

Lab Report

Lab Assignment 3, 2023

Subroutines with RPi Pico



Authors:

Robin G. Hansen
Michael Daun.

Course:

Computer Technology 1
1DT301

Task 1

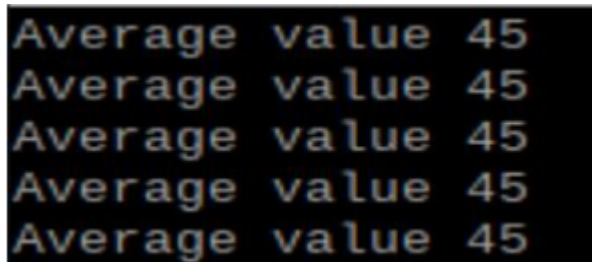
Write a program to calculate the average value of 8 numbers defined in the .data section. Then, show the result in the terminal (i.e. Minicom).

Solution

We used the template code provided in the task description. Our function is hardcoded to exclusively work with an array with 8 elements in total. This is because of the need to divide the sum of the integers in the array before we return it. This is solved with the shift-right command and is in our code set to #3, which means that we divide the sum by 2^3 (8).

The main idea of our subroutine is to simply load in every integer and add it to a sum that gets divided by 8 before we return the average of the array in the R0 register.

Our result looks like this:



```
Average value 45
Average value 45
Average value 45
Average value 45
Average value 45
```

Code with comments

```
.thumb_func @neccessary because sdk uses BLX
.global main @ Provide program starting address to linker.

main:
BL  stdio_init_all @ initialize uart or usb
loop:
LDR r0, =my_array
```

```
MOV r1, #8 @ 8 elements in the array
BL average @ call the subroutine average, with parameter r0 and r1

@Print string and average value
MOV r1, r0 @ Move average value to the printf parameter r1
LDR R0, =message_str @ load address of helloworld string
BL printf @ call pico_printf
B loop @ loop forever

@Returns average of numbers stored in array
@Subroutine average takes the parameters:
@r0 - Memory address to first element of integer array
@r1 - Number of integers in the array
@r0 - Return value (integer average value)
average:
LDR r2, [r0] @Load the first element in array
MOV r5, #0 @Set to 0 to compare.
MOV r3, #0 @This register must be set to zero manually, learnt the
hard way
loop2:
ADD r3, r3, r1 @Add the loaded element to sum
LDR r2, [r0, #4] @Load the next element in array
ADD r0, r0, #4 @Move to next memory space
SUB r1, r1, #1 @Decrement counter
CMP r1, r5
BNE loop2 @If counter not 0, loop again
LSR r3, r3, #3 @Divide sum with 8
MOV r0, r3 @Move sum of average to r0 according to assembler
convention
bx lr

.data
```

```
.align 4 @ necessary alignment
message_str: .asciz "Average value %d\n"
.align 4 @ necessary alignment
my_array: .word 10, 20, 30, 40, 50, 60, 70, 80
```

Task 2

Connect an LED to GP0 and two pushbuttons: One connected to GP1 and one to GP2. [...] Then, write a program with the following functionality:

If the pushbutton on GP1 is down, turn on the LED. If the pushbutton on GP2 is pushed down, turn off the LED.

Solution

We begin by assigning the pins we are going to use. Then, we initialize each one and set their appropriate directions using the C-function "set_dir." We configure our on and off button-pins to be used as inputs and our LED pin to be used as an output.

```
.EQU LED_BUTTON_ON, 1
.EQU LED_BUTTON_OFF, 2
.EQU LED, 0

.thumb_func
.global main    @ Provide program starting address
main:

    @ Initialize and sets direction for on button.
    MOV R0, #LED_BUTTON_ON
    BL gpio_init
    MOV R0, #LED_BUTTON_ON
    MOV R1, #0
    BL link_gpio_set_dir

    @ Initialize and sets direction for off button.
    MOV R0, #LED_BUTTON_OFF
    BL gpio_init
    MOV R0, #LED_BUTTON_OFF
    MOV R1, #0
    BL link_gpio_set_dir

    @ Initialize and sets direction for the LED.
    MOV R0, #LED
    BL gpio_init
    MOV R0, #LED
    MOV R1, #1
    BL link_gpio_set_dir
```

