Chào mừng đến với khóa học Lập trình hệ thống nhúng. Tên tôi là Miro Samek và trong bài học này tôi sẽ chỉ cho bạn cách cải thiện chương trình "blinky" bằng cách sử dụng bộ xử lý trước C và từ khóa "volatile".

Như thường lệ, chúng ta hãy bắt đầu bằng cách tạo một bản sao của dự án "bài học 4" trước đó và đổi tên thành "bài học 5". Nếu bạn mới tham gia khóa học, bạn có thể tải xuống các dự án trước đó từ state-machine.com/quickstart.

Vào thư mục "lesson5" mới và nhấp đúp vào tệp không gian làm việc để mở bộ công cụ IAR. Nếu bạn không có bộ công cụ IAR, hãy quay lại "bài học 0".

Vậy đây là chương trình bạn đã tạo trong "bài học 4". Nó hoạt động theo cách có thể làm đèn LED ĐỎ trên bảng Stellaris Launchpad nhấp nháy, nhưng chắc chắn là không dễ đọc vì nó chứa đầy những con số khó hiểu và không có bình luận nào giải thích những gì đang diễn ra.

Để cải thiện khả năng đọc mã, sẽ rất tuyệt nếu sử dụng tên cho các thanh ghi thay vì các số khó hiểu. Một cách để bạn có thể thực hiện được điều này là sử dụng bộ xử lý trước C, cho phép bạn định nghĩa bất kỳ đoạn mã nào thành một macro.

PREPROCESSOR

Ví dụ, hãy định nghĩa một macro cho thanh ghi đầu tiên mà bạn ghi vào. Bắt đầu một dòng mới bằng dấu # theo sau là từ "define" (xác định), tiếp theo là tên của macro. Bảng dữ liệu gọi thanh ghi này là Thanh ghi điều khiển cổng xung nhịp chế độ chạy cho GPIO, vì vậy chúng ta hãy đặt tên cho macro này là RCGC-GPIO.

Sau tên, bạn chỉ cần dán đoạn mã mà macro này sẽ thay thế.

Sau khi định nghĩa macro, bạn có thể sử dụng nó thay cho đoạn mã gốc.

Nhấn F7 để kiểm tra xem trình biên dịch có chấp nhận mã của bạn hay không.

Bộ xử lý trước C được gọi như vậy vì về mặt khái niệm, nó là bước đầu tiên riêng biệt trong quá trình thay thế văn bản đơn giản trước khi biên dịch thực sự. Bộ xử lý trước sẽ xóa tất cả các dòng bắt đầu bằng dấu #, do đó trình biên dịch sẽ không nhìn thấy chúng. Ví dụ, bạn có thể định nghĩa một macro là bất kỳ thứ gì, nhưng miễn là nó không được sử dụng trong mã thì điều đó không quan trọng và mã vẫn được biên dịch.

Ngoài ra, bộ xử lý trước chỉ thay thế các macro thực sự được sử dụng trong mã, do đó trình biên dịch chỉ nhìn thấy chuỗi ký tự thay thế chứ không nhìn thấy tên macro.

Điều này có nghĩa là một macro không cần phải là một phần tử hoàn chỉnh của ngôn ngữ C. Ví dụ, macro FOO chỉ là một phần của biểu thức ép kiểu con trỏ, nhưng miễn là văn bản thay thế cho macro có ý nghĩa trong ngữ cảnh cụ thể thì trình biên dịch sẽ vui vẻ chấp nhận, vì thực tế trình biên dịch không thể phân biệt được sự khác biệt.

Hệ quả của tất cả những điều này là bạn cần phải cẩn thận khi định nghĩa macro, để ý nghĩa của chúng không thay đổi đột ngột tùy thuộc vào ngữ cảnh mà chúng được thay thế. Ví dụ, để tránh bất ngờ, bạn nên đặt macro như RCGC-GPIO trong dấu ngoặc đơn để luôn có nghĩa là hủy tham chiếu đến con trỏ trong bất kỳ ngữ cảnh nào có thể sử dụng.

