НИУ «МЭИ»
представляет учебный курс:

Проектирование цифровых систем управления на базе отечественного микроконтроллера **НИИЭТ К1921ВК01Т**

Москва 2017

Лекция 3.2

- Знакомство с методами измерения производительности вычислений (затраченного на вычисления процессорного времени).
- Практическая работа по определению производительности работы написанного программного кода с разными форматами вычислений.
- Демонстрация влияния заданной степени оптимизации компилятора на производительность вычислений.

Для чего нужно знать производительность?

- Система управления (СУ) должна вызываться с заданной дискретностью по времени.

- Чем частота вызова расчета СУ выше, тем быстрее должны закончиться вычисления.

- Чем частота вызова расчета СУ выше, тем лучше реакция СУ на изменение задания и возмущения.

Вывод: в замкнутых системах управления всегда стремятся повысить частоту расчета СУ, чтобы улучшить показатели регулирования, ускорить переходные процессы, повысить быстродействие.

Как измерять производительность?

Несколько способов:

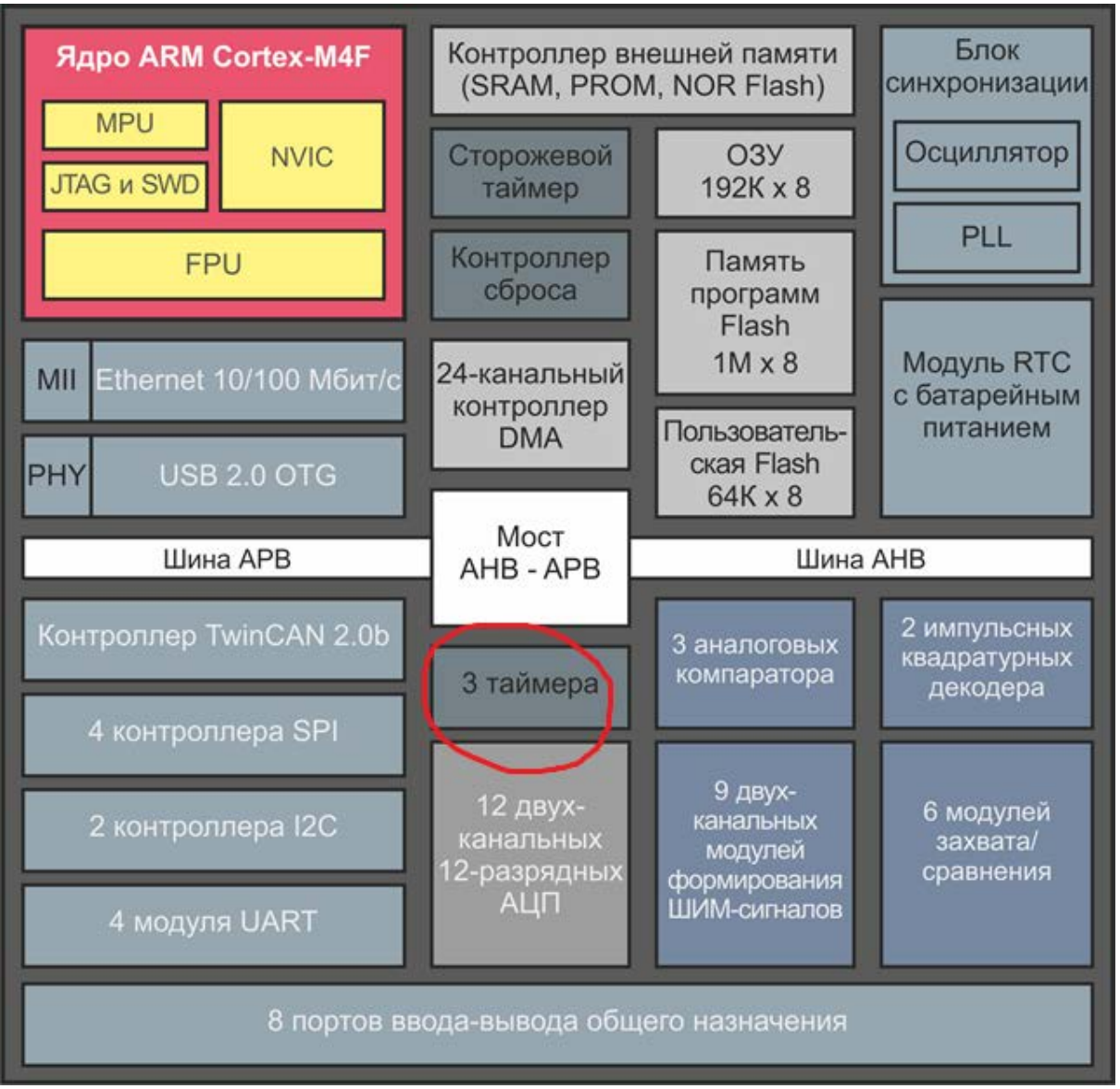
- Посчитать число ассемблерных команд в сгенерированном коде. Подход не работает, все команды занимают разное время, много условных переходов.

