Stephen Smith's Blog

Musings on Machine Learning…

Bit-Banging the Raspberry Pi Pico's GPIO Registers

with 5 comments

i
12 Votes

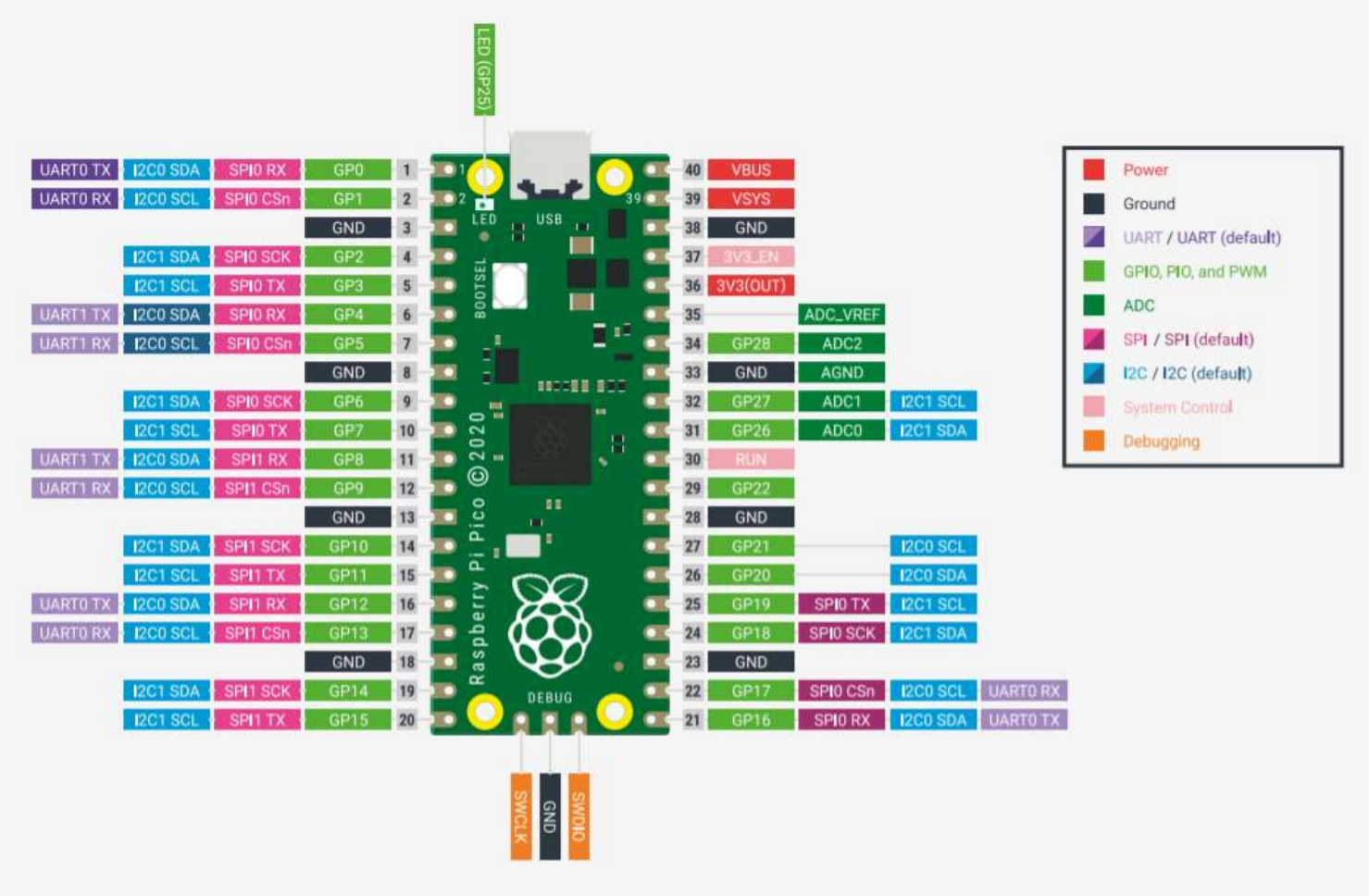
Introduction

Last week, I introduced my first Assembly Language program for the Raspberry Pi Pico. This was a version of my flashing LED program that I implemented in a number of programming languages for the regular Raspberry Pi. In the original article, I required three routines written in C to make things work. Yesterday, I showed how to remove one of these C routines, namely to have the main routine written in Assembly Language. Today, I'll show how to remove the two remaining C routines, which were wrappers for two SDK routines which are implemented as inline C functions and as a consequence only usable from C code.

