µP Tutorial 2

Introduction to HW/SW

F4 discovery board, peripheral drivers, C programming and documenting

BY BEN NAHILL (ORIGINAL AUTHOR), ASHRAF SUYYAGH

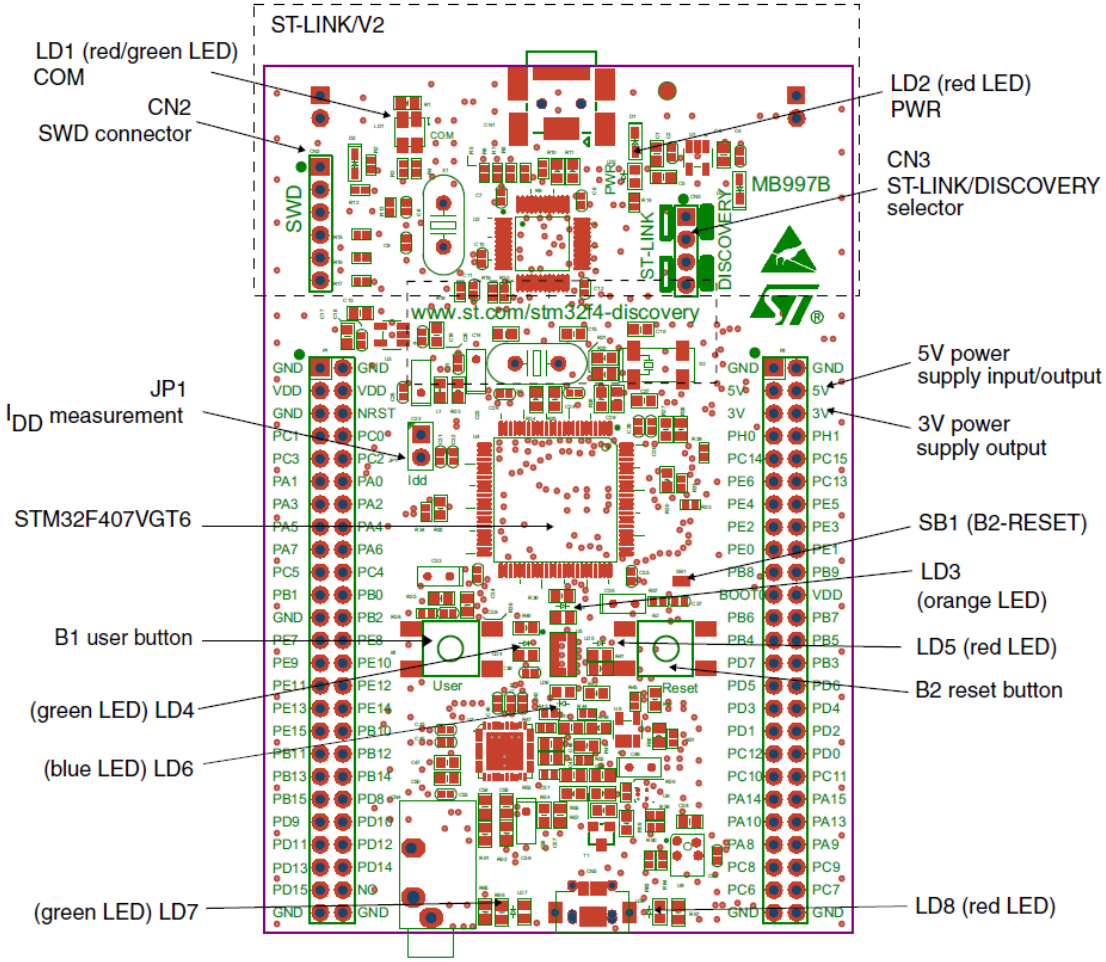
VER 1.3 FALL 2013 (MINOR REVISION – ASHRAF SUYYAGH)

VER 1.0 WINTER 2013 (BEN NAHILL)