- Пройти замеряемую программу по шагам, засекать такты средой разработки: чаще всего есть возможность замерить число тактов процессора от точки останова до точки останова.

- При заходе в измеряемую часть кода выставить произвольную ножку микроконтроллера в единицу, при выходе обнулить. Засечь время между фронтами внешним осциллографом.

- Использовать системный таймер микроконтроллера, «тикающий» с частотой ядра. Засечь число тактов на входе и на выходе измеряемой части кода, посчитать разницу в конце.

Как измерять производительность?



Три таймера общего назначения, которые все равно «ничего не делают».

Запустим один на постоянную работу, чтобы засекать такты.

```
NT_TIMER2->RELOAD = 0xFFFFFFFF;
NT_TIMER2->CTRL = (1 << 0);
```

Как измерять производительность?

```
158 //! Прервыние, вызываемое по таймеру с частотой 10кГц
159 //! \memberof MAIN_C
160 void TIM0_IRQHandler(void) {
161     CpuTimerIsr10 = NT_TIMER2->VALUE; //Засекается время выполнения функции
162     sm_sys.fast_calc(&sm_sys); //расчет 10кГц всего
163 
164     FastCounter++;
165     TIsr10 = (CpuTimerIsr10 - NT_TIMER2->VALUE) & 0xFFFFFF; //время выполнения функции
166     if (TIsr10 > 9700) {
167         sm_prot.bit_fault1 |= F_PROGRAM_10K; //если расчет слишком долгий, ошибка
168     }
169     NT_TIMER0->INTSTATUS_INTCLEAR_bit.INT = 1;  //сброс прерывания
170 }
```

- Засекаем значений таймера `NT_TIMER2->VALUE` перед запуском измеряемого кода в переменную `CpuTimerIsr10.`
- Вызываем функцию, требующую измерения числа тактов.
- Вычитаем из того что было текущие показания (таймер считает вниз).
- Наблюдаем результат.
- Благодаря тому, что таймер считает от 0xFFFFFFFF, специальный учет переполнения не нужен, вычисление разницы всегда будет правильным при наложении маски, отсекающей знак.

Если измеряемое число тактов ожидается малым

```
158 int32 VarA=_IQ(0.5);
159 int32 VarB=_IQ(2.0);
160 int32 VarC=0;
161
162 void TIM0_IRQHandler(void) {
163     CpuTimerIsr10 = NT_TIMER2->VALUE; //Засекается время выполнения функции
164     VarC=_IQmpy(VarA,VarB);
165     TIsr10 = (CpuTimerIsr10 - NT_TIMER2->VALUE) & 0xFFFFFF; //время выполнения функции
166
167     NT_TIMER0->INTSTATUS_[illegible]EAR_bit.INT = 1;  //сброс прерывания
168 }
```

Так делать нельзя!

