

Gambling shield

Input and output

The IO-pins on our development board are bidirectional and individually configurable. The driver package includes functions for setting pin direction and for reading or writing the pin state. Our processor has three IO ports that are called PIO0, PIO1 and PIO2. Each port has 32 pins.

Study Board_LED-functions board.c in lpc_board_nxp_lpcxpresso_1549-project for an example of configuring and using pins. In the Arduino receptacles schematic diagram (later in this document) you see wires marked with PIOx_y, where x is the IO port number and y is the pin number.

The IO pin direction is configured with Chip_GPIO_SetPinDIROutput and Chip_GPIO_SetPinDIRInput. Both functions take three parameters: pointer to IO configuration registers, port number and pin number.

For example:

```
Chip_GPIO_SetPinDIROutput(LPC_GPIO, 1, 8);  
Chip_GPIO_SetPinDIRInput(LPC_GPIO, 0, 24);
```

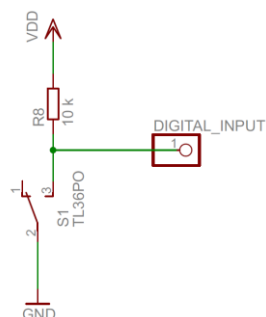
The above configures [port 1, pin 8] as an output and [port 0, pin 24] as an input.

When a pin is configured as an input some additional configuration may be required depending on what is connected to the pin.

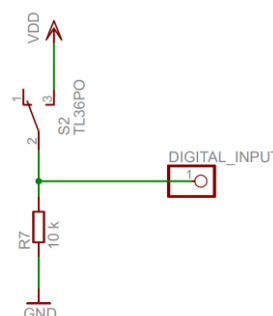
When simple (single pole) push button is connected to an input, a pull up or pull down resistor is required to ensure that a pin always has a stable value. Pull up/pull down resistor will force the pin into a default state if the pin is left floating (unconnected).

A pin with a button and a pull up resistor has default value of one. When the button is pressed the pin goes low and reads zero.

A pin with a button and a pull down resistor has default value of zero. When the button is pressed the pin goes high and reads one.



Picture 1 Pull up resistor



Picture 2 Pull down resistor

Our gambling shield has grounding buttons and switches so we need to use pull up resistors in the input pins. Most modern microcontrollers have integrated pull up/pull down resistors that can be enabled (=connected) with software. Our LPC1549 makes no exception to this, so we need to configure pull ups to read gambling shield pins.

Some pins can be used as analog inputs or connected to internal peripherals (for example UART).

The following example shows how configure one pin as a digital input with pull up resistor and inverted input. The default value when a button is not pressed is one (high) with a pull up resistor. LPC1549 allows inputs to be inverted before the value is passed CPU. When inversion is enabled a pressed button reads one and not pressed reads zero.

```
// Set Port0.Pin8 as a digital input with pull up
Chip_IOCON_PinMuxSet(LPC_IOCON, 0, 8, (IOCON_MODE_PULLUP | IOCON_DIGMODE_EN | IOCON_INV_EN));
Chip_GPIO_SetPinDIRInput(LPC_GPIO, 0, 8);
```

Exercise 1 - Simple blink sequence

Follow instructions in 00_How_to_create_and_debug_a_project.pdf to import periph_blinky project to your workspace. The board comes with an RGB-led where each color is controlled by an IO-pin. **Find which pins are driving each of the colors.** You can find the information from the example project and the chip and board libraries.

Create a new C++ project. Configure the systick interrupt to run at 1000 Hz. See blinky example for detail on how to configure systick frequency. The systick configuration takes place on the lines 75-89 in systick.c of periph_blinky example.

Add code from the following following page to get a simple delay function to your project.

Write a program that reads the state of button SW1. When SW1 is pressed the program switches each of the leds on for one second in a sequence, first red then green and then blue. After the sequence is finished all leds are switched off and the program starts reading SW1 again.

To read SW1 you need to configure the button as an input with pullup. SW1 is connected to Port 0, Pin 17.

A simple delay function

Systick timer and a function that generates a fairly accurate delay. Systick timer is assumed to run at 1000 Hz (once per millisecond). See `periph_blinky` project for example on configuring the systick timer.