Bạn cũng có thể định nghĩa macro bằng cách sử dụng các macro khác. Ví dụ, nếu bạn định nghĩa macro GPIOF\_BASE theo như được chỉ định trong Bảng dữ liệu, bạn có thể sử dụng macro này trong định nghĩa của các macro khác,

chẳng hạn như GPIOF\_DIR cho thanh ghi hướng chân với độ lệch 0x400 so với địa chỉ cơ sở,

GPIOF\_DEN cho phép kỹ thuật số với độ lệch 0x51C so với địa chỉ cơ sở

và GPIOF\_DATA với độ lệch 0x3FC so với địa chỉ cơ sở.

Cuối cùng, bạn nên thêm chú thích vào mã của mình. Bình luận chỉ dành cho con người, những người đọc mã của bạn và bị trình biên dịch bỏ qua hoàn toàn.

Tiêu chuẩn C99 hỗ trợ hai loại chú thích: chú thích C truyền thống được phân cách bằng /\* \*/, như chú thích này:

và chú thích theo kiểu C++ bắt đầu bằng // và kết thúc bằng cuối dòng, như những chú thích này.

Chú thích được phép ở mọi nơi mà bạn có thể đặt khoảng trắng hợp pháp và trên thực tế, tất cả các chú thích đều được thay thế bằng một khoảng trắng trước khi biên dịch. Cả hai loại chú thích đều có thể được sử dụng trong định nghĩa macro.

Vì vậy, bây giờ sẽ rất thú vị khi xem liệu mã này có còn làm đèn LED nhấp nháy hay không. Để thực sự thấy được điều này, tôi sẽ sử dụng bảng Stellaris Launchpad . Nhưng nếu bạn không có bo mạch, hãy cấu hình Trình gỡ lỗi cho Trình mô phỏng và làm theo.

Tuyệt, đèn LED nhấp nháy như trước, nghĩa là mọi thay đổi của bạn đều có hiệu quả.

Chúng ta hãy cùng xem xét chi tiết cách trình biên dịch dịch macro GPIO\_DATA và xem liệu nó có gây ra bất kỳ chi phí phát sinh nào không. Rốt cuộc, bạn có thể lo ngại rằng CPU cần phải thêm địa chỉ cơ sở vào vị trí bù trừ khi chạy.

Nhưng khi bạn xem qua mã, bạn có thể thấy ngay rằng lệnh LDR.N tải trực tiếp địa chỉ đầy đủ 0x400253FC vào R0 mà không thực hiện bất kỳ phép cộng nào.

Nói cách khác, mã vẫn hiệu quả như trước vì trình biên dịch sẽ gộp mọi hằng số có thể tính toán được tại thời điểm biên dịch và tránh các tính toán không cần thiết khi chạy.

Cuối cùng, thật thú vị khi xem lệnh nào thực sự bật đèn LED. Thì ra đây chính là lệnh STR. Nói cách khác, theo quan điểm của CPU, việc giao tiếp với thế giới bên ngoài về cơ bản rất đơn giản và chỉ cần ghi một giá trị cụ thể vào một địa chỉ cụ thể.

MACROS

Được rồi, chương trình của bạn vẫn hoạt động và hiệu quả như trước. Nhưng tôi không muốn để lại cho bạn ấn tượng rằng bạn phải tự mình xác định macro cho tất cả các thanh ghi. Trên thực tế, thông thường bạn không cần phải làm như vậy vì các nhà cung cấp vi điều khiển, chẳng hạn như Texas Instruments trong trường hợp bo mạch Stellaris, đã cung cấp các macro cho bạn trong một tệp riêng, mà tôi đã sao chép vào thư mục lesson5.

