# Lab Assignments

- Preparation for Labs
- GPIO - LED and Switch Interface
- Flash Accelerator Module
- DMA - Memory to Memory transfer
- Clock Configuration
- Hardware Timer
- DMA and Timer Integration
- UART using GPIO
- EEPROM Driver

# Preparation for Labs

C programming basics

- Functions
- Structures
- Pointers

- Bit Masking Tutorial
- Basic parts and toolkit
  - Look for any basic starter kit on amazon - mostly Breadboard, Jumper cables, LEDs, Potentiometer, Buttons, Resistors
  - Something like this

---

## Part 0: Compile the sample project

In this part, the objective is simply to compile your project, and to make sure you can load the compiled image onto your board.

- Erase the contents of your existing main.c and use the code given below
- Use the Google Chrome based serial terminal to confirm the output

```c
// file: main.c
#include <stdio.h>
// Separate include files for Clang to sort separately
#include "delay.h"
void main(void) {
  unsigned counter = 0;

  while (1) {
    printf("Running: %u\n", counter);
```

```
    ++counter;
    delay__ms(1000U);
  }
}
```

## Part 1: Get familiar with GPIO Blinky API

The objectives of this part is:

- Learn how to use existing API to blink LEDs
- Get familiar with how to develop clean C APIs
- Explore the code of the existing APIs to assess how it works

```c
#include <stdio.h>
#include "board_io.h"
#include "delay.h"
#include "gpio.h"
/* This is just a sample
 * Cross-reference the schematic, and browse through the header files included above
 * to go above and beyond, and in general play with existing APIs
 */
int main(void) {
  gpio_s led0 = board_io__get_led0();
  while (1) {
    gpio__toggle(led0);
    delay__ms(500U);
  }
  return 0;
}
```

## Conclusion

This exercise was mostly to get familiar with the board, and be excited about how your software can

control hardware. In the following weeks, you will be developing your own code by directly manipulating the microcontroller's memory to achieve your objectives.

# GPIO - LED and Switch Interface

## Objective

The objective of the assignment is to access microcontroller's port-pins to manipulate LEDs that are connected to a few on-board LEDs of the SJ2 board.

Please reference this article for good hints related to this assignment.

---

## Part 0:

In this part, we will setup basic skeleton of the code before we access the memory map to manipulate an LED.

```c
// file: main.c
#include <stdio.h>
// Separate include files for Clang to sort separately
#include "delay.h"
void main(void) {
  unsigned counter = 0;

  while (1) {
    printf("Running: %u\n", counter);
    ++counter;
    delay__ms(1000U);
  }
}
```

# Part 1:

In this part, we will actually reference the LPC user manual and manipulate one of the on-board LEDs.

You can refer Chapter 8 - Table 94 of the user manual.

```c
// file: main.c
#include <stdio.h>
// Separate include files for Clang to sort separately
// Add header files required
#include "delay.h"
void blinky_leds(void){
// 1. Refer to the datasheet and configure the direction and pin using the memory address
uint32_t *port1_pin_register = (uint32_t *)(MEMORY_ADDRESS);
    ...
// Set directions to the pins
...
while (1) {
delay__ms(100);
METHOD1: Use bitmasking techniques to set and reset the pins
```

```
  //METHOD2: Use the set and clear registers
    }
}
int main(void)
{
  blinky_leds();
  return 0;
}
```

---

# Part 2:

In this part, we will use the LPC memory map to manipulate the on-board LEDs. This will reduce us from cross referencing the LPC user manual.

 Use LPC40xx MCU Memory Map as reference

```
// file: main.c
#include <stdio.h>
// Separate include files for Clang to sort separately
// Add header files required
#include "board_io.h"
#include "delay.h"
#include "gpio.h"
#include "lpc40xx.h"
void blinky_leds(void) {
 // Step 1: Choose pin as GPIO
 LPC_IOCON->(port_pin_number) &= ~7;

 // Step 2: Enable the direction pin
 while (1) {
 delay__ms(100);
 // Use set and clear registers to set and clear pins accordingly
 }
}
```

---

# Part 3:

Create a comprehensive GPIO driver and manipulate the on-board LED's using the on-board switches.
You could use PART1: GPIO Driver as reference.

---

# Extra Credit:

Do something creative with your implemented GPIO driver.  You could have the LED blink three times
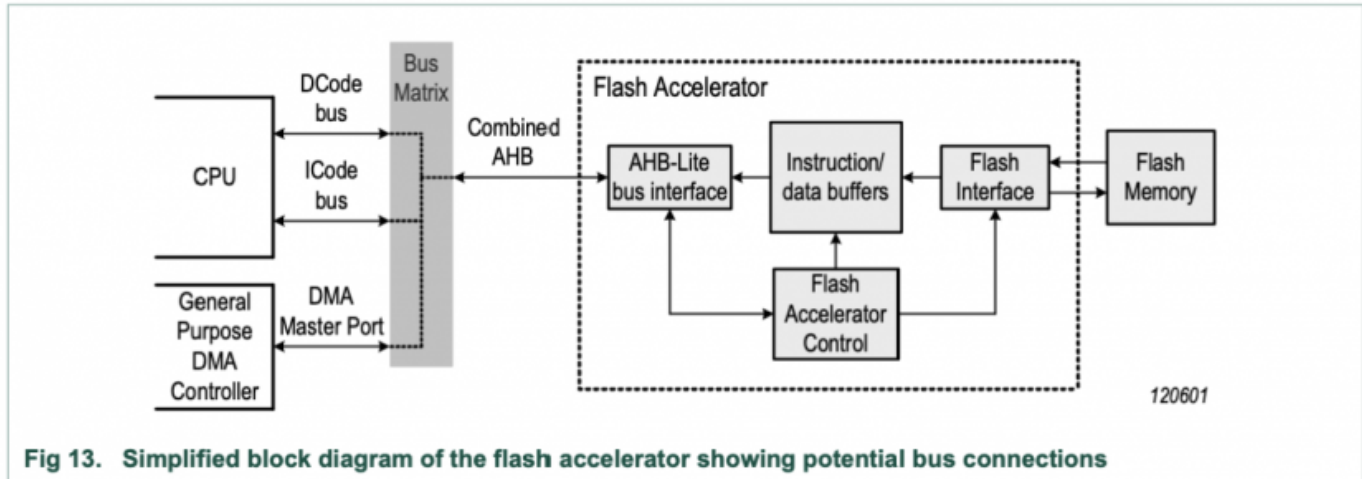when the switch is pressed.

---

# Go above and beyond

Want to have even more fun and experience multi-tasking functionality of your board's software package? Then follow this link:

- GPIO Assignment

# Flash Accelerator Module

Chapter 4 of LPC40xx controller outlines the information about the flash accelerator module.