You can add this to your project if you need to generate delays.

```
#include <atomic>

static volatile std::atomic_int counter;

#ifdef __cplusplus
extern "C" {
#endif
/**
 * @brief    Handle interrupt from SysTick timer
 * @return    Nothing
 */
void SysTick_Handler(void)
{
    if(counter > 0) counter--;
}
#ifdef __cplusplus
}
#endif

void Sleep(int ms)
{
    counter = ms;
    while(counter > 0) {
        __WFI();
    }
}
```



This puts the CPU in sleep mode until an interrupt (= next counter decrement) occurs.

`SysTick_Handler` is called by the timer hardware at the frequency specified by the systick timer configuration. Note that you must not call `SysTick_Handler` from your program and that the name of the function must be spelled exactly as in the example above. The `#ifdef __cplusplus` is an essential part of the code and may not be omitted.

The delay generation is simple – just call `Sleep` with the required delay time as parameter. For example to generate 200 ms delay do `Sleep(200)`.

Note: Hardware takes care of calling `SysTick_Handler`. Do not call `SysTick_Handler` in your code. Do not modify the handler code or `Sleep()`.

Exercise 2 – IO pin class

Implement a class that abstracts DigitalIoPins and a program that uses the class.

The class must have the following interface:

```
class DigitalIoPin {
public:
    DigitalIoPin(int port, int pin, bool input = true, bool pullup = true, bool invert = false);
    DigitalIoPin(const DigitalIoPin &) = delete;
    virtual ~DigitalIoPin();
    bool read();
    void write(bool value);
private:
    // add these as needed
};
```

Callouts for the class interface:

- Constructor parameters:**
 - `bool input`: true → input, false → output
 - `bool pullup`: true → enable pullup, false → no pullup. Has no effect with output pin.
 - `bool invert`: true → invert, false → normal. Applies only to input.
- `DigitalIoPin(const DigitalIoPin &) = delete;`: Makes the object non-copyable
- `bool read();`: Reads the state of the pin
- `void write(bool value);`: Sets the state of pin. Has no effect when pin is configured as an input.

Note that when a pin is configured as an input you can enable (or disable) a hardware inverter for the pin.

Write a program that creates three DigitalIoPin objects, one for each on-board button. The buttons are connected to the following pins:

- SW1 → PIO 0.17
- SW2 → PIO 1.11
- SW3 → PIO 1.9

All three switches require that pullups are enabled. With pullups and the way the buttons are connected, the button reads zero when it is pressed and one when not pressed. This can be changed by enabling the inverter on the IO-pins.

Create the objects locally (do not use new!). For example:

```
DigitalIoPin sw1(0,17,true, true, false);
```

The program monitors the state of the pins and switches leds when a button is pressed. Check the button states at 10 ms intervals. When SW1 is pressed program the red led is on while the button is pressed. SW2 switches on green led and SW3 switches on blue led. The leds stay on as long as the controlling button is pressed. When the button is released the led switches off after one second. It must be possible to release buttons at different times.

Hint: First rewrite Exercise 1 to use your IO pin class to verify that you are able to read and write pins.

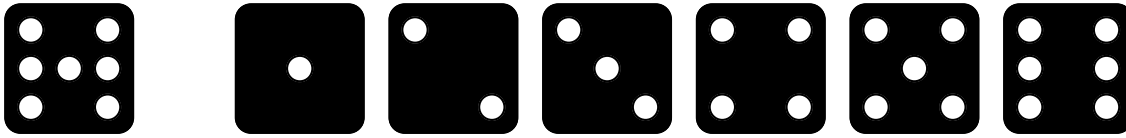
Exercise 3 – Electronic dice

Design electronic dice according to the specification below.

Electronic dice consists of seven leds and two buttons.

One button is a test button which turns on all the leds and keeps the leds on until the button is released. When the test button is released all leds are switched off.

The second button is an operate button which when pressed and released randomly displays a number between 1 and 6. Number is displayed until user operates dice again or if test button is pressed. All leds must be off while the operate button is pressed.



Outputs of test button and numbers 1 – 6 are shown in the picture above.

Gambling shield is described in more detail after the assignment at the end of the document.

Implementation requirements

Use the class from Exercise 2 for input and output.

To get a random number make a variable run in a loop between 1 - 6 at high frequency, while the button is pressed. High counting frequency prevents user from picking numbers by releasing the button at a specific time. When the button is released display the number that was left in the counter when button was released. See flow chart on the next page for more details.

Write a class for displaying a number with dice shaped led pattern.

