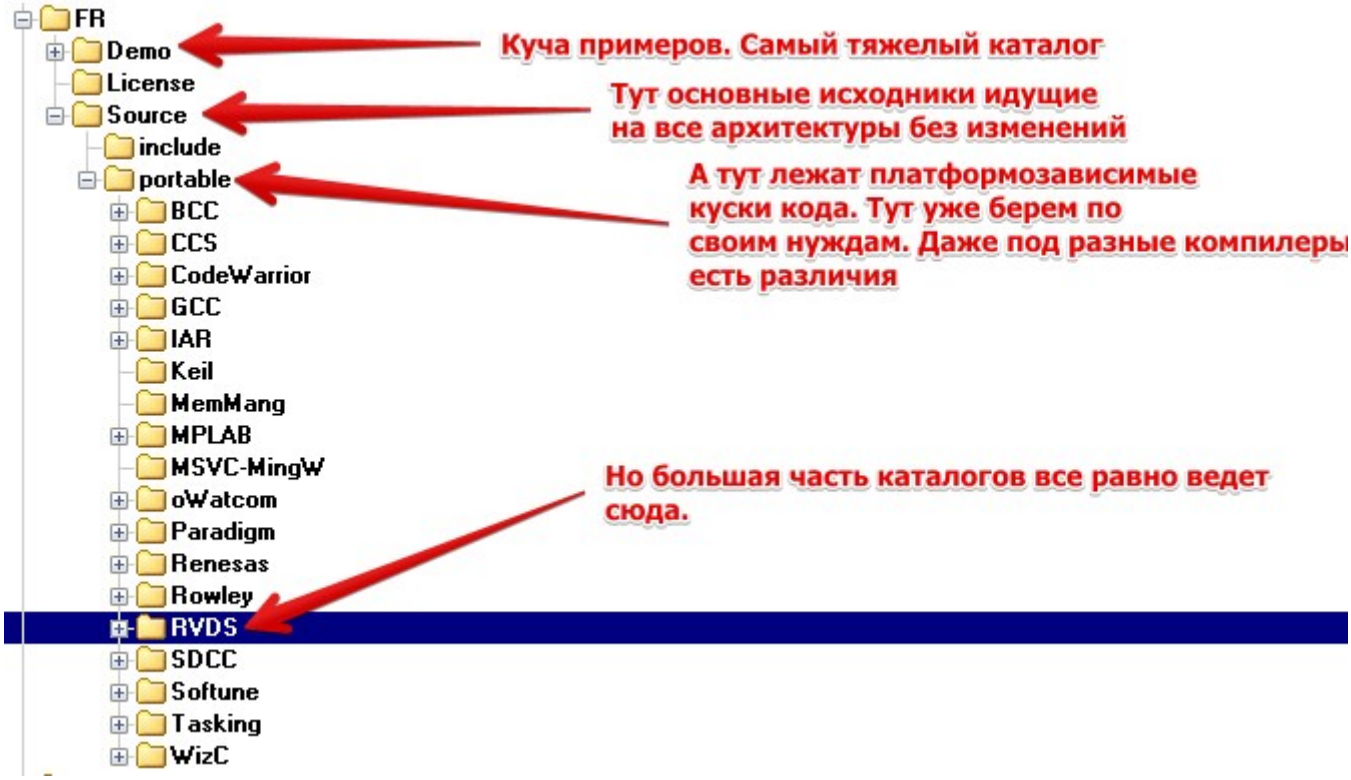
Установка и конфигурация FreeRTOS

ARM. Учебный курс | 25 Июнь 2014 | DI HALT | 49 Comments

На самом деле это скорей интеграция ее в проект. С технической точки зрения выглядит как подключение библиотек. Как той же CMSIS или SPL. Добавляем инклюдники, добавляем файлы в проект и все. Можно взять готовый пример и переколхозить, но в этом случае есть шанс прозевать какие-нибудь детали и получить странные эффекты. Поэтому начну с нуля, в качестве основы будет модуль Pinboard STM32F103C8T6 и Keil uVision. Под него все мы и соберем.

Качаем ОС

Тащим архив с freertos.org. Это довольно толстая солянка где 99% занимают примеры под разные архитектуры. Вот его примерная структура:



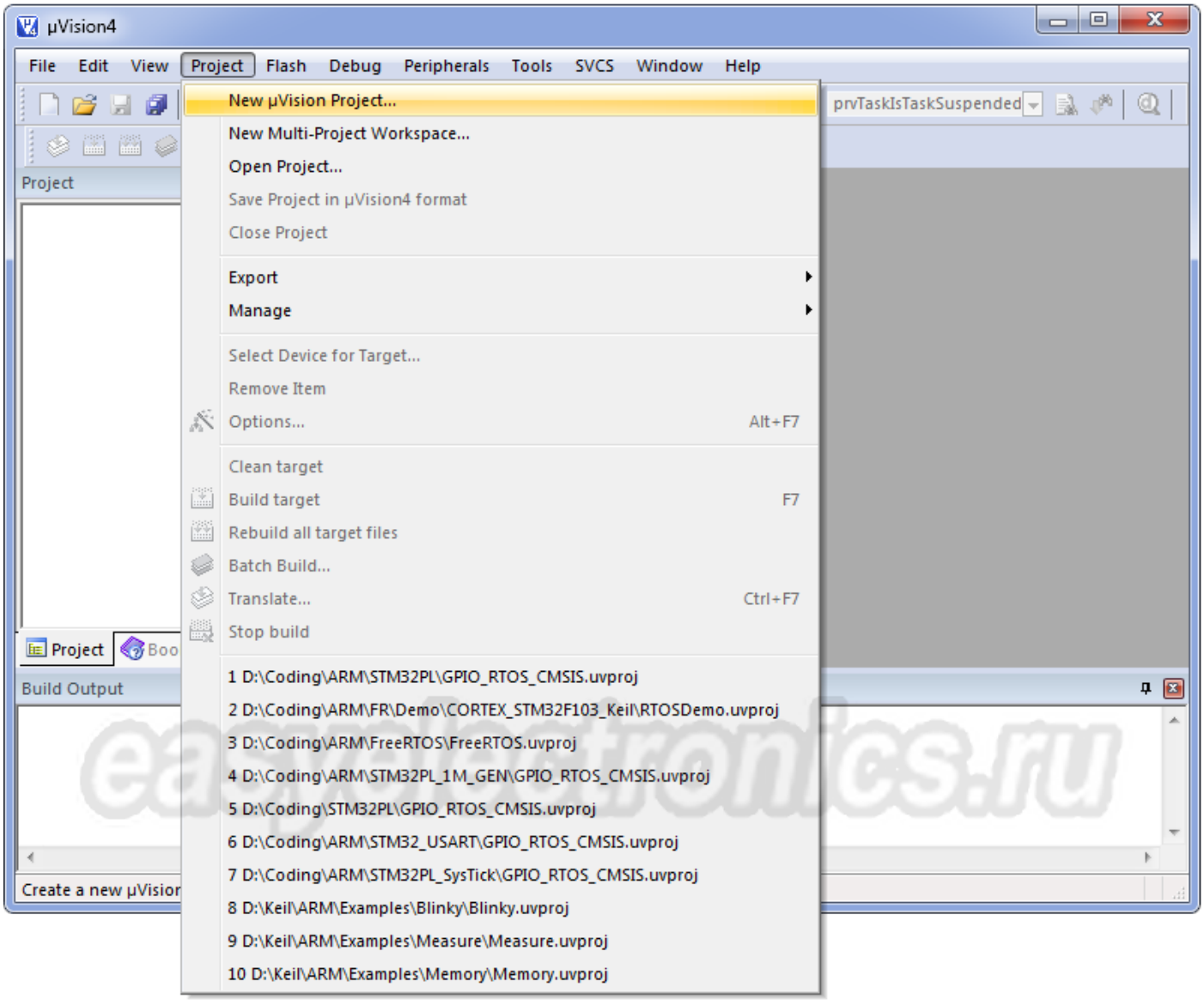
Вся ОС по большей части прячется вот в этих нескольких файлах:

