CAB202 Topic 7 – Introduction to Teensy

Lawrence Buckingham, Queensland University of Technology.

Contents

References

Recommended reading:

- Practical C Programming, Chapter 11 Bit Operators

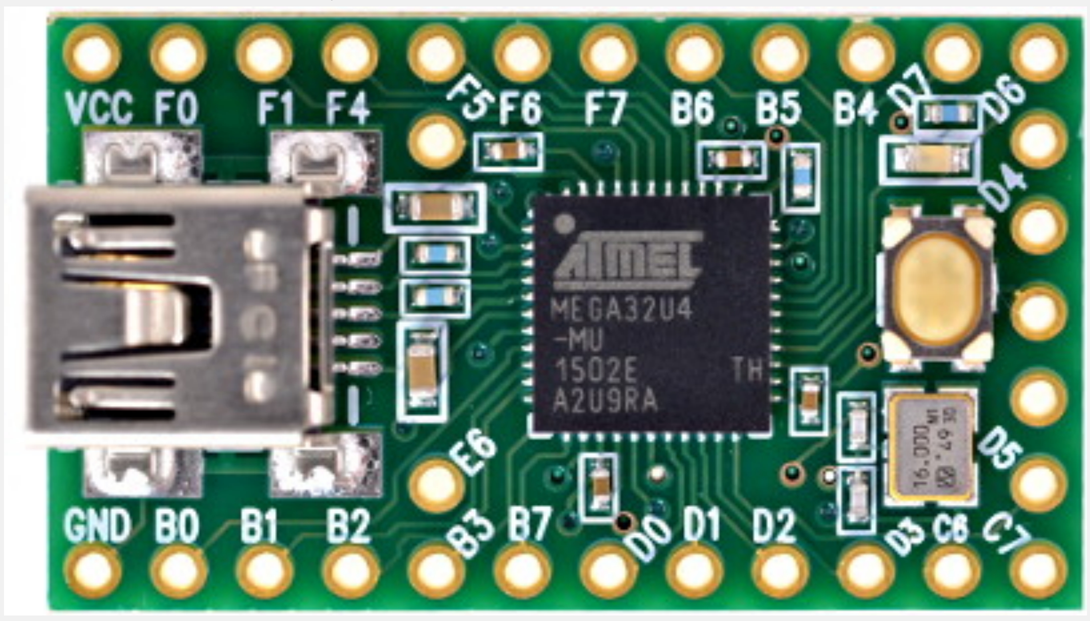
What's a Teensy?

According to the manufacturer's web site, a Teensy is a "… *complete USB-based microcontroller development system, in a very small footprint, capable of implementing many types of projects*.".

- https://www.pjrc.com/teensy/

Teensy uses a range of ATMEL AVR microcontroller chips, which are packaged on a small development board with integrated USB connector for power and serial communication.

Our board is Teensy 2.0:

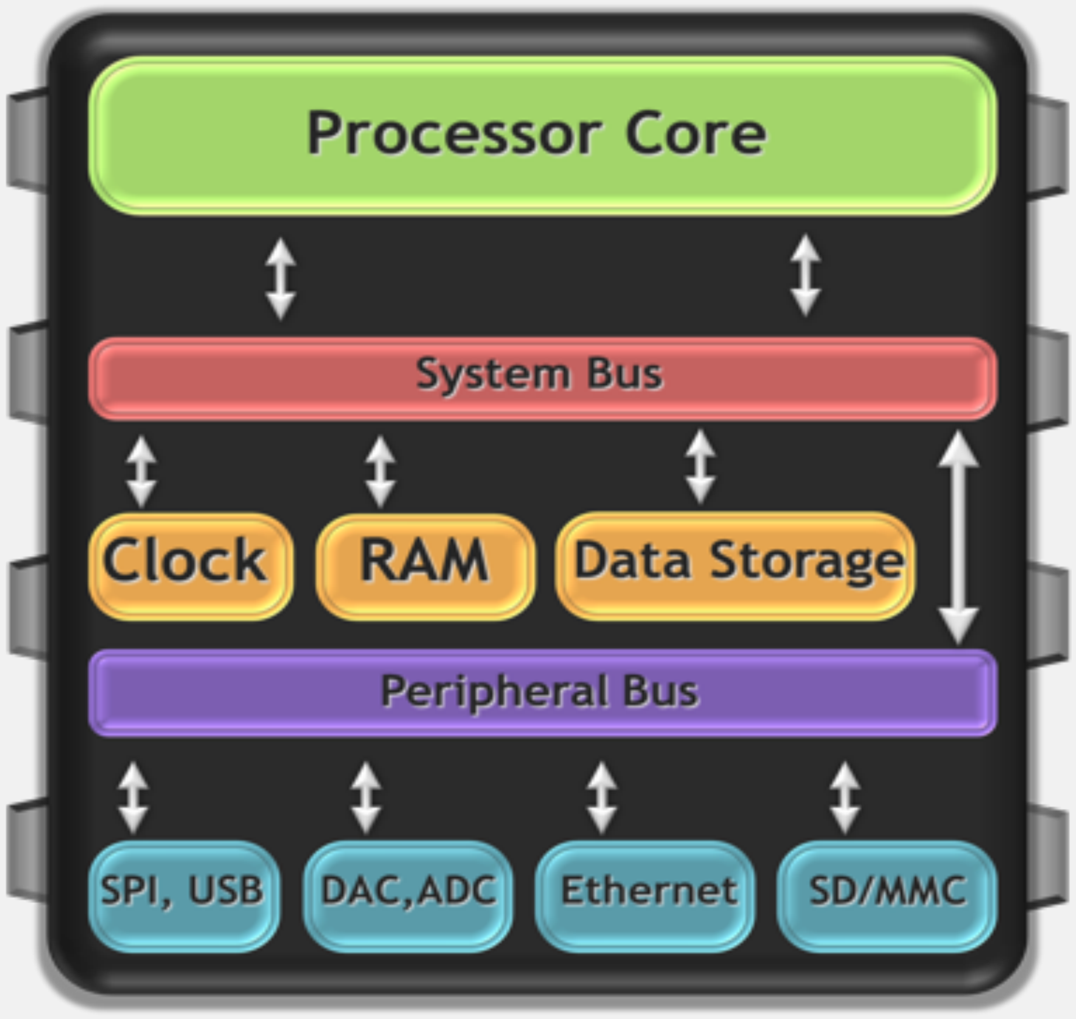


The dark grey square in the middle of the board is an ATMega32U4 microcontroller.

A microcontroller is a small computer with multiple components integrated into a single chip. It includes:

- A microprocessor (CPU) containing: arithmetic and logic unit (ALU); program control system which fetches, decodes and executes instructions; and a bank of registers in which the ALU carries out operations.
- Memory: Flash storage which holds the program; EEPROM (electrically erasable programmable read-only memory); and RAM.

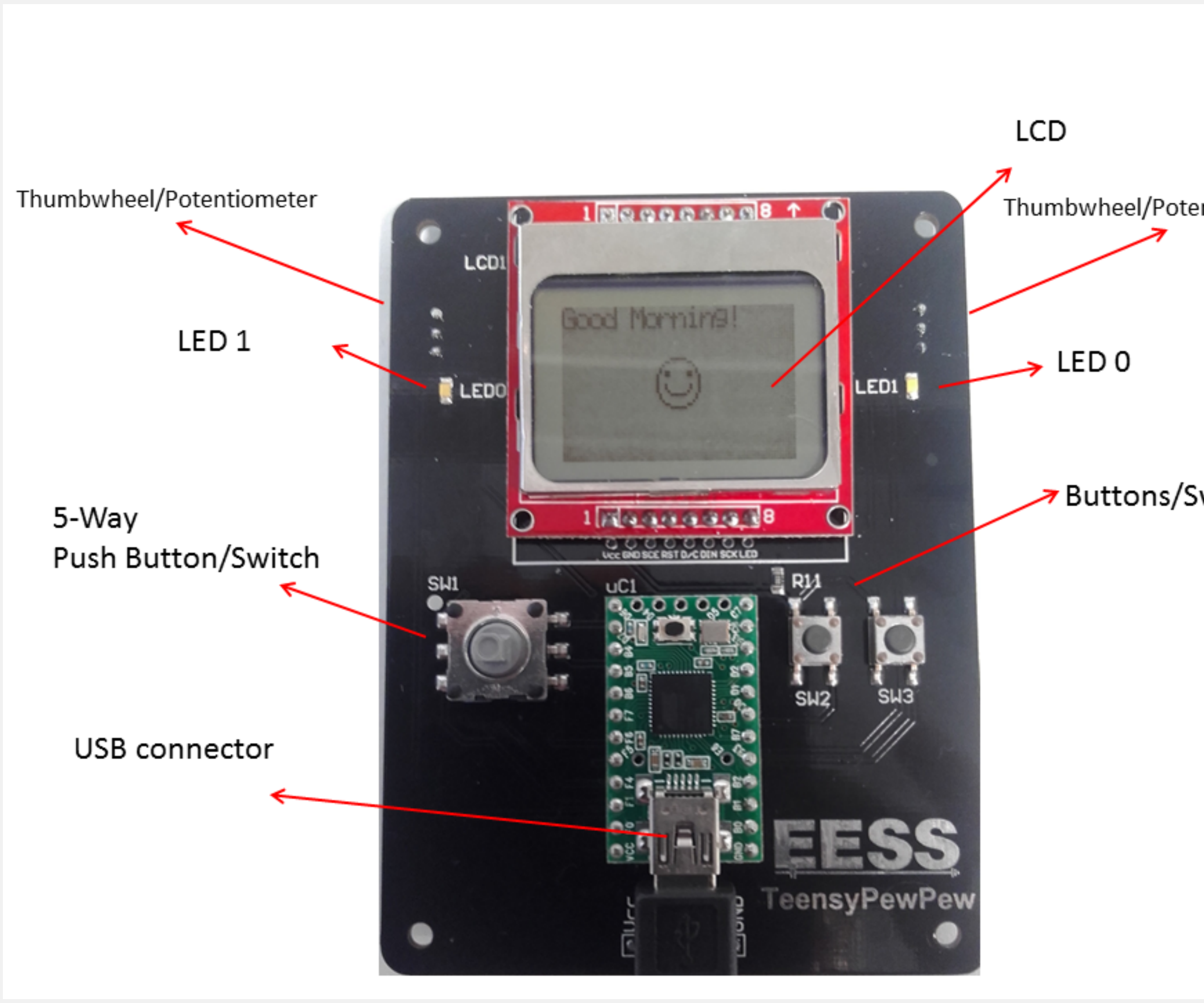
- Clock(s)
- Data connections: digital; analogue; network; memory cards.