- Накладные расходы на операции замера числа тактов (забор значения таймера) сравнимы с числом тактов измеряемого кода.
- Все переменные входа и выхода, с которыми работает замеряемая функция, должны быть volatile. Иначе компилятор может просто «выбросить» измеряемый код и ничего не считать.
- Нужно гарантировать, что во время выполнения измерений вычисления не прервет другое прерывание.

Если измеряемое число тактов ожидается малым

```
158 volatile int32 VarA=_IQ(0.5);
159 volatile int32 VarB=_IQ(2.0);
160 volatile int32 VarC=0;
161
162 void TIM0_IRQHandler(void) {
163     CpuTimerIsr10 = NT_TIMER2->VALUE; //Засекается время выполнения функции
164     VarC=_IQmpy(VarA,VarB);
165     VarC=_IQmpy(VarA,VarB);
166     VarC=_IQmpy(VarA,VarB);
167     VarC=_IQmpy(VarA,VarB);
168     VarC=_IQmpy(VarA,VarB);
169
170     VarC=_IQmpy(VarA,VarB);
171     VarC=_IQmpy(VarA,VarB);
172     VarC=_IQmpy(VarA,VarB);
173     VarC=_IQmpy(VarA,VarB);
174     VarC=_IQmpy(VarA,VarB);
175     TIsr10 = (CpuTimerIsr10 - NT_TIMER2->VALUE) & 0xFFFFFF; //время выполнения функции
176
177     NT_TIMER0->INTSTATUS_INTCLEAR_bit.INT = 1;  //сброс прерывания
178 }
```

- При многократном последовательном вызове замеряемой операции накладные расходы «растают» в общем числе тактов. Результат нужно поделить на число вызовов.
- Квалификатор volatile запрещает компилятору оптимизации, связанные с записью и чтением этой переменной. Все операции будут выполнены.

Практическая работа

- Открыть проект MotorControlDemo в VectorIDE, в котором ранее был реализован код инерционного звена первого порядка.
- Организовать замер числа тактов десяти последовательных вызовов для одной из реализаций фильтра. Наблюдать результат.
- Добавить замер тактов для других реализаций фильтра, включая реализацию в плавающей точке.
- Вывести результаты всех четырех замеров в свободные переменные для наблюдения в UniCON (можно использовать готовые Debug3, Debug4.
- Выключить оптимизацию полностью. Удалить ключ компиляции «*-fsingle-precision-constant*» в Properties->C/C++ Build-> Settings->Cross ARM C Compiler->Miscellaneous->Other compiler flags.
- Замерить число тактов с каждым из уровней оптимизации.
- Заменить умножение в фильтре целочисленной математики на сдвиг, замерить такты.
- Заменить умножение на переменную в фильтре с плавающей точкой на константу (число прямо в коде программы), замерить.

Особенности вычислений во float

- Перед использованием float нужно убедиться, что поддержка аппаратной поддержки float включена в настройках компилятора (а по умолчанию она обычно выключена). Иначе вычисления будут идти программно, медленно, используя целочисленную имитацию float.

- При использовании констант в выражении нужно задавать их внимательно. По умолчанию компилятор стремится все константы трактовать не как float (32 бита), а как double (64 бита): двойной точности.
- Но аппаратный FPU в Cortex M4F поддерживает только поддержку float (32 бита). Что тогда происходит, когда константа double?
- Компилятор снова использует **программную реализацию** вычислений, вместо аппаратной: вычисляет double через больше число целочисленных операций: вместо умножения за один такт получается умножение за 30 тактов.

Особенности вычислений во float

```
volatile float VarA=0;
volatile float VarB=0;
volatile float VarC=0;

void TIM0_IRQHandler(void) {
    CpuTimerIsr10 = NT_TIMER2->VALUE;

    VarC=VarA*1.3+VarB;
```

Эта константа трактуется как double, а значит компилятор обязан все выражение посчитать как double (программно), а потом сохранить всё равно результат во float. Нельзя этого допускать!

Что делать?