The algorithm is a simple one: one on-loop, one off-loop. We use the C-function "get," which takes the pin as its parameter and returns whether the pin is high or low. We then use this to make a comparison to determine if our button has been pressed (1) or not (0). If it's pressed, we move to the new loop and repeat.

```
loop_off:
    MOV R0, #LED
    MOV R1, #0
    BL link_gpio_put

    MOV R0, #LED_BUTTON_ON
    BL link_gpio_get
    CMP R0, #1

    BEQ loop_on
    BNE loop_off

loop_on:
    MOV R0, #LED
    MOV R1, #1
    BL link_gpio_put

    MOV R0, #LED_BUTTON_OFF
    BL link_gpio_get
    CMP R0, #1

    BEQ loop_off
    BNE loop_on
```

For our C-functions we used a wrapper class with the following functions:

```
#include "hardware/gpio.h"

void link_gpio_set_dir(int pin, int dir)
{
    gpio_set_dir(pin, dir);
}

void link_gpio_put(int pin, int value)
{
    gpio_put(pin, value);
}

bool link_gpio_get(int pin)
{
    return gpio_get(pin);
}
```

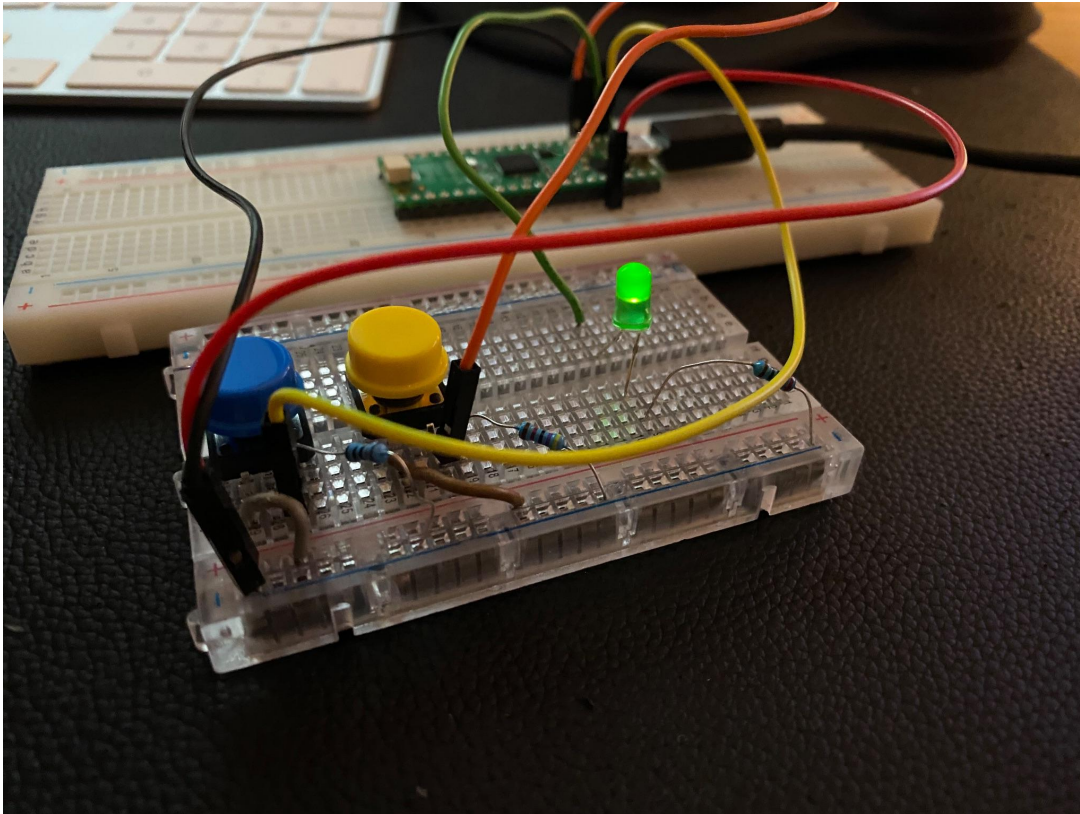
"set_dir" is used to configure a pin as an input or an output.