- **queue.c** — функции очередей и мутексов
- **tasks.c** — функции работы с задачами

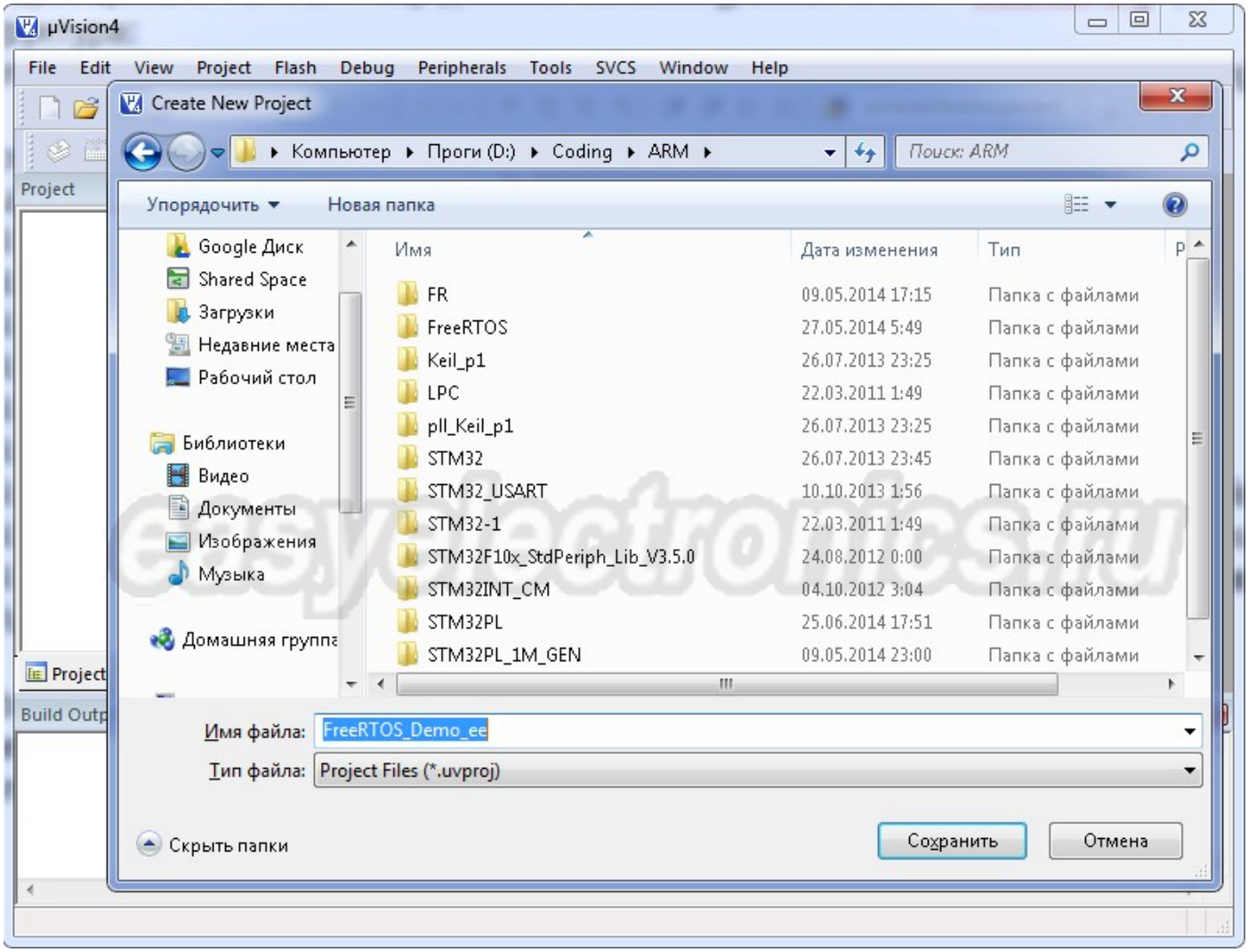
- **timers.c** — функции работы с таймерами
- **croutine.c** — функции работы с сопрограммами
- **event_groups.c** — функции работы с флагами
- **list.c** — тут все для отладки
- **port.c** — платформозависимые параметры. У каждого МК этот файл свой
- **portmacro.h** — настройки платформы. Тоже индивидуальный для каждого типа МК
- **FreeRTOSConfig.h** — настройки ОС. Платформозависимо, а еще зависит от целей и проекта

Создаем проект в Keil

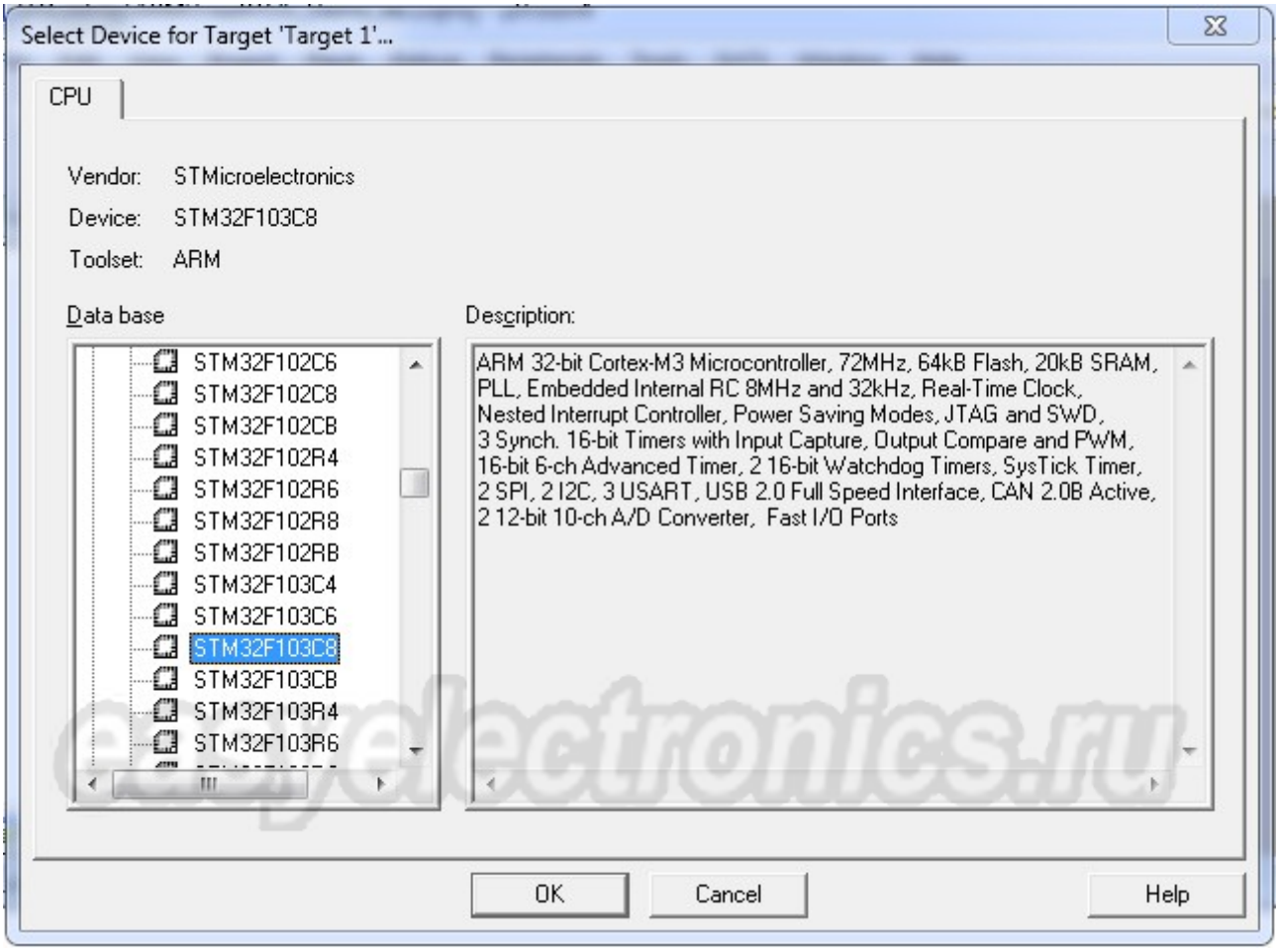
Запускаем Keil и там NewProject:



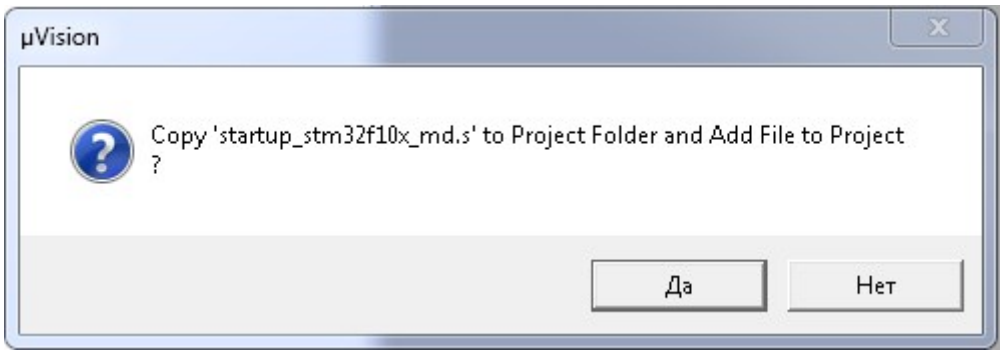
Создаем каталог под проект:



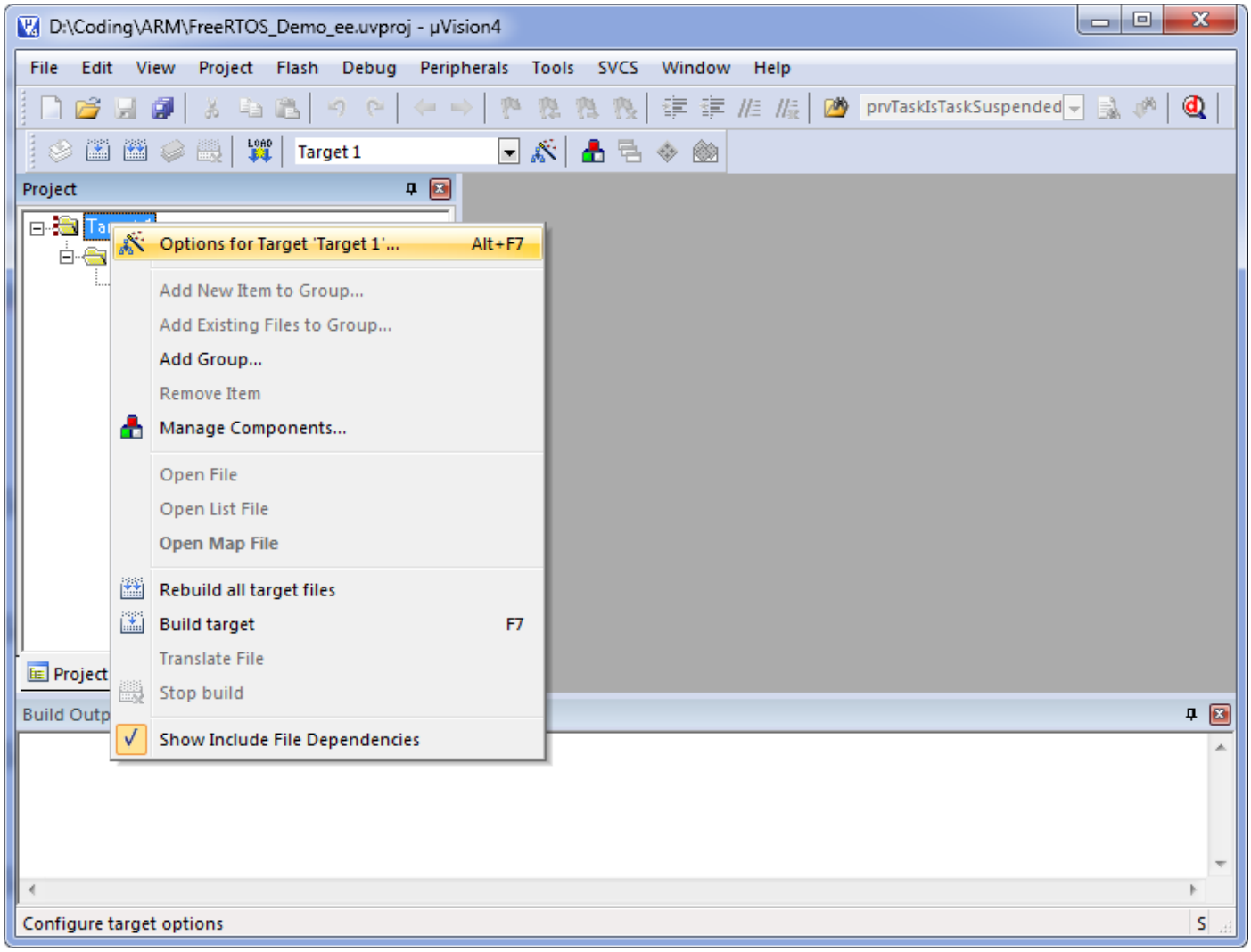
Выбираем проц:



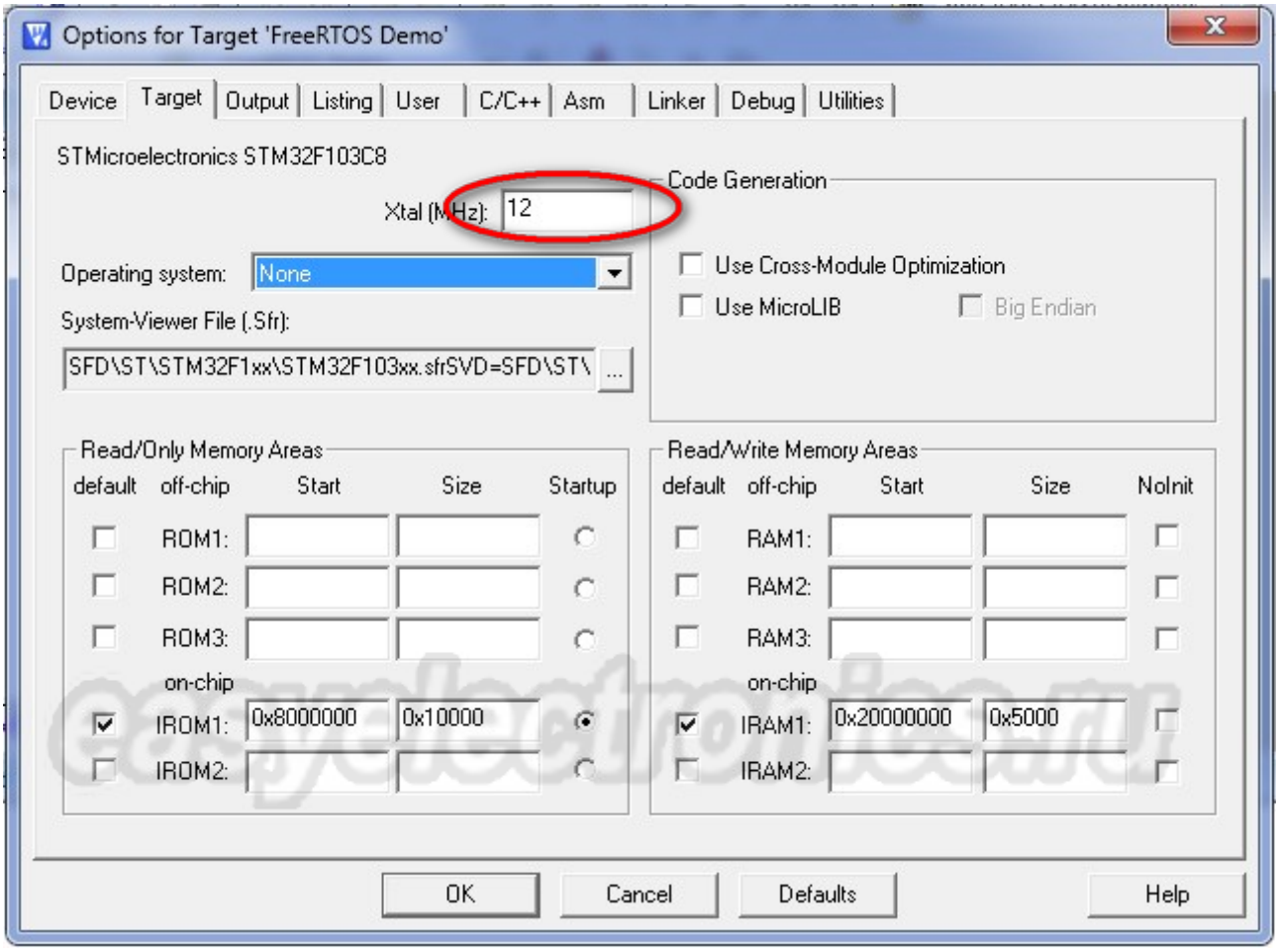
Соглашаемся добавить стартап файлы:



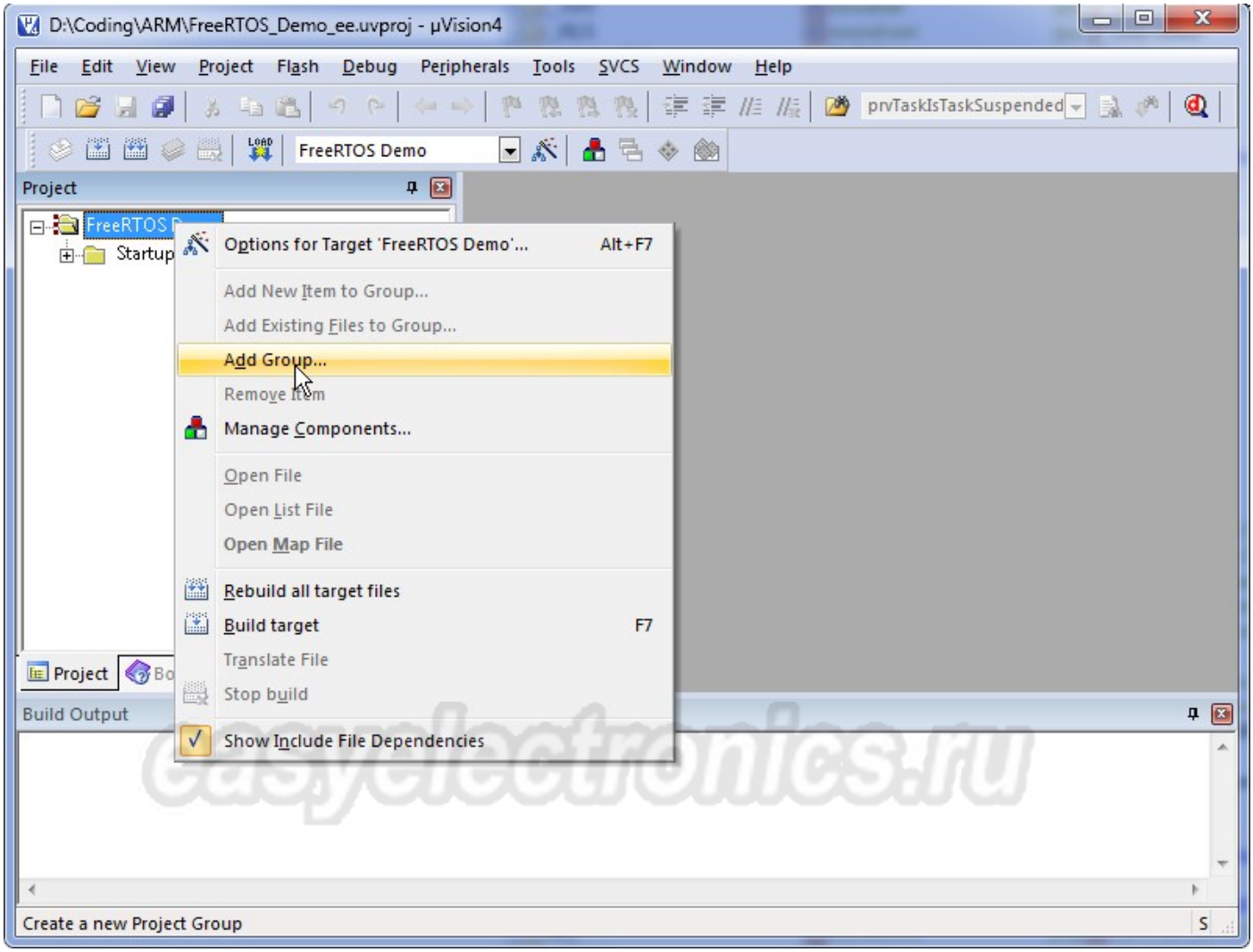
Лезем в настройки проекта:

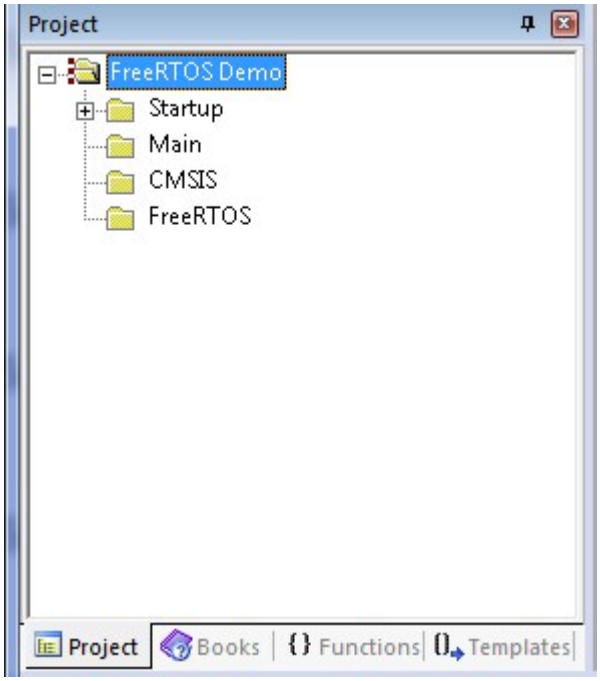


Выставляем там частоту:



Создаем три группы исходников. Для нашего кода, для CMSIS и FreeRTOS. Если юзаете SPL, то и под него можно. Это опционально конечно, но так порядку больше.

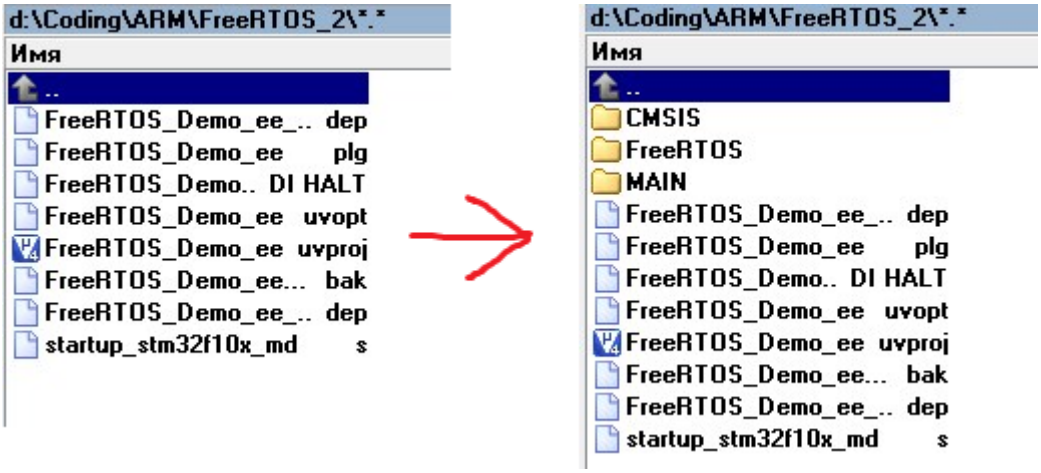




Все это сохраняем и начинаем набивать наш проект файлами:

Файло

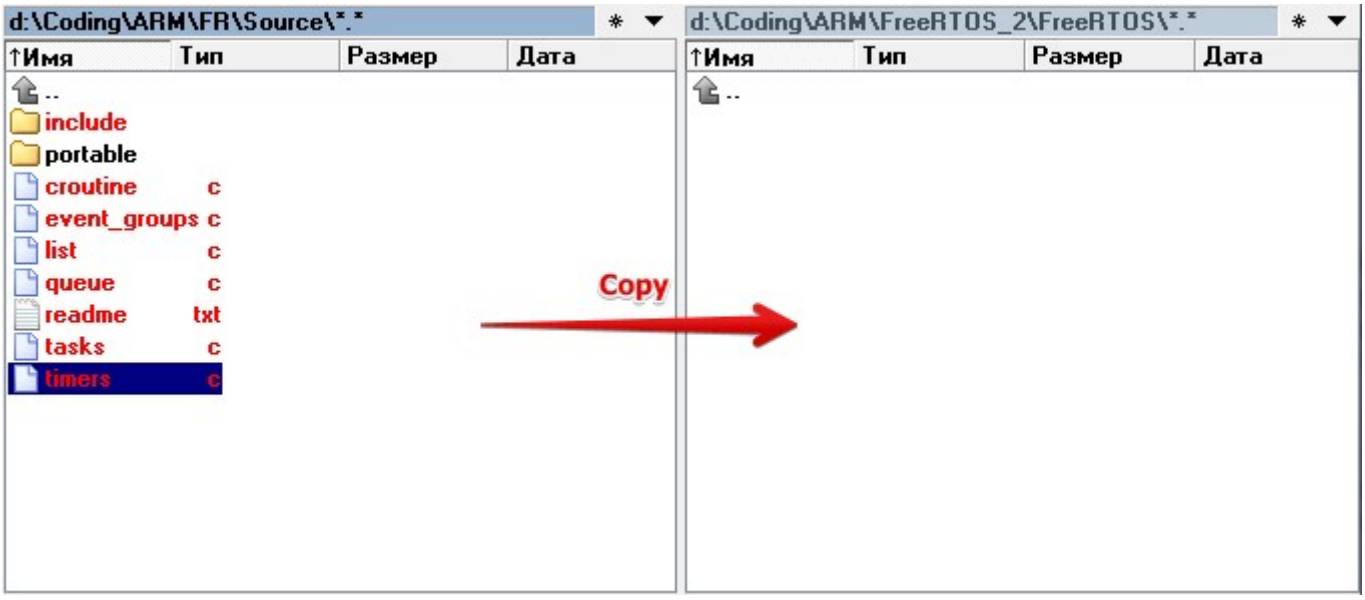
Идем в каталог нашего проекта, видим там помойку служебных файлов проекта Keil и создаем три папки под сорцы, в соответствии с нашими группами:



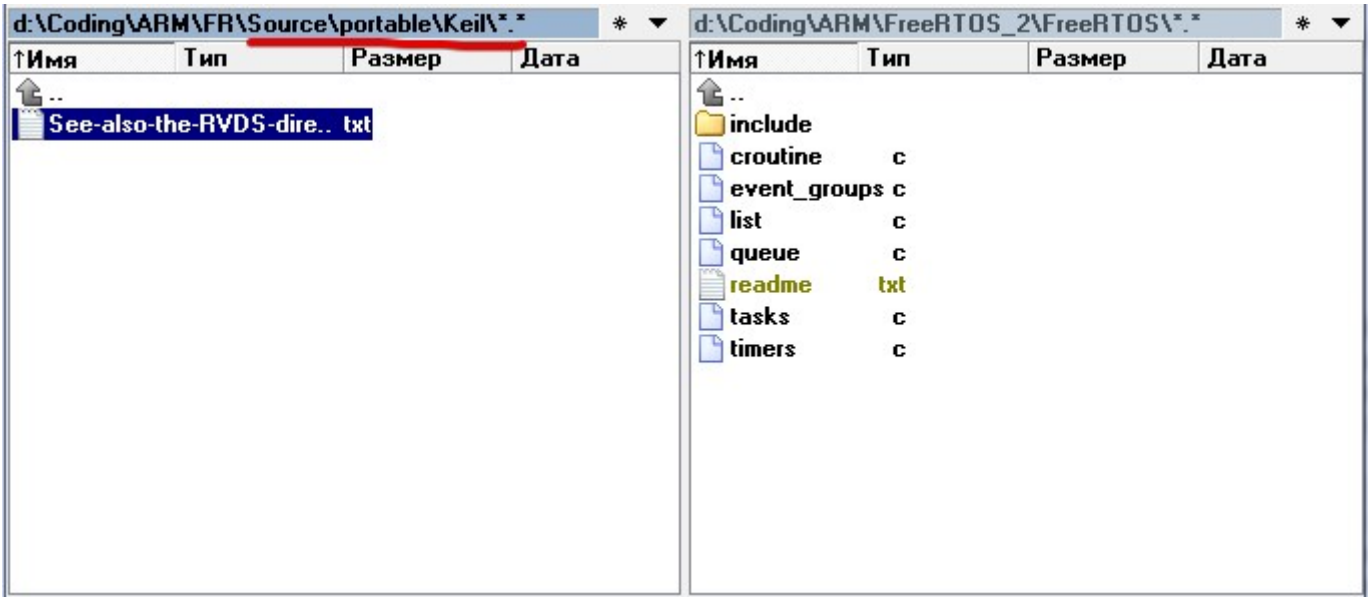
В CMSIS кидаем файлы CMSIS для нашего проекта. Как подцеплять CMSIS я уже расписывал однажды.

В MAIN кладем пустые пока что **main.c**

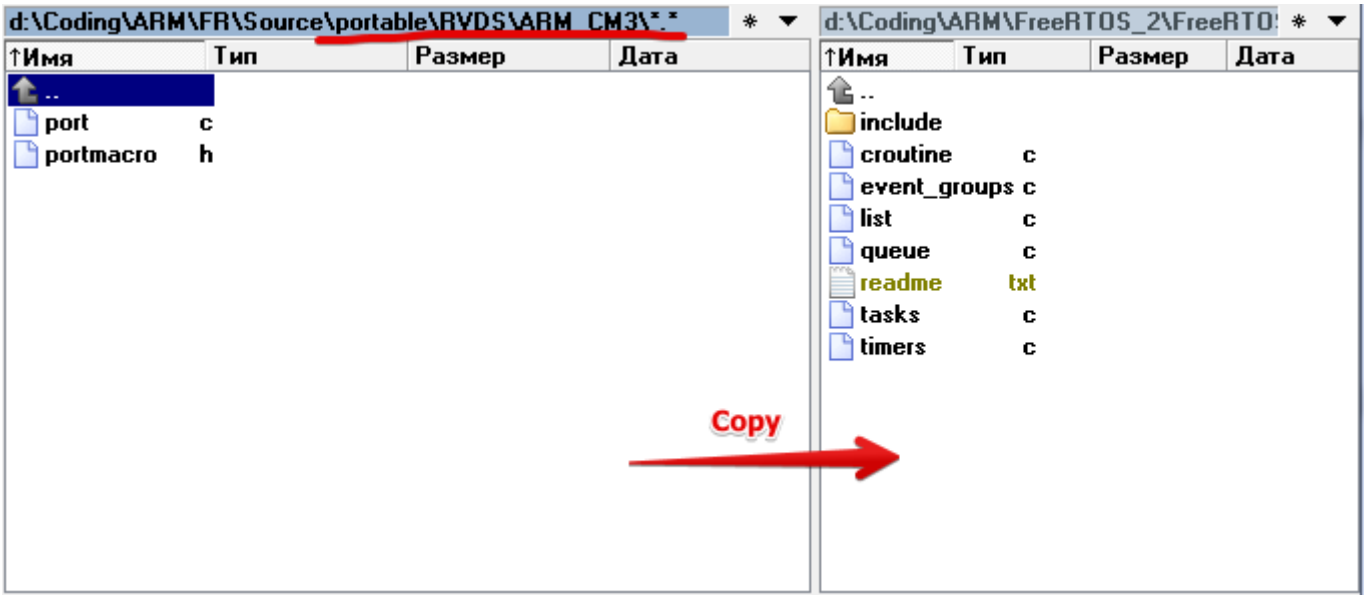
Начинаем копировать необходимые файлы FreeRTOS в ее каталог. Открываем дистрибутив FreeRTOS и тащим все исходники туда, кроме папки Portable:



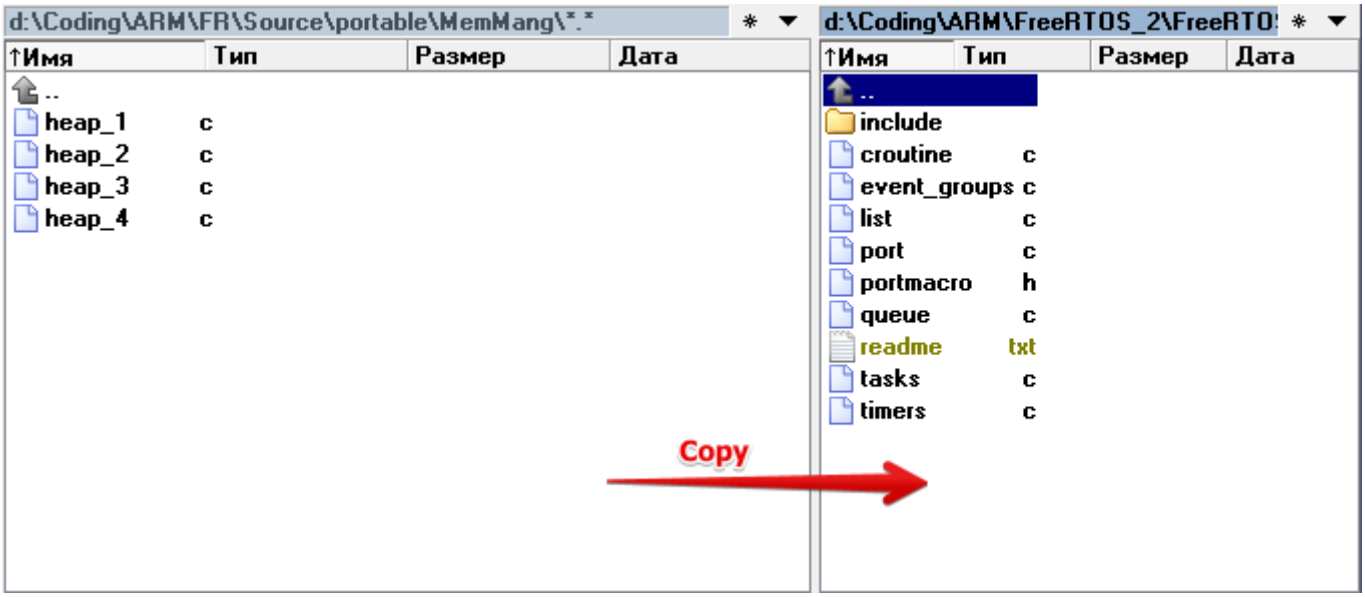
Заходим в Portable и смотрим папку со своей средой. У нас Keil.