- The class must have a member function for displaying a number. When the display member function is called with 0 all leds are switched off. When it is called with a number that is greater than six then all leds are switched on. When the number is between 1 and 6 the number is displayed as specified in the picture above
- The constructor must configure the direction and mode of the output pins and switch all leds off
- **The display class may not configure or provide functions for handling the buttons. The responsibility of this class must be limited to displaying numbers on the shield.**

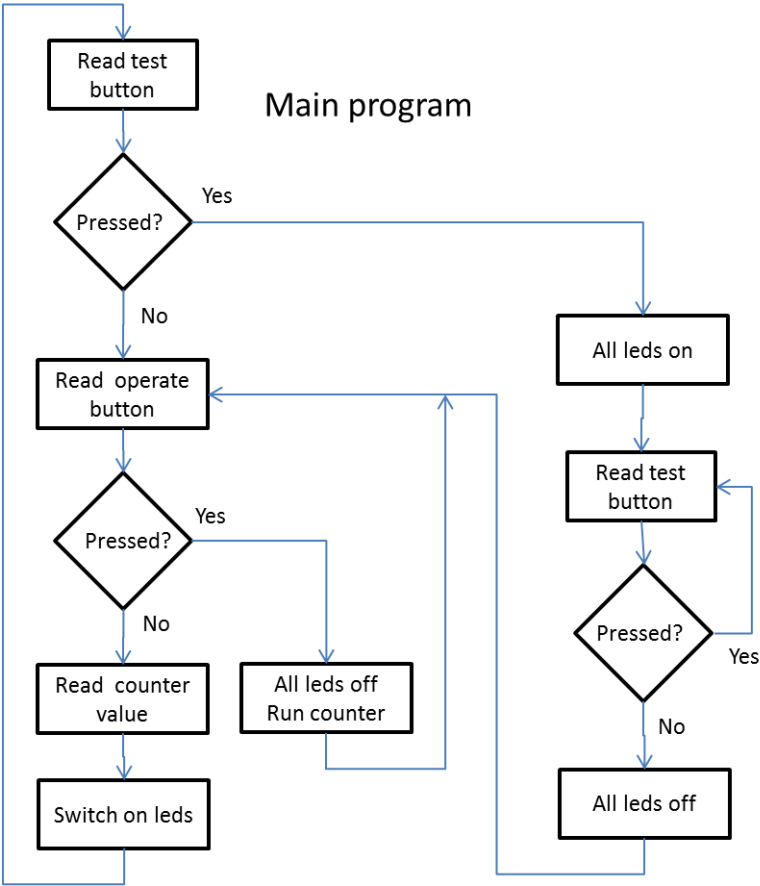
```
class Dice {  
public:  
    Dice();  
    virtual ~Dice();  
    void Show(int number);  
private:  
    /* you need to figure out yourself the private members that you need */  
}
```

This is the public interface. Do not change it!