QUT Teensy Pew Pew

This section incorporates material created by Dr Luis Mejias.

TeensyPewPew is QUT designed and manufactured. A Teensy 2 board is connected to a range of switches, lights, potentiometers and a 84×48 pixel monochrome Nokia5110 LCD display.



The ATMega32U4 chip has three 8-pin digital IO ports.

- Ports B, D, and F.
- Each port has 8 data pins, which map to a bank of 8-bit storage locations in RAM.
- Each port is bidirectional – input or output.
- Each pin within a port is independently configurable as either input or output.
    - How they are configured depends on the hardware they are connected to.

Ports C and E also exist but only limited number of pins are available.

Rigorous definitions of pin assignments, memory addresses, and programming notes are covered in the ATMega32U4 Data Sheet.

- Microcontroller Resources on Blackboard

First Program on the Teensy

Get things set up

1. Windows users: **do not** uninstall Cygwin. You still need it.
2. Install the `avr-gcc` cross-compiler, using instructions in Tutorial 7 worksheet.
3. Download the `cab202_teensy` archive from Topic 7 Learning Resources in Blackboard.
    - If you have already downloaded an earlier version of this archive, please replace it with the latest version.
4. Extract the folders from the archive and place the `cab202_teensy` folder next to ZDK in your file system.
5. Open a terminal window; `cd` to the `cab202_teensy` folder; run `make rebuild`.
6. If all goes well, you should see a file called `libcab202_teensy.a` generated. This is the Teensy equivalent of the ZDK.

Create (or in some other way get) a `c` source file

Several example programs are provided in the present document; use `teensy_hello.c`:

You will include line 11 in the start-up code of every program you write. This is required to ensure that the CPU clock speed matches the clock speed used when compiling your code with the makefile. If the CPU speed is wrong, your program may not work correctly.

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>

#include <graphics.h>
#include <macros.h>

void setup(void) {
        set_clock_speed(CPU_8MHz);
        lcd_init(LCD_DEFAULT_CONTRAST);
        clear_screen();
        show_screen();
}

void process(void) {
        char *hello = "Hello Teensy!";
        clear_screen();
        draw_string(0, 0, hello, FG_COLOUR);
        draw_string(0, 8, hello, BG_COLOUR);
        draw_string(0, 16, hello, FG_COLOUR);
        draw_string(0, 24, hello, BG_COLOUR);
        draw_string(0, 32, hello, FG_COLOUR);
        draw_string(0, 40, hello, BG_COLOUR);
        show_screen();
}