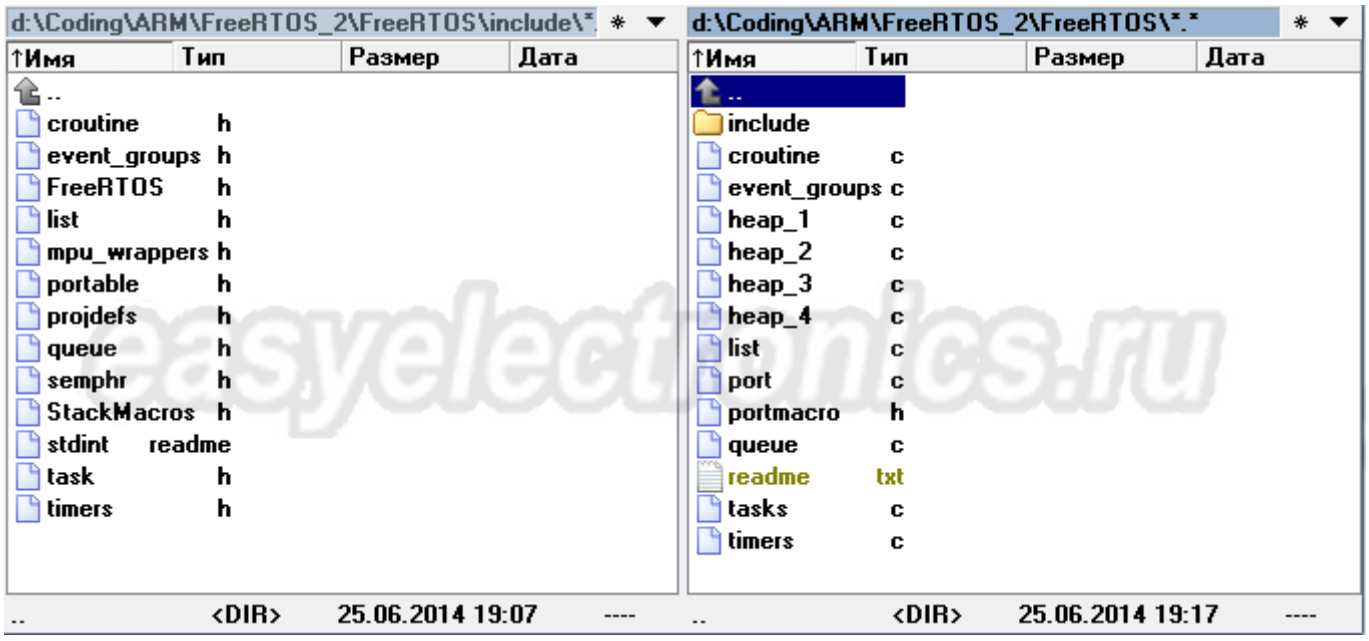
Там нас ждет файл пустышка, намекающий, что все необходимое лежит в папке RVDS. Идем туда, а там лежат папки рассортированные по архитектурам. Нам нужна ARM Cortex M3. Т.е. папка ARM_CM3 и оттуда мы возьмем файлы **portmacro.h** и **port.c**. portmacro.h лучше скопировать в папку include, в кучу ко всем заголовкам.



Также надо не забыть скопировать себе файлы управления памятью **heap_*.c** . Они лежат в Portable/MemMang



В результате у вас будет вот такой набор файлов.



Осталось где-нибудь стырить конфиг :))) Предлагаю утащить его из папки Demo. Лезем в дистрибутив и в папке \Demo\CORTEX_STM32F103_Keil\ находим файл **FreeRTOSConfig.h** его мы копируем куда-нибудь себе. Например в папку MAIN. Все же это уже больше к нашей проге относится чем к ОС. Правда я бы рекомендовал сходить на официальный сайт и добавить в него недостающих строчек. Т.к. он там какой то кастрированный, видимо таскается с дремучих времен, а ОС развивается и появляются новые параметры и их в конфиге может и не быть, а они могут быть критичными или нужными.

FreeRTOSConfig.h — Конфигруция ОС

Открываем этот хидер… Там будет достаточно много комментов, я их вырежу, оставлю только мясо:

Использовать вытесняющую многозадачность?

```
1 #define configUSE_PREEMPTION 1
```

Если этот параметр поставить в 0, то получим кооперативку, с требованием вызывать **taskYIELD()** для отдачи управления диспетчеру.

```
1 #define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
```

Использовать ли платформенно зависимые способы переключения задач. Некоторые контроллеры имеют разные хитрые аппаратные механизмы заточенные под работу с ОС и ОС может их использовать, но для этого ей надо это разрешить.

```
1 #define configUSE_TICKLESS_IDLE 0
```

Использовать ли IDLE без тиков. В этом режиме таймер в IDLE глушится и не генерирует тики зря, полезно малопотребляющих решений.

Есть ли функция перехватывающая IDLE?

```
1 #define configUSE_IDLE_HOOK 0
```

В FreeRTOS мы можем поставить перехват IDLE, т.е. при попадании в Idle будет выполнена некая функция. Например загон проца в спячку или еще какая нужна вещь. Если Hook на

IDLE не используется, то 0.

```
1 #define configUSE_TICK_HOOK    0
```

Аналогичная функция, но про перехват диспетчера. Т.е. можно поставить зацепку за диспетчер и выполнять какую-нибудь фигню каждый тик.

```
1 #define configCPU_CLOCK_HZ     ( ( unsigned long ) 72000000 )
```

Частота на которой будет тикать проц. Нужна для вычисления задержек. Я разгоняю до 72МГЦ потому выставляю тут такое значение.

```
1 #define configTICK_RATE_HZ       ( ( TickType_t ) 1000 )
```

Частота системных тиков ОС.

```
1 #define configMAX_PRIORITIES    ( 5 )
```

Количество приоритетов задач

```
1 #define configMINIMAL_STACK_SIZE    ( ( unsigned short ) 128 )
```

Минимальный размер стека. Вообще он не влияет ни на что кроме тех мест где мы создаем задачи. Это просто макроопределение.

```
1  #define configTOTAL_HEAP_SIZE ( ( size_t ) ( 17 * 1024 ) )
```

Размер суммарной кучи. Области памяти задачи, где лежит и стек и локальные переменные какие то. Т.е. тут мы укзываем сколько вообще у нас места. Зависит от доступной оперативки. У нас в STM32F103C8T6 ее 20кБ, но может быть некий запас под глобальные переменные какие. Потому 3кБ оставляем себе, а остальное отдаем ОС.

```
1  #define configMAX_TASK_NAME_LEN ( 16 )
```

Для отладочных целей можно задавать каждой задаче символическое имя. Это имя естественно ест немного памяти. Вот тут можно указать строку какой длины мы допускаем для этого безобразия.

```
1  #define configUSE_TRACE_FACILITY  0
```

Используем ли мы отладочные примочки ОС.

```
1  #define configUSE_16_BIT_TICKS      0
```

Разрядность таймера ОС. 1—16 бит, 0—32 бита. У STM32 SysTick 24 разрядный, поэтому оставляем тут 0.

```
1  #define configIDLE_SHOULD_YIELD   1
```

Этот параметр определяет поведение задач с приоритетом IDLE болтающихся одновременно с IDLE ядра. Если стоит 1, то ядро будет отдавать управление сразу же как только задача станет готовой к запуску. А если оставить 0, то время будет равномерно делиться между задачами с приоритетом IDLE и IDLE ядра.

```
1 #define configUSE_MUTEXES              1
2 #define configUSE_RECURSIVE_MUTEXES    1
3 #define configUSE_COUNTING_SEMAPHORES  1
```

Это конфигурация на использование мутексов и семафоров. Т.е. можно поотрубать ненужное, чтобы память не ело зря.

```
1 #define configUSE_CO_ROUTINES          0
2 #define configMAX_CO_ROUTINE_PRIORITIES ( 2 )
```

Используются ли сопрограммы и приоритеты для них.