Bạn có thể thêm tệp này vào dự án bằng cách nhấp chuột phải vào dự án và chọn tùy chọn menu Thêm->Thêm tệp. Tên tệp là "lm4f120h5qr.h", tương ứng chính xác với loại vi điều khiển trên bo mạch Stellaris Launchpad của bạn. Phần mở rộng tệp ".h" có nghĩa là tệp này là tệp tiêu đề, được thiết kế riêng để đưa vào các tệp ".c", chẳng hạn như main.c.

VOLATILE QUALIFIER

Khi bạn mở tệp tiêu đề, bạn có thể thấy nó chứa một loạt các macro rất giống với những gì bạn đã tự định nghĩa. Tuy nhiên, các phép ép kiểu con trỏ được sử dụng trong tệp tiêu đề lại khác biệt đáng kể, điều này cần được giải thích thêm.

Để tôi lấy một macro từ tệp tiêu đề và sao chép vào tệp main.c để so sánh.

Sự khác biệt đầu tiên là loại con trỏ. Các macro trong main.c sử dụng "unsigned int", trong khi tệp tiêu đề sử dụng "unsigned long". Tôi sẽ thảo luận về các kiểu dữ liệu trong một bài học riêng, nhưng hiện tại tôi chỉ muốn nói rằng trên máy 32 bit, như bộ xử lý ARM, kiểu int có độ rộng 32 bit, kiểu long cũng vậy. Vì vậy, "unsigned int" và "unsigned long" là tương đương nhau.

Vì vậy, sự khác biệt thực sự nằm ở từ "volatile" . Nó thông báo cho trình biên dịch rằng đối tượng được con trỏ trỏ tới có thể thay đổi một cách tự phát.

***Khi bạn khai báo một đối tượng là " volatile ", bạn đang cho trình biên dịch biết rằng đối tượng đó có thể thay đổi ngay cả khi không có câu lệnh nào trong chương trình làm thay đổi nó.*** Ví dụ, hai bit trong thanh ghi GPIOF trên bo mạch Launchpad được kết nối với công tắc người dùng. Các bit này có thể thay đổi khi người dùng nhấn hoặc nhả công tắc, điều này rõ ràng không phải do bất kỳ lệnh chương trình nào gây ra. Do đó, thanh ghi GPIOF và thực tế là hầu hết các thanh ghi I/O khác đều là thanh ghi không ổn định.

***Điều này rất quan trọng vì trình biên dịch có thể tối ưu hóa quyền truy cập vào các đối tượng không biến động bằng cách đọc giá trị của đối tượng vào thanh ghi CPU, làm việc với thanh ghi đó trong một thời gian và cuối cùng ghi giá trị vào thanh ghi đó trở lại đối tượng.***

***Trình biên dịch KHÔNG được phép thực hiện thao tác tối ưu hóa này với các đối tượng "volatile". Mỗi khi chương trình nguồn yêu cầu đọc từ hoặc ghi vào một đối tượng "volatile", trình biên dịch phải thực hiện việc đó.***

***Rõ ràng là thuật ngữ "volatile" hữu ích cho các thanh ghi I/O như GPIOF. Nhưng nó cũng có thể rất hữu ích cho các biến "bình thường" để ngăn chặn các tối ưu hóa mà trình biên dịch có thể thực hiện.***

Ví dụ, biến "counter" chỉ được sử dụng trong hai vòng lặp trễ. Nhưng nếu bạn xem xét các vòng lặp này từ góc nhìn của trình biên dịch, chúng không đóng góp gì cho phép tính toán, vì giá trị cuối cùng của "bộ đếm" sẽ bị ghi đè hoặc bị loại bỏ.

Trong trường hợp này, trình biên dịch được phép tối ưu hóa cả hai vòng lặp trễ "xa". Trên thực tế, bạn có thể dễ dàng thấy điều này bằng cách cho phép mức tối ưu hóa cao hơn như sau.