"gpio_put" is used to assign a specific value to the target pin.

"gpio_get" is used to read the value of the target pin.

Hardware setup

In the image below we have the circuit we used for both tasks 2 & 3. We used breadboards to connect the pushbuttons and the led with our Raspberry Pico. The power rail and ground rail are connected to their respective pins on the Pico. The led is connected in series with a protective 330 Ω resistor to ensure that there is a limited current flow. The pushbuttons are connected with pull down resistors valued at 47k Ω each. We used the pull down resistor to ensure that the idle voltage for the Picos input remains at 0 V when the buttons are not actively in use.



Task 3

Connect a LED to GP0 and two pushbuttons: One connected to GP1 and one to GP2. This is the same setup as Task2.

This time, you are not allowed to use C functions to read from the input pin or to turn the LED on/off! You have to use reads and writes to and from hardware registers.

Solution

Since we already had a working algorithm for the similar task 2, our goal was to replicate the code and replace necessary functions with their assembly equivalents. We used the suggested [example from the course book](#) as a template for our code. Therefore, most of the essential functions required

for our algorithm were already in place. The missing component was the equivalent to ‘**gpio_get**’ used in task 2. As in the previous task we wanted this function to return a value indicating whether the button had been pushed or not. By examining the example, we could see that, in addition to the base memory address for the Pico's GPIO (‘**gpiobase**’), we also needed an offset to access the register that handles the input value of the pin.

By searching the source code provided by Raspberry Pi, we discovered that in the [sio.h file](#), there is a section called ‘**SIO_GPIO_IN**’ that provided us with the proper offset for extracting the input value at a given pin. With this knowledge we created a function called ‘**gpio_read**’.

```
@ Read the state of GPIO pin
gpio_read:
LDR R2, gpiobase @ Load the base address of GPIO registers into R2
LDR R3, [R2, #SIO_GPIO_IN_OFFSET] @ Read the input value of the pin
and store it in R3
LSR R3, R3, R0 @ Right-shift the value in R3 by the PIN specified
in R0
MOV R0, R3 @ Return the result in R0
BX LR
```

This function returns a number that indicates if the pin is receiving voltage or not. When we investigated the output of our function we found that it returns the following values:

- **ON_BUTTON** (pin 1) OFF: 8388608 ➡ In hexadecimal: 0x800000
- **ON_BUTTON** (pin 1) ON: 8388609 ➡ In hexadecimal: 0x800001
- **OFF_BUTTON** (pin 2) OFF: 4194304 ➡ In hexadecimal: 0x400000
- **OFF_BUTTON** (pin 2) ON: 4194305 ➡ In hexadecimal: 0x400001

Since we couldn't figure out how to use bitmask in the .thumb mode, we could not smoothly extract the least significant bit needed to determine whether the button was pushed or not. To solve this we used the rather unorthodox solution to store the “0” equivalent value of the output in a ‘**.EQU**’ assembler directive called ‘**SUB_SUM_ON**’ and ‘**SUB_SUM_OFF**’. These values would, when subtracted from the return value from the ‘**gpio_read**’ return either “1” or “0” giving us an easier number to work with in the main loop. The result can be seen in the code section below.

Code, main sections and on/off loops.

```
main:
BL stdio_init_all @ initialize uart or usb
@ Init each of the three pins and set them to output
MOV R0, #LED
BL gpioinit
MOV R0, #ON_BUTTON
BL gpioinit
MOV R0, #OFF_BUTTON
BL gpioinit

loop_off:
MOV R0, #LED
BL gpio_off
MOV R0, #ON_BUTTON
BL gpio_read
LDR R2, =SUB_SUM_ON
SUB R0, R0, R2
CMP R0, #1
BEQ loop_on
B loop_off

loop_on:
MOV R0, #LED
BL gpio_on
MOV R0, #OFF_BUTTON
BL gpio_read
LDR R2, =SUB_SUM_OFF
SUB R0, R0, R2
CMP R0, #1
BEQ loop_off
B loop_on
```