**Fig 13. Simplified block diagram of the flash accelerator showing potential bus connections**

Typically in a microcontroller, there is either an internal or external flash memory that is interfaced over a serial bus (SPI). This memory needs to be memory mapped to the processor and hence the intent of the flash accelerator module is to provide an abstraction between the CPU and the serial memory.

# Critical Thinking Questions

**1. Flash Accelerator Functionality**

a. What is the main purpose of the flash accelerator in the LPC408x/407x microcontroller?

b. Describe how the flash accelerator improves CPU performance when accessing flash memory.

**2. Flash Accelerator Blocks**

a. Identify the main functional blocks of the flash accelerator.

b. What are the roles of the I-code and D-code buses in the context of the flash accelerator?

**3. Prefetch Mechanism**

a. Explain the difference between a "fetch" and a "prefetch" in the flash accelerator.

b. Why does the flash accelerator prioritize data access over instruction fetch?

**4. Flash Programming Issues**

a. Why can't the flash memory be accessed during programming or erase operations?

b. What steps must be taken to prevent a system failure during flash memory programming?

## 5. Flash Accelerator Configuration Register

a. What does the FLASHCFG register control in the flash accelerator?

b. How does changing the FLASHCFG register affect the flash accelerator's operation?

## 6. CPU Stalls and Flash Access

a. What happens if a flash instruction fetch and a flash data access occur simultaneously?

b. Under what conditions would the CPU experience a stall during flash memory access?

# DMA - Memory to Memory transfer

## Objective

- Copy data from one memory block to another memory block using DMA controller
- Use implemented driver to compare the performance between DMA copy and CPU copy

---

## Part 0: Read the DMA chapter

The first step is to familiarize yourself with the DMA peripheral. Read the LPC user manual multiple times before you start the assignment.

## 35.1 Basic configuration

The GPDMA is configured using the following registers:

1. Power: In the PCONP register (Section 3.3.2.2), set bit PCGPDMA.

   **Remark:** On reset, the GPDMA is disabled (PCGPDMA = 0).

2. Clock: The GPDMA operates at the AHB bus rate, which is the same as the CPU clock rate (CCLK).

3. Interrupts: Interrupts are enabled in the NVIC using the appropriate Interrupt Set Enable register.

4. Programming: see Section 35.6.

5. Select the DMA channel alternate requests in the DMA channel request select register in the system control block. See Section 3.3.7.7.

## Part 1: Basic DMA driver

In this portion of the lab, you will implement GPDMA driver by choosing one of the DMA channels (0-7, preferably 7) for performing memory-to-memory copy between 2 arrays. You will be modifying DMA registers available in `lpc40xx.h`

```
#include <stdbool.h>


#include "lpc40xx.h"
#include "lpc_peripherals.h"
typedef enum {
  DMA__CHANNEL_0 = 0,
  DMA__CHANNEL_1,
  DMA__CHANNEL_2,
  DMA__CHANNEL_3,
  DMA__CHANNEL_4,
  DMA__CHANNEL_5,
```

```c
    DMA__CHANNEL_6,
    DMA__CHANNEL_7,
} dma__channel_e;
LPC_GPDMACH_TypeDef *dma_channels[] = {LPC_GPDMACH0, LPC_GPDMACH1, LPC_GPDMACH2, LPC_GPDMACH3,
                                       LPC_GPDMACH4, LPC_GPDMACH5, LPC_GPDMACH6, LPC_GPDMACH7};
void dma__initialize(void) {
  // 1. Power on GPDMA peripheral; @see "lpc_peripherals.h"
  lpc_peripheral__turn_on_power_to(...); // Fill in the arguments
}
void dma__copy(dma__channel_e channel, const void *dest, void *source, uint32_t size_in_bytes);
  // 2. Enable GPDMA - Set the Enable bit in Config register

  // 3. Clear any pending interrupts on the channel to be used by writing to the IntTCClear
  // and IntErrClear register. The previous channel operation might have left interrupt active
  // 4. Write the source address into the CSrcAddr register.
  dma_channels[channel]->CSrcAddr = (uint32_t)source;

  // 5. Write the destination address into the DestAddr register.

  // 6. Write the control information into the Control register in one instruction
  // transfer size [11:0]
  // source burst size [13:12]
  // dest burst size [15:14]
  // source transfer width [20:18]
  // destination transfer width [23:21]
  // source address increment [26]
  // destination address increment [27]
  // TCI enable
  // Enable channel by setting enable bit for the channels CConfig register
  // IMPORTANT:
  // Poll for DMA completion here
}
bool check_memory_match(const void *src_addr, const void *dest_addr, size_t size) {
  // Write code to check the data of source and destination arrays
  // Hint: You may use memcmp()
}
```

```
int main(void) {
  const uint32_t array_len = _____; //specify a length (< 2^12)
  // Be aware that you only have 64K of RAM where stack memory begins
  // You can make these 'static uint32_t' or make them global to not use stack memory
  uint32_t src_array[array_len];
  uint32_t dest_array[array_len];
  // Initialize the source array items to some random numbers
  // Choose a free DMA channel with the priority needed.
  // DMA channel 0 has the highest priority and
  // DMA channel 7 the lowest priority
  dma__initialize();
  dma__copy(...);        // fill in the arguments

  const bool memory_matches = check_memory_match(...); // fill in the arguments
  while (true) {
  }
  return 1; // main() shall never return}
```

---

## Part 2: Compare Performance of various methods

Reuse code from Part 1. Reference the code below to measure the time taken for DMA copy, programmatic copy and standard library function `memcpy()`

```
#include <string.h>
// Declare source and destination memory for the lab
static uint8_t memory_array_source[4096];
static uint8_t memory_array_destination[4096];
// Re-initialize the memory so they do not match
static void reset_memory(void) {
  for (size_t i = 0; i < sizeof(memory_array_source); i++) {
    memory_array_source[i] = i;
    memory_array_destination[i] = i + 1;
  }
}
```

```c
// Copy memory using standard library memcpy()
static uint32_t memory_copy__memcpy(void) {
  const uint32_t start_time_us = sys_time__get_uptime_us();
  {
    memcpy(...); // TODO: Fill in the parameters
  }
  const uint32_t end_time_us = sys_time__get_uptime_us();

  return (end_time_us - start_time_us);
}
// Copy memory using the DMA
static uint32_t memory_copy__dma(void) {
  const uint32_t start_time_us = sys_time__get_uptime_us();
  {
    dma__copy(...); // TODO: Fill in the parameters
  }
  const uint32_t end_time_us = sys_time__get_uptime_us();

  return (end_time_us - start_time_us);
}
static uint32_t memory_copy__loop(void) {
  const uint32_t start_time_us = sys_time__get_uptime_us();
  {
    // TODO: Use a for loop
  }
  const uint32_t end_time_us = sys_time__get_uptime_us();

  return (end_time_us - start_time_us);
}
int main()
{
  dma__initialize(...);
  // Test 1:
  reset_memory();
  const uint32_t dma_us = memory_copy__dma();
  if (!check_memory_match(memory_array_source, memory_array_destination, sizeof(memory_array_source))
```

```
    puts("ERROR: Memory copy did not work");
  }
  // TODO: Reset memory, then perform memory copy, and check if memory matches
  // Test 2:
  const uint32_t loop_us = memory_copy__loop();


  // Test 3:
  const uint32_t memcpy_us = memory_copy__memcpy();


  printf("DMA copy completed in: %lu microsec\n", dma_us);
  printf("For loop completed in: %lu microsec\n", loop_us);
  printf("memcpy completed in  : %lu microsec\n", memcpy_us);


  while (true) {
  }
  return 1; // main() shall never return}
```