int main(void) {
        setup();
        for ( ;; ) {
                process();
                _delay_ms(10);
        }
}
```

Create makefile; add target to makefile

A **`makefile`** is a plain text document which contains instructions for the **`make`** program.

- **`make`** is a general purpose tool for managing projects which consist of a collection of inter-dependent documents.
- **`make`** automates the build process.
- **`make`** is the most reliable way to build your Teensy programs.
- **`make`** is good. Learn **`make`**. Use **`make`**.

The following **`makefile`** has been designed for use with the CAB202 Teensy Graphics library.

- It contains pre-defined targets for all the examples in this document.
- Each target is the name of a **`.hex`** file. Teensy executable programs are stored in **`.hex`** files.
- We need to provide a **`C`** source file to match each of these targets, or modify the targets so they match our available source files.
- For example, the target **`teensy_hello.hex`** will be compiled from a source file called **`teensy_hello.c`**. If **`teensy_hello.c`** is not found, compilation will fail.

```
# CAB202 Teensy Makefile
# Lawrence Buckingham, September 2017.
# Queensland University of Technology.

# Replace these targets with your target (hex file) name, including the .hex part.

TARGETS = \
        teensy_hello.hex \
        teensy_lines.hex \
        teensy_pixels.hex \
        turn_on_led.hex

# Set the name of the folder containing libcab202_teensy.a

CAB202_TEENSY_FOLDER = ../cab202_teensy

# ---------------------------------------------------------------------------
#       Leave the rest of the file alone.
# ---------------------------------------------------------------------------

all: $(TARGETS)

TEENSY_LIBS = -lcab202_teensy -lprintf_flt -lm
TEENSY_DIRS =-I$(CAB202_TEENSY_FOLDER) -L$(CAB202_TEENSY_FOLDER)
TEENSY_FLAGS = \
        -std=gnu99 \
        -mmcu=atmega32u4 \
        -DF_CPU=8000000UL \
        -funsigned-char \
        -funsigned-bitfields \
        -ffunction-sections \
        -fpack-struct \
        -fshort-enums \
        -Wall \
        -Werror \
        -Wl,-u,vfprintf \
        -Os

clean:
        for f in $(TARGETS); do \
                if [ -f $$f ]; then rm $$f; fi; \
                if [ -f $$f.elf ]; then rm $$f.elf; fi; \
                if [ -f $$f.obj ]; then rm $$f.obj; fi; \
        done

rebuild: clean all

%.hex : %.c
        avr-gcc $< $(TEENSY_FLAGS) $(TEENSY_DIRS) $(TEENSY_LIBS) -o $@.obj
        avr-objcopy -O ihex $@.obj $@
```

Build and run

After running **`make`** the target(s) should be created.

Use the Teensy USB loader to transfer the program to your Teensy.

When you reboot, the **`.hex`** file is downloaded to Teensy and the program starts.

Library Support

Graphics

The Teensy graphics library in **`cab202_teensy/graphics.h`** is a bit simpler than **`ZDK/cab202_graphics.h`**.

- The LCD display connected to our Teensy is a monochrome
- Where ZDK works with characters, in Teensy we operate at pixel level.
- Two colours are available for drawing: **`FG_COLOUR`** and **`BG_COLOUR`**.
- We use one bit per pixel to encode image data: **`FG_COLOUR == 1`**, and **`BG_COLOUR == 0`**.

Similar to ZDK, we use buffered graphics.

- Drawing operations update an off-screen memory region, which is copied to the display by calling **`show_screen`**.
- The screen buffer is declared in **`cab202_teensy/graphics.h`**, and defined in **`cab202_teensy/graphics.c`**.
- The buffer occupies **`84×48÷8 == 504`** bytes.
- The detailed layout and use of the screen buffer will be covered in Topic 8.

```
/*
 *  CAB202 Teensy Library: 'cab202_teensy'
 *      graphics.h
 *
 *      B.Talbot, September 2015
 *  L.Buckingham, September 2017
 *      Queensland University of Technology
 */
#ifndef GRAPHICS_H_
#define GRAPHICS_H_

#include <stdint.h>

#include "ascii_font.h"
#include "lcd.h"

/*
 *  Size of the screen_buffer, measured in bytes. There are LCD_X
 *      columns, and LCD_Y rows of pixels. Pixels are packed vertically
 *      into bytes, with 8 pixels in each byte.
 */
#define LCD_BUFFER_SIZE (LCD_X * (LCD_Y / 8))

/**
 *      Enumerated type to define colours. We have two colours:
 *      FG_COLOUR - foreground.
 *      BG_COLOUR - background.
 */
typedef enum colour_t {
        BG_COLOUR = 0,
        FG_COLOUR = 1
} colour_t;

/*
 *  Array of bytes used as screen buffer.
 *  (accessible from any file that includes graphics.h)
 */
extern uint8_t screen_buffer[LCD_BUFFER_SIZE];

