *  @param row The address to which the current cursor
 *         row will be written.
 *  @param col The address to which the current cursor
 *         column will be written.
 *  @return Void.
 */
void get_cursor(int* row, int* col);

/** @brief Hides the cursor.
 *
 *  Subsequent calls to putbytes do not cause the
 *  cursor to show again.
 *
 *  @return Void.
 */
void hide_cursor();

/** @brief Shows the cursor.
 *
 *  If the cursor is already shown, the function has no effect.
 *
 *  @return Void.
 */
void show_cursor();

/** @brief Clears the entire console.
 *
 * The cursor is reset to the first row and column
 *
 *  @return Void.
 */
void clear_console();

/** @brief Prints character ch with the specified color
 *         at position (row, col).
 *
 *  If any argument is invalid, the function has no effect.
 *
 *  @param row The row in which to display the character.
 *  @param col The column in which to display the character.
 *  @param ch The character to display.
 *  @param color The color to use to display the character.
 *  @return Void.
 */
void draw_char(int row, int col, int ch, int color);
```

```
/** @brief Returns the character displayed at position (row, col).
 *  @param row Row of the character.
 *  @param col Column of the character.
 *  @return The character at (row, col).
 */
char get_char(int row, int col);

#endif /* _CONSOLE_H */
```

```
/** @file kernkbd.c
 *
 *  @brief Keyboard routines
 *
 *  This file contains routines related to the keyboard interrupt handler.
 *  For the installation of the handler, see "handler_install.c"; for
 *  the assembly wrapper around the handler, see "wrappers.S".
 *
 *  @author Arthur O'Dwyer (ajo)
 *  @bug Not implemented
 **/

#include <interrupts.h>
#include "kernkbd.h"
#include <keyhelp.h>
#include <stdio.h>
#include <x86/proc_reg.h>

/*
   The keyboard buffer is implemented as a circular queue.
   The user is allowed to specify the behavior when the queue
   fills up --- are the first-received scancodes dropped, or
   do we simply stop storing scancodes until some input has
   been consumed? The |kbd_buffer_mode| flag controls this
   behavior, via the |set_kbd_buffer_mode| function.
*/
#define BUFFER_SIZE 256

#define KBD_SAVEFIRST 0
#define KBD_SAVELAST  1

static int kbd_buffer_mode = KBD_SAVELAST;
static unsigned int keybuffer[BUFFER_SIZE];
static unsigned int buffront = 0;
static unsigned int bufback = 0;
static unsigned int bufempty = 1;

/** @brief Push scancodes onto the back of the queue.

   This routine's job is to push scancodes onto the keyboard
   buffer. Data is pushed on the back of the queue and read
   off the front. If the queue is full (that is, if |buffront==bufback|
   and not |bufempty|), then our behavior depends on the current
   setting of |kbd_buffer_mode|.
*/
void c_kbd_bottom(unsigned int scancode)
{
    disable_interrupts();
    /* Drop incoming data if the buffer is full. */
    if (kbd_buffer_mode == KBD_SAVEFIRST) {
        if (!bufempty && (buffront == bufback))
          return;
    }
    /*
       In all other cases, the new data goes into the queue.
       If we need to drop some data, drop it from the front.
    */
    if (kbd_buffer_mode == KBD_SAVELAST) {
        if (!bufempty && (buffront == bufback))
          buffront = (buffront+1) % BUFFER_SIZE;
    }
    bufempty = 0;
    keybuffer[bufback] = scancode;
    bufback = (bufback+1) % BUFFER_SIZE;
    enable_interrupts();
    return;
}
```

```

/** @brief Read a character from the keyboard.

   This function reads through the keyboard buffer's queue, pulling
   off scancodes and examining them. Any scancode that corresponds
   to a character is processed and returned; other scancodes are
   processed for changes to the keyboard state (e.g., depressing the
   Shift key) and then discarded. Keyboard state processing happens
   inside |process_scancode|.