## Requirements

- Part 1 and Part 2 must be completed and fully functional.
  - You are encouraged to ask questions for any line of code that is not well understood (or magical).
- Try changing source burst size and destination burst size bits to understand performance improvements
  - Note the time taken in each case and turn in the screenshots
- Should be able to make it work for any DMA channels(0-7) or array size
  - We may ask you to change the channel or array length and then recompile and re-flash your board to and prove it works
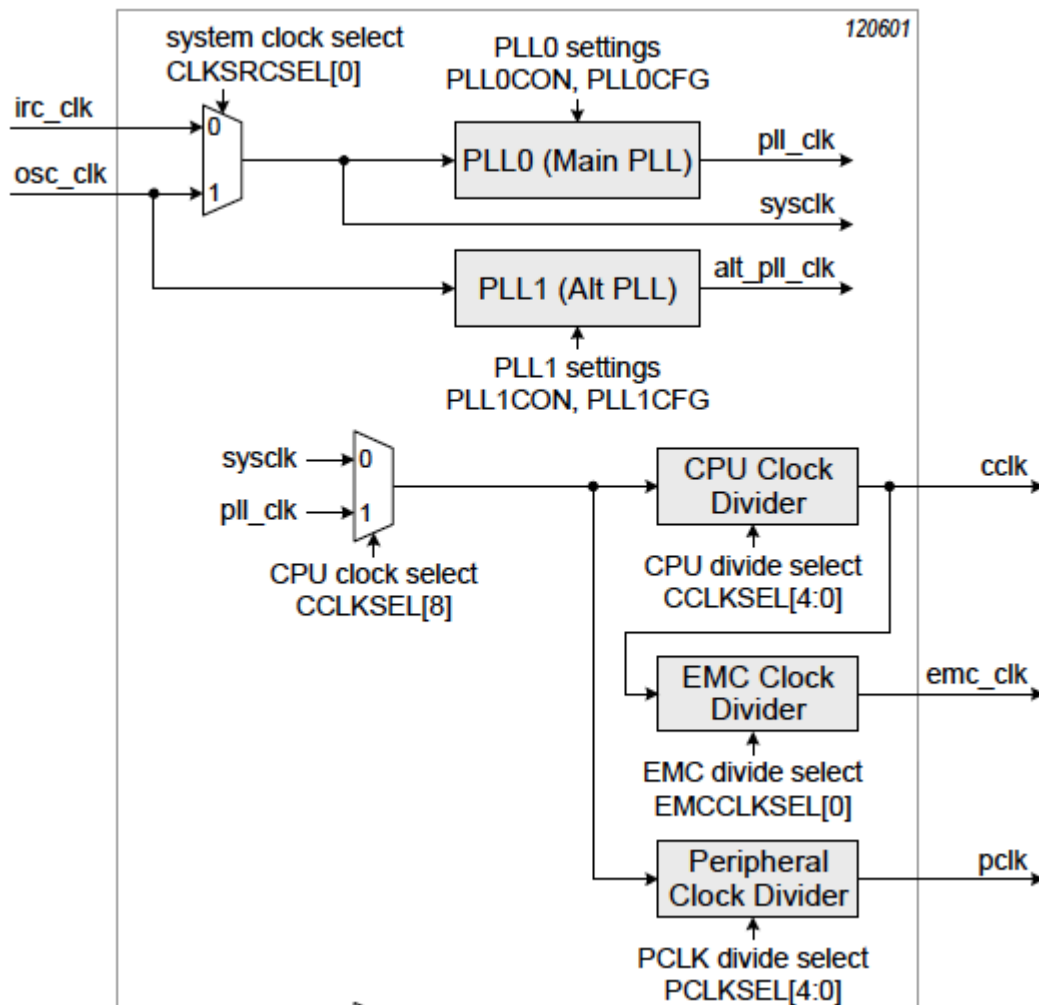
## What to turn in:

- Turn in the **screenshots** of terminal output for different burst size

> ? **Extra Credit - Make program to choose DMA channels using onboard buttons and run DMA copy process**

# Clock Configuration

One of the first aspects of a startup process is CPU clock configuration. Usually, a microcontroller would have an internal "RC" oscillator, which is referenced in the diagram below as an IRC. This internal oscillator allows the CPU to boot before the application code can switch to another clock source if desired.



# Exercise

Note that because you will be altering the clock source, your UART driver which is used to output printf() data may not work.

1. OSC clock
   1. Verify if you have an external oscillator installed on your board
   2. Switch the CLCKSEL[8] to switch to SYSCLK
   3. Modify CLKSRCSEL[0] to switch to OSC clock
   4. Validate that you are able to execute code by blinking an LED

2. Revert your changes from the previous part to complete this part
   1. Divide the Peripheral clock 2 using PCLKSEL[4:0]
   2. Use 38400 in webserial application and check what is output when you hit RESET on your board
   3. Slow down the webserial communication rate by 2 (ie: 19200 vs. 38400)
      1. Validate that you are able to see the printf() output
      2. Explain your observations