Очень важной опцией идет группа настройки приоритетов прерываний. Я в прошлом посте это описывал, но штука крайне важная, поэтому я продублирую еще раз.

Суть в чем. Есть у нас прерывание диспетчера. И в этом прерывании он что-то делает со своими структурами (очередями, задачами и прочей системной фигней). Но диспетчер это не самая важная вещь на свете, поэтому его часто вешают на самое зачуханное прерывание с самым низким приоритетом. И его все могут перебивать.

Но есть нюанс. Допустим диспетчер что-то делает с очередью, смотрит кого там можно разблокировать по ее состоянию, пишет туда что-то свое, а тут опа прерывание и в нем некая [****]FromISR записала в ту самую очередь. Что у нас будет? У нас будет лажа. Т.к. на выходе из прерывания диспетчер похерит все что предыдущее прерывание туда писало — он то запись до конца не провел. Т.е. налицо классическое нарушение атомарности. Чтобы этого не было, диспетчер на время критических записей запрещает прерывания. Чтобы ему никто не мог помешать делать свое дело. Но запрещает он не все прерывания, а

только определенную группу. Скажем все прерывания с приоритетом (меньше число, старше приоритет) 15 по 11, а 1 по 10 нет. В результате на 1 по 10 мы можем вешать что то ну очень сильно важное и никакой диспетчер это не перебьет. Но пользоваться API RTOS в этих (1-10) прерываниях ни в коем случае уже нельзя — они могут диспетчер скукожить. Для настройки этих групп есть конфиг специальный. Для STM32 он выглядит так:

```
1  /* This is the raw value as per the Cortex-M3 NVIC.  Values can be 255 (lowest) to 0 (1?)
2  (highest). */
3  #define configKERNEL_INTERRUPT_PRIORITY         255
4
5  /* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!! See
6  http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
7  #define configMAX_SYSCALL_INTERRUPT_PRIORITY      191 /* equivalent to 0xb0, or
8  priority 11. */
9
   /* This is the value being used as per the ST library which permits 16 priority values, 0 to
   15.  This must correspond to the configKERNEL_INTERRUPT_PRIORITY setting.  Here
   15 corresponds to the lowest NVIC value of 255.
   Шняга для SPL, короче. Эти утырки там приоритеты по другому обозначают*/
   #define configLIBRARY_KERNEL_INTERRUPT_PRIORITY        15
```

Мы указываем приоритет ядра KERNEL = 255. И задаем планку максимального приоритета для прерываний которые имеют право юзать [****]FromISR API функции = 191. В статье о NVIC я писал, что у STM32 в байте с приоритетом играет роль только старшая тетрада, т.е. у нас есть 16 уровней приоритетов от старшего к младшему: 0x00, 0x10, 0x20, 0x30…0xF0

Т.е. 255 это уровень 0xF0 — самый младший, а 191 уровень 0xB0 и, таким образом, все прерывания в которых мы можем использовать API фукнции должны быть сконфигурированы с приоритетом от 0xF0 до 0xB0, не старше. Иначе будет трудноуловимый глюк. Прерывания же не использующие API могут быть с каким угодно приоритетом от самого низкого до самого старшего.

Следующая группа позволяет включать-выключать ненужные функции, сокращая размер который ОС занимает во флеше.

```
#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_xResumeFromISR 1
#define INCLUDE_vTaskDelayUntil 1
#define INCLUDE_vTaskDelay 1
#define INCLUDE_xTaskGetSchedulerState 1
#define INCLUDE_xTaskGetCurrentTaskHandle 1
#define INCLUDE_uxTaskGetStackHighWaterMark 0
#define INCLUDE_xTaskGetIdleTaskHandle 0
#define INCLUDE_xTimerGetTimerDaemonTaskHandle 0
#define INCLUDE_pcTaskGetTaskName 0
#define INCLUDE_eTaskGetState 0
#define INCLUDE_xEventGroupSetBitFromISR 1
#define INCLUDE_xTimerPendFunctionCall 0
```

И, напоследок, мы впишем в конфиг следующие строки:

```
#define xPortSysTickHandler SysTick_Handler
#define xPortPendSVHandler PendSV_Handler
#define vPortSVCHandler SVC_Handler
```

Тем самым, мы привяжем вектора прерываний на диспетчер ОС. Как вы можете видеть SysTick используется как таймер службы диспетчера. У другого МК тут можно повесить Другой таймер.

Более подробно можно прочесть в официальной доке, правда на варварском наречии, но нам не привыкать.

Вот что получилось в итоге:

```
1  /*
2     FreeRTOS V8.0.1 - Copyright (C) 2014 Real Time Engineers Ltd.
3     All rights reserved
4     VISIT http://www.FreeRTOS.org TO ENSURE YOU ARE USING THE LATEST
5  VERSION.
6     1 tab == 4 spaces!
7  */
8
9  #ifndef FREERTOS_CONFIG_H
10 #define FREERTOS_CONFIG_H
11
12
13 #define configUSE_PREEMPTION                    1
14 #define configUSE_PORT_OPTIMISED_TASK_SELECTION      0
15 #define configUSE_TICKLESS_IDLE                 0
16 #define configCPU_CLOCK_HZ              ( ( unsigned long ) 72000000 )
17 #define configTICK_RATE_HZ              ( ( TickType_t ) 1000 )
18 #define configMAX_PRIORITIES            5
19 #define configMINIMAL_STACK_SIZE        128
20 #define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 17 * 1024 ) )
21 #define configMAX_TASK_NAME_LEN             16
22 #define configUSE_16_BIT_TICKS              0
23 #define configIDLE_SHOULD_YIELD             1
24 #define configUSE_MUTEXES               1
25 #define configUSE_RECURSIVE_MUTEXES             1
26 #define configUSE_COUNTING_SEMAPHORES       1
27 #define configUSE_ALTERNATIVE_API           0 /* Deprecated! */
28 #define configQUEUE_REGISTRY_SIZE           10
29 #define configUSE_QUEUE_SETS                0
30 #define configUSE_TIME_SLICING              0
31 #define configUSE_NEWLIB_REENTRANT          0
32 #define configENABLE_BACKWARD_COMPATIBILITY      0
33
34 /* Hook function related definitions. */
35 #define configUSE_IDLE_HOOK             0
36 #define configUSE_TICK_HOOK             0
37 #define configCHECK_FOR_STACK_OVERFLOW          0
38 #define configUSE_MALLOC_FAILED_HOOK            0
39
40 /* Run time and task stats gathering related definitions. */
41 #define configGENERATE_RUN_TIME_STATS           0
42 #define configUSE