/*
 *  Copy the contents of the screen buffer to the LCD.
 *      This is the only function that interfaces with the LCD hardware
 *  (sends entire current buffer to LCD screen)
 */
void show_screen(void);

/*
 * Clear the screen buffer (all pixels set to BG_COLOUR).
 */
void clear_screen(void);

/**
 *      Draw (or erase) a designated pixel in the screen buffer.
 *
 *      Parameters:
 *              x - The horizontal position of the pixel. The left edge of the screen
 *                      is at x=0; the right edge is at (LCD_X-1).
 *              y - The vertical position of the pixel. The top edge of the screen is
 *                      at y=0; the bottom edge is at (LCD_Y-1).
 *              colour - The colour, FG_COLOUR or BG_COLOUR.
 */
void draw_pixel(int x, int y, colour_t colour);

/**
 *      Draw a line in the screen buffer.
 *
 *      Parameters:
 *              x1 - The horizontal position of the start point of the line.
 *              y1 - The vertical position of the start point of the line.
 *              x2 - The horizontal position of the end point of the line.
 *              y2 - The vertical position of the end point of the line.
 *              colour - The colour, FG_COLOUR or BG_COLOUR.
 */
void draw_line(int x1, int y1, int x2, int y2, colour_t colour);

/**
 *      Render one of the printable ASCII characters into the screen buffer.
 *
 *      Parameters:
 *              x - The horizontal position of the top-left corner of the glyph.
 *              y - The vertical position of the top-left corner of the glyph.
 *              character - The (ASCII code of the) character to render. Valid values
 *                      range from 0x20 == 32 == 'SPACE' to 0x7f == 127 == 'BACKSPACE'.
 *              colour - The colour, FG_COLOUR or BG_COLOUR. If colour is BG_COLOUR,
 *                      the character is rendered as an inverse video block.
 */
void draw_char(int top_left_x, int top_left_y, char character, colour_t colour);

/**
 *      Render a string of printable ASCII characters into the screen buffer.
 *
 *      Parameters:
 *              x - The horizontal position of the top-left corner of the displayed
 *                      text.
 *              y - The vertical position of the top-left corner of the displayed
 *                      text.
 *              text - A string to render. Valid values for each element range from
 *                      0x20 == 32 to 0x7f == 127.
 *              colour - The colour, FG_COLOUR or BG_COLOUR. If colour is BG_COLOUR,
 *                      the text is rendered as an inverse video block.
 */
void draw_string(int top_left_x, int top_left_y, char *text, colour_t colour);

#endif /* GRAPHICS_H_ */
```

When you examine the **`.h`** file you will notice a new numeric data type: **`uint8_t`**.

- This represents an unsigned 8-bit integer. Also known as a **`byte`**.
- **`uint8_t`** permits values ranging from 0 to 255 inclusive.

We also define an enumerated type: **`colour_t`**.

- This represents the values **`FG_COLOUR`** and **`BG_COLOUR`** – foreground and background colour, respectively.

To see how to render pixels to the screen, examine **`teensy_pixels.c`**:

```
#include <stdint.h>
#include <stdlib.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>
```

```
#include <graphics.h>
#include <macros.h>

#define N 200
int x[N];
int y[N];
int current_pixel = 0;

void init_pixels(void) {
        for ( int i = 0; i < N; i++ ) {
                x[i] = rand() % LCD_X;
                y[i] = rand() % LCD_Y;
        }
}

void draw_pixels(void) {
        for ( int i = 0; i < N; i++ ) {
                draw_pixel(x[i], y[i], FG_COLOUR);
        }
}

void update_pixels() {
        x[current_pixel] = rand() % LCD_X;
        y[current_pixel] = rand() % LCD_Y;
        current_pixel = (current_pixel + 1) % N;
}

void setup(void) {
        set_clock_speed(CPU_8MHz);
        lcd_init(LCD_DEFAULT_CONTRAST);
        clear_screen();
        init_pixels();
        draw_pixels();
        show_screen();
}

void process(void) {
        clear_screen();
        update_pixels();
        draw_pixels();
        show_screen();
}

int main(void) {
        setup();