STM32F4-Discovery Board

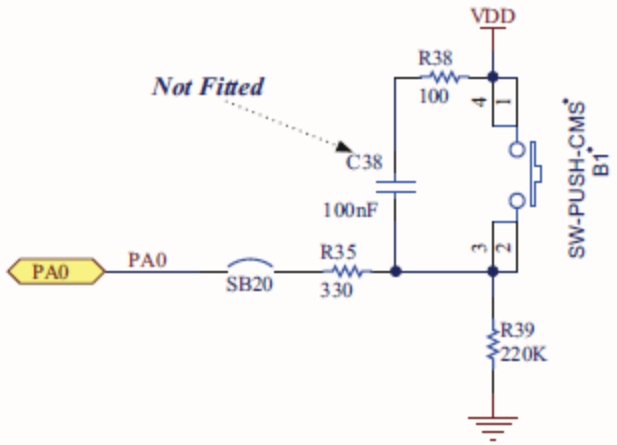
An integrated STM32F4 development platform

- Embedded ST_LINK/V2 programmer with Integrated (Serial Wire Debug) SWD debugger
  - Lightweight alternative to JTAG (2 vs 5 lines)
  - Just for ARM stuff
- To determine to which pin of the STM32 processor is each of these components connected, you need to refer to board schematics (pp. 31-36 of STM32F4-DISCOVERY user manual). Posters provided in lab.
- Sample schematic next page