3. Try setting the CCLKSEL[4:0] to 0
   1. Is your LED still blinking?
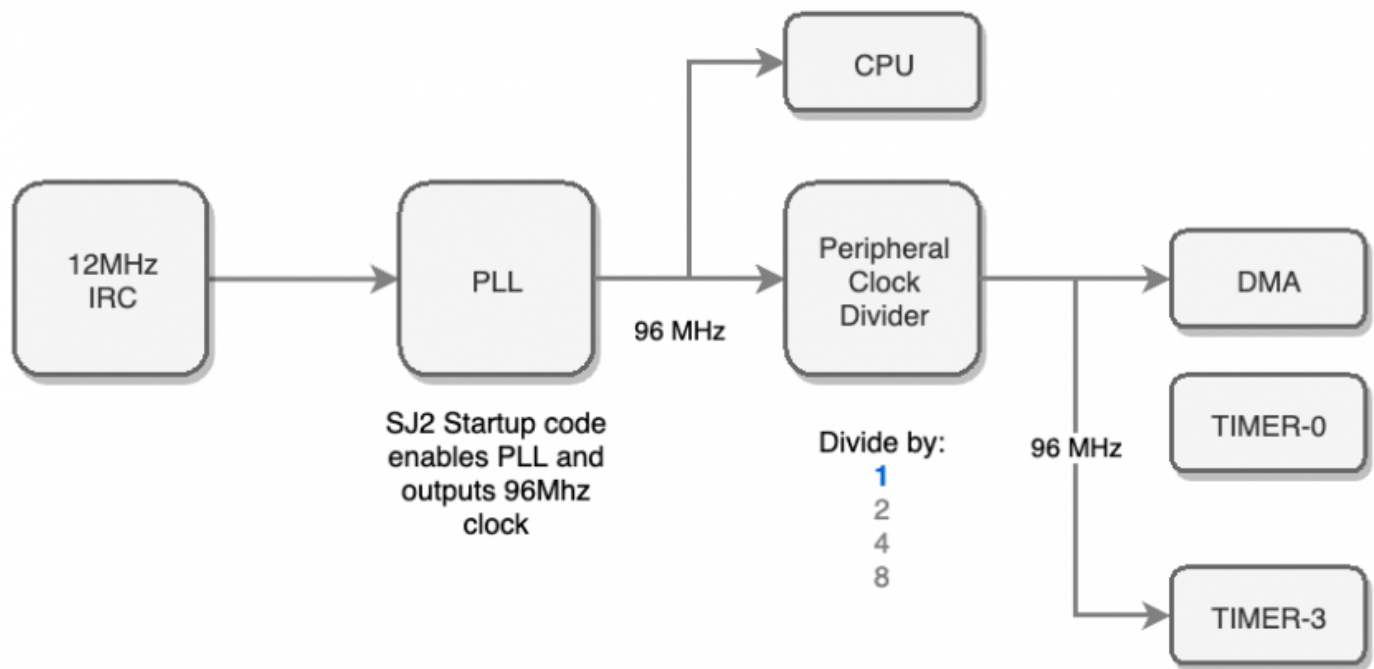   2. Explain your results

# Hardware Timer

A hardware timer is a time tracking peripheral that runs independent of the foreground CPU instructions. For example, you can start to count the clock cycles using a hardware timer independent of the instructions the CPU is processing. Fundamentally, a hardware timer is a CPU peripheral, but you can consider "CPU peripheral" as an independent CPU, or co-processor.

## Part 0: Fundamentals

Objective of this part is to fully understand the clock system, and the registers of the Timer Peripheral.

First thing to recognize is that all peripherals are configured to receive the PLL clock. As a reminder, `12MHz` clock on the SJ2 (NXP chip) is multiplied to `96MHz` , and this clock is then sourced to the hardware timer peripheral. What this means is that you can digitally count the number of clock cycles of the CPU which is `96Mhz` , and each timer register ticking up (or incrementing) is at the rate of about `10.4ns` .

CPU

12MHz
IRC → PLL → Peripheral Clock Divider → DMA

96 MHz

SJ2 Startup code enables PLL and outputs 96Mhz clock

Divide by:
1
2
4
8

96 MHz → TIMER-0

96 MHz

TIMER-3

The LPC User manual chapter of interest is:

# UM10562

## Chapter 24: LPC408x/407x Timer0/1/2/3

Rev. 3 — 12 March 2014                                    User manual

The register of relevance:

- PC
  - This register will count the input clock which is the PCLK (peripheral clock)
  - When it equals to the value of (PR + 1) then it resets back down to zero, and increment TC by 1
- PR
  - This register provides means to divide the clock, and increment the TC register slower than the input clock of 96Mhz. So for example, `if (PR == 95)`, then input clock of 96Mhz will increment the PC register, and when that equals to 95, the TC will increment by 1. So in this example, we will configure such that TC register will increment at 1Mhz rather than 96MHz
- Match Registers
  - When TC matches one of the four registers, you can generate events such as resetting the timer
- Capture Registers
  - These allow you to capture events such as an external GPIO (switch or a sensor) giving data by means of signal pulses. The capture registers can capture the time when a GPIO goes low, or it

goes high, and in the end you can use this peripheral to read pulse width modulation signals

| | | | 0x4009 0000 (TIMER2), 0x4009 4000 (TIMER3)) | | |
|---|---|---|---|---|---|
| Name | Access | Address offset | Description | Reset value[1] | Section |
| IR | R/W | 0x000 | Interrupt Register. The IR can be written to clear interrupts. The IR can be read to identify which of eight possible interrupt sources are pending. | 0 | Table 540 |
| TCR | R/W | 0x004 | Timer Control Register. The TCR is used to control the Timer Counter functions. The Timer Counter can be disabled or reset through the TCR. | 0 | Table 541 |
| TC | R/W | 0x008 | Timer Counter. The 32 bit TC is incremented every PR+1 cycles of PCLK. The TC is controlled through the TCR. | 0 | Table 542 |
| PR | R/W | 0x00C | Prescale Register. When the Prescale Counter (PC) is equal to this value, the next clock increments the TC and clears the PC. | 0 | Table 543 |
| PC | R/W | 0x010 | Prescale Counter. The 32 bit PC is a counter which is incremented to the value stored in PR. When the value in PR is reached, the TC is incremented and the PC is cleared. The PC is observable and controllable through the bus interface. | 0 | Table 544 |
| MCR | R/W | 0x014 | Match Control Register. The MCR is used to control if an interrupt is generated and if the TC is reset when a Match occurs. | 0 | Table 545 |
| MR0 | R/W | 0x018 | Match Register 0. MR0 can be enabled through the MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt every time MR0 matches the TC. | 0 | Table 546 |
| MR1 | R/W | 0x01C | Match Register 1. See MR0 description. | 0 | Table 546 |
| MR2 | R/W | 0x020 | Match Register 2. See MR0 description. | 0 | Table 546 |
| MR3 | R/W | 0x024 | Match Register 3. See MR0 description. | 0 | Table 546 |
| CCR | R/W | 0x028 | Capture Control Register. The CCR controls which edges of the capture inputs are used to load the Capture Registers and whether or not an interrupt is generated when a capture takes place. | 0 | Table 547 |
| CR0 | RO | 0x02C | Capture Register 0. CR0 is loaded with the value of TC when there is an event on the CAPn.0 input. | 0 | Table 548 |
| CR1 | RO | 0x030 | Capture Register 1. See CR0 description. | 0 | Table 548 |
| EMR | R/W | 0x03C | External Match Register. The EMR controls the external match pins. | 0 | Table 549 |
| CTCR | R/W | 0x070 | Count Control Register. The CTCR selects between Timer and Counter mode, and in Counter mode selects the signal and edge(s) for counting. | 0 | Table 550 |

# Part 1: Bring up the Timer

In this part, we will configure the Timer peripheral to make sure that it is incrementing as expected.