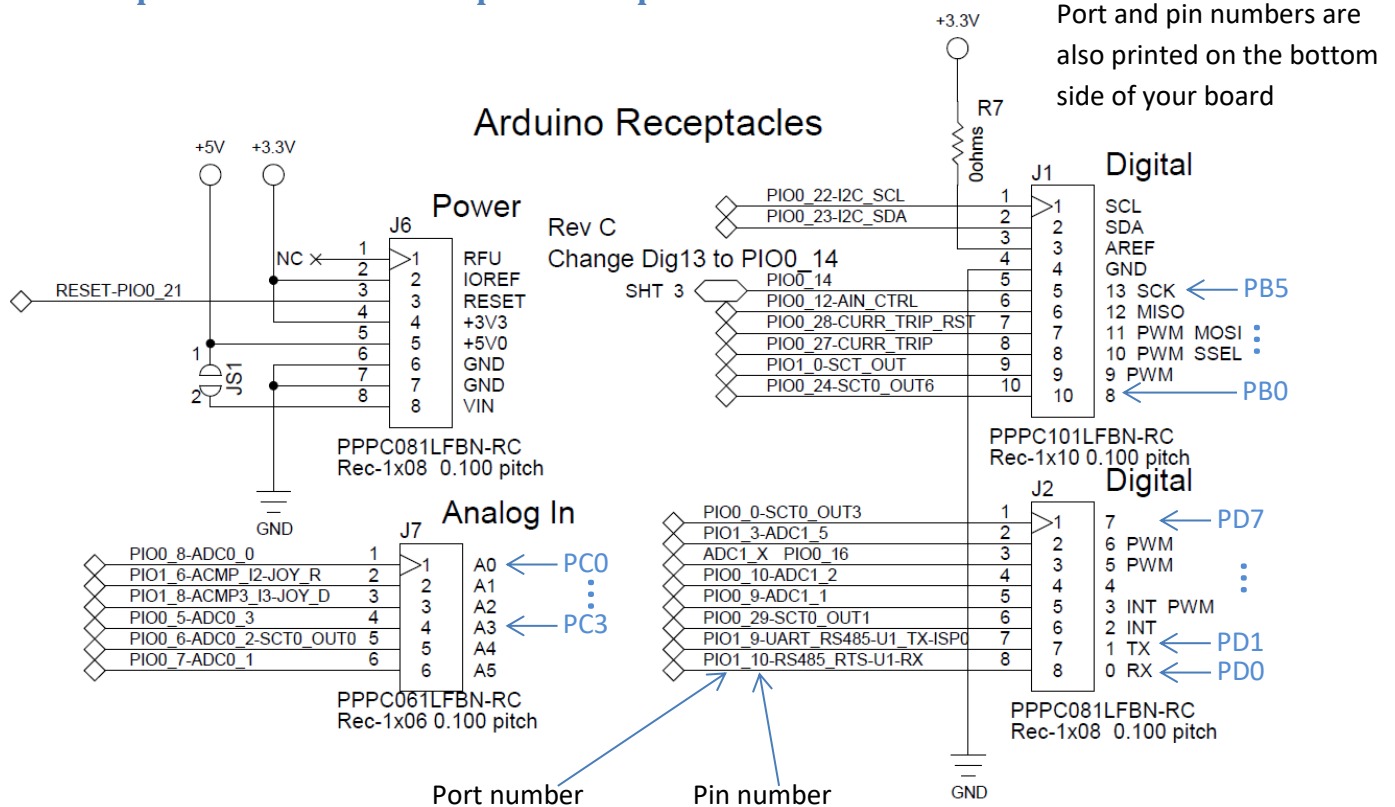
The classes help in raising the abstraction level of a program by hiding the implementation details behind the class interface. Remember that class interface should publish only the member functions that are needed to use the services that it provides and the rest of the members must be private.

MCUXpresso IDE has a wizard for creating a class. The wizard creates header and code templates and adds them to your project.

Flow chart



How map leds and buttons to ports and pins



The schematic above indicates correspondence of gambling shield markings and Arduino receptacles. The blue texts on the right-hand side of connector symbols are the markings found on the gambling shield. To the left of each connector symbols you can find which processor pin each Arduino pin is connected to. The first digit after PIO indicates port number (0 – 2) and the number after the underscore indicates pin number (0 – 31). This pair is needed to configure the pin mode and direction in your program.

Remember to enable pull ups and digital mode on the input pins when gambling shield is used.

Gambling shield

Gambling shield is an Arduino compatible shield that goes in to the Arduino receptacles on the LPCXpresso board. You must match the markings on the shield with the processor pins in the schematic below. Each led, button and switch has an identifier next to it and same identifiers are used to indicate which pin in Arduino pin headers they are connected to.

Dice shaped leds are connected to PD1 – PD7. Card suit leds are connected to port BB2 –PB5. Switches and buttons are connected to PB and PC. See shield printing for connection details.

PD1 – 7 and PB2 – 5 must be configured as outputs and all pins with a button or switch must be configured as inputs

