Intro to C Programming

ECSE 426 - Winter 2013

Ben Nahill

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

No built-in dynamic memory management

So Why C?

- Universal
  - Tools exist for nearly every platform
- Control
  - Code size vs. performance
  - Variable and function locations in memory
- Performance
  - *Can* be very high

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, Go

Embedded C

Not really a different language

- There is a standard for it, but nobody cares

Unofficial extensions of C

- Extensions for better memory control
- Support for function calls that don't require preservation
- More specific datatypes
  - Fractional

Hello World

```c
// Include components from other files
#include "stdio.h"

// Declare "global" variables
int a_number;

// This will be the entry point of your program
int main(void){
     // Assign a value
     a_number = 15;
     // Print with a pointless substitution
     printf("Hello World - %d\n", a_number);
     // Return
     return 0;
}
```

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;

// Fastest unsigned integer of at least 32 bits
uint_fast32_t f;

// 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.

Casting -- Pointer types

```
// Type safety hazard! Continue at your own risk!
// Otherwise extraordinarily useful for effective generic programming

typedef struct {
     void (*callback)(void *);
     void * pointer_arg;
} action_t;

void a_callback(void * device){
     // Pretend that void pointer is a different kind of pointer (but of same
size)
     ((SPI_TypeDef *)device)->DR = 0;
}

action_t spi_action = {
     .callback = a_callback,
     .pointer_arg = SPI1
};

void run_callback(action_t * action){
     action->callback(action->pointer_arg);
}
```

Unions

```
// An alternative to casting among a small set of types
// Define a set of types sharing the same space in memory

union {
     uint32_t a_big_int;
     uint8_t smaller_numbers[4];
} more_flexible_int;

more_flexible_int.a_big_int = 5000;

// If we need to send this byte-by-byte
int i;
for(i = 0; i < 4; i++){
     transmit(more_flexible_int.smaller_numbers[i]);
}

// Anonymous unions -- If no name is given, allow the internal name to be
identifier
union { uint32_t big, uint8_t smaller[4] };
big = 25;
smaller[0] = 1;
```

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"
```

C Compilation Process

```
src/            (Your original source files
  main.c
  accelerometer.c
  startup.s
build/          ("object files", Keil might suffix these ".obj")
  main.o
  accelerometer.o
  startup.o
  final_exectuable.axf (or .elf, .hex, .coff)
```

Each object built independent of all other objects

All shared information must be in header files, including types, function prototypes, constants, and global variables.

C Compilation Process - Step 1: The Preprocessor

This occurs on a per-C-file basis

- All of the find/replace operations occur
- Additional 'defines' may be provided by command line
- Basic syntax errors that prevent parsing will cause immediate failure

C Compilation Process - Step 2: Object Compilation

Compile each C file to assembly (still on a per-C-file basis) then assemble it

- Optimization will be performed here if requested
- Global symbols (functions/variables) will be given a space in the file's global symbol table
- Local symbols will be in a separate table

C Compilation Process - Step 2b: Static Libraries

If creating a static library (.a or .lib), the process stops here

- That .a or .lib file is just an archive of object files
- A header file indicates the public interface to symbols defined therein

Static libraries can be good for isolating functional units, possibly with different compilation options (see CMSIS-DSP)

C Compilation Process - Step 3: Linking

Assign symbols to locations in memories, replace symbols with correct locations in code

- Only at this stage will your files have any real interaction
- Symbols that aren't actually defined will now throw errors
- You may also get errors about certain variables or functions not having a place

C Compilation Process - Symbols?

In a symbol table, each symbol is listed by name along with requirements:

- Size
- Specific place in memory if required
- More generic place otherwise
  - Executable? Constant?
- Does it require runtime initialization?

Global symbols must be unique

Local symbol names never leave the file

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!

The Other 'Static'

The *static* keyword at the top-level means simply that the variable or function is local to the file

```
static void some_internal_function(void){

}
```

This is useful for small helper functions to enforce that they are only used locally.

In more complex applications, it keeps from cluttering the final symbol table.

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
```

<u>Wherever possible, use this instead of macros!</u>

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
```

Other Important Keywords -- inline

This is a hint to the compiler that it should inline a function where possible

- Like a macro function but with some type safety
    - Type safety is a good thing for making sure you don't do anything stupid so use it
        - *Exception:* when polymorphism is required

OO Design

Can create "classes" and "methods"

```
// In person.h
typedef struct {
      uint8_t age;
      char name[32];
} person_t;