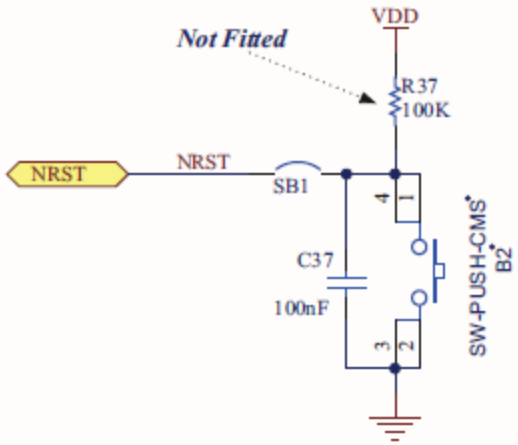


STM32F4-Discovery Board

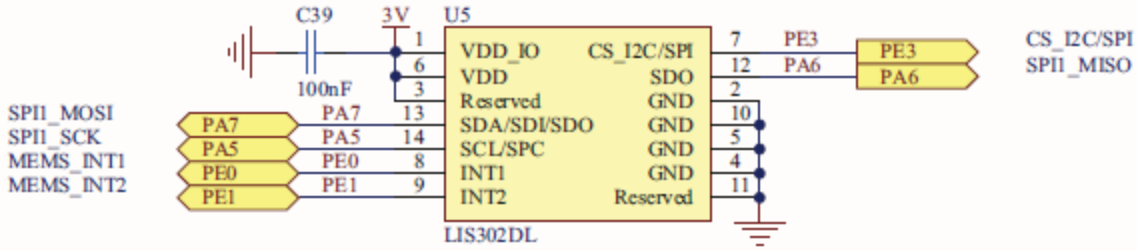
Sample schematic of LEDs and buttons



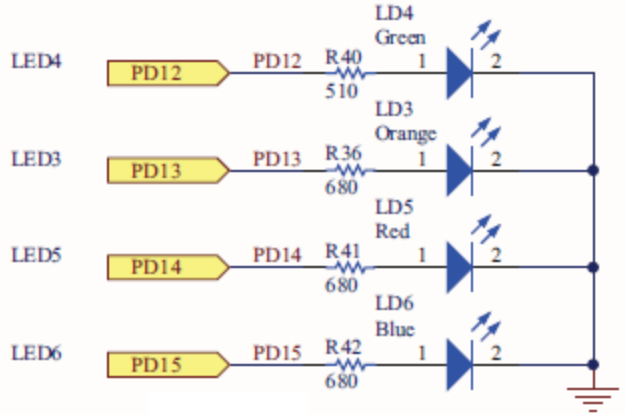
USER & WAKE-UP Button



RESET Button



MEMS



LEDs

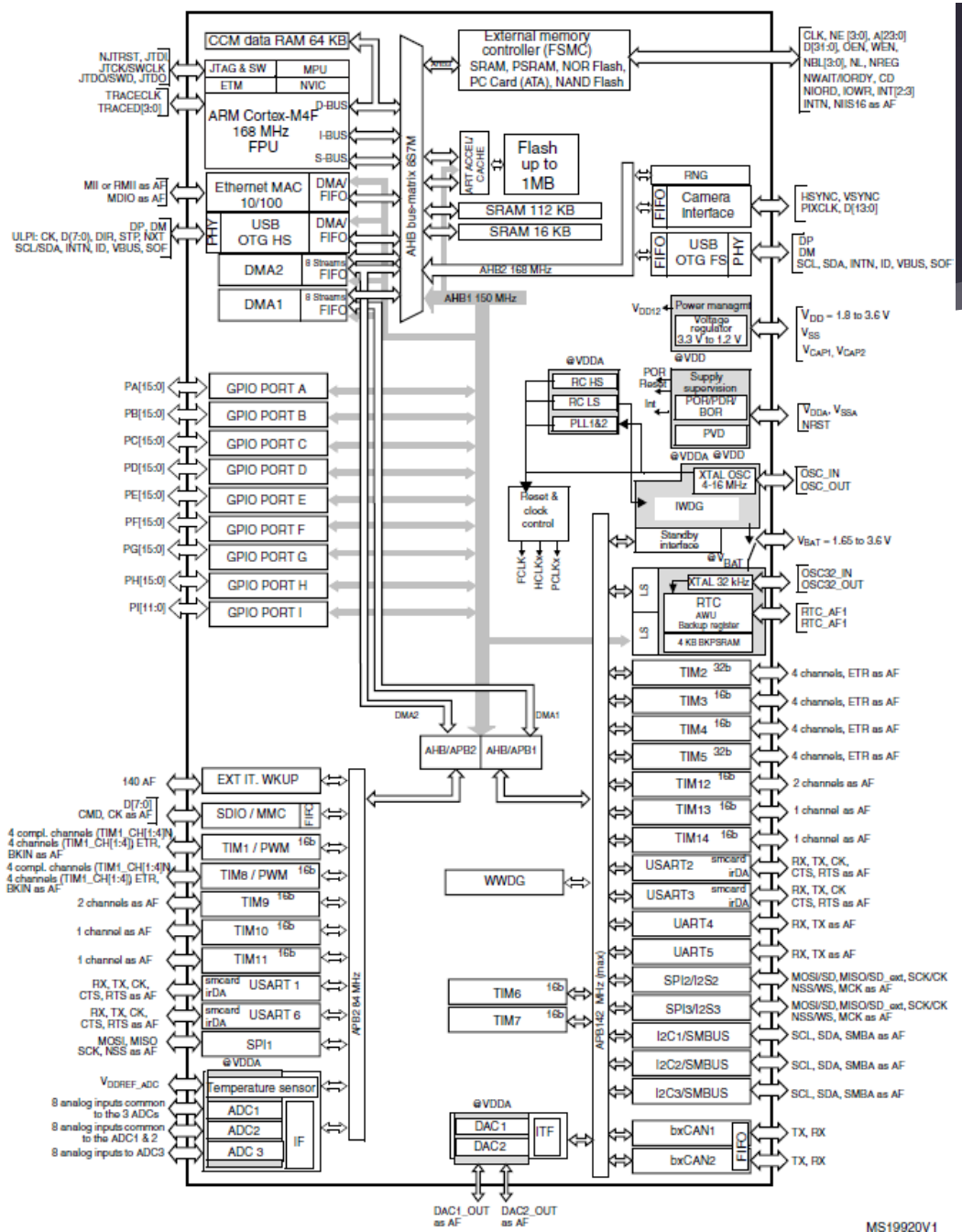
STM32F4 processor chipset

ARM's licensed IP core:

- ARM Cortex-M4F + FPU
- NVIC
- Debugging (JTAG and SW)
- Buses (AHB, APB) and others

Vendor added peripherals:

- Memories (SRAM. FLASH)
- Peripherals, GPIO, Timers, ADC, SPI, I2C ..etc



Documentation

- STM32F4-Discovery User Guide
  - Schematics and port maps for board
- STM32F4 Family Reference Manual
  - Peripheral documentation
- STM32F40x Datasheet
  - Interrupt vector mapping
  - Pin mapping (some overlap with Discovery doc)
- STM32F4-Discovery Library
  - Example applications for your exact hardware
  - Accelerometer, audio DAC drivers
- STM32F4 Peripheral Library
  - Lots of examples

