C FOR THE EAGLE PROJECT

PANTELIS SOPASAKIS AND PANOS PATRINOS

Abstract. This document is not intended as a full tutorial on C. It is rather a hitchhiker's guide to C programming for the EAGLE project including a collection of good programming practices and an exposition of the structure of the project framework. We do not address dynamic memory allocation as it will not be needed in this project.

Contents

1. Introduction to C

If you are a beginner in C programming, we recommend that you follow an online tutorial such as the one at tutorialpoints.com or cprogramming.com.

1.1. Header files.

Header files: this is where development begins. Header files contain all the information that the end-user needs to know about the underlying implementation which is found in a `.c` file. There are several things one finds in a header file

`#include` statements: a header file may depend on other header files. Following the interface segregation principle, unnecessary dependencies should be eliminated.

Guard macros: If a header file is included twice (which is not at all difficult to occur), the compile will give an error. To remedy that, the standard trick is to wrap the header file contents around *guard macros* also known as *controlling macros*. Such headers are called *once-only* headers. This is an example of header file `attitude.h`:

```
#ifndef ATTITUDE_H
#define ATTITUDE_H

/* file content */

#endif
```

(P. Sopasakis and P. Patrinos) KU Leuven, Department of Electrical Engineering (ESAT), STADIUS Center for Dynamical Systems, Signal Processing and Data Analytics & Optimization in Engineering (OPTEC), Kasteelpark Arenberg 10, 3001 Leuven, Belgium. Email addresses: pantelis.sopasakis@kuleuven.be and panos.patrinos@esat.kuleuven.be.

Lecture Notes for the 3rd-year BSc project "EAGLE" at KU Leuven. version 1.0.3. Last updated: September 8, 2017.

Function declarations: these are function prototypes lacking an implementation. They explain how the function should be invoked (number and type on input/return arguments) and they are also accompanied by documentation. This is an example of a function declaration:

```
void controller_flying(
     float thrust,
     float rot_x,
     float rot_y,
     float rot_z);
```

Global variable definitions: these are variables which are accessible by anyone who includes the header file. This is exactly why it is a bad idea to use global variables: not all functions should be allowed to modify their values. To remedy that `static` variables should be used instead (see Section 1.4).

Type definitions such as definitions of structure types:

```
typedef struct {
    float w;
    float x;
    float y;
    float z;
} Quat32;
```

Last, but not least, header files may contain `#define`s and macros (details about preprocessor directives can be found in Section 1.5). Macros are pieces of code which are given a name; similar to the use of `#define` to define constant variables, macros are used to define general statements. These, at compile time, will be inserted *in situ* into the code. Here are a few examples:

```
/*
 * creating an alias for an existing function
 * We wil now be able to call `hsdap8723` using
 * its alias, `update_data`
 */
void hsdap8723();
#define update_data hsdap8723

/*
 * This is a macro we use in the EAGLE project
 * and takes three arguements. Note that a backslash
 * must be used to break between lines.
 */
#define CLAMP_INPLACE(x, lo, hi) { \
     if((x) < ((lo))) \
          (x) = (lo); \
     if((x) > ((hi))) \
          (x) = (hi); \
}
```

1.2. **Anatomy of a function.** Functions need to be structured as follows:

```
int myFunction(int a, int b){
    /* Declaractions  */
    float x;    /* explain what `x` is */
    float y[2]; /* all variables must be documented */

    /* Implementation */

    /* Return result */
    return 0;
}
```

The implementation of a function should be separated from all declarations. The following is a bad practice and should be avoided:

```
void badPractice(int a, int b){
    int sumOfArguments;
    sumOfArguments = a + b; /* implementation */
```

```
    int anotherVariable; /* declaration */
    anotherVariable = sumOfArguments - 3;
}
```

But even the following is bad practice although it raises no warnings (at least in `gcc`):

```
void badPractice(int a, int b){
  int sumOfArguments = a + b; /* declaration and definition */
  int anotherVariable = sumOfArguments - 3; /* declaration and definition */
}
```

The reason is that declarations are not separated from definitions.

Moreover, variables should be initialised in separate lines and each line should be documented explaining what the variable corresponds to:

```
/* GOOD PRACTICE */
float position[3];  /* (x,y,z)-position in m   */
float velocity[3];  /* (x,y,z)-velocity in m/s */
```

```
/* BAD PRACTICE */
float position[3], velocity[3];   /* position and velocity */

/* WORSE PRACTICE (Cryptic variable names) */
float pos[3], vl[3];
```

1.3. **Documentation and comments.** Your source code needs to be well documented. Use comments to make your code readable. This applies to auto-generated code as well. Comments in plain ANSI C are wrapped in `/* ... */` as shown in the previous section too. Do not use `//` as these are not compatible with the ANSI C standard.

Documentation should always precede a function or variable declaration in a header file. In `.c` files there should not be any function or variable documentation — there should be comments, nevertheless.

We propose to use the Doxygen standard to document your C code. Here are a few examples:

```
/**
 * Documentation for this function, explaining how it works.
 *
 * Note the double asterisk in the beginning.
 */
void someFunction();

/**
 * Explain what this function does.
 *
 * @param beta scalar
 * @param y vector y
 * @param n size of y
 *
 * @return returns beta times the norm of vector y.
 *
 */
float functionWithParameters(
    float beta,
    const float * y,
    size_t n);
```

If you need to write some very short documentation for a function or a variable (typically, for variables), you may document your code as follows:

```
static float posCameraMeas[2]; /**< (x,y)-position from image processing */
```

There are two advantages in documenting your functions and variables as above. Firstly, you will able to use `Ctrl+Space` in Vivado to fire up a dialogue box showing the documentation for that function or variable, thus facilitating the development. Secondly, you may use Doxygen to automatically generate HTML code which organises your documentation nicely.

1.4. **Scope and initialisation of variables.** Using global variables is, in general, not advisable and unless it is absolutely necessary (it should not be), should rather be avoided. Instead, `static` file-scope variables should be used. If their value is needed in some other scope, we should implement a function that exports them. Static variables can be initialised in place. However, this is not possible on the Zybo platform. Instead, we have to implement an initialiser. Here is an example — this is the header file `attitude.h`:

```
/* File: attitude.h */
#ifndef ATTITUDE_H
#define ATTITUDE_H

    /**
     * Initialiser
     */
    void initialise();

    /**
     * Returns that value of `alpha` which is stored
     * internally.
     */
    int getAlpha();

#endif /* ATTITUDE_H */
```

and this is `attitude.c` where the static variable is declared:

```
/* File: attitude.c */
#include "attitude.h"

/** some static variable*/
static int msAlpha;

void initialise() {
    msAlpha = 100; /* initialise the value of msAlpha */
}

int getAlpha(){
   return msAlpha;
}
```

When the user uses this functionality, they will not be able to directly access and modify the value of `msAlpha`. However, `msAlpha` is available inside `attitude.c` and can be read and modified by functions therein. Note that if we define `msAlpha` in the header file, this will be accessible to any other scope where `attitude.h` is included.

It is very important to underline once again that the following code where we have the simultaneous *declaration* and *definition* of a global or static variable is not allowed and an initialiser should be used instead:

```
/*
 * NOT ALLOWED ON ZYBO.
 */
static int someVariable = 1;
```

In C, variables and functions have a *scope* — the context in which they can be identified and used (read, written, executed). Variables can have (i) *global* scope when they are defined in a header file — then any file that includes that header file can access the variable, (ii) *function* scope, that is, it is declared and can be used within a particular function only, (iii) *file* scope when it can be accessed by all functions in a `.c` file, (iv) *reduced* scope — not a legitimate term — when the scope of the variable is smaller than that of a function. Let us examine the following example:

```
/* File: attitude.h */
#ifndef ATTITUDE_H
#define ATTITUDE_H

/* This is a global variable */
```

```
float sonarMeasurement;

/* This function will be available to
 * all those functions in files that include
 * attitude.h
 */
void controller_flying (
    float thrust,
    float rot_x,
    float rot_y,
    float rot_z );

#endif
```

```
/* File: attitude.c */
#include "attitude.h"

/* File-scope variable */
static float sonarFilteredValue;

/* File-scope function declaration. It can only be used
 * by other functions in this file.
 */
static void controllerHelper(
    float thrust,
    float rot_x,
    float rot_y,
    float rot_z );

static void controllerHelper(
    float thrust,
    float rot_x,
    float rot_y,
    float rot_z ) {
    /* Implementation */
}
```

Variables may also have a reduced scope, for example, that of a for loop. These are *local variables*, but with a reduced scope compared to that of a function. Here is an example:

```
/* File: example.h */
#include <stddef.h> /* Definition of size_t */
```

```
/* File: example.c */
#include "example.h"

void someFunction() {
    size_t i; /* Declaration of variable i */
    i = 1; /* Definition of i */
    while (i <= 5) {
        size_t j; /* local declaration */
        j=1;
        while (j <= i ) {
            printf("%u ",(unsigned)j);
            j++;
        }
        printf("\n");
        i++;
    }
    /* Variable j is out of scope here */
}
```

This function prints:

```
1
```

```
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

1.5. **Preprocessor directives.** We already presented the preprocessor directive `#include` which is used to define dependencies on other header files and `#define` which we used to define variables and reuseable pieces of code. Furthermore, we saw that `#define`s can be used to define guards for once-only header files. There is another interesting use of `#define`s that allows us to write highly portable code which can be executed both on a PC and on Zybo. This will allow you to write code in such a way that you can compile it and test it on your PCs and use the same exact code on Zybo.

For example, the following code will execute only if the compiler is gcc:

```
#ifdef __GNUC__
  /* This code will be compiled only if the compiler is GCC */
#endif /* __GNUC__ */
```

There are certainly pieces of code that cannot be executed on a PC and these might hinder the testing of the code. As an example, we have the function `PWMOutput`. A possible way to overcome this limitation is to first define a macro variable in your Zybo project which will be visible by all header files (e.g., in `main.h`)

```
#define __EAGLE_ZYBO_PROJECT__
```

Then, wrap functions such as `PWMOutput` with guards:

```
void controller_flying(float thrust, float rot_x, float rot_y, float rot_z) {

#ifdef __EAGLE_ZYBO_PROJECT__
  /*
   * This code will execute ONLY when compiled
   * on Zybo.
   */
  PWMOutput(
    (float)(CTR_OUT_LOW + (CTR_OUT_HIGH - CTR_OUT_LOW)*v[0]),
    (float)(CTR_OUT_LOW + (CTR_OUT_HIGH - CTR_OUT_LOW)*v[1]),
    (float)(CTR_OUT_LOW + (CTR_OUT_HIGH - CTR_OUT_LOW)*v[2]),
    (float)(CTR_OUT_LOW + (CTR_OUT_HIGH - CTR_OUT_LOW)*v[3]) );
#else
  /*
   * Running on PC - Do something else.
   */
#endif /*  __EAGLE_ZYBO_PROJECT__  */
}
```

1.6. **Datatypes.** Here is a list of datatypes and comments on their use:

(1) `int`: signed integer; use *only* if negative values are allowed. Example: status codes where `0` corresponds to success, positive values provide information about the execution status (still implying successful execution) and negative values are error codes.
(2) `size_t`: size type defined in `stddef.h`; this is an unsigned type used to represent the size of objects. Example: Array sizes must always be of type `size_t`. Iterators over the elements of an array must also be of this type.
(3) `float`: single-precision floating-point numbers.
(4) `double`: double-precision floating-point numbers.
(5) pointers: addresses in memory where some value is stored. We will use pointers sparingly in this project, yet they are inevitable, so we recommend that you read a C tutorial.
(6) Custom types. You may define custom types using `typedef`. There are several reasons to do so. First, if you create a custom type called `real_t` which is the same type as `float`, you can build your software based on `float`. If at some point you want to test what happens if you change `float` to `double`, you will have to change a single line of code. The second motivation for using `typedef` is to create aliases for long type names such as `unsigned int` or `unsigned long` (you may call these `uint_t` and `ulong_t`). A third reason to use typedef, is that it is very convenient to define types for structures (see Section 1.8).

1.7. **Working with Arrays.** An array is an indexed collection of data items of the same data type accessible by a common name. There exist one-dimensional arrays (lists) and multi-dimensional ones, but we will not be using such constructs in this project. One-dimensional arrays can be thought of as vectors. In the EAGLE project it is the case that all involved arrays have a fixed and known size. That said, arrays are declared as follows:

```
#define NX_ATTITUDE 9   /**< Number of attitude states */
#define NY_ATTITUDE 6   /**< Number of attitude outputs */

static float xAttEstimate[NX_ATTITUDE]; /**< State estimate for attitude */
static float yAtt[NY_ATTITUDE]; /**< Attitude meas. (3D quat., omega) */
```

It is good practice to use `#define` directives to define the size of arrays instead of hard-coding the sizes (e.g., `static float xAttEstimate[9]`). According to ISO C90, it is not possible to allocate arrays inside functions using variable lengths, even when the length is defined as `const`.

```
void wrongFunction() {
  const size_t len = 10;
  int myArray[len]; /* this is wrong */
}
```

Note that when it comes to writing for loops you might have encountered code such as the following:

```
for (size_t i = 0; i < 10; ++i) {
  /* do stuff */
}
```

This is only allowed in C99 and should not be used in this project. As we discussed in Section 1.4, it is not possible to initialise variables in place and this holds for arrays too.

The elements of an array are accessed using square brackets and using indices which start at zero and go up to one less than the size of the array. This is an example where we define and print

```
#include <stddef.h> /* definition of size_t */

#define LEN 3

void someFunction() {
  /* Declarations */
  size_t i;
  float myArray[LEN];

  /* Definition */
  myArray[0] = 1.453;
  myArray[1] = 5.845;
  myArray[2] = 3.142;

  for (i = 0; i < LEN; ++i) {
    printf("array[%u] = %2.4f\n", (unsigned) i, myArray[i]);
  }
}
```

If we pass an array to a function, there is no way the function can retrieve the size of that array. The reason is that when we pass an array to a function, we essentially provide a pointer to its first element — no other information. The following two function prototypes are equivalent:

```
void fooBar(float x[]){
  /*
   * A pointer to the first element of an array is
   * provided to this function. There is no way to
   * tell the number of elements of the array here.
   */
}

void fooBar2(float * x){
  /*
   * Here, we receive a pointer-to-float. This is
   * just a position in memory; nothing more.
```

```
    */
}
```

For that reason, when we need to pass an array to a function, we also need to specify the number of its elements. Here is a simple example:

```
void arrayReader(float * x, size_t numElements){
  size_t i;
  for (i = 0; i < numElements; ++i) {
    /* do something */
  }
}
```

1.8. **Passing structure types to functions.** Structures allow the combination of different types of data with different names. First, it is convenient to define structures types using `typedef`. For example:

```
typedef struct time_struct {
    unsigned int hour;
    unsigned int minute;
} mytime_t;
```

There are two ways in which we can pass such a structure to a function. The first one is to pass it *by value* as in the following example:

```
/* Adds one minute to the provided time */
mytime_t plusOneMinute(mytime_t d){
    int over;
    d.minute += 1;
    over = d.minute / 60;
    d.minute %= 60;
    d.hour += over;
    return d;
}
```

We should note a couple of things about this implementation. First, a *copy* of `d` is created and is passed to `plusOneMinute`. This bears two important implications. First, an — often — unnecessary use of memory and, second, the fact that the input argument `d` cannot be modified. Have a look at the following example

```
void updateTimePlusOneMinute(mytime_t d){
    int over;
    d.minute += 1;
    over = d.minute / 60;
    d.minute %= 60;
    d.hour += over;
}
```

The above function does *nothing*; it updates a copy of `d`, so the original variable does not get updated. It is, therefore, preferred to pass `d by reference`, that is, pass a pointer to `d`:

```
void updateTimePlusOneMinute(mytime_t * d){
    int over;
    d->minute += 1;
    over = d->minute / 60;
    d->minute %= 60;
    d->hour += over;
}
```

The fields of pointers-to-structure can be accessed using `->`, not the dot operator. This is how this function can be used:

```
mytime_t d;
d.hour = 12;
d.minute = 59;
updateTimePlusOneMinute(&d); /* pass pointer to d */
```

2. C for the EAGLE project

The C framework for the EAGLE project which is available on ESAT's gitlab where you may login using you student account. In folder `BareMetal` there is a framework that allows you to obtain measurements from the various sensors of EAGLE, read the incoming signals from the RC, communicate with the Python code and send PWM signals to the ESCs.

The starting point is the README file inside this folder which will guide you through the contents of that folder and explains where exactly you have to implement your controllers and observers. Feel free to modify the structure of any of the files in `BareMetal/src/control/` by adding static functions, modifying the visibility of variables (by removing them from the header files and declaring them in the C file as `static`) etc.

2.1. **Performance.**

A few hints for higher performance code: Avoid calling a function — which will return the same result at every invocation — in a loop or more than once:

```
void multipleFunctionCalls(float[] myArray){
    size_t i;
    for (i = 0; i < N; ++i) {
      /*
       * Here function `getX` is called at every iteration of
       * the loop.
       */
      myArray[i] += getX();
    }
}
```

This is a better way to go:

```
void multipleFunctionCalls(float[] myArray){
    size_t i;
    const x = getX(); /* Called once outside the loop */
    for (i = 0; i < N; ++i) {
      myArray[i] += x;
    }
}
```

The same principle holds for structure. Avoid the following:

```
/* parameters_t is some structure type */
float multipleStructureCalls(parameters_t * params){
    size_t i;
    float result;
    result = 0.0f;
    for (i = 0; i < params->numMotors; ++i) {
      result += params->motors[i]->weight;
    }
    return result;
}
```

Instead, prefer the following:

```
float multipleStructureCalls(parameters_t * params){
    size_t i;
    float result;
    motor_t * myMotors;
    result = 0.0f;
    myMotors = params->motors;
    for (i = 0; i < params->numMotors; ++i) {
      result += myMotors[i]->weight;
    }
    return result;
}
```

3. Best practices

(1) Export only the functionality that is needed:
   i. Everything in a source file that can be hidden from the outside world should be. Only the documented external interfaces should be exposed.
   ii. All exposed functionality must be declared in a header file.
   iii. The header must be included whenever the functionality is needed.
   iv. Headers are once-only (use header guards).
(2) Use a private source code versioning system (preferably based on git) and actively collaborate using it. Add README files, make branches when necessary, write meaningful commit messages. Do not forget to add a `.gitingore` file and do not track binary files.
(3) Document your functions properly and clearly. If you change the function template (inputs/outputs), update the documentation too.
(4) Write comments as you code.
(5) Do not comment-out code (use git).
(6) Write unit tests and target 100% coverage.
(7) Do not use globals. Use file-scope static variables and functions.
(8) Establish a meaningful naming convention (and make sure all team members abide by it). Variables should be as self-explanatory as possible (e.g., if you use the camel-case standard, a variable name should be `angularVelocity` rather than `angV` or `w`).
(9) Create milestones and use an issue tracker to know at every moment what needs to be done and who is responsible for what.
(10) Use valgrind to detect memory leaks and bugs related to memory management.

4. Self-evaluation checklist

Preparation

- ☐ Everyone has read this document

Code versioning system

- ☐ Git repository is set up
- ☐ Git repository is regularly used and updated (total number of commits: ___)
- ☐ README file(s)
- ☐ `.gitignore` file(s)
- ☐ No binary files (If any, explain how you deal with conflicts)
- ☐ Branches created: ____
- ☐ Pull requests: ____
- ☐ First pull request reviewed; merge performed.

Project management

- ☐ Use of issue tracker
- ☐ Team members are aware of all issues
- ☐ Use of trello or similar project management system

Documentation and comments

- ☐ All functions are documented
- ☐ Documentation based on the Doxygen standard
- ☐ All global or static variables are documented
- ☐ All local function-scope variables have explanatory comments
- ☐ All implementations contain comments

Interfaces

- ☐ No unnecessary functionality is exported
- ☐ No unused function declarations in header files

Good coding practices

- ☐ Adoption of naming convention
- ☐ Separation of declarations from code
- ☐ Declarations in separate lines (see Section 1.2)
- ☐ No initialisations of global or static file-scope variables
- ☐ No hard-coded sizes of arrays (`#define` is used instead)
- ☐ No structures passed by value (see Section 1.8)
- ☐ No commented-out code or print statements for debugging purposes

Testing

- ☐ Unit tests
- ☐ Memory check with valgrind
- ☐ Tests are performed before every commit