   If the queue is empty, or contains only non-character scancodes,
   then this function returns $-1$. It does not block waiting for
   input.

   @return The first character read, or $-1$ if the queue is empty
*/
int readchar(void)
{
    int rc;

    if (bufempty) return -1;

    do {
        disable_interrupts();
        rc = keybuffer[buffront];
        buffront = (buffront+1) % BUFFER_SIZE;
        enable_interrupts();
        rc = process_scancode(rc);
    } while ((rc == -1) && (buffront != bufback));

    disable_interrupts();
    if (buffront == bufback)
      bufempty = 1;
    enable_interrupts();

    return rc;
}


/** @brief Read a scancode from the keyboard.

   This function reads a single scancode and returns it, uninterpreted,
   to the caller. |process_scancode| is called, and the result discarded,
   just so that the state of the Shift key and so on is maintained
   properly. This should allow the mixing of calls to |readchar| and
   |read_scancode|, but I still think that would be a bad idea.

   If the queue is empty, then this function returns $-1$. It does not
   block waiting for input.

   @return The first scancode available, or $-1$ if the queue is empty
*/
int read_scancode(void)
{
    int rc;

    if (bufempty) return -1;

    do {
        disable_interrupts();
        rc = keybuffer[buffront];
        buffront = (buffront+1) % BUFFER_SIZE;
        enable_interrupts();
        (void)process_scancode(rc);
    } while ((rc == -1) && (buffront != bufback));
    disable_interrupts();
    if (buffront == bufback)
      bufempty = 1;
    enable_interrupts();

    return rc;
}
```

```
#ifndef H_KERNKBD
 #define H_KERNKBD

/*
   This function returns the next character from the keyboard, without
   blocking. It returns $-1$ if no characters are in the input buffer.
*/
int readchar(void);

/*
   This function returns the next /scancode/ from the keyboard, without
   blocking, or $-1$ if the buffer is empty. It differs from |readchar|
   in that it will return non-negative values for events such as the
   Shift key being depressed or released. Mixing calls to |read_scancode|
   and |readchar| is probably a bad idea.
*/
int read_scancode(void);

#endif
```

```c
/** @file kerntimer.c
 *
 *  @brief Timer routines
 *
 *  This file contains routines related to the timer interrupt handler.
 *  For the installation of the handler, see "handler_install.c"; for
 *  the assembly wrapper around the handler, see "wrappers.S".
 *
 *  @author Arthur O'Dwyer (ajo)
 *  @bug Not implemented
 **/

#include <interrupts.h>
#include <x86/pio.h>
#include <timer_defines.h>
#include "kerntimer.h"
#include <kerndebug.h>

extern void tick(unsigned int nticks);

static unsigned int global_turbo_rate = 0;


/*
   Set up the number of timer cycles between interrupts.
   That speed is controlled via |global_turbo_rate|.
*/
void set_turbo_rate(int new_rate)
{
    if (new_rate < 0) return;
    global_turbo_rate = new_rate;

    outb(TIMER_MODE_IO_PORT, TIMER_SQUARE_WAVE);
    outb(TIMER_PERIOD_IO_PORT, global_turbo_rate >> 8);
    outb(TIMER_PERIOD_IO_PORT, global_turbo_rate & 0xFF);
    return;
}


int get_turbo_rate(void)
{
    return global_turbo_rate;
}


void c_timer_bottom(void)
{
    static unsigned int nticks = 0;
    tick(nticks++);
    return;
}
```

```
#ifndef H_KERNTIMER
 #define H_KERNTIMER

void set_turbo_rate(int new_rate);
int get_turbo_rate(void);

#endif
```

```
/*
   This file is a drop-in replacement for "bomb_game.c". I used it
   to test the new |process_scancode| function and my console driver.