Setting up projects for STM32F4 Discovery board

- Introduction to real hardware!
  - Differences in setting up a project
  - debugging... (driver files, c and header)
- Basic hardware configuration
  1. Setting up the clock to the peripheral
  2. Declaring an isntant of the declartion struct and filling it with parameteres
  3. Passing this configuration struct to the enabling subroutine
- This is generally true, some peripherals might needs extra steps, all in the driver documentation
- Example: use of GPIO pins, connecting the pieces toghther

STM32 Peripherals / RCC

- Uses clock gating for low power
  - Must first enable power to a peripheral before use

```
void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState);
    void RCC_AHB2PeriphClockCmd(uint32_t RCC_AHB2Periph, FunctionalState NewState);
    void RCC_AHB3PeriphClockCmd(uint32_t RCC_AHB3Periph, FunctionalState NewState);
    void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState NewState);
    void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);


// ex: Enable the clock for the GPIOA port
// This will allow use of the GPIO pins in bank A (PA0, PA1, ....)
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
```

- Check stm32f4xx_rcc.h for more peripheral names

GPIO w/ ST Peripheral Library

```
typedef struct{
      uint32_t GPIO_Pin;            // Pin mask -- these are the pins to configure
      GPIOMode_TypeDef GPIO_Mode;   // The GPIO mode (in, out, alternate, analog)
      GPIOSpeed_TypeDef GPIO_Speed; // The maximum slew rate of the pin
      GPIOOType_TypeDef GPIO_OType; // Push-pull, open-drain
  GPIOPuPd_TypeDef GPIO_PuPd;       // Normal, weak pull-up/down
    }GPIO_InitTypeDef;
// Ex:
GPIO_InitTypeDef gpio_init_s;

GPIO_StructInit(&gpio_init_s);
gpio_init_s.GPIO_Pin   = GPIO_Pin_4;        // Select pin 4
gpio_init_s.GPIO_Mode  = GPIO_Mode_OUT;     // Set as output
gpio_init_s.GPIO_Speed = GPIO_Speed_100MHz; // Don't limit slew rate
gpio_init_s.GPIO_OType = GPIO_OType_PP;     // Push-pull
gpio_init_s.GPIO_PuPd  = GPIO_PuPd_NOPULL   // Not input, don't pull

// Actually configure that pin
GPIO_Init(GPIOA, &gpio_init_s);

GPIO_SetBits   (GPIOA, GPIO_Pin_4);
GPIO_ResetBits (GPIOA, GPIO_Pin_4);
GPIO_WriteBit  (GPIOA, GPIO_Pin_4, Bit_SET); (or Bit_RESET )
GPIO_Write     (GPIOA, 0x16);
```

What happens on a low level?

```
// Write a 1
GPIOA->ODR |= 1 << 4;
GPIOA->BSRRL = 1 << 4;

// Write a 0
GPIOA->ODR &= ~(1 << 4);
GPIOA->BSRRH = 1 << 4;

// Read at GPIOA->IDR & (1 << 4)
```

What does that even mean?

SysTick Timer

- Basic interval timer for providing consistent timebase
- Sets an interrupt when the timer expires
  - Vector table associates interrupt vectors with a software function handler
    - Vector table is declared in startup file
  - With ST base project, function is SysTick_Handler()
- Interrupts execute with high priority and their run time ***must*** be minimized
  - Otherwise system becomes less responsive for other tasks and events

SysTick Usage Example -- Asynchronous Handler

```
static volatile uint_fast16_t ticks;

void main(){
    ticks = 0;
    // Configure for 10ms period
    // NOTE: argument here must be less than 0xFFFFFF; //(24 bit timer)
    // At 168MHz, this just a bit slower than 100Hz
         SysTick_Config(10 * SystemCoreClock / 1000);  //Number of ticks between two interrupts
                                                       // or 1/Freq * SystemCoreClock


         while(1){
          // Wait for an interrupt
          while(!ticks);


          // Decrement ticks
          ticks = 0;

          // Do something!
         }
}

 //Interrupt handler for system tick
 //This should happen every 10ms
void SysTick_Handler(){
    ticks = 1;
}
```