Два способа:

- Писать всем константам на конце f, типа 1.3f, что трактуется как константа во float. Обязательно где-то забудете, не надо на это полагаться!
- Заставить компилятор трактовать константы как float всегда. Для GCC это делается ключом ***-fsingle-precision-constant***

Почувствуйте разницу!

main.c

```
156 Uint16 TIsr10 = 0;
157
158 volatile float VarA=7;
159 volatile float VarB=9;
160 volatile float VarC=0;
161
162 //! Прерывние, вызываемое по таймеру с частотой 10кГ
163 //! \memberof MAIN_C
164 void TIM0_IRQHandler(void) {
165     CpuTimerIsr10 = NT_TIMER2->VALUE; //Засекается в
166     VarC=VarA*1.3+VarB;
167
168
169     TIsr10 = (CpuTimer  10 - NT_TIMER2->VALUE) & 0x
170
171
172     sm_sys.fast_cal      10кГц всего
173
174     FastCounter++;
175     if (TIsr10 > 9700)
176         sm_prot.bit_fault1 |  _PROGRAM_10K; //если
177     }
```

354 такта

Disassembly

Enter location here

```
0000ba94: 0xc2f20002   movt r2, #8192  ; 0x2000
0000ba98: 0x42f6f023   movw r3, #10992 ; 0x2af0
0000ba9c: 0xc2f20003   movt r3, #8192  ; 0x2000
0000baa0: 0x00001760   str r7, [r2, #0]
166                      VarC=VarA*1.3+VarB;
0000baa2: 0xd3f80080   ldr.w r8, [r3]
0000baa6: 0xf4f7b5fe   bl 0x814 <__extendsfdf2>
0000baaa: 0x000023a3   add r3, pc, #140        ; (adr r3,
0000baac: 0xd3e90023   ldrd r2, r3, [r3]
0000bab0: 0xf4f704ff   bl 0x8bc <__muldf3>
0000bab4: 0x00000446   mov r4, r0
0000bab6: 0x00004046   mov r0, r8
0000bab8: 0x00000d46   mov r5, r1
0000baba: 0xf4f7abfe   bl 0x814 <__extendsfdf2>
0000babe: 0x00000246   mov r2, r0
0000bac0: 0x00000b46   mov r3, r1
0000bac2: 0x00002046   mov r0, r4
0000bac4: 0x00002946   mov r1, r5
0000bac6: 0xf4f747fd   bl 0x558 <__aeabi_dadd>
0000baca: 0xf5f731f9   bl 0xd30 <__truncdfsf2>
0000bace: 0x42f6f423   movw r3, #10996 ; 0x2af4
0000bad2: 0xc2f20003   movt r3, #8192  ; 0x2000
```

main.c

```
157
158 volatile float VarA=7;
159 volatile float VarB=9;
160 volatile float VarC=0;
161
162 //! Прерывние, вызываемое      той 10кГ
163 //! \memberof MAIN_C
164 void TIM0_IRQHandler(voi    {
165     CpuTimerIsr10 = NT_  IMER2->VALUE; //Засекается в
166     VarC=VarA*1.3f+VarB;
167
```

12 тактов

Disassembly

Enter location here

```
0000ba8a: 0xc2f20001   movt r1, #8192  ; 0x2000
0000ba8e: 0xc2f20002   movt r2, #8192  ; 0x2000
165                      CpuTimerIsr10 = NT_TIMER2->VALU
0000ba92: 0x00005d68   ldr r5, [r3, #4]
166                      VarC=VarA*1.3f+VarB;
0000ba94: 0x9fed1f7a   vldr s14, [pc, #124]    ; 0xbb14
0000ba98: 0xd1ed006a   vldr s13, [r1]
0000ba9c: 0xd2ed007a   vldr s15, [r2]
0000baa0: 0xe6ee877a   vfma.f32 s15, s13, s14
165                      CpuTimerIsr10 = NT_TIMER2->VALU
0000baa4: 0x42f6e022   movw r2, #10976 ; 0x2ae0
```

Особенности вычислений во float

В зависимости от того, какой компилятор используется, вычисление выражения вида

```
float Var1 = sin(Var2);
```

может вылиться во что угодно, включая вычисление синуса целочисленной математикой в формате double.