        for ( ;; ) {
                process();
                _delay_ms(10);
        }
}
```

To see how to render lines to the screen, examine **teensy_lines.c**:

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>

#include <graphics.h>
#include <macros.h>

void setup(void) {
        set_clock_speed(CPU_8MHz);
        lcd_init(LCD_DEFAULT_CONTRAST);
        clear_screen();
        show_screen();
}

void process(void) {
        static double fraction = 0.0;
        clear_screen();

        for ( int y = 0; y < LCD_Y; y++ ) {
                draw_line(0, y, LCD_X - 1, y, FG_COLOUR);
        }

        int x1 = LCD_X * fraction;
        int x2 = LCD_X - x1;
        int y1 = LCD_Y * fraction;
        int y2 = LCD_Y - y1;
        fraction += 0.01;

        if ( fraction >= 0.5 ) fraction = 0.0;

        draw_line(x1, y1, x2, y1, BG_COLOUR);
        draw_line(x1, y2, x2, y2, BG_COLOUR);
        draw_line(x1, y1, x1, y2, BG_COLOUR);
        draw_line(x2, y1, x2, y2, BG_COLOUR);

        show_screen();
}

int main(void) {
        setup();

        for ( ;; ) {
                process();
                // _delay_ms(10);
        }
}
```

Digital Input and Output

Setting, Clearing, and Reading Bits

We saw earlier that the 8-pin physical IO ports are connected to memory locations, and that each pin can be independently configured as input or output.

Each physical IO port maps to a set of three bytes in RAM. These locations are called I/O Registers.

Consider port B:

- We use a variable called **`DDRB`** – the *data direction register* – to specify whether each physical pin is to be used as input or output.
- We use a variable called **`PORTB`** – the *output register* for port B – to send data to the physical pins that are configured for output.
- We use a variable called **`PINB`** – the *input register* for port B – to get signals from physical pins that are configured for input.
- NB: **`PORTB`** *is not a port*, it's a register. **`PINB`** *is not a pin*, it's a register.

Similarly for ports D and F, we have (**`DDRD`**, **`PORTD`**, **`PIND`**) and (**`DDRF`**, **`PORTF`**, **`PINF`**).

See page 66 of the ATMega32U4 data sheet for a formal detailed coverage of this topic.

To configure physical pin *i* of port B as an output, we *set* bit *i* of **`DDRB`** (that is, we make that bit have a value of 1). Here *i* is between 0 and 7 inclusive.

- **`cab202_teensy/macros.h`** contains helpful macros for this.
- For example, use **`SET_BIT(DDRB, i)`** to set bit *i* of the **`DDRB`** register.
- **`SET_BIT`** works with any register.

To configure physical pin *i* of port B as an input, we *clear* bit *i* of **`DDRB`** (making it have a value of 0).

- Use **`CLEAR_BIT(DDRB, i)`** to clear bit *i* of the **`DDRB`** register.
- **`CLEAR_BIT`** works with any register.

To switch on pin *i* of port B when configured for output, we set bit *i* of **`PORTB`**.

- **`SET_BIT(PORTB, i)`** will do this.

To read the value of pin *i* of port B when configured for input, we must extract bit *i* from **`PINB`**.