ADC Configuration

- STM32 has 12-bit SAR ADC
  - 16 external channels
  - 3 internal to VBat, temperature, and Vrefint
- ADC quantizes voltage referenced between GND and VDD
  - 0 is GND, 0xFFF is VDD
  - Actual voltage is dependent on VDD

ADC Example

```
ADC_InitTypeDef       adc_init_s;
ADC_CommonInitTypeDef adc_common_init_s;

RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
adc_common_init_s.ADC_Mode = ADC_Mode_Independent;
adc_common_init_s.ADC_Prescaler = ADC_Prescaler_Div2;
adc_common_init_s.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;
adc_common_init_s.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles;
ADC_CommonInit(&adc_common_init_s);


adc_init_s.ADC_Resolution          = ADC_Resolution_12b;
adc_init_s.ADC_ScanConvMode        = DISABLE;
adc_init_s.ADC_ContinuousConvMode  = DISABLE;
adc_init_s.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
adc_init_s.ADC_DataAlign = ADC_DataAlign_Right;
adc_init_s.ADC_NbrOfConversion = 1;
ADC_Init(ADC1, &adc_init_s);

ADC_Cmd(ADC1, ENABLE);
ADC_RegularChannelConfig(ADC1, ADC_Channel_12, 1, ADC_SampleTime_480Cycles);

ADC_SoftwareStartConv(ADC1);
while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET); //Could be through interrupts (Later)
ADC_ClearFlagStatus(ADC1, ADC_FLAG_EOC);
ADC_GetConversionValue(ADC1); // Result available in ADC1->DR
```

The C Programming Language

- General purpose procedural language
  - Everything from <1kB micros to huge desktop applications
- Allows relatively low-level machine interaction
  - Second only to assembly

Why *You* Should *MASTER* C

- It is *the* language of embedded programming (currently)
  - Most of you will be asked to write C professionally at some point
- Small microcontrollers are ridiculously cheap and are replacing basic digital logic everywhere
  - These require people programming them and doing so safely and efficiently
- Several more powerful languages are based heavily on it: see C++, C#, Objective-C

Flow Control in C

```
uint_fast8_t i;
for(i = 0; i < 25; i++){
          // Do something 25 times, incrementing the variable 'i' each time
}

if(i == 25){
          // Do something if i == 25
          // This would occur only if the for loop executed completely
} else if(i == 24) {
          // Do something else
} else {
          // Something CRAZY
}

while(i){
          // Now just run i back down to 0
          // Conditioning on an integer check for equality with 0
          i -= 1; // or i--
}

do {
          // Do this at least once, but guarantee that i <= 25 afterwards
} while(++i < 25)
```

Types -- Integer

```
// Platform dependent length, usually signed
int i;
// An 8-bit also usually-signed value
char c;

//////
// Generally for embedded applications, DON'T USE THOSE AS NUMBERS
// -- char is still fine for text
//////

// Include a set of better-defined integers
#include "stdint.h"

// Unsigned 32-bit integer
uint32_t u;
// Signed 32-bit integer
int32_t s;

// Literal options for these are numeric types in decimal or hex and also ASCII
// characters in single quotes 'a', 'b', or escaped values like newline '\n'
```

Types -- Floating point

```
// 32-bit IEEE754 single-precision float
float f = 0.0;

// 64-bit IEEE754 double-precision float
double d = 0.0;
```

Don't usually try comparing for equality

- Rounding errors will likely get you
- Compare for a range instead

```
// Don't:
if(f == 2.4)
    ...
// Do:
if((f < 2.41) && (f > 2.39))
    ...
```

Types -- Enumerated

- Enumerated types hold a fixed number of values
  - Natural application is state machines
  - Internally, they are integers and can be used as such

```
enum {
    STATE_OFF = 0, // Assigned values are optional
    STATE_ON = 1
} current_state;


switch(current_state){
case STATE_OFF:
    // do something
    break;
case STATE_ON:
    // Something else
    break;
default:
    break;
}
```