There are four HW timers on the NXP LPC40xx. We will use the Timer-3 peripheral for this assignment; we could theoretically use any HW timer peripheral, but the Timer-0 is already utilized by the sample

project specifically to support the delay APIs, and to get system uptime.

```c
#include "lpc40xx.h"
#include "lpc_peripherals.h"
void hw_timer3__initialize(void) {
  // 1. Power on TIMER peripheral as per the timer_num argument; @see "lpc_peripherals.h"
  lpc_peripheral__turn_on_power_to(...); // Fill in the arguments
}
void hw_timer3__enable_timer(void) {
  // 2. Configure the PR register


  // 3. Configure the PC register
  // 4. Configure the TCR register
}
uint32_t hw_timer3__get_timer_counter(void) {
  // Return the TC register value
}
int main(void) {
  hw_timer3__initialize(....); // fill in the arguments
  hw_timer3__enable_timer();
  printf("Timer3 MR0 value: %u\n", LPC_TIM3->MR0);
  // TODO: Print other registers:
  // TCR
  // PR
  // PC

  while (true) {
    □  delay__ms(1000);
      printf("TC value: %u\n",hw_timer3__get_timer_counter());
    }
  }
  return 1; // main() shall never return}
```

# Part 2: Delay API

In this part, we will build an API to delay the CPU execution with precision of within a few tens of nanoseconds.

Reuse code in Part 1 and create a new api called `delay__ns(uint32_t nanos)`. Note that in the sample code below, we use an API (`delay__us`) to estimate that our nanosecond delay API is *roughly* correct. The `delay__us` API is not very precise and may easily be inaccurate by 1-2uS.

```c
#include "lpc40xx.h"
#include "lpc_peripherals.h"
void delay__ns(uint32_t nanos) {
  // Use TC as reference and keep looping inside this function until delay is done


}
void hw_timer3__enable_timer(void) {
  // From Part 1
}
int main(void) {
  const uint32_t nanos = _____; //fill in the nanoseconds (Ex. 500)

  // Initialize
  hw_timer3__initialize(...);
  hw_timer3__enable_timer();

  uint32_t start_time_us = 0;
  uint32_t end_time_us = 0;
  while (true) {
      start_time_us = sys_time__get_uptime_us();
      delay__ns(nanos);
      end_time_us = sys_time__get_uptime_us();

      printf("Diff in microsecs: %u\n", (end_time_us - start_time_us));
      delay__ms(1000);
    }
```

```
    }
    return 1; // main() shall never return}
```

## Part 3: Make Delay API precise

In this part, we will apply a software instruction cycle compensation to the delay function you built.

- You will inspect the generated `*.lst` file (in your build directory)
- Assuming average of 2 clock cycles per instruction, you will compensate for this for the delay
- If the delay API is called with fewer nanoseconds than minimum, you can exit the function immediately
- Note that in case you cannot find your function in the lst file, please try using GCC "no inline" attribute.
  - `void delay_ns() __attribute__ ((noinline));`

```
void delay__ns(uint32_t nanos) {
  const uint32_t function_invocation_approximate_cycles = 10; // You determine the number

  // At 96MHz, it is approximately 10ns per clock cycle
  const uint32_t function_invocation_approximate_nanos = 10 * function_invocation_approximate_cycles;

  // Compensate...
  if (nanos > function_invocation_approximate_nanos) {
    nanos -= function_invocation_approximate_nanos;
  }}
```

## Requirements

- Part 1 and Part 2 must be completed and fully functional.
  - You are encouraged to ask questions on any of the register settings
- The delay function you build must compensate for software delay (Part 3)
- Change seconds/nanos values and check the output prints

- Turn in the print screenshots

## What to turn in:

- Code snippets
- Turn in the **screenshots** of terminal output for different seconds/nanoseconds values

---

# Extra credit: Going above and beyond:

- You can use the PC and the TC registers to form a 64-bit counter
  - The only trick is that you will have to read them in a way to make sure that you do not read one register value while the other one has overflows
  - See reference code below for hints

```
uint64_t sys_time__get_uptime_us(void) {
  uint32_t before = 0;
  uint32_t after = 0;
  uint32_t tc = 0;
  /**
   * Loop until we can safely read both the rollover value and the timer value.
   * When the timer rolls over, the TC value will start from zero, and the 'after' value will be less
   * value in which case, we will loop again and pick up the new rollover count.
   */
  do {
    before = LPC_TIM3->PC;
    tc = LPC_TIM3->TC;
    after = LPC_TIM3->PC;
  } while (before < after);
  uint64_t bit64 = ((uint64_t)after << 32) | tc;}
```
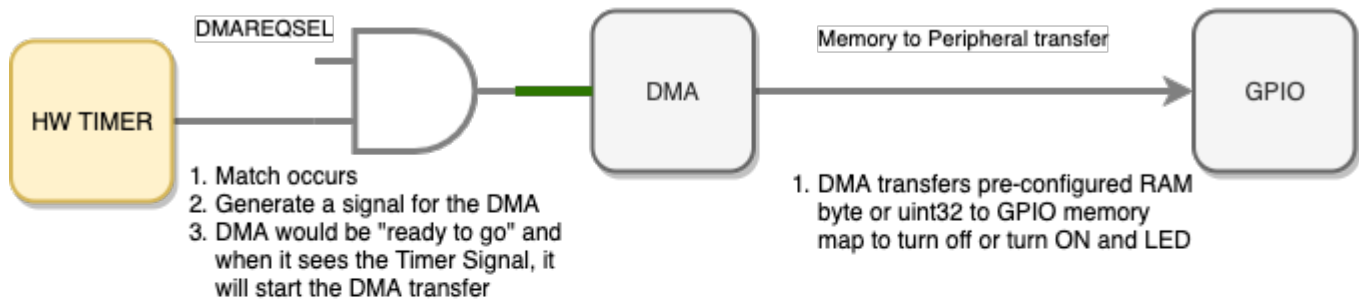
# DMA and Timer Integration

In this assignment, we will use multiple CPU peripherals and allow them to talk to each other per the design of the NXP chip. This means that you can only "program" the hardware to do what it is capable of, and we cannot arbitrarily have peripherals talk to each other unless NXP chip provides such capability.
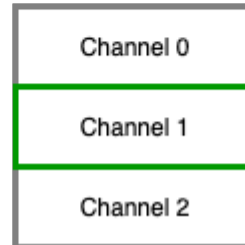
---