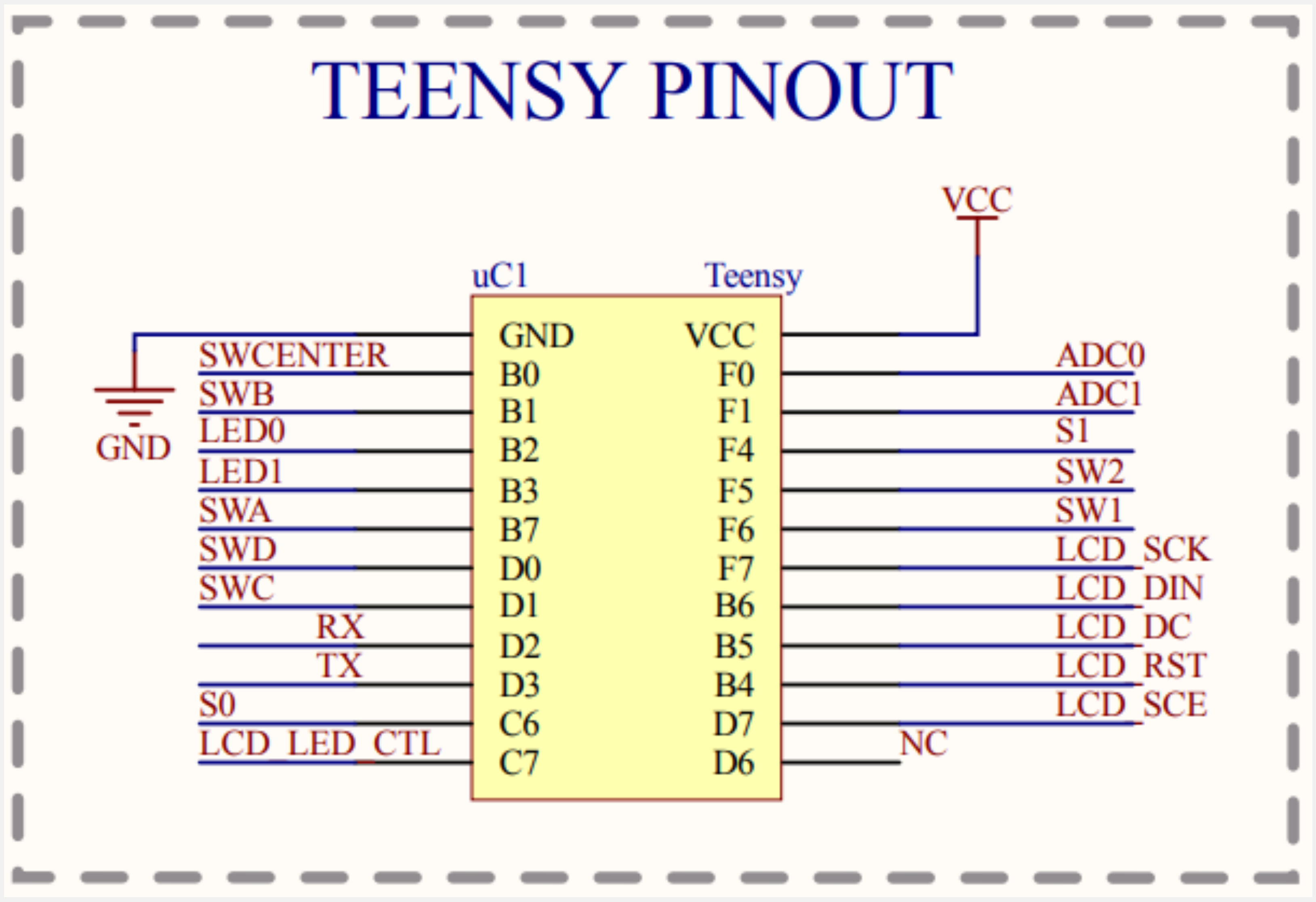
- **`BIT_VALUE(PINB, i)`** gets the status of the physical pin.
- **`BIT_IS_SET(PINB, i)`** asks the question: "Is bit *i* of register **`PINB`** set?" This is usually more useful.

Digital Output: Turn on LED0

In this exercise, we turn on the left-hand LED as soon as the program starts. To do this, we need to:

- Find out which pin the LED is connected to.
- Configure that pin as an output so we can turn the LED on.
- Turn the LED on.

Looking on the Teensy, we see that the left LED is LED0. Consulting the TeensyPewPew schematic diagram:



we find that LED0 is connected to B2: pin 2 of port B.

Therefore, we will need to set bit 2 of DDRB to configure it for output, and we will also need to set bit 2 of PORTB to turn it on.

This can be done with the following program, **`turn_on_led.c`**:

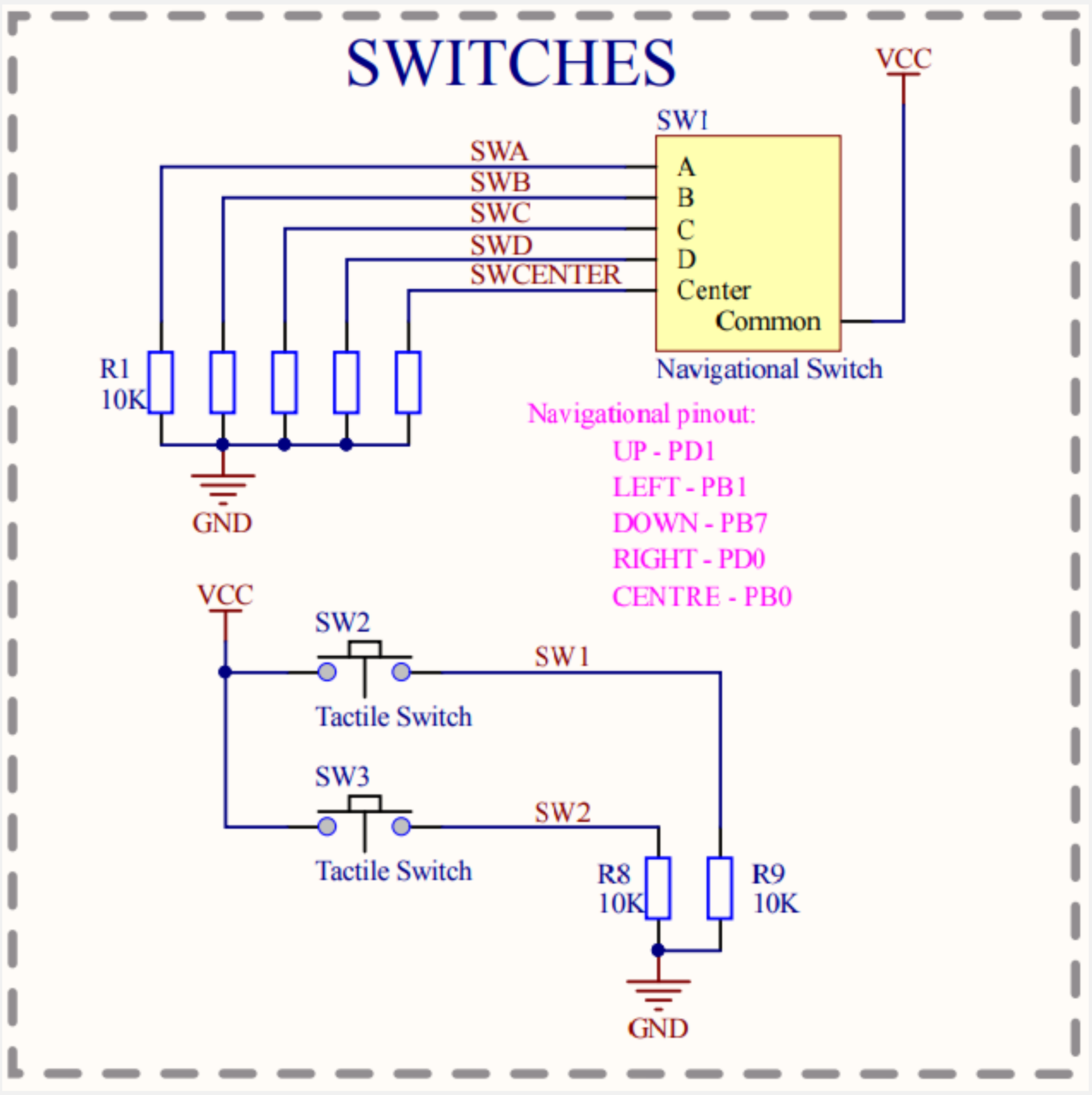
```
// Include the AVR IO library
#include <avr/io.h>

// Include the CPU Speed information
#include "cpu_speed.h"
#include "macros.h"

void setup() {
        set_clock_speed(CPU_8MHz);

        // Enable left LED for output
        SET_BIT(DDRB, 2);

        // Turn the LED on
        SET_BIT(PORTB, 2);
}

void process() {
        // Do nothing this time.
}

int main(void) {
        setup();

        for ( ;; ) {
                process();
        }
}
```

Digital Input: Read switches

In this exercise, we write a program that turns on the left-hand LED when the left button is clicked, and turns it off when the right button is clicked.

To do this, we must determine what pin each of the buttons is connected to. Which is a bit messy, because the schematic contains apparently contradictory labels.

Looking at Teensy, we see that the left button is labelled **SW2** and the right button is labelled **SW3**.



We find that **SW2** morphs into **SW1**, which is connected to **F6**. **SW3** turns into **SW2**, which is connected to **F5**.

During the setup phase, we will have to clear bits 6 and 5 of **DDRF**.

In the process phase, we must ask if each button is pressed, and if so, turn the light on or off. This can be done with **BIT_IS_SET**.

Clearing bit 2 of the **PORTB** will turn the light off.

The final program is **turn_on_off_led.c**:

```
// Include the AVR IO library
#include <avr/io.h>

// Include the CPU Speed information
#include "cpu_speed.h"
#include "macros.h"

void setup() {
        set_clock_speed(CPU_8MHz);

        // Output to left LED
        SET_BIT(DDRB, 2);

        // Input from the left and right buttons
        CLEAR_BIT(DDRF, 5);
        CLEAR_BIT(DDRF, 6);
}

void process() {
        // If left button is pressed, Turn on left LED.
        if ( BIT_IS_SET(PINF, 6) ) {
                SET_BIT(PORTB, 2);
        }

        // Otherwise, if right button is pressed, turn off left LED.
        else if ( BIT_IS_SET(PINF, 5) ) {
                CLEAR_BIT(PORTB, 2);
        }
}
```

```
int main(void) {
        setup();

        for ( ;; ) {
                process();
        }
}
```

The End