// In person.c
#define MAX_AGE 115
void person_init(person_t *person, uint8_t age, char *name){
      person->age = age;
      strcpy(person->name, name);
}

// Some "private" method, called only from "person.c"
static void person_set_age(person_t *person, uint8_t age){
      person->age = age;
}

// Static method
uint8_t person_get_max_age(){
      return MAX_AGE;
}
```

It's just not quite as pretty as in true OO languages.

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_fast8_t lis302dl_init(lis302dl_t * acc);

// More comments
uint_fast8_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_fast8_t lis302dl_spi_read(lis302dl_t * acc, uint8_t addr,
                                      uint8_t * buff, uint8_t num_bytes);

lis302dl_t acc1 = {...};

uint_fast8_t lis302dl_init(lis302dl_t * acc){...}

uint_fast8_t lis302dl_read(lis302dl_t * acc, reading_t * reading){...}

static uint_fast8_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"

Modularity (Naming Conventions)

- C has only one namespace:
  - Within a module, should prefix all symbols with a module identifier
  - Enum labels are visible everywhere that includes them so they *MUST* have unique names
    - enum {LIS302_ST_ON, LIS302_ST_OFF}
    - Or keep them contained in the C file only
- Common naming scheme makes it easy to guess a function name in absence of a nice autocomplete IDE

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

CMSIS Core

- Helper functions to use architecture-specific and SIMD instructions
  - __disable_irq(), __enable_irq()
- Well documented using Doxygen
- SysTick
  - A very simple periodic timer intended as a scheduler tick
- Cross-toolchain support
  - Inline assembly isn't exactly a standard...

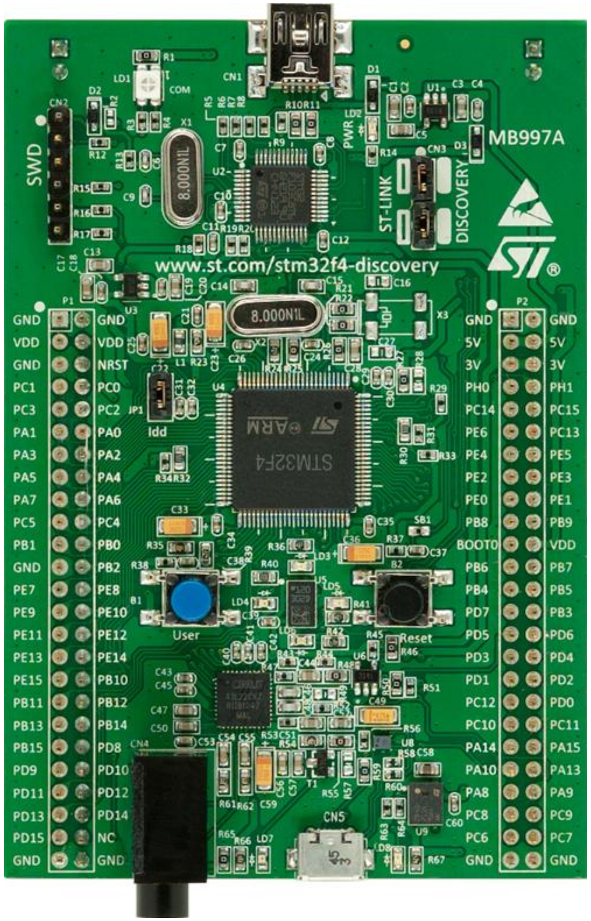
CMSIS DSP

- A powerful DSP library supporting many different types for many operations
- Highly optimized for each processor
- [Link]

STM32F4-Discovery

An integrated STM32F4 development platform

- Integrated SWD debugger
  - Lightweight alternative to JTAG
  - Just for ARM stuff
- Pushbutton, accelerometer, audio DAC, USB OTG, 4 LEDs



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

Lab 2

- Introduction to real hardware!
  - & debugging...
- Basic hardware configuration
- Use of the ADC to sample a voltage
- Use of timer to provide regular sampling interval
  - Basic interrupt handling
- Use of GPIO pins

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   = 1 << 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);

// 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)
```

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 ***<u>must</u>*** 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;
     //       At 168MHz, this just a bit slower than 10Hz
     SysTick_Config(10 * SystemCoreClock / 1000);

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

          // Decrement ticks
          ticks -= 1;

          // 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);
ADC_ClearFlagStatus(ADC1, ADC_FLAG_EOC);
// Result available in ADC1->DR
```