Modularizing C Code: Managing large projects

Dean Camera

September 12, 2014

****

Contents

There comes a time when a project has outgrown the scope of a single file. Perhaps it's the point at with you need to add more RAM to your computer just to open your project, or when you realize you've just spent 45 minutes tracking down the timeout function you swore you wrote two weeks ago. Whatever the reason, it's time to split your project up into several manageable files, which you can then keep better track of. This tutorial will give several pointers about the intricacies of how to accomplish such a task.

1 Identify Common Routines

First of all, it makes sense to group all the common routines into their own files — eg. one for USART management, one for your protocol layer handler, one for your main code. You should determine which functions can be moved into a separate files while still retaining some connection with one another. Let's take the following example of routines:

```
GetUSARTByte()
TurnOnStatusLed()
SetupSpeaker()
TurnOffStatusLed()
SetupUSART()
main()
CheckTempSensorVal()
ProcessByte()
Beep()
EnableUSART()
SleepMode()
CheckADCChannel()
```

We can split these routines up into several small files — one for the USART, one for the main (and misc) routines, one for the speaker and one for the ADC (and related processing).

Let's take a look at our proposed new structure:

USART.c

```
SetupUSART()
EnableUSART()
GetUSARTByte()
ProcessByte()
```

Speaker.c

```
Code:
SetupSpeaker()
Beep()
```

ADC.c

```
CheckADCChannel()
CheckTempSensorVal()
```

Main.c

```
main()
SleepMode()
TurnOnStatusLed()
TurnOffStatusLed()
```

Ok, looks pretty good! We'll cut and paste those routines into the appropriate files in the same project directory, so now we have a bunch of C files containing related functions.

2 Adding the new files to your makefile

Even though you've made no functional changes to your code outside moving certain routines to a different file, you still need to tell the compiler, linker and associated GCC tools where everything is now located. Open up your makefile, and you should find some lines similar to the following (if your makefile is based off the WinAVR template):

```
# List C source files here. (C dependencies are automatically generated.)

SRC = $(TARGET).c
```

What we now need to do is add the file names of the newly created files. We'll take our above example here again for consistency. Our new extra files are called `ADC.c`, `Speaker.c` and `USART.c`, so we need to add those to the SRC line of the makefile.

```
# List C source files here. (C dependencies are automatically generated.)

SRC = $(TARGET).c ADC.c USART.c Speaker.c
```

Now, that'll work, but it's a real pain for future expansion. To make our life easier, we'll place the filenames on their own line in alphabetical order. To indicate a line continuation in a makefile, we need to place a "\" at the end of the continued lines:

```
# List C source files here. (C dependencies are automatically generated.)

SRC = $(TARGET).c \
      ADC.c       \
      Speaker.c   \
      USART.c
```

NB: Watch the file case! To GCC and its related utilities, `MAIN.C` is not the same as `Main.c`. Make sure you put down the file names in the exact same case as they appear inside explorer, or you'll get build errors!

3 Naming the routines

One problem with our multi-file setup remains; routines are still hard to find across the files. One might easily identify the file location of our mythical `CheckADCChannel()` routine, but what about `SleepMode()`? Obviously a naming convention becomes important.

Now, this is where it gets tricky. There's really no set standard for function names, and every one prefers something different. It is important to choose which one you prefer, but it is ten times more important to remain consistent in your scheme across your entire project.

The first word, symbol or acronym in your function name should indicate the file it is located in. For example, all the ADC functions will have "ADC" at their start, and all the speaker-related functions in `Speaker.c` would have "Speaker" at their start. You can eliminate repetition in a name as it becomes self-explanatory what the function is referring to due to the prefix — thus `CheckADCChannel()` would become `ADCCheckChannel()` (or similar), rather than the superfluous `ADCCheckADCChannel()`.

I'll use our example again and put here a possible function naming scheme. Note that the `main()` function's name remains unchanged as it is mandated by the C standard:

USART.c

```
USARTSetup()
USARTEnable()
USARTGetByte()
USARTProcessByte()
```

Speaker.c

```
SpeakerSetup()
SpeakerBeep()
```

ADC.c

```
ADCCheckChannel()
ADCCheckTempSensorVal()
```

Main.c

```
main()
MainSleepMode()
MainTurnOnStatusLed()
MainTurnOffStatusLed()
```

This new scheme makes finding the location of a routine quick and easy, without the need for a multi-file search utility. Don't forget to change the function prototypes in your header file to the match your new function names!

4 Making functions static

Static functions is possibly something you've never heard of up to now. The process of declaring a function to be static indicates to the compiler that its scope is limited to the source file in which it is contained - in essence it makes the function private, only accessible by other function in the same C file. This is good practice from two standpoints; one, it prevents outside code from calling module-internal functions (and reduces the number of functions exposed by the file) and two, it gives the compiler a better chance to optimize the function in a better way that if it was assumed to be able to be called by any other source file.

Identify which of your functions are module-internal, and add in the static keyword. In our example, let's say that the `USARTEnable()` function was only called by `USARTSetup()`, and never by any other source file. Assume that the USARTEnable function is declared as the following:

```
void USAREnable(void);
```

We'll make the function static to reduce its scope to inside `USART.c`:

```
static void USARTEnable(void);
```

Note that the static keyword should be added to both the prototype, as well as to the function in the C source file. However, there's now an additional important step: we need to move the (now) static function prototype out of our header file and back into the C source file. This seems counter-intuitive at first; after all, header files are supposed to contain function prototypes. The reason is due to the limited scope of static functions; since they can only be called from the C source file where they are defined, there's no need to have a global prototype in the header files that other C source files could potentially include. In fact, if you *do* leave the static function prototype in the header file, you will receive compiler warnings about undefined functions if the static function prototype is visible to other C source files.

5 Global variables

Your project probably has quite a few global variables declared at the top of your main source file. You need to cut-and-paste them each into the most appropriate source file. Don't worry if it's referenced by more than one file; dealing with that case will be handled later in the tutorial. For now, just put it in the C file you deem to be the best for it.

Just in case you've done this, remember the following golden rule: header files should declare, not define. A header file should not itself result directly in generated code - it is there to help the compiler and yourself link together your C code, located in C files. If you've put any globals inside your header file, move them out now.

6 Splitting the header file

This is where the hard part begins. So far we've successfully split up the contents of our project into several C files, but we're still stuck with one large header file containing all our definitions. We need to break our header file up into separate files for each of our C modules.

First, we'll create a bunch of .h header files of the same names as our `.c` files. Next, we'll move the obvious things from the master header file, the function prototypes. Copy each of the prototypes into the respective header files.

Next, the macros. Again, move the macros into the header file where they are used the most. If you have any generic macros that are used in the majority of your source files, create a new `GlobalDefinitions.h` header file and place them there. Do the same for any typedefs you may have in your header file.

Let's assume our above example has the following original header file:

```
#define StatusLedRed          (1 << 3)
#define StatusLedGreen        (1 << 2)
#define ADCTempSenseChannel   5
#define BitMask(x)            (1 << x)

typedef unsigned char USARTByte;

USARTByte GetUSARTByte(void);
void      TurnOnStatusLed(void);
void      SetupSpeaker(void);
void      TurnOffStatusLed(void);
void      SetupUSART(void);
int       main(void);
int       CheckTempSensorVal(void);
void      ProcessByte(USARTByte);
void      Beep(void);
void      EnableUSART(void);
void      SleepMode(void);
int       CheckADCChannel(char);
```

We can split this into our new header files like thus:

USART.h

```
typedef unsigned char USARTByte;

void        USARTSetup(void);
static void USARTEnable(void);
USARTByte   USARTGetByte(void);
void        USARTProcessByte(USARTByte);
```

Speaker.h

```
void SpeakerSetup(void);
void SpeakerBeep(void);
```

ADC.h

```
#define ADCTempSenseChannel 5

int ADCCheckChannel(char);
int ADCCheckTempSensorVal(void);
```

Main.h

```
#define StatusLedRed     (1 << 3)
#define StatusLedGreen   (1 << 2)

int main(void);
void MainSleepMode(void);
void MainTurnOnStatusLed(void);
void MainTurnOffStatusLed(void);
```

Pretty straightforward - using our renamed functions, the job becomes easy. The only issue is the orphaned macro `BitMask()`, which we'll assume is used by all the files. As discussed, we'll place that into a separate header file called `GlobalDefinitions.h`.

7 Renaming the macros

As with the function names, the macros should also be renamed to indicate where they are located. Again, use whatever convention you prefer.

8 Global variables revisited

Now comes the time to fix up those global variable references from before. You're faced with a problem; your global variable is declared in one file, but it's used in two or more. How do you tell the other files to use the declaration you already made?

The answer is the use of the "extern" keyword. This keyword indicates to the compiler that a global of the specified name and type has been declared elsewhere, and so the references are fixed up by the linker when the project is built. Inside the header file of the C module containing the global, add the same line of code (sans any initialization value) but add the extern keyword. Say we have the following:

USART.c

```
unsigned char LastCommand = 0x00;
```

We can add a reference to the global in `USART.h` to make it visible in our other C modules:

USART.h

```
extern unsigned char LastCommand;
```

Do this for each of your global variables that are accessed in C files other than the one in which it is contained.

9 Including the header files

Our final task is to link our header files to the relevant C files. Firstly, each `.c` file should include its own header file. In the C language, for header files located in the project's current directory, we use quotes around the file names:

USART.c

```
#include "USART.h"
```

Speaker.c

```
#include "Speaker.h"
```

ADC.c

```
#include "ADC.h"
```

Main.c

```
#include "Main.h"
```

Next, we need to include the library header files relevant to each C module. For example, our `Main.c` and `Speaker.c` files both make use of the `<avr/io.h>` header files. System headers should have angled brackets instead of quotes:

Speaker.h

```
#include <avr/io.h>
```

Main.h

```
#include <avr/io.h>
```

Rinse and repeat for each of the required system header files inside the header of each C module.

The last task of this step is to include the header files used by each module. For example, if a function in Main.c calls a function in `ADC.c`, we need our Main.h header file to include the `ADC.h` header file.

10 Header file protection

The last thing to do before our conversion is complete, is to protect our header files from multiple inclusion. Take the following example. Say that Main and ADC both refer to each other:

Main.h

```
#include "ADC.h"
```

ADC.h

```
#include "Main.h"
```

What happens when this is compiled? The preprocessor will look at `Main.h`, then include `ADC.h`. However, `ADC.h` includes `Main.h`, which again includes `ADC.h`, etc...

To guard against this problem, we can use preprocessor defines. The following code snippet is the basic protection setup:

```
#ifndef MAIN_H
#define MAIN_H

// Header file contents

#endif
```

This construct, when applied to each of your header files, will protect against multiple inclusions. As each C file is compiled, the associated header file is included, as well as any other referenced header files (via includes in the C file's header file). As each header is included, a check is performed to see if the header's unique token is already defined, and if so the inclusion halts to prevent recursion. If the token is not already defined, the preprocessor defines it and looks at the remainder of the header file's contents. By giving each header file a different token (typically the header's filename in `ALL CAPS`, and the period replaced by an underscore), this system will prevent any preprocessor troubles.

Let's take a final look at how our mythical C file's headers might look like:

USART.h

```
#ifndef USART_H
#define USART_H

typedef unsigned char USARTByte;

void        USARTSetup(void);
static void USARTEnable(void);
USARTByte   USARTGetByte(void);
void        USARTProcessByte(USARTByte);

extern unsigned char LastCommand;

#endif
```

Speaker.h

```
#ifndef SPEAKER_H
#define SPEAKER_H

void SpeakerSetup(void);
void SpeakerBeep(void);

#endif
```

ADC.h

```
#ifndef ADC_H
#define ADC_H

#define ADC_TEMPSENSORCHANNEL 5

int ADCCheckChannel(char);
int ADCCheckTempSensorVal(void);

#endif
```

Main.h

```
#ifndef MAIN_H
#define MAIN_H

#define MAIN_STATLED_RED     (1 << 3)
#define MAIN_STATLED_GREEN   (1 << 2)

int  main(void);
void MainSleepMode(void);
void MainTurnOnStatusLed(void);
void MainTurnOffStatusLed(void);

#endif
```

Voila, we now have a separated, maintainable project!