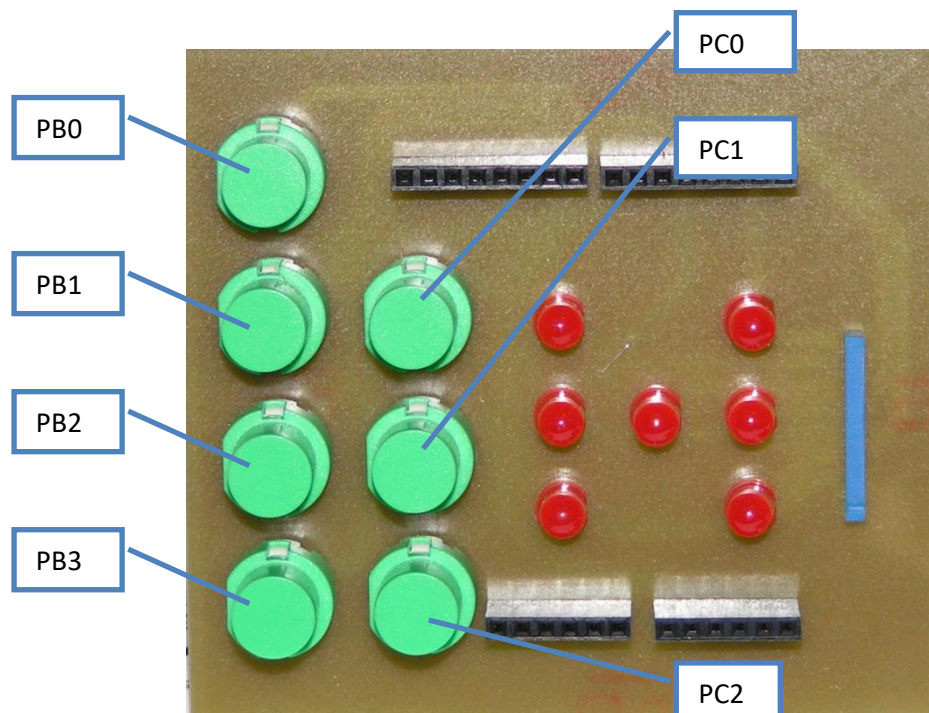


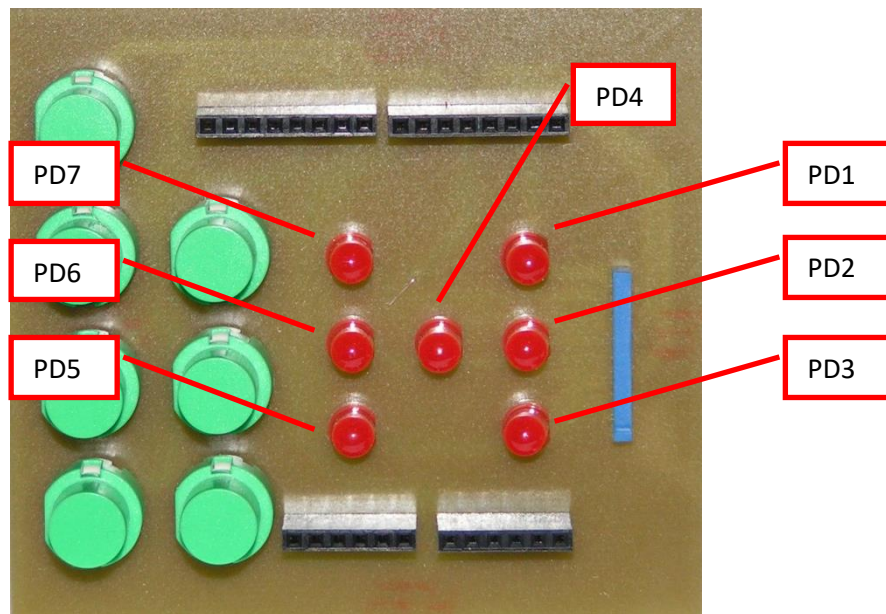
Gambling shield configuration:

- Input: D.0, B.0, B.1, C.0, C.1, C.2, C.3
- Outputs: D.1 – D.7, B.2 – B.5

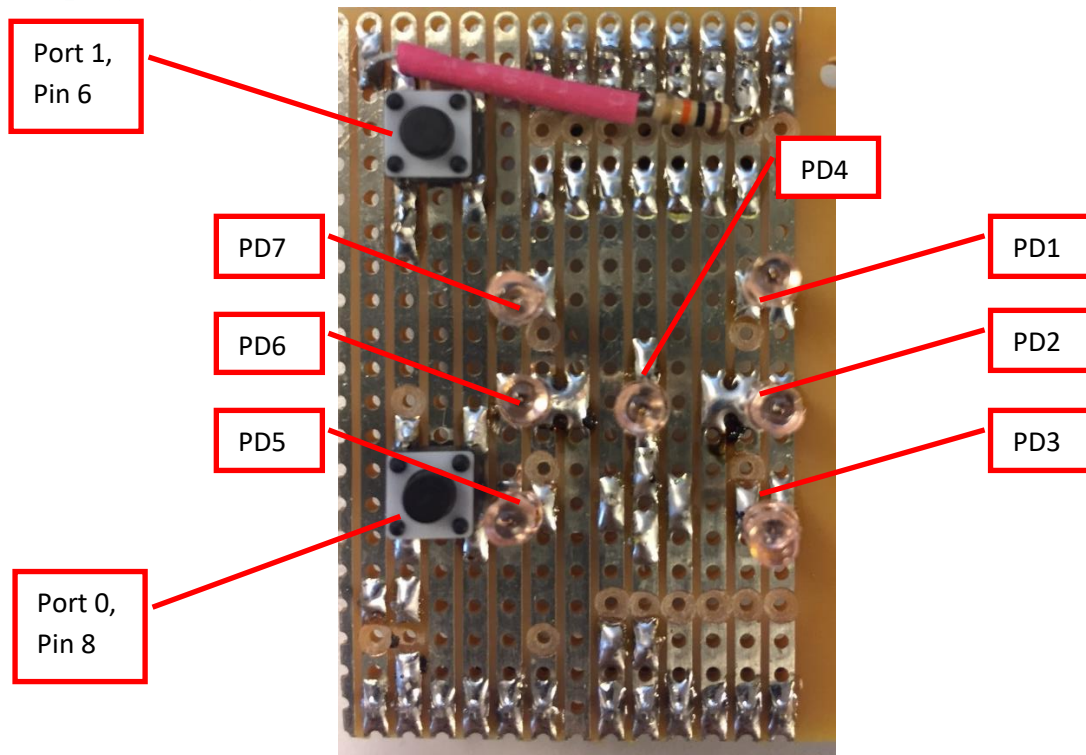
Old gambling shield

The old version of the gambling shield has more buttons and a slightly different button layout. Note that using buttons PC0, PC1 or PC2 allows you to test your dice with either of the gambling shield types.





