// Това са бележки, които съдържат информация събирана от различни икзточници, относно //

// по-доброто писане на код за състезанието Renesas MCU Car Rally. //

1. Int е най-добрият тип на променлива за 32-битов процесор.

2. Char и Short се използват, само когато се нуждаем от точен брой на битове.

- Пр. Когато използваме I\O, или масив със много елементи.

3. Възможно най-малко извиквания на функция: r = foo(12); това е добра практика!

- Весто да я викаме постоянно, когато ни трябава, то просто и взимаме стойността и боравим с нея.

4. Да не се оставят никакви цикли, от които няма нужда.

5. При настройка на компилатора:

- The conclusion is that one should always try to compile a program with different optimization

settings and see what happens. Some compilers allow you to adjust the aggressiveness of

individual optimizations—explore that possibility, especially for inlining.

- It is often worthwhile to use different compilation settings for different files in a project:

put the code that must run very quickly into a separate file and compile that for minimal

execution time (maximum speed), and the rest of the code for minimal code size.

This will give a small program, which is still fast enough where it matters.

Some compiler allow different optimization settings for different functions in the same

source file using #pragma directives.

6. Кога искаме Unsigned типа.

- The conclusion is that, if you think about a value as never going below zero, make it unsigned.

If the purpose of a variable is to manipulate it as bits, make it unsigned.

Otherwise, operations like right shifting and masking might do strange things to the value of

the variable.

Source code Optimized Code

int a; int a;

a /= 2; if( a < 0) /\* compensate for \*/

a++; /\* sign of a \*/

a >>= 1;

unsigned int a; unsigned int a;

a /= 2; a >>= 1;

7. Use limited versions of standard functions.

- For instance, the standard printf() is a very big function.

Unless you really need the full functionality, you should use a limited version

that only handles basic formatting or ignores floating point.

- It is not possible for the compiler to automatically figure out which parts

of the function your program needs.

8. Ето с какви пойнтери трябва да работим.

- Use the smallest pointers you can, and avoid any form of generic pointers unless necessary.

Remember to check the compiler default pointer type (used for unqualified pointers).

In many cases it is a rather large pointer type.

9. Опитвай се да избягваш кастването!

- Casting from a smaller and a bigger signed integer type will introduce sign extend operations,

and casting to and from floating point will force calls to the floating point library.

- It is not uncommon that function pointers are larger than int—on a 16-bit machine,

int will be 16 bits, while code pointers may well be 24 bits.

Casting an int to a function pointer will lose information in this case.

- A good way to avoid implicit casts is to use consistent typedefs for all types used

in the application. Otherwise, it is easy to start mixing different-size integers

or other types. Some lint-type tools can also be used to check for type consistency.

10. Ето какви параметри да използваме:

- Some programmers use global variables to pass information between functions.

Register allocation has a hard time with global variables.

If you want to improve register allocation, use parameters to pass information

to a called function. They will often be allocated to registers both in the calling

and called function, leading to very efficient calls.

11. Да се избягва взимането на адреси и то най-вече на локални променливи!

- Thus, you should only take the address of a local variable if you really

must (it is very seldom necessary). If the taking of addresses is used to

receive return values from called functions (for instance, from scanf()),

introduce a temporary variable to receive the result, and then copy the value

from the temporary to the real variable. This should allow the real variable

to be register allocated.

12. Static е ДОБРО !

- Making a global variable static is a good idea (unless it is referred to in another file),

since this allows the compiler to know all places where the address is taken, potentially

leading to better code.

13. Пример относно, защо да не изполваме адреси.

- An example of when not to use the address-of operator is the following, where the use

of addresses to access the high byte of a variable will force the variable to the stack.

The good way is to use shifts to access parts of values.

Bad example Good example

#define highbyte(x) (\*((char \*)(&x)+1)) #define highbyte(x) ((x>>8)&0xFF)

short a; short a;

char b = highbyte(a); char b = highbyte(a);

14. Прави прототипи на функции. - ЗАПОМНИ!

15. Ако използвате глобална променлива във функция много пъти е по-добре стойността и да се придаде

на локална, понеже глобалната не се намира във регистър!!!

- If you are accessing a global variable several times over the life of a function,

it might pay to copy the value of the global into a local temporary. This temporary

has a much higher chance of being register allocated, and if you call functions,

the temporary might remain in registers while a global variable would have to be

written to memory. Note that this assumes that you know that the functions called

will not modify your variables.

16. Групирай извикването на функциите по възможност в началото или края на функция.

- Function calls are bad for register allocation, since they force write-back of global

variables to memory and increase the demand for registers (since registers are used for

parameters and return values, and the called function is allowed to scratch certain registers).