Types -- Aggregate

```
typedef struct {
    uint8_t age;
    char name[32];
} person_t;


// Old-style initialization
person_t ben = {65, "Ben"};


// C99-style initialization
person_t ben = {.name = "Ben", .age = 65};


// Bitfields!
typedef struct {
    uint32_t packet_length  : 5; // Original type must be larger than field
    uint32_t packet_id      : 16;
    enum {
      TYPE_DATA = 0,
      TYPE_SYNC = 1
          } packet_type           : 3;
          uint32_t little_data    : 8;
} header_t;
// The total size of the above is only 32 bits
```

Pointers

```
typedef struct {
    uint8_t age;
    char name[32];
} person_t;


person_t ben;


person_t * ben_ptr = &ben; // Pointer to ben
ben.age = 65; // Assignment to structure value
ben_ptr->age = 65; // Assignment to structure value through a pointer


void print_person(person_t * person){
    printf("%s is %d years old\n", person->name, person->age);
    // Could also modify person as needed here and caller would see that
}


// Call a function with a pointer argument
print_person(&ben);


// Or equivalently
print_person(ben_ptr);
```

Null Pointers

A pointer to the address 0 is called a *null pointer*

- This usually is to indicate that there is nothing there
- Reading from or writing to there will result in an error
  - Segfault on PC, HardFault on ARM

Linked lists use null pointers to mark end

Arrays

```
char name[32]; // 32 char values in memory, presumably 0-terminated
               // AKA a string...
// The symbol name is now of type (char []) pointing at the first element
// This is the same as (char * const) but with the bonus of having a known length


// Function that takes an array argument
void cut_string(char * str, char cut_at){
    char * iter = str;
    while(*iter != 0){
     if(*iter == cut_at){
          *iter = 0;
          break;
     }
     iter += 1; // Or iter++
         }
}


// To assign multiple values, must be done at initialization
char last_name[] = "Nahill"; // Length doesn't need to be provided in this case


// Otherwise assign one element at a time
last_name[0] = 'M';
```

Pointer Arithmetic

```
// Pointer of type (some_type *) is actually an integer
// Adding 1 to it will increase the internal value by the size of some_type


person_t this_class[29];


person_t * iter;
// Iterate until the first invalid element
for(iter = this_class; iter != &this_class[29]; iter++){
    // Sabotage
    iter->age *= 2;
}


// This can be faster than indexing by an integer if you don't actually need the
// integer value at any point
```

Casting -- Numeric types

```
int32_t some_int;
float some_float;


// Lets say we want to convert a float between -1 and 1 to the full range of a
// 32-bit integer


// INT32_MAX provided from stdint.h
some_int = (int32_t)(some_float * INT32_MAX);


// And reversing the result...
some_float = ((float)some_int) * (1 / INT32_MAX);
// Always avoid division where possible...
```

In this case, a conversion is performed in the casting. For other casting cases, this doesn't happen.

Type Definitions

```
// Introduce "struct person" as a type
struct person {
    uint8_t age;
};


// Give it a nicer alias, "person_t"
typedef struct person person_t;


// Create a person
struct person person1;
// Create another one
person_t person2;


// For linked list:
typedef struct person_ll {
    uint8_t age;
    struct person_ll * next;
} person_ll_t;


// person_t isn't available at the time that it is needed so use intermediate
// struct type
```

Arithmetic Operators

```
uint32_t a, b, c;
a = b + c;    a = b - c;    a = b * c;    a = b / c;

// Operations round down. If you want real rounding, cast to float and add 0.5

a++; // Evaluate a, then increment it (post-increment)
++a; // Increment a, then evaluate it (pre-increment)

a += 1;
a *= 10;
a /= 25;

a >>= 2; // Shifts will sign-extend if necessary
a <<= 1;

// Comparison:
a > b;
a >= b;
```

Bitwise Operators

```
uint32_t a, b, c;
//   AND            OR             XOR        NOT
a = b & c;     a = b | c;     a = b ^ c;     a = ~b;


// Clear all but 2 LSbs
a &= 0x03;


// Set MSb
b |= 0x80000000;


// Clear MSb
b &= ~0x80000000;
```