   Arthur O’Dwyer, 1 February 2006
*/

#include <console.h>
#include <ctype.h>
#include <kerndebug.h>
#include "kernkbd.h"
#include "kerntimer.h"
#include <keyhelp.h>
#include <levels.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bomb_game.h"

volatile struct game_state_t global_game_state;

void game_run(void)
{
    int i;

    global_tick_behavior = TB_NOTHING;
    global_game_state.seed = 1;

    set_term_color(7);
    clear_console();
    for (i=0; i < 18; ++i)
      printf("Line %d\n", i);

    while (1) {
        int rc = readchar();
        if (rc == -1) continue;
        putbyte(rc);
        if (rc == 27) {
            int col;
            get_term_color(&col);
            col++;
            set_term_color(col);
        }
    }
}
```

```
/** @file tick.c
 *  @brief Tick function, to be called by the timer interrupt handler
 *
 *  Fill in this tick function with any processing your game needs
 *  to do on each timer interrupt. This function should be called from your
 *  timer interrupt handler, even if it is empty.
 *
 *  Function declared in 410_reqs.h
 *
 *  @author Arthur O’Dwyer (ajo)
 *  @bug Not implemented
 **/

#include <410_reqs.h>
#include <console.h>
#include "bomb_game.h"

unsigned int global_tick_behavior;

void
tick(unsigned int numTicks)
{
    switch (global_tick_behavior)
    {
        case TB_PAUSE:
            global_game_state.fuse -= 1;
            if (numTicks % 4 == 0)
              global_game_state.seed += 1;
            return;
        case TB_PLAYING:
            if (numTicks % global_game_state.ticks_per_fuse == 0) {
                global_game_state.fuse -= 1;
                draw_char(FUSE_Y, FUSE_X+global_game_state.fuse, ’*’, 14);
                draw_char(FUSE_Y, FUSE_X+global_game_state.fuse+1, ’ ’, 0);
            }
            return;
        case TB_SET_TURBO:
            global_game_state.ticks_per_fuse += 1;
            break;
        case TB_NOTHING:
        default:
            return;
    }
}
```

```
/** @brief Handle the bookkeeping of timer and keyboard interrupts
 *  before passing control on to the |c_foo_bottom| routines for
 *  queueing of the actual data. See "kerntimer.c" and "kernkbd.c"
 *  for the implementation of the meat of these interrupts.
 *
 *  @author Arthur O'Dwyer (ajo)
 *  @bug None known.
 */

/*
   These #defines must match the values #defined in "keyhelp.h"
   and "interrupts.h".
*/
#define INT_CTL_DONE   0x20
#define INT_CTL_REG    0x20
#define KEYBOARD_PORT  0x60

.globl asm_timer_wrapper
asm_timer_wrapper:
    /*
       The stack on entry to this routine contains the following
       data:
         EFLAGS
         CS
         EIP
    */
     cli
    pusha                                 # Push the register set.
    call    c_timer_bottom                # Handle the interrupt.
    movb    $INT_CTL_DONE, %al
    outb    %al, $INT_CTL_REG             # All done!
    popa                                  # Pop the register set.
    sti
    iret

.globl asm_kbd_wrapper
asm_kbd_wrapper:
    cli
    pusha                                 # Push the register set.
    xor     %eax, %eax
    inb     $KEYBOARD_PORT, %al           # Read a byte from the device.
    push    %eax
    call    c_kbd_bottom                  # Handle that byte.
    pop     %eax
    movb    $INT_CTL_DONE, %al
    outb    %al, $INT_CTL_REG             # All done!
    popa                                  # Pop the register set.
    sti
    iret

.globl asm_set_A20
asm_set_A20:
    movl    $0x2401, %eax
    int     $0x15
    ret
```

```
#ifndef H_WRAPPERS
 #define H_WRAPPERS

void asm_timer_wrapper(void);
void asm_kbd_wrapper(void);
void asm_set_A20(void);

#endif
```