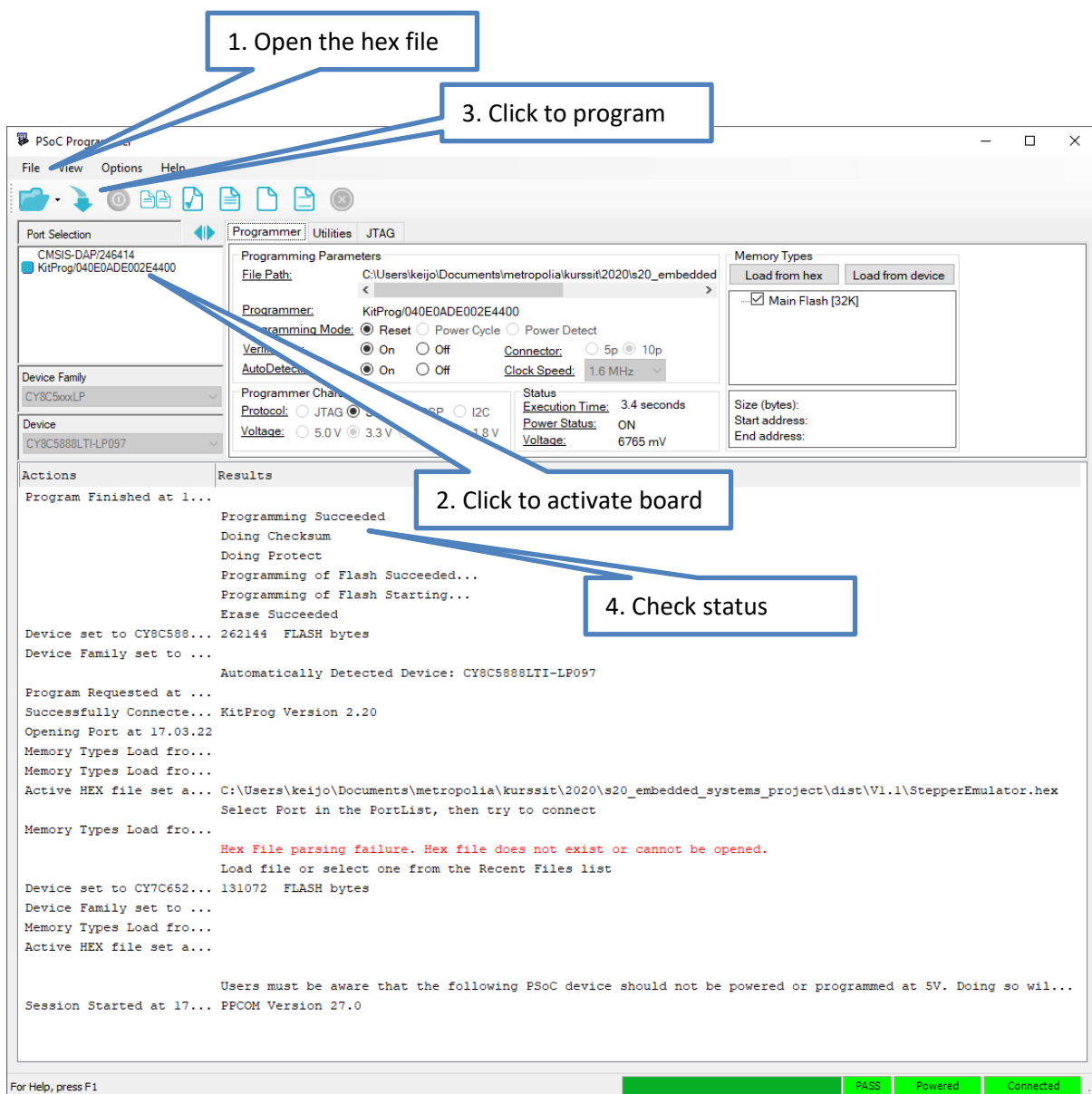
Prototype gambling shield

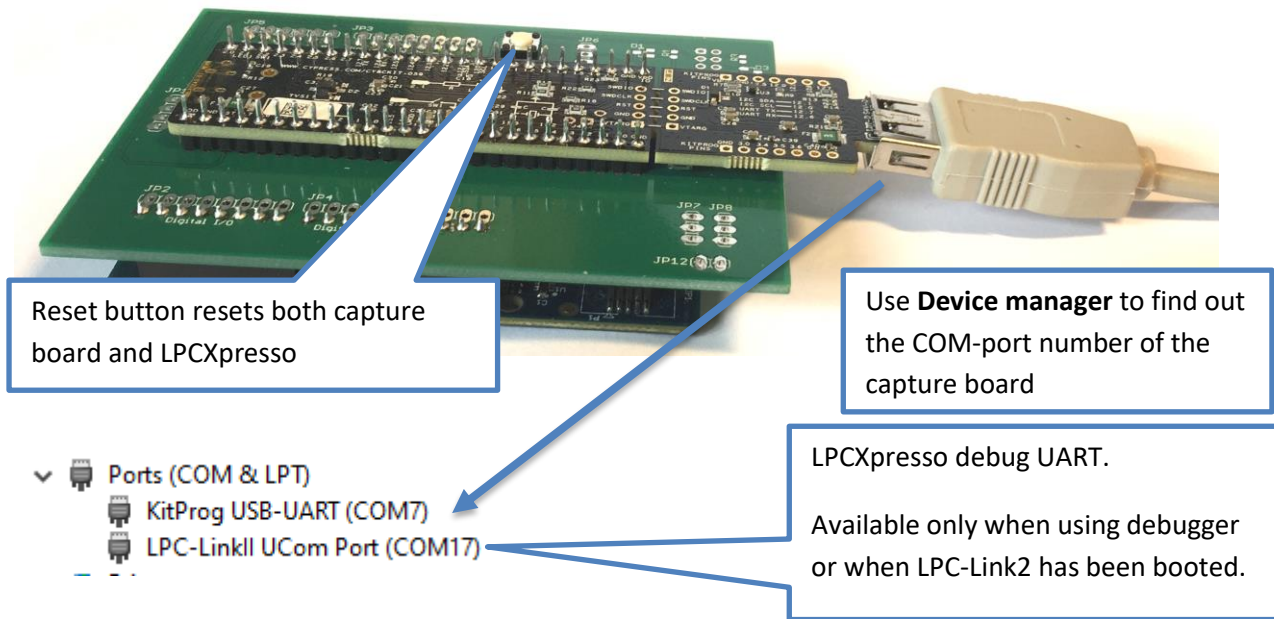


Signal capture board

On the following pages you can find how to program the capture board. Unfortunately, PSoC programmer runs only on windows. When the board is programmed, we use Kitprog USB-UART which has been tested to work also on OSX and Linux. Once the board has been programmed the rest of the exercise is operating system independent: MCUXpresso IDE is available for Windows, Linux and OSX.