Почему? Вызовется «стандартная» библиотечная реализация синуса, которая «неизвестно» с какой точностью и для какой задачи была написана.

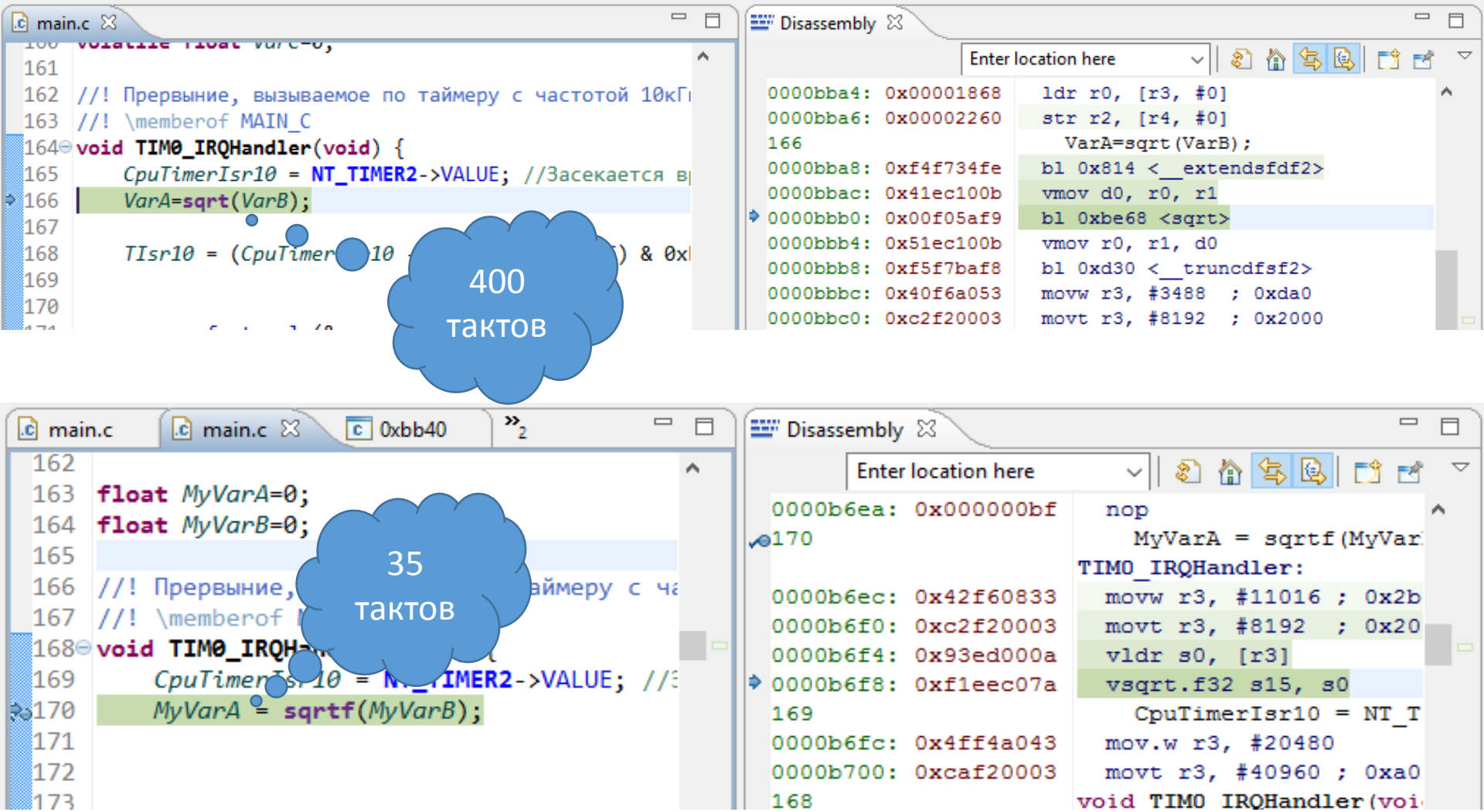
Никогда не доверяйте непроверенным библиотечным функциям! Всегда смотрите в окне дизассемблера, что там вызывается и засекайте, как долго это считается.

