Stop Go Code Notes

This document seeks to provide notes on the stop go series of programs.

When the TM4C123GXL board powers up, most ports are configured as digital inputs. In the stop_go.c code, after the C initialization is performed, main() is called and the function initHw() is called, whose purpose is to configure the microcontroller to work with with our hardware.

The first line of the initHw() function and a function called initSystemClockTo40Mhz(). This function configures the chip to run with a 40 MHz instruction cycles, meaning that up to 40 million instructions per second (40 MIPS) can be executed by controller. This register operation will be detailed in the Hardware Guide discussion at a later time.

For the stop go programs, the hardware configuration is as follows:
- Red LED connected to port F, bit 1 (bit='1' turns on)
- Green LED connected to port F, bit 3 (bit='1' turns on)
- Push button connected to port F, bit 4 (push button makes the pin read as '0')

To use these devices connected to port F, we need to enable the clocks for GPIO port F, by setting bit 5 in the following read-modify(set bit 5)-write operation:

```
SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R5;
```

where `SYSCTL_RCGCGPIO_R5` is equal to 32.

After this bit is set, we need to wait 3 clocks for the GPIO port F module to start up as required by Section 10 of the datasheet:

```
_delay_cycles(3);
```

Since most GPIO ports are configured as digital inputs at power up, we need to configure the red and green LED pins (bits 3 and 1) as digital outputs with the following read-modify(set bits 3 and 1)-write operation:

```
GPIO_PORTF_DIR_R |= GREEN_LED_MASK | RED_LED_MASK;
```

where `GREEN_LED_MASK` equals 8 (2^3) and `RED_LED_MASK` equals 2 (2^1).

To make sure the push button is configured as a digital input (it will be on powerup, but to make the code more clear and allow the code to work properly if the hardware had been configured differently before initHw() is called), we clear bit 4 with the following read-modify(clear bit 4)-write operation:

```
GPIO_PORTF_DIR_R &= ~PUSH_BUTTON_MASK;
```

where `PUSH_BUTTON_MASK` equals 16 (2^4) and ~`PUSH_BUTTON_MASK` equals 0xFFFFFFEF.

Next, we set the output pins to the 2mA maximum current (this again is the default) with the read-modify(set bits 3 and 1)-write operation:

```
GPIO_PORTF_DR2R_R |= GREEN_LED_MASK | RED_LED_MASK;
```

Finally, we enable digital operations on all three pins with the read-modify(set bits 4, 3, and 1)-write operation:

```
GPIO_PORTF_DEN_R |= PUSH_BUTTON_MASK | GREEN_LED_MASK | RED_LED_MASK;
```

There is one remaining issue to solve. When the push button is pressed, portF, bit 4 is grounded, so it reads as a '0' to software. When the push button is not pressed, the pin floats and the value is not guaranteed. We need something to make the pin read as a '1' when the push button is not pressed. To do this, we could add an external register to pull the pin up to 3.3V (a 10 kohm resistor from the pin to 3.3V would be adequate), but to save the cost and space on the board, it is possible to turn on a pull up register (PUR) that connects the pin through a weakly-biased FET transistor that creates a little current flow just like an external pull up resistor would. Why a FET? -- It takes too much space to create a high-value resistor on the die of an integrated circuit, so a weakly-biased FET serves the same purpose. The pull-up register is enabled on the push button input with the following read-modify(set bit 4)-write operation:

```
GPIO_PORTF_PUR_R |= PUSH_BUTTON_MASK;
```

Now the hardware is initialized for our circuit in this project.

In main(), to turn off the green LED (clear bit 3) and turn on the red LED (set bit 1), the following two read-modify-writes operations are made:

```
GPIO_PORTF_DATA_R &= ~GREEN_LED_MASK;
GPIO_PORTF_DATA_R |= RED_LED_MASK;
```

Now, we call the waitPbPress() function, which is a blocking function (it only returns when the push button is pressed).

In waitPbPress(), this code executes continuously until the push button input reads as '0':

```
while(GPIO_PORTF_DATA_R & PUSH_BUTTON_MASK);
```

In this line of code, the entire port F register is read. Since we only care about bit 4 (the push button input), we "mask out" all but bit 4 by ANDing the value read from the 32-bit register with only bit 4 set (0x00000010 = 2^4 = 16), resulting in a value of 0 if the push button is pressed or 16 if the push button is not pressed. While the result is not zero (the AND result is 16), the while loop continues.

After waitPbPress() returns to main(), the red LED is turned off and the green LED is turned on with these two read-modify-writes:

```
GPIO_PORTF_DATA_R &= ~RED_LED_MASK;
GPIO_PORTF_DATA_R |= GREEN_LED_MASK;
```

The next line in main() is critical, as it prevents main() from returning, which would reset the microcontroller:

```
while(true);
```

This always needs to be at the end of your main() function.

In the bit banded version of stop go (stop_go_bitband.c), the initHw() function is identical, but the read-modify-write operations to control the LEDs are changed to write operations that allow the LEDs to be controlled individually through the bit-banded registers. The push button can also be tested in waitPbPressed by reading the single bit-band register without the need for masking out bits other than bit 4.

The LED read-modify-write operations are converted to:

```
GREEN_LED = 0;
RED_LED = 1;
```

and

```
RED_LED = 0;
GREEN_LED = 1;
```

and the push button test is just:

```
while(PUSH_BUTTON);
```

The bit banded addresses are as follows:

```
#define RED_LED      (*((volatile uint32_t *)(0x42000000 +
(0x400253FC-0x40000000)*32 + 1*4)))
#define GREEN_LED    (*((volatile uint32_t *)(0x42000000 +
(0x400253FC-0x40000000)*32 + 3*4)))
#define PUSH_BUTTON  (*((volatile uint32_t *)(0x42000000 +
(0x400253FC-0x40000000)*32 + 4*4)))
```

The address is calculated as the bit-banded based address (0x4200 0000) +
the offset of the peripheral address of PORTF_DATA_R from the start of the peripheral memory space (0x4000 0000) times 32 plus the bit number of the port times 4.

Note 1: The address of PORTF_DATA_R (APB) assumes access to all bits of the register, so bits 9:2 are set (0x40025000 | 0x000003FC = 0x400253FC) – see the DATA register in 10.5 for more information and the stop_go_masking.c example.

Note 2: A 32-bit register is assigned to control each bit in the peripheral space. The value written to this 32-bit register is either 1 or 0, turning on or off a corresponding peripheral

register bit. For each bit, there is a corresponding 32-bit register, so the address is offset by 4 bytes for each bit (bit times 4 in the calculation above). For each byte (8 bits), the address is offset by 8x4=32 bytes (offset times 32 in the equation above).

The last example, stop_go_masking using the bits 9:2 of the PORTF_DATA_R to create an address that only allows bits 3 and 1 to be updated with a write. The allows a single write to change both the red and green LED status at once. The address of PORTF_DATA_R controlling only bits 3 and 1, is:

```
#define LEDS          (*((volatile uint32_t *)(0x40025000 + 0x0A*4)))
```

where the address is 0x40025000 | 0x00000028 = 0x40025028.

The write operation to turn-on the red LED and turn-off the green LED is:

```
LEDS = RED_LED_MASK;
```

and the write operation to turn-on the green LED and turn-off the red LED is:

```
LEDS = GREEN_LED_MASK;
```

This makes it possible for different parts of the code to manipulate the different pins of port F without changing the status of other bits.

Both the masking and bit banding allow atomic operations (the bit change operation is immediate with a write so it is not divided into steps as in the read-modify-write opeations of stop_go.c). This will be very important as we move to more complicated topics.