The PSoC-board needs to be programmed with the capture software. The hex-file of the capture software and the programmer can be downloaded from the course workspace. When you install the programmer, PSoC programmer, you get drivers for both the programmer (KitProg) and the USB-UART. USB UART drivers are needed to communicate with the PC-based simulator. PSoC programmer is not needed after the capture board has been programmed.

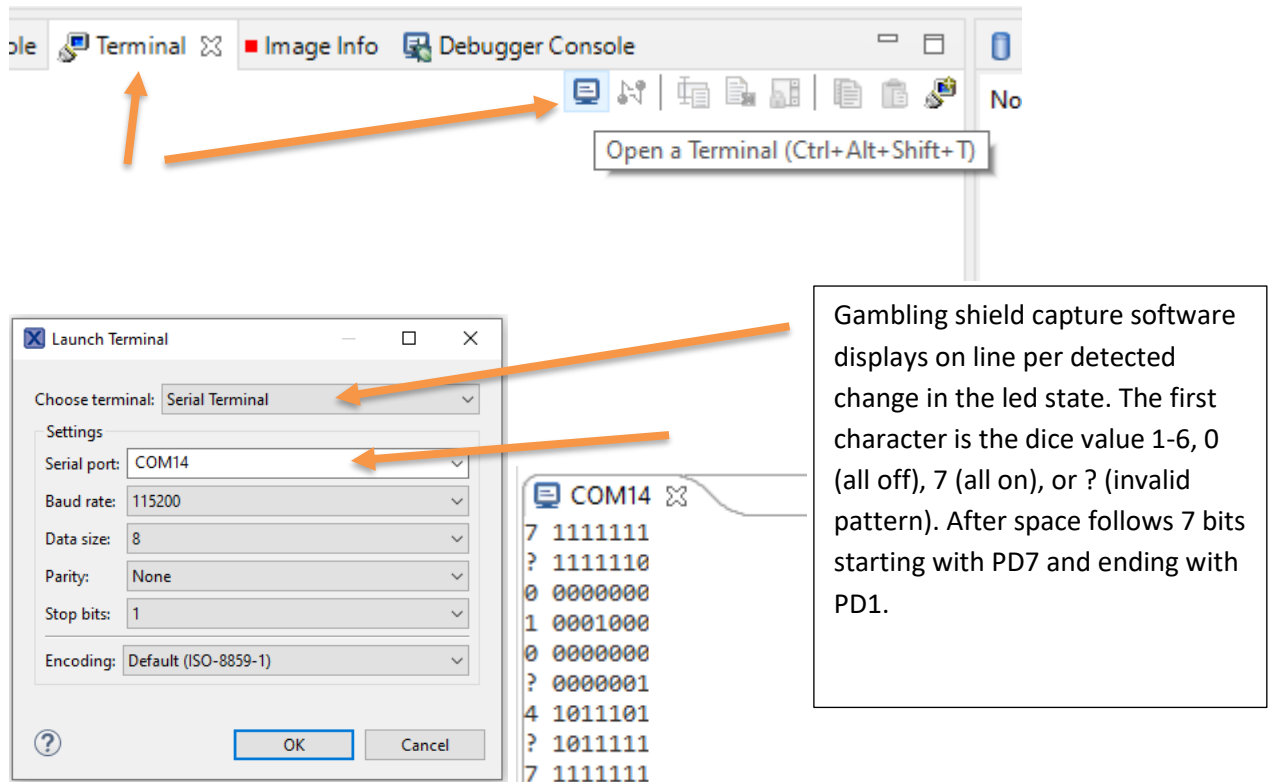




Note that because reset pin of PSoC and LPCXpresso are connected together the PSoC board is reset always when LPCXpresso is reset (by LPCXpresso debugger or by reset button).

If your capture board shows up as a mass storage device press and hold reset for about 5 seconds. When the board is in mass storage programmer mode the green led near USB-connector blinks. Solid green means that the board is in standard mode. USB UART works only in standard mode. Note that mass storage mode can't be used to program the capture board.

You can connect to a com-port directly from MCUXpresso IDE.



In addition to capturing signals the board can simulate button presses and other signals.

Type 's' to press the operate button of the gambling shield and type 'w' to release the button. Signal capture board connects operate button to PC_0 (LPC Port 0, Pin 8).

Typing 'a' will press the test button and typing 'q' will release it. Signal capture board connects test button to PC_1 (LPC Port 1, Pin 6).