COMPILER OPTIMIZATION

Nhấp vào tùy chọn Project, chọn mục "C/C++ Compiler" và nhấp vào tab "Optimizations". Chọn mức tối ưu hóa "High" và nhấp vào OK.

Biên dịch lại và chạy chương trình trên bảng Launchpad.

Như bạn có thể thấy, đèn LED sáng lên và liên tục sáng...

Khi bạn thực hiện từng bước trong mã, bạn có thể thấy rằng các hướng dẫn để bật và tắt đèn LED vẫn còn đó, nhưng các vòng lặp trễ giữa chúng đã biến mất.

VOLATILE KEYWORD

Nhưng bây giờ, khi bạn đã biết về từ khóa "volatile" , bạn sẽ biết cách ngăn trình biên dịch tối ưu hóa vòng lặp trễ.

Bạn cần phải làm cho biến "counter" trở nên "volatile".

Nhân tiện, từ khóa "volatile" có thể được đặt trước kiểu (int,long..v.v), như trong macro từ tệp tiêu đề, hoặc sau kiểu. Tôi khuyên bạn nên ***đặt nó sau*** (int,long long ..vv..).

Chúng ta hãy nhanh chóng kiểm tra xem bộ đếm "volatile" có thực sự khắc phục được vấn đề hay không.

Có, đèn LED nhấp nháy...

Bạn cũng có thể thấy vòng lặp trễ khi bạn thực hiện từng bước một qua mã.

USING THE HEADER FILE

Bây giờ, chúng ta hãy sử dụng tệp tiêu đề .h bằng cách đưa nó vào chương trình chính.

Một lần nữa, bạn sử dụng bộ xử lý trước cho việc này.

Để bao gồm một tệp, bạn bắt đầu một dòng mới bằng dấu #, theo sau là từ "include" và cuối cùng là tên tệp trong dấu ngoặc kép.

Hãy đặt tệp tiêu đề cạnh chương trình chính để thay thế tất cả các macro bạn đã xác định cho đến nay bằng các macro từ tệp tiêu đề.

Tệp tiêu đề do nhà cung cấp vi điều khiển cung cấp sử dụng tên thanh ghi từ Bảng dữ liệu, do đó bạn sẽ không gặp khó khăn khi nhận ra các thanh ghi thú vị, chẳng hạn như:

GPIO\_PORTF\_DATA

GPIO\_PORTF\_DIR

và GPIO\_PORTF\_DEN

Nếu bạn có bất kỳ nghi ngờ nào về tên chính xác của điện trở, bạn luôn có thể kiểm tra địa chỉ của nó để đảm bảo rằng đây là tên bạn muốn.

Sau khi thay thế tất cả các macro, bạn có thể xóa định nghĩa của riêng mình và biên dịch lại mã.

Hãy kiểm tra lại mã lần cuối để xem đèn LED có còn nhấp nháy hay không...

Bài học về bộ tiền xử lý C và từ khóa "volatile" kết thúc tại đây. Từ bây giờ, bạn có thể viết các chương trình chạy chính xác ở mọi cấp độ tối ưu hóa. Xin chúc mừng!

Trong bài học tiếp theo, bạn sẽ học cách sử dụng toán tử OR và AND theo từng bit để nhấp nháy các màu khác của đèn LED tổng hợp và bạn cũng sẽ tìm hiểu về các tính năng nâng cao hơn của thanh ghi GPIO.

English Text:

Welcome to the Embedded Systems Programming course. My name is Miro Samek and in this lesson I'll show you how to improve the "blinky" program by using the C preprocessor and the "volatile" keyword.

As usual, let's start with making a copy of the previous "lesson4" project and renaming it to "lesson5". If you are just joining the course, you can download the previous projects from state-machine.com/quickstart.

Get inside the new "lesson5" directory and double-click on the workspace file to open the IAR toolset. If you don't have the IAR toolset, go back to "lesson0".