## Part 0: Fundamentals

Let us first understand the overall block diagram of various different components we will be programming. The objectives to understand are:

- HW Timer will generate a "trigger" signal for the DMA
  - We will use a "Match" register to generate the trigger signal
- DMA channel will be pre-configured to transfer a value from RAM to the GPIO peripheral

  - The channel `CONFIG` register bits, particularly the `DESTPERIPHERAL` chooses which channel will receive what trigger request

DMAREQSEL

HW TIMER

1. Match occurs
2. Generate a signal for the DMA
3. DMA would be "ready to go" and
   when it sees the Timer Signal, it
   will start the DMA transfer

DMA

Memory to Peripheral transfer

GPIO

1. DMA transfers pre-configured RAM
   byte or uint32 to GPIO memory
   map to turn off or turn ON and LED

Configuration of the Channel CONFIG:

1. ENABLE the channel, but the TIMER will trigger
   the actual transfer
2. **DESTPERIPHERAL** bits need to be set to
   Timer3-MRx to actually have the timer trigger the
   DMA channel
3. TRANSFERTYPE will be "Memory (RAM) to
   Peripheral (GPIO)"

Channel 0

Channel 1

Channel 2

---

# Part 1: Program DMA to write GPIO

In this part, we will configure the DMA and test it **independently of the HW timer** to ensure that it is able to write to our GPIO register correctly. The objective is to program the DMA to turn on or turn off an LED, and we will trigger the DMA manually in software (rather than having the HW timer do that).

We will need to:

1. Choose a persistent (global) RAM variable as source of data (`static` variable)
   - This is the value we want to write to either the GPIO SET or CLR register. For example, if you are to set or clear led number 13, then you will set the variable's value to `(1 << 13)`
   - Note that the LEDs are active LOW, so if the LED is lit by default (before the `main()` function), then you would set DMA destination address to write to the GPIO SET register

2. Set DMA source as Memory (RAM)
3. Set DMA destination as Peripheral (GPIO)
   - For this part it may not matter, but once the HW timer is used as a trigger, then you have to set the destination as a peripheral type

4. Set source size as one non-incrementing byte address
5. Set destination size as non-incrementing byte address
6. Transfer type would be memory-to-memory because the destination is set to address of GPIO

memory register

By the end of this portion of the lab, you should prove yourself that you can use the DMA to write the GPIO peripheral to turn off an LED. The register settings may be a bit confusing, so here is a note from a previous student:

> When setting the Destination peripheral note that you are not setting the location where the transfer is going to but you are setting the source of the DMA request and in the case where the source is the timer match then you will need to set a the transfer type as a memory to memory.

```c
#include <stdbool.h>

#include "lpc40xx.h"
#include "lpc_peripherals.h"
// All DMA related functions and global variables to be re-used from previous DMA lab
void dma__setup_ram_to_peripheral() {
  // Reuse code from previous lab dma__copy() function for register configuration
  // In CControl register, do not increment the destination address

  .

  .

  .

  // In CConfig register
  // Set TIMER3 MR0 as DESTPERIPHERAL
  // Set Memory-to-peripheral as TRANSFER TYPE
  // Example: We wish to turn on P1.26
  // We are trying to do this: 'LPC_GPIO1->CLR = (1 << 10)'
  // Use 'static' such that source memory never goes out of scope
  static const uint32_t value_from_ram_you_want_to_write = (1 << 26);
  [CH2]->SRC = (uint32_t) &value_from_ram_you_want_to_write;
  [CH2]->DST = &(LPC_GPIO1->CLR);
}
int main(void) {
  LPC_SC->DMAREQSEL = _____; // Set Timer3 Match Compare 0 as DMA request signal
  // DMA initialization to initialize common GPDMA registers
  dma__initialize();
```

```
   // Use SET register to turn off LED P1.26
   // Startup code should have enabled the LED, so you will disable (turn it off)
   dma__setup_ram_to_peripheral();


   while (true) {
   }
   return 1; // main() shall never return}
```

## Part 2: Configure HW Timer to trigger DMA

In this part, we will enable the HW timer to automatically kick the DMA to do the transfer of data from memory to the GPIO peripheral. Since the DMA is already configured to transfer a fixed byte from our RAM to the GPIO peripheral, the only thing to really figure out is how to have the HW timer kick a specific DMA channel.

General steps:

1. HW timer (TC register) starts at value of zero
2. HW timer is pre-programmed to kick the DMA upon a "match"
   - Which means that we need to choose our desired value of the "match register"

3. DMA should be programmed to receive DMA start request from your HW timer

You can consider a few approaches:

- HW Timer match can stop the TC when it occurs (and it will still generate DMA trigger signal)
- HW Timer match can reset TC back to zero to make it a periodic DMA request

By the end of this portion of the lab, you should ensure that your timer is working, and upon the match register value match, that it does what you designed the timer to do (such as reset the TC, and generate DMA)

```
 // Reuse the TIMER3 code from previous lab
 // Set MR0 register to peripheral clock frequency
 // Set PR register to 95, such that each TC increments every 1uS (good for easy calculations)
 void setup_timer3_match0_dma_trigger(void) {
```

```
    // TODO: complete this
}
```

# Part 3: Integration

In this part, we will ensure that after you prove that the Timer can kick pre-programmed DMA channel, there should be a graceful terminal condition such that the transfer does not happen again unless requested by you as a programmer. The objective is to bring it all together, and use Part 0 - Part 2 to turn off an LED by using the timer and the DMA.

# Part 4: Blinky LED

To earn bonus points, create an application that will:

- Setup two channels of DMA
  - Match 0 triggers DMA Channel 0
  - March 1 triggers DMA Channel 1
- Setup HW timer to trigger CH0 every odd second (1, 3, 5, etc.)
  - Use "Match 0"
- Setup HW timer to trigger CH1 every even second (2, 4, 6, etc.)
  - Use "Match 1"
- Create an application that will blink an LED at 1Hz without writing the GPIO peripheral

# Requirements

- Source code submitted to Canvas
- Any accompanying screenshots or videos

# UART using GPIO

The objective of this assignment is to emulate UART in software. You will use a GPIO pin to transmit data to another UART receiver.
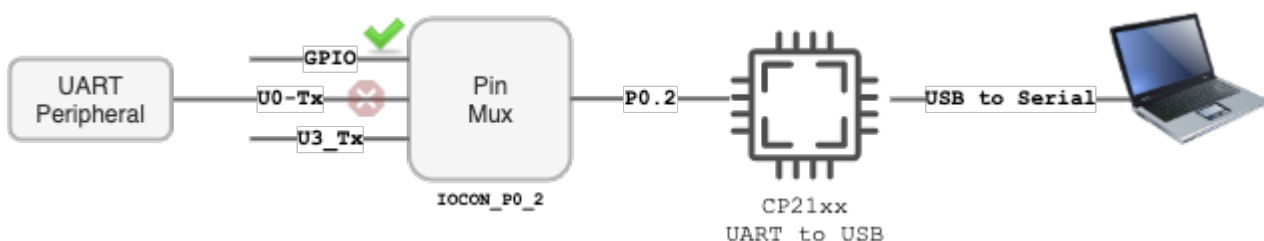
Here is another bookstack page to reference:

- http://books.socialledge.com/books/sjsu---embedded-drivers-rtos/page/uart

---

## Part 0: Fundamentals

In this part, you will understand the fundamentals of what you are trying to accomplish in this assignment. We will disconnect the UART peripheral that transmits data, and take control of the pin using the GPIO peripheral.