Один неверный синус – и вам будет казаться, что производительность микроконтроллера для вашей задачи нужна раза в два выше!

Особенности вычислений во float

- Найдите быструю реализацию тригонометрии во float и вставьте себе в проект. Например, для начала можно посмотреть на реализацию в CMSIS-DSP.
- Замерьте и запишите в заметках, сколько тактов какая функция занимает и какую точность реализует. Точность выше 16 значимых разрядов на практике обычно не нужна! Если функция считает точнее, найдите ту, которая вместо этого считает быстрее!
- Квадратный корень и деление во float в CortexM4F реализуется аппаратно! Для вызова аппаратного квадратного корня в GCC достаточно вызвать функцию `sqrtf()` (но не `sqrt()`, она медленная!). Деление автоматически будет реализовано аппаратно.
- Даже если вы все сделали правильно, убедитесь что используется аппаратная реализация, открыв окно дизассемблера.

Почувствуйте разницу!



Практическая работа

- Замерить, какое количество тактов занимает реализация стандартной библиотечной функции синус sin().
- Найти в интернете быструю реализацию функции синуса в формате float (табличную, аппроксимированную, из какой-то библиотеки – любую, чем она быстрее, тем лучше). Подсказка – можно посмотреть на sinf.
- Перенести в проект, замерить количество тактов (при полной оптимизации).
- Сравнить число тактов с целочисленной функцией синуса с фиксированной точкой в формате IQ8.24 _IQ24sinPU и _IQ24sinPU_accurate из файла IQmath.c.
- Найти максимальную ошибку (отклонение от «идеального» синуса) для трех реализаций функции синуса (найденной в интернете, _IQ24sinPU и _IQ24sinPU_accurate). За эталон значения синуса взять функцию sin() из стандартной библиотеки. Аргумент перебрать с шагом 1%. Будьте внимательны к форматам данных!