So, this is the program you created in "lesson4". It works in that it manages to blink the RED led on the Stellaris Launchpad board, but it is certainly is not very readable, because it is full of cryptic numbers and there are no comments to explain what's going on.

To improve the readability of the code, it would be very nice to use names for the registers instead of the criptic numbers. One way you can achive it, is by means of the C preprocessor, which lets you define any piece of code as a macro.

For example, let's define a macro for the first register you write-to. Start a new line with the #-sign followed by the word "define", followed by the name of the macro. The data sheet calls the register Run-Mode Clock Gating Control Register for GPIO, so let's name the macro RCGC-GPIO.

After the name you simply paste the piece of code this macro is going to replace.

Once the macro is defined, you can use it instead of the original piece of code.

Press F7 to check if the compiler accepts your code so far.

The C pre-processor is called that way, because it is conceptually a separate first step of simple text substitution before the real compilation. The pre-processor removes all lines starting with the #-sign, so the compiler does not see them at all. For example, you can define a macro to be anything, but as long as it is not used in the code, it doesn't matter and the code still compiles.

Also, the preprocessor replaces only the macros that are actually used in the code, so the compiler sees only the replacement sequences of characters and never the macro names themselves.

This means that a macro does not need to be any complete element of the C language. For example, a macro FOO is just a piece of the pointer-cast expression, but as long as the text substituted for the macro makes sense in the specific context, the compiler will happily accept it, because really the compiler cannot tell the difference.

The corollary of all this is that you need to be careful how you define your macros, so that their meaning will not change unexpectedly depending on the context in which they are substituted. For example, to avoid surprises, it is always a good idea to enclose the macro like RCGC-GPIO in parentheses, so that it always means de-referencing of a pointer in any context it might be used.

It is also possible to define macros using other macros. For example, if you define the macro GPIOF\_BASE as it is specified in the Data Sheet, you can use it in the definitions of other macros,

such as GPIOF\_DIR for the pin direction register with the offset 0x400 from the base address,

GPIOF\_DEN for digital enable with the offset 0x51C from the base address ,

and GPIOF\_DATA with the offset 0x3FC from the base address.

Finally, it is always highly recommended to add comments to your code. Comments are only for the benefit of humans, who read your code, and are completely ignored by the compiler.

The C99 Standard supports two types of comments: the traditional C comment delimited by /\* \*/, such as this one:

and the C++-style comment starting with // and terminated by the end of line, such as these ones

Comments are allowed everywhere where you could legally place a space, and in fact, all comments are replaced with a single space before the compilation. Both types of comments can also be used in the macro definitions.

So, now it will be interesting to see if this code still blinks the LED. To actually see this, I'm going to use the Stellaris Launchpad board. But If you don't have the board, configure the Debugger for Simulator and follow along.

Great, the LED blinks as before, so all your changes seem to work.

Let's examine in some detail how the compiler translated the GPIO\_DATA macro and weather or not it introduced any overheads. After all, you might be concerned than now the CPU needs to add the base address to the offset at runtime.

But when you step through the code, you can immediately see that the LDR.N instruction loads directly the full address 0x400253FC into R0 without performing any additions.

In other words, the code is as efficient as before, because the compiler folds any constants computable at compile time and avoids unnecessary computations at runtime.

Finally, it is very interesting to see which single instruction actually turns the LED on. It turns out that this is the STR instruction. In other words, from the CPU point of view, talking to the outside world is fundamentally very simple and boils down to writing a specific value into a specific address.

OK, so your program still works and is as efficient as before. But I don't want to leave you with the impression that you have to define macros for all the registers yourself. In fact, typically you don't need to, because the microcontroller vendors, such as Texas Instruments in the case of the Stellaris board, already provide the macros for you in a separate file, which I've copied to the lesson5 directory.

You can add this file to the project by right-clicking on the project and choosing the Add->Add Files menu option. The file name is "lm4f120h5qr.h", which corresponds exactly to the microcontroller type on your Stellaris Launchpad board. The file extension ".h" means that it is a header file, specifically designed for inclusion into ".c" files, such as main.c.