For this reason, it is a good idea to avoid function calls for as long stretches of code as

possible. To minimize the effects, try to group function calls together.

- Note that grouping function calls has no effect if the functions become inlined.

17. Прави функциите static.

- Making a function static tells the compiler that the function will not be called

from outside the module, giving more information to the inliner, enabling it to be

more aggressive and make better decisions about when to inline.

- Note that only functions located within the same module can be inlined (otherwise,

the source code of the function is not available).

- A risky technique to take advantage of inlining is to declare a number of small helper

functions in a header file and making them static. This will give each compiled module

its own copy of every function, but since they are small and static, they are quite

likely to be inlined. If the inlining succeeds, you might save a lot of code space and

gain speed. This approach is similar to declaring inline functions in C++ header files,

which is also just a hint to the compiler.

18. Прави глобалните променливи static.

- By the same logic as local functions, variables that are global but only used within one

file should be made static. This has the double gain of explicitly hiding the variable from

other source files and giving the compiler more information about the variable. Since the

compiler sees a whole module (file) at the same time, it will know all accesses to the

variable, including whether its address is taken, and can thus optimize the code accessing

the variable more efficiently.

- If a variable is not marked as static, the compiler has to assume that someone outside the

current file accesses it, forcing it to be more conservative in optimizing code working with

the variable.

19. Не вграждай асемблер.

- If you need to use assembler, the best solution is to split it out into assembly source files,

or at least into functions containing only inline assembly. Do not mix C code and assembly code

in the same function!

20. Не пиши "умен код", понеже освен, че е труден за разбиране от човек, е труден и за компилатора.

- Some C programmers believe that writing fewer source code characters and making clever use

of C constructions will make the code smaller or faster. The result is code which is harder

to read, and which is also harder to compile. Writing things in a straightforward way helps

both humans and compilers understand your code, giving you better results.

“Clever” solution Straightforward solution

unsigned long int a; unsigned long int a;

unsigned char b; unsigned char b;

/\* Move bits 0..20 to positions 11..31 /\* Straight-forward if statement \*/

\* If non-zero, first ! gives 0 \*/ if( (a & 0x1FFFFF) != 0)

b |= !!(a << 11); b |= 0x01;

---------------------------------------------------------------------------------

“Clever” solution Straightforward solution

int bar(char \*str) int bar(char \*str){

{ /\* Calculating with result of \*/ if(\*str=='+')

/\* comparison. \*/ str++;

return foo(str+(\*str=='+')); } return foo(str);

}

21. Пишете добри за компилатора цикли.

- Common optimization done by progammers is to change from counting up (“for(i=0;i<100;++i)”)

to counting down to zero (“for(i=100;i>0;--i)”), since this is more efficient on some machines.

This can also be done automatically by a compiler.

- To enable the compiler to perform such optimizations, you should write your loops so that they

are easy to understand by the compiler: change the value of the index variable only in the

loop header.

- Do not use the loop index variable after the loop unless this is the purpose of the loop,

since this will make it harder to turn the counting.

- Write easy-to-understand loops, and leave the compiler to optimize the code for a certain target.

22. Map Peripherals to Variables.

- In most ARM® embedded systems, peripherals are located at specific addresses in memory.

It is often convenient to map a C variable onto each register of a memory-mapped peripheral,

and then use a pointer to that variable to read and write the register. In your code, you must

consider not only the size and address of the register, but also its alignment in memory.

23. Use Switch for Jump Tables.

- If you want a jump table, see if you can use a switch statement to achieve the same effect.

It is quite likely that the compiler will generate better and smaller code for the switch

rather than a series of indirect function calls through a table. Also, using the switch makes

the program flow explicit, helping the compiler optimize the surrounding code better.

24. Достигай до елементите на структира по реда им!

- Jumping around inside a structure will force more address calculations to be generated.

- Note that this conflicts with some other advice in this paper about grouping function calls,

so check and see which change gives the best effects. Accessing in order is probably less

important, but it all depends on your particular program, compiler, and platform.

Original Code Rewritten Code

char x, y; char x, y;

volatile char port; volatile char port;

s.D = port; s.A = bar(x,y);

s.B = s.D + 4; s.B = port;

s.A = bar(x,y); s.C = x;

s.C = x; s.D = s.B – 4;

25. Split Structures.

26. Изследвайте Bit Fields преди да ги ползвате.

27. Използвай различни компилатори.

- Different compilers for the same chip are different. Some are better at generating

fast code, other at generating small code, and some may be no good at all.

////////////////////////////////////////////////////////////////////////////////////////////////

As a general rule, code that is easy to understand for a fellow human programmer—and thus easy

to maintain and port—is also easier to compile efficiently.

////////////////////////////////////////////////////////////////////////////////////////////////