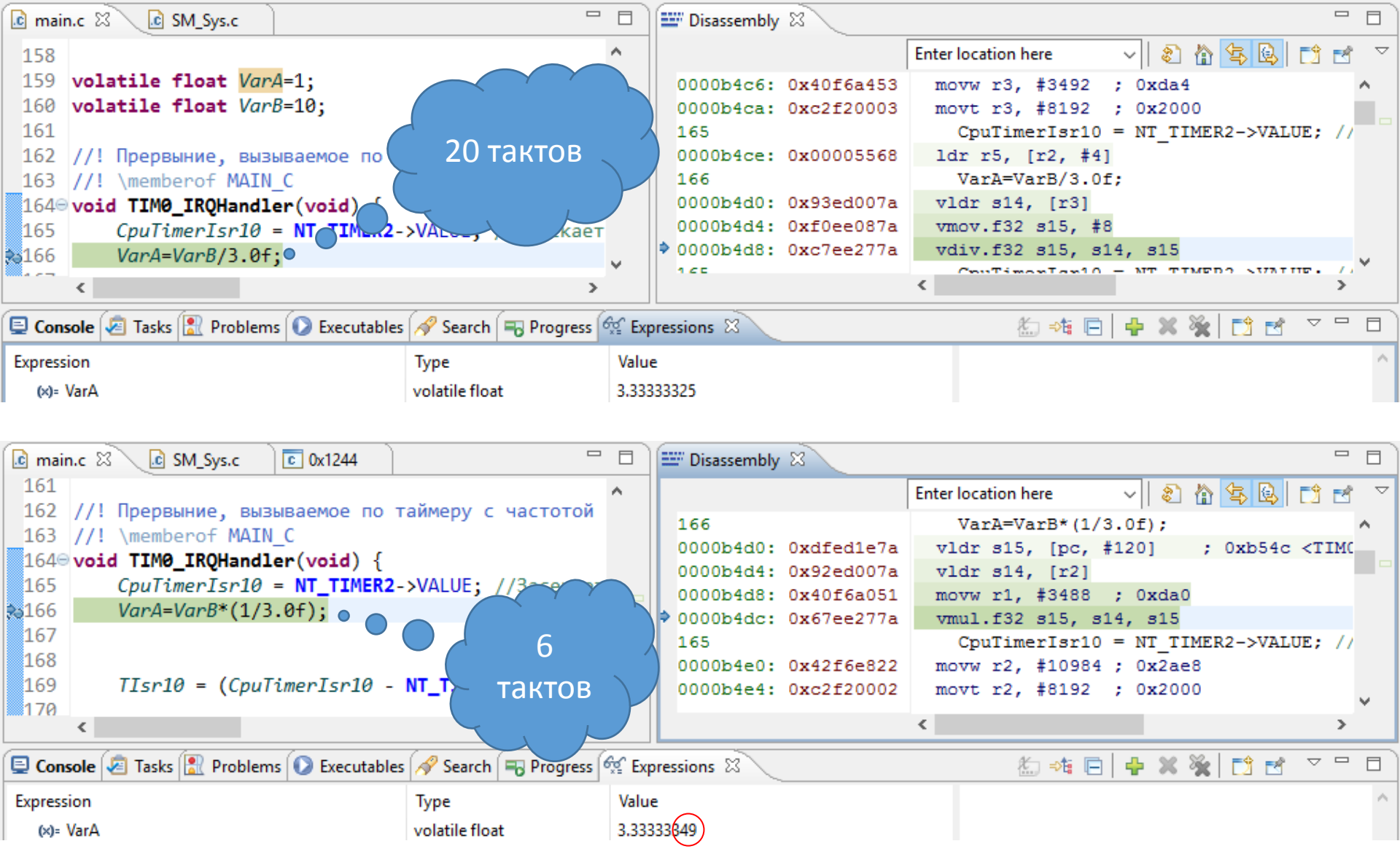
Общие рекомендации повышения производительности

1. При использовании вычислений с фиксированной точкой «проблем» с неиспользованием аппаратной поддержки нет: при включенной оптимизации как правило всё будет считаться оптимально.
2. При использовании формата плавающей точки всегда использовать одинарную точность (float), а не двойную (double): в системах управления электроприводами и источниками питания float более чем достаточен. Если нет, то вы что-то неверно делаете идеологически.
3. При использовании float всегда проверяйте, какой ассемблерный код генерирует компилятор. «Визитная карточка» аппаратных операций float на CortexM4F это использование ассемблерных команд начинающихся с буквы **v**, а также регистров с буквой **s**.
4. Помните, что CortexM4F имеет аппаратную поддержку float операций сложения, вычитания, умножения, деления, квадратного корня, умножения с накоплением и ряд вспомогательных. Если при этих операциях работают не инструкции «v», то ищите проблему!

Общие рекомендации повышения производительности

1. Старайтесь не использовать деление там, где можно обойтись умножением. Умножение выполняется за 1 такт, а деление на порядок дольше (в зависимости от формата данных). Всегда заменяйте деление переменной на константу умножением на обратное число.
2. Если для работы алгоритма деление необходимо, подумайте, как часто меняется величина, на которую вы делите? Если это вводимая в алгоритм константа или параметр, выполните деление в фоновом цикле или один раз при инициализации, подготовьте обратную переменную и на неё умножайте в основном теле алгоритма управления.
3. Если реализуете расчет в целочисленной математике, подумайте, нельзя ли заменить умножение и деление на операцию сдвига. Сдвиг – всегда более быстрая операция.
4. Разделяйте структуру управления на вычисление в разных потоках (прерываниях). Например, контура токов можно реализовать в более быстром прерывании, а всё остальное в более медленном.

Почувствуйте разницу!



А если нет разницы, зачем считать дольше?