In this article, we'll look at the structure for the GPIO registers on the RP2040 and how to access these. The procedure we are using is called bit-banging because we are using one of the two M0+ ARM CPU cores to loop banging the bits in the GPIO registers to turn them on and off. This isn't the recommended way to do this on the RP2040. The RP2040 implements eight programmable I/O (PIO) co-processors that you can program to offload this sort of thing from the CPU. We'll look at how to do that in a future article, but as a first step we are going to explore bit-banging mostly to understand the RP2040 hardware better.

The RP2040 GPIO Hardware Registers

There are 28 programmable GPIO pins on the Pico. There are 40 pins, but the others are ground, power and a couple of specialized pins (see the diagram below).



This means that we can assign each one to a bit in a 32-bit hardware register which is mapped to 32-bits of memory in the RP2040's address space. The GPIO functions are controlled by writing a 1 bit to the correct position in the GPIO register. There is one register to turn on a GPIO pin and a different register to turn it off, this means you don't need to read the register, change one bit and then write it back. It's quite easy to program these since you just place one in a CPU register, shift it over by the pin number and then write it to the correct memory location. These registers start at memory location 0xd0000000 and are defined in sio.h. Note there are two sio.h files, one in hardware_regs which contains the offsets and is better for Assembly Language usage and then one in hardware_structs which contains a C structure to map over the registers. Following are the GPIO registers, note that there are a few other non-GPIO related registers at this location and a few unused gaps in case you are wondering why the addresses aren't contiguous.

Register	Address
gpio_in	0xd0000004
gpio_hi_in	0xd0000008
gpio_out	0xd0000010
gpio_set	0xd0000014
gpio_clr	0xd0000018
gpio_togl	0xd000001c
gpio_oe	0xd0000020
gpio_oe_set	0xd0000024
gpio_oe_clr	0xd0000028

gpio_togl	0xd000002c
gpio_hi_out	0xd0000030
gpio_hi_set	0xd0000034
gpio_hi_clr	0xd0000038
gpio_hi_togl	0xd000003c
gpio_hi_oe	0xd0000040
gpio_hi_oe_set	0xd0000044
gpio_hi_oe_clr	0xd0000048
gpio_hi_oe_togl	0xd000004c

Notice that there are a number of _hi_ registers, perhaps indicating that Raspberry plans to come out with a future version with more than 32 GPIO pins.

In the SDK and my code below we just write one bit at a time, I don't know if the RP2040's circuitry can handle writing more bits at once, for instance can we set all three pins to output in one write instruction? Remember hardware registers tend to have minimal functionality to simplify the electronics circuitry behind them so often you can't get too complicated in what you expect of them.