Each I/O pin can be selected for a specific function. By default in the SJ2 software code, the `P0.2` is selected for UART transmission. We will disconnect this from the UART peripheral, and take control of this pin using the GPIO peripheral.



Sample code below; please inspect the function calls to figure out what register it may be writing, and correlate this with the LPC user manual. For example, confirm that `gpio__construct_as_output()` is writing to the `DIR` register.

```
#include "gpio.h"
```

```
const uint32_t pin2 = 2;
// See board_io.c for reference
static void disconnect_uart_peripheral(void)
  gpio__construct_with_function(GPIO__PORT_0, pin2, GPIO__FUNCTION_0); // P0.2 - Uart-0 Tx
  // Construct P0.2 as an output pin and set to logic HIGH to "IDLE" level of the UART signal
  gpio__construct_as_output(GPIO__PORT_0, pin2);
  LPC_GPIO0->SET = (1 << pin2); // Set P0.2 as HIGH (3.3 logic "1")}
```
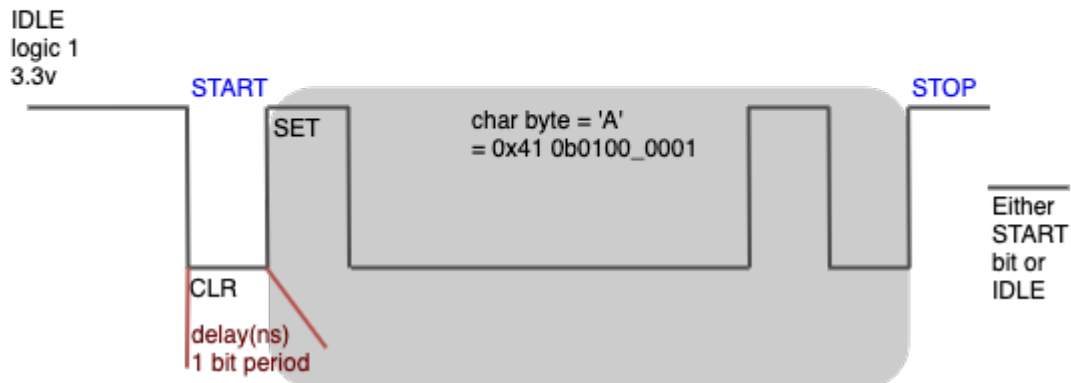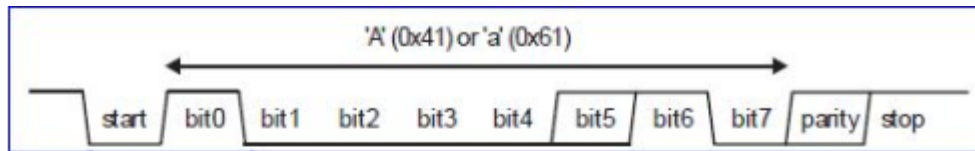
# Part 1: UART in `main()`

In this part, you will design your software based UART in your main function and transmit a string to your computer. You will use the HW timer delay you build in one of the previous labs.

Note that since you took over the `P0.2` pin in Part 0, you can control this pin using the GPIO peripheral. UART idles high on startup, so you should design such that the pin is left with logic high when idle. So far, your skeleton of the code should resemble something like the following code snippet below. The one thing you lose is that because we have taken over `P0.2` (UART peripheral is no longer controlling this pin), we can no longer `printf()` or output any data to the serial console anymore, but this is only temporary until the next Part 2.

At the end of this Part, you should be able to convince yourself that you are able to fully emulate the UART using the GPIO. Instead of peripheral doing the work for you, you are using your CPU to do the work, so of course things are not as efficient as they can be.

What we will try to do is send out data by directly controlling the `P0.2`. You may have built your HW timer function which has accuracy of maybe a few tens, or even a couple of hundred nanoseconds. UART can compensate for about 3% error, so what this means is that if you are sending data at 9600bps, the other receiver may be able to read your data even if it is running 9870bps.

'A' (0x41) or 'a' (0x61)

start | bit0 | bit1 | bit2 | bit3 | bit4 | bit5 | bit6 | bit7 | parity | stop

IDLE
logic 1
3.3v

START

SET

char byte = 'A'
= 0x41 0b0100_0001

STOP

Either
START
bit or
IDLE

CLR

delay(ns)
1 bit period

```c
static const uint32_t pin2 = (1U << 2);
// Make this function static to make it 'private' to this file only
static void disconnect_uart_peripheral(void)
  gpio__construct_with_function(GPIO__PORT_0, 2, GPIO__FUNCTION_0); // P0.2 - Uart-0 Tx
  gpio__construct_as_output(GPIO__PORT_0, 2);
  LPC_GPIO0->SET = pin2;
}
// Make this function non static on purpose
// static
void gpio_uart_send_byte(char byte) {
  // Set LOW for the start bit
  LPC_GPIO0->CLR = pin2; your_hw_delay_ns(baud_rate_delay_ns);

  // TODO:
  // Send 1 bit at a time of the 'byte' with LSB first
  // Use a for loop or similar to send all 8 bits using the P0.2 GPIO
  // Set HIGH for the stop bit
  LPC_GPIO0->SET = pin2; your_hw_delay_ns(baud_rate_delay_ns);
}
int main(void) {
  disconnect_uart_peripheral();
  while (1) {
    gpio_uart_send_byte('G');
    gpio_uart_send_byte('O');
```

```
    delay_ms(1000);

  }

  return 0;}
```

# Part 2: Connect `printf()` to UART

In this part, you will replace the underlining function that uses hardware based UART peripheral with your software based UART function.