Logical Operators

```
// AND
if(a && b) // Both a and b must evaluate to true (non-zero usually)

if(a || b) // Either a or b evaluates to true

if(!a || b) // Either (not a) or b
```

C Preprocessor

Preprocessor does a simple find/replace before compilation happens

```
// Parentheses not necessary but good practice in a lot of cases
#define PI (3.14)
// Type checks are done only after substitution
#define MULT_BY_PI(x) (PI*x)
#define MULT_BY_PI_ON_SEVERAL_LINES(x) \
    (PI*x)
#define WE_SHOULD_MAKE_B 1


#ifdef WE_SHOULD_MAKE_B
#if WE_SHOULD_MAKE_B
float b = MULT_BY_PI(2);
#endif
#endif


// This means "replace this line with the contents of accelerometer.h"
#include "accelerometer.h"
```

Variable Allocation -- Stack

Memory through levels of function calls uses a stack:

```
void one_function(){
    uint32_t one_var;
    ...
}


void another_function(){
    uint32_t another_var;
    one_function();
}


void yet_another_function(){
    uint32_t yet_another_var;
    another_function();
}
```

```
STACK:
<-Stack pointer before first call
[yet_another_var]
[other info preserved from yet_another...]
[another_var]
[other info preserved from another_function]
[one_var (could also just be in a reg)]
<-Current stack pointer
```

A consequence of this is that a function can be called many times inside itself new *uninitialized* variables will be allocated as needed.

Variable Allocation -- Static

If you need persistent information for a function across calls, declare the variable *static*.

```
uint32_t count(){
    static uint32_t counter = 0;
    return counter++;
}
```

The initialization will be done at startup and never again, but that variable won't be visible outside the function.

Variable Allocation -- Dynamic

Use a heap to allocate memory at run time

```
// Try to allocate the space for a person
person_t * person = (person_t *)malloc(sizeof(person));


// If unsuccessful, the result will be null
if(person){
    // Then you have to free it
         free(person);
}


// C doesn't have garbage collection so you have to free stuff
```

Don't do this unless you have to! There is no reason for anyone to have to do this in this class!

Other Important Keywords -- extern

Extern indicates that the compiler should assume that this variable exists and will be found later by the linker.

```
// In accelerometer.c

acc_t some_accelerometer = {

        ....

};


// In accelerometer.h

extern acc_t some_accelerometer;
```

Now anyone including accelerometer.h knows about *some_accelerometer* and can use it.

Other Important Keywords -- const

const indicates that an item is constant

- This allows for optimization
- This enforces safety and documentation
- This confuses people

const refers to the type *to its left* unless there is nothing to its left, else right

```
const int i; // A constant integer
int const * i; // A pointer to a constant integer
int * const i; // A constant pointer to an integer
int const * const i; // A constant pointer to a constant integer
```

More on const

If your function doesn't intend to modify its pointer arguments, make them const:

```
// Example from standard library
char *strcpy(char *dest, const char *src);


// The contents of the string src aren't being modified so let the compiler know
// that.
```

Other Important Keywords -- volatile

- Opposite of const
  - Assume this might change outside the normal program flow and always re-read it

Example: A memory-mapped GPIO port

```
typedef struct {
    uint32_t ODR;
    uint32_t IDR;
    ....
} GPIO_TypeDef;


GPIO_TypeDef volatile * const GPIOA = (GPIO_TypeDef volatile *)0x4000010;
// (or whatever the address is)


// This is a constant pointer now to a structure that is assumed to be able to
// change outside of the normal program flow


// Unfortunately stupid ST uses #define instead to declare these things
```

Style Guide

- Style and consistency are important
  - Code must be readable (to others too)
  - Variables and functions should have obvious names
  - Modular design *will* make your life easier
- Many details and areas of personal preference
  - Module breakdown
  - Function and variable name schemes
  - Indentation and brackets
  - Documentation style

Modularity

Generally:

- One C file per 'module'
- Module is a self contained unit with its own functions, datatypes, and/or variables with a well-defined API.
- Examples: spi.c, uart.c, lis302dl.c, framebuffer.c