Bit-Banging the Registers in Assembly

Below is the new updated program that doesn't require the C file. In our routines to control the GPIO pins, we pass the pin number as parameter 1, which means it is in **R0**. We place 1 in **R3** and then shift it left by the value in **R0** (the pin number). This gives the value we need to write. We then load the address of the register we need, which we specified in the .data section and write the value. Note that we need two **LDR** instructions, once to load the address of the memory address and then the second to load the actual value.

```
@
@ Assembler program to flash three LEDs connected to the
@ Raspberry Pi GPIO port using the Pico SDK.
@
@

.EQU LED_PIN1, 18
.EQU LED_PIN2, 19
.EQU LED_PIN3, 20
.EQU sleep_time, 200

.thumb_func
.global main            @ Provide program starting address to linker

.align  4 @ necessary alignment

main:
```

```
@ Init each of the three pins and set them to output

MOV R0, #LED_PIN1
BL gpio_init
MOV R0, #LED_PIN1
BL gpiosetout
MOV R0, #LED_PIN2
BL gpio_init
MOV R0, #LED_PIN2
BL gpiosetout
MOV R0, #LED_PIN3
BL gpio_init
MOV R0, #LED_PIN3
BL gpiosetout

loop:

@ Turn each pin on, sleep and then turn the pin off

MOV R0, #LED_PIN1
BL gpio_on
LDR R0, =sleep_time
BL sleep_ms
MOV R0, #LED_PIN1
BL gpio_off
MOV R0, #LED_PIN2
BL gpio_on
LDR R0, =sleep_time
BL sleep_ms
MOV R0, #LED_PIN2
BL gpio_off
MOV R0, #LED_PIN3
BL gpio_on
LDR R0, =sleep_time
BL sleep_ms
MOV R0, #LED_PIN3
BL gpio_off

B       loop @ loop forever

gpiosetout:
@ write a 1 bit to the pin position in the output set register
movs r3, #1
lsl r3, r0 @ shift over to pin position
ldr r2, =gpiosetdiroutreg @ address we want
ldr r2, [r2]
str r3, [r2]
bx lr
```

```
gpio_on:
movs r3, #1
lsl r3, r0 @ shift over to pin position
ldr r2, =gpiosetonreg @ address we want
ldr r2, [r2]
str r3, [r2]
bx lr

gpio_off:
movs r3, #1
lsl r3, r0 @ shift over to pin position
ldr r2, =gpiosetoffreg @ address we want
ldr r2, [r2]
str r3, [r2]
bx lr

.data
    .align  4 @ necessary alignment
gpiosetdiroutreg: .word   0xd0000024 @ mem address of gpio registers
gpiosetonreg: .word   0xd0000014 @ mem address of gpio registers
gpiosetoffreg: .word   0xd0000018 @ mem address of gpio registers
```

Having separate functions for gpio_in and gpio_out simplifies our code since we don't need any conditional logic to load the correct register address.

We loaded the actual address from a shared location. We could have loaded the base address of 0xd000000 and then stored things via an offset, but I did this to be a little clearer. If you look at the disassembly of the SDK routine, it does something rather clever to get the base address. It does:

```
movs r2, #208 @ 0xd0
lsl r2, r2, #24 @ becomes 0xd0000000
```

And then uses something like:

```
str r3, [r2, #40] @ 0x28
```

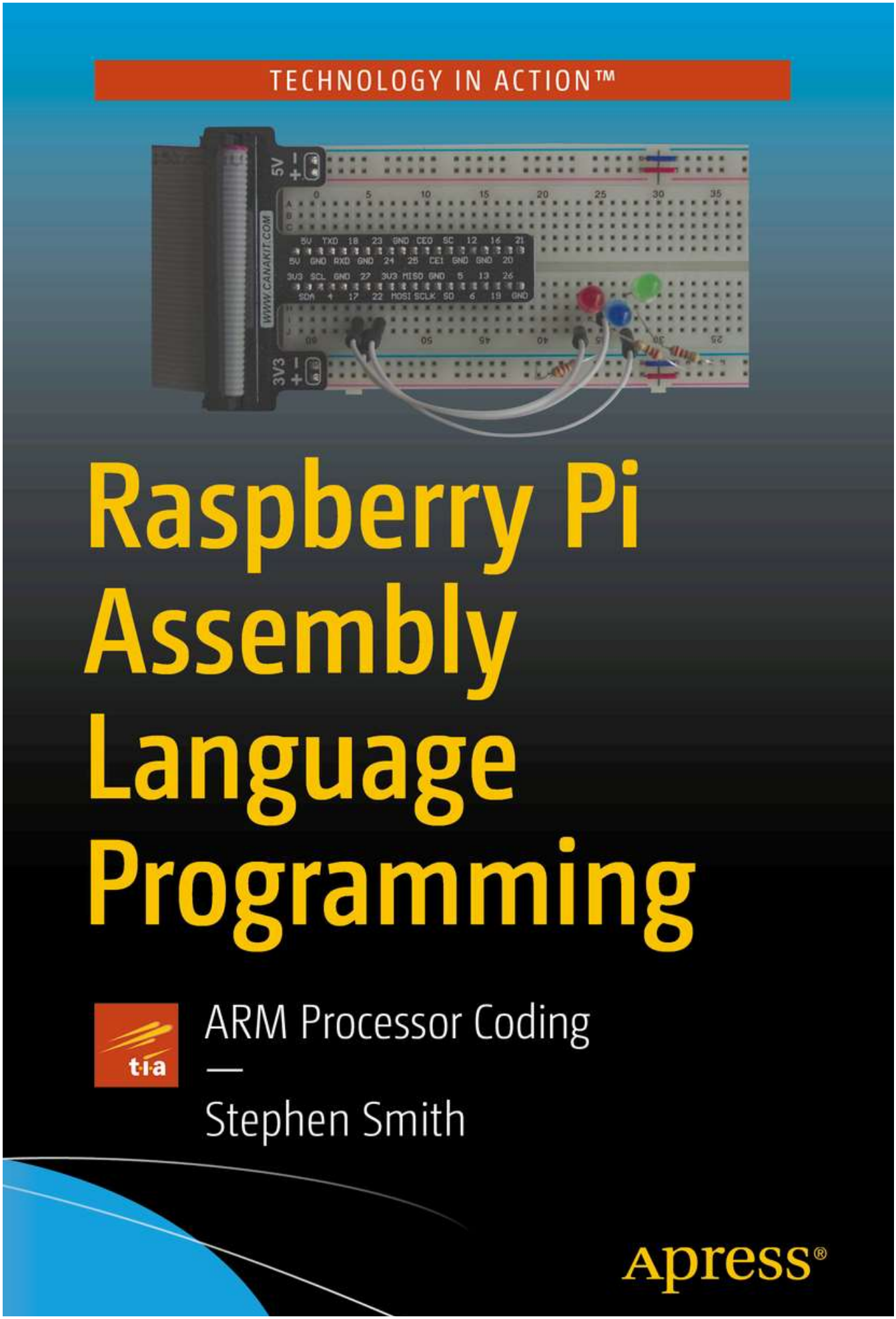
To store the value using an index which is the offset to the correct register. I thought this was rather clever on the C compiler's part and represents the optimizations that the ARM engineers have been adding to the GCC generation of ARM code. This technique takes the same time to execute, but doesn't require saving any values in memory, saving a few bytes which may be crucial in a larger program.