Please note the following:

- `printf()` is invoked before the `main()` function starts to run

- If you have any dependency on your HW timer initialization before your `main()`, then you may have to perform a workaround

- The `_write()` function is meant to send multiple bytes according to `const char *ptr, int bytes_to_write`)

The key is to alter `system_calls.c` file and re-direct its output to our UART function. Check the following for reference.

```
// file: system_calls.c
// Declare your GPIO based UART output function here so we can invoke it
// The compiler's linker will "find" this function even without an inclusion of a header file
void gpio_uart_send_byte(char byte);
int _write(int file_descriptor, const char *ptr, int bytes_to_write) {
  if (_isatty(file_descriptor)) {
    // ...
    if (rtos_is_running && transmit_queue_enabled && !is_standard_error) {
      //Instead of calling this function, call your GPIO UART based function
      //system_calls__queued_put(ptr, bytes_to_write); // replace with gpio_uart_send_byte()
    } else {
      //system_calls__polled_put(ptr, bytes_to_write); // replace with gpio_uart_send_byte()
    }
```

```
  } else {
    system_calls__print_and_halt("ERROR: Call to _write() with an unsupported handle");
  }
  return bytes_to_write;}
```

Of course, after altering the file, test it out by now using `printf()` in your `main()` function.

---

# Extra Credit

To go above and beyond, you can use a timer interrupt to latch one bit at a time and minimize your CPU consumption. The psuedo-algorithm is as follows:

```
char byte_to_transmit;
size_t bit_number;
void transmit_byte(char byte) {
  while (timer_is_running) {
    ; // Wait for previous transmission to complete
  }
  bit_number = 0;
  byte_to_transmit = byte;
  // Initialize HW timer
  // 1. Set Match register interrupt for 104uS (assuming 9600bps)
  // 2. Enable timer
}
// System interrupt will invoke this function every 104uS
void timer_match_interrupt(void) {
  switch (bit_number) {
    case 0: CLR = ?; break; // start bit
    case 1 ... 8: GPIO = byte_to_transmit & 0x01; byte_to_transmit >>= 1;
      break;

    case 9: // stop bit
```

```
        GPIO = 1; // SET Register

        // TODO: Disable timer

        break;

    }

    bit_number++;}
```

For this portion of the assignment, you should use `hw_timer.h` which has existing API to enable timer and an interrupt callback.

```
// Reference hw_timer.h
// Reference how sys_timer.c generates match interrupt
// Inside the interrupt, you also have to acknowledge and clear interrupt
/**
 * Enables and starts the timer
 * @param prescalar_divider This divider is applied to the clock source into the timer
 *                          This is offset by 1, so 0 means divide by 1, and 1 means divide by 2
 *
 * @param isr_callback      The ISR callback for the timer, including all Match-Register interrupts
 * @note The isr_callback may be NULL if the timer will not be configured for any match interrupts
 */
void hw_timer__enable(lpc_timer_e timer, const uint32_t prescalar_divider, function__void_f isr_callba
```

# EEPROM Driver

In this assignment, we will build up a driver to write the EEPROM on the NXP processor.

> " EEPROM is a non-volatile memory mainly used for storing relatively small amounts of data, for example for application settings. The EEPROM is indirectly accessed through address and data registers, so the CPU cannot execute code from EEPROM memory

## Part 0: EEPROM Chapter

### 37.2 Description

EEPROM is a non-volatile memory mainly used for storing relatively small amounts of data, for example for application settings. The EEPROM is indirectly accessed through address and data registers, so the CPU cannot execute code from EEPROM memory.
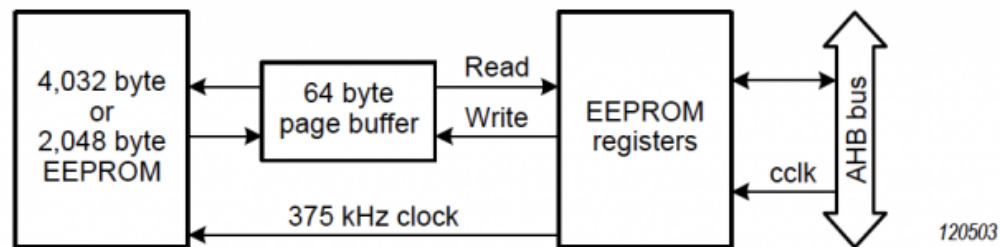


**Fig 169. EEPROM block diagram**

## Part 1: Driver Skeleton

To simplify the driver interface, we will deal with "pages" and read and write the entire page at a time rather than reading or writing singular bytes.

```
#include "lpc40xx.h"
typedef struct {
  uint8_t bytes[64];
} eeprom_page_s;
void eeprom__initialize(void) {
  // Configure PWRDWN, CLKDIV, WSTATE registers as per datasheet instructions
}
uint8_t eeprom__read_page(eeprom_page_s *page, uint8_t page_number) {
    //1) Read data from RDATA register
    //2) Wait for read to finish
}
uint8_t eeprom__write_page(const eeprom_page_s *page, uint8_t page_number) {
    //1) Write byte_data to WDATA register
    //2) Issue another CMD to erase/program and wait for process to finish
}
int main(void)
{
  // For 4K EEPROM - 64 pages limit and each page can hold 64 bytes of data
  // TODO: Setup your test harness, such as:
  // Write a page of known data
  // Read a page
  // Compare the pages together
  while(1)
  {
  }

  return 0;}
```

## Part 2: Implementation with page data read/write

Reuse the above code and implement writing/reading stream of data with page offset

```
void eeprom__read_data(uint32_t page_num, uint32_t page_offset, void *buff, size_t bytes_to_read) {
    LPC_EEPROM->ADDR = (page_num << 6) | page_offset;
```

```c
  //Configure READ operation

  uint32_t index = 0;
  for(....) {
    //copy data to buffer
  }
}
void eeprom__write_data(uint32_t page_num, uint32_t page_offset, void *buff, size_t bytes_to_write) {
  //a) Set ADDR with page address. Refer API above

  //b) Configure WRITE operation

  uint32_t index = 0;
  for(....) {
    //copy buffer to WDATA register
  }
}
int main(void) {
  //For 4K EEPROM - 64 pages limit and each page can hold 64 bytes of data

  //1) Create a char buffer with your name
  const char name[] = "_____"; //fill in
  uint32_t name_len = _____; //fill in

  //2) Initialize EEPROM

  //3) Copy name buffer to EEPROM page 5 offset 10 and program/flush data to EEPROM

  //4) Verify the data on page 5 by reading into a buffer and print

  while(1)
  {
  }

  return 0;}
```

## Part 3: Extra Credit

To earn bonus points, enhance the READ/WRITE APIs that will:

- Use Interrupt status during read/write/Program/Erase operation
- Make reads/writes to take in 8-bit/16-bit/32-bit configuration as argument

## Requirements

- Source code and terminal output screenshots submitted to Canvas