When you open the header file, you can see that it contains a whole bunch of macros very similar to what you have defined yourself. However, the pointer casts used in the header file are significantly different, which requires some explanation.

Let me grab one macro from the header file and copy it to the main.c file for comparison.

The first discrepancy is the pointer type. The macros in main.c use "unsigned int", while the header file "unsigned long". I will discuss data types in a separate lesson, but for now let me only say that on a 32-bit machine, like the ARM processor, the int-type is 32-bit wide, so is the long-type. So, "unsigned int" and "unsigned long" are equivalent.

So, the real difference is the "volatile" qualifier. It informs the compiler that the object pointed-to by the pointer might change spontaneously.

When you declare an object to be "volatile", you are telling the compiler that the object might change even though no statements in the program appear to change it. For example, two bits in the GPIOF register on the Launchpad board are connected to the user switches. These bits can be changed when the user presses or releases the switches, which is obviously not caused by any program instruction. Therefore, the GPIOF register, and in fact most other I/O registers, are volatile.

This is important, because the compiler can optimize access to non-volatile objects by reading an object's value into a CPU register, working with that register for a while, and eventually writing the1 value in the register back to the object.

The compiler is NOT permitted to do this soft of optimization with "volatile" objects. Every time the source program says to read-from or to write-to a "volatile" object, the compiler has to do so.

So clearly, the "volatile" qualifier is useful for I/O registers such as GPIOF. But it can also be very useful for "normal" variables to prevent optimizations that the compiler otherwise might do.

For example, the "counter" variable is used only in the two delay loops. But if you look at these loops from the compiler's perspective, they make no contribution whatsoever to the computation, because the final value of the "counter" is either overwritten or discarded.

In this case, the compiler is permitted to optimize both delay loops "away".

You can actually see this quite easily by allowing2 a higher-level of optimization as follows.

Click on the Project options, choose "C/C++ Compiler" section and click on the "Optimizations" tab. Select "High" optimization level and click OK.

Recompile and run the program on the Launchpad board.

As you can see, the LED lights up and stays on, all the time...

When you single-step through the code, you can see that the instructions for turning the LED on and off are still there, but the delay loops between them are gone.

But now, that you know about the "volatile" keyword, you know how to prevent the compiler from optimizing the delay loops away.

You need to make the "counter" variable "volatile".

By the way, the "volatile" keyword can be placed either before the type, like in the macro from the header file, or after the type. I recommend placing it after the type.

Let's quickly test weather the "volatile" counter indeed fixed the problem.

Yes, the LED blinks...

You can also see the delay loop when you single-step through the code.

So now, let's make use of the .h header file by actually including it into the main program.

Again, you use the preprocessor for this.

To include a file, you start a new line with the #-sign, followed by the word "include" followed by the name of the file in double quotes.

Let's put the header file side-by-side with the main program to replace all the macros you've defined so far with the macros from the header file.

The header file provided by the microcontroller vendor uses the register names from the Datasheet, so you should have no trouble to recognize the interesting registers, such as:

GPIO\_PORTF\_DATA

GPIO\_PORTF\_DIR

and GPIO\_PORTF\_DEN

If you have any doubts about the correct name of the resister, you can always check it's address to make sure that this is what you want.

After replacing all the macros, you can remove your own definitions and recompile the code.

Let's test the code one last time to check if the LED will still blink...

This concludes this lesson about the C preprocessor and the "volatile" keyword. From now on, you are able to write programs that will run correctly at any level of optimization. So congratulations!

In the next lesson you will learn how to use the bit-wise OR and AND operators to blink other colors of the composite LED and you will also learn about the more advanced features of the GPIO register.

If you like this channel, please subscribe to stay tuned. You can also visit state-machine.com/quickstart for the class notes and project file downloads.