Each module has private functionality and public functions

- ONLY THE PUBLIC ONES GO IN THE HEADER FILE

Modularity (example)

```
// lis302dl.h
// Multiple-include guard
#ifndef __LIS302DL_H_
#define __LIS302DL_H_


// Comments
typedef struct {
    SPI_TypeDef * const spi;
    reading_t current_reading;
          GPIO_TypeDef * const nss_gpio;
          uint32_t       const nss_mask;
} lis302dl_t;


// Declare that we are going to have an instance of this accelerometer
extern lis302dl_t acc1;


/*!
 @brief Initialize an LIS302DL accelerometer
 @param, @return ..
 */
uint_t lis302dl_init(lis302dl_t * acc);


// More comments
uint_t lis302dl_read(lis302dl_t * acc, reading_t * reading);


#endif // __LIS302DL_H_
```

Modularity (example cont'd)

```
// lis302dl.c
#include "lis302dl.h"


// Declare private functions


/*!
 @brief Read from the LIS302DL using SPI bus
 @param ....
 @return
 */
static uint_t lis302dl_spi_read(lis302dl_t * acc, uint8_t addr,
                                uint8_t * buff, uint8_t num_bytes);


lis302dl_t acc1 = {...};


uint_t lis302dl_init(lis302dl_t * acc){...}


uint_t lis302dl_read(lis302dl_t * acc, reading_t * reading){...}


static uint_t lis302dl_spi_read(lis302dl_t * acc, uint8_t addr,
                                uint8_t * buff, uint8_t num_bytes){...}
```

Modularity (main.c)

- Main.c is generally the entry point for your program
  - It has no public interface (eg. no main.h)
  - Logically, including main.h doesn't make sense
- Sometimes you may need global parameters
  - Don't belong logically in any one module
    - Global flags and constants
  - I suggest "common.h", or "yourapp.h"

Indentation

- MANY ways to deal with this
  - Big argument -- tabs vs spaces
    - Tabs aren't same size on all machines
    - Spaces are a pain to deal with
- If working on a project already well established:
  - Follow their convention. Don't mess it up.
- If starting a new projects:
  - Tabs for indentation, spaces for alignment

```
while(1){
TAB|if(some_condition | some_other_condition |
TAB|   another_condition | more_conditions){  // Align the beginning of this
TAB|TAB|//Some stuff that may happen conditionally
TAB|TAB|
TAB|}
}
```

Documentation Style

- Whatever your style, you *must* document
  - We read your code and want to know what happened
  - If we have to go through each line to figure out what's going on, you will be penalized
- I recommend Doxygen
  - Structured, human readable
  - Also machine readable for automatically generated documentation
    - Bosses really like this part => valuable skill
    - Don't worry about generating documents for us

Doxygen

```
/*!
 @file spi.h
 @brief This is the public interface for a SPI driver
 */


// This part is for the automatic documentation generator
//! @addtogroup SPI
//! @{


/*!
 @brief Initialize a SPI driver
 @param spi The SPI driver
 @return 0 if successful
 @pre GPIO clocks must be enabled
 @post SPI will be setup


 This is some longer documentation string about what this function does
 */
uint_fast8_t spi_init(spi_t * spi);


//! This is a driver for SPI1
extern spi_t spi1;


//! @} // SPI
```

Documentation

- Documentation goes with the prototype
  - ***All*** public interface documentation in header
  - That documentation is for the person looking to call it
    - Doesn't want to look at the inner workings
  - Static (private) function documentation goes with the prototype at the top of the C file
- Break declarations up into sections
  - Makes it easy to find what you want in your file
  - Example ->

Documentation (Example)

```
/*!
 @file lis302dl.c
 Some header documentation
*/


// Includes


// Private defines


// Private type definitions


// Public variables


// Private variables


// Private function prototypes


// Function bodies
```

Libraries for STM32F4

ST and ARM both offer libraries to ease your development process

- ARM CMSIS
  - Collection of support for Cortex-M series processors
  - Includes CMSIS DSP library
- ST STM32F4 Peripheral Library
  - Simple abstraction for peripherals on STM32F4
  - Not great as abstraction since you still need to know the peripherals well
  - Real value is in examples