Summary

Writing to the hardware registers directly on the Raspberry Pi Pico is a bit simpler than the Broadcom implementation in the full Raspberry Pi. With these routines we wrote our entire program in Assembly Language. There is still C code in the SDK which will be linked into our program and we are still calling

both gpio_init and sleep_ms in the SDK. We could look at the source code in the SDK and reimplement these in Assembly Language, but I don't think there is any need. Between the RP2040 documentation and the SDK's source code it is possible to figure out a lot about how the Raspberry Pi Pico works.

For people playing with the Raspberry Pi Pico or another RP2040 based board, you can program in 32-bit ARM Assembly Language and might want to consider my book "Raspberry Pi Assembly Language Programming".


TECHNOLOGY IN ACTION™
Raspberry Pi Assembly Language Programming
ARM Processor Coding
—
Stephen Smith
tia
Apress®

Written by smist08

April 24, 2021 at 11:50 am

Posted in assembly language, raspberry pi, rp2040

Tagged with assembly language, gpio pins, raspberry pi, raspberry pi pico, rp2040

5 Responses

Subscribe to comments with RSS.

> I just went through your youtube video https://www.youtube.com/watch?v=GBRdzaAxHB8&t=40s . I learnt everything in so much of time and you concluded that in 40 minutes video. I think I need to keep in touch with you.
>
> **Puneet**
>
> April 28, 2021 at 11:51 am
>
> Reply

[…] Last time we looked at how to access the RP2040's GPIO registers directly from the CPU in Assembly Language. This is a common technique to access and control hardware wired up to a microcontroller's GPIO pins; however, the RP2040 contains a number of programmable I/O (PIO) coprocessors that can be used to offload this work from the main ARM CPUs. In this article we'll give a quick overview of the PIO coprocessors and present an example that moves the LED blinking logic from the CPU over to the coprocessors, freeing the CPU to perform other work. There is a PIO blink program in the SDK samples, which blinks three LEDs at different frequencies, we'll take that program and modify it to blink the LEDs in turn so that it works the same as the examples we've been working with. […]

I/O Co-processing on the Raspberry Pi Pico | Stephen Smith's Blog

April 30, 2021 at 10:02 am

Reply

ldr r2, [r2] should be removed.

James

January 9, 2022 at 4:22 pm

Reply

> Yes, since we just write r3 into that memory location rather than combining it with the previous value.
>
> **smist08**
>
> January 9, 2022 at 7:07 pm

Reply

Removing = before gpio’s addresses and the line below give the same result. Also many thanks posting these kind of projects.

O2

April 7, 2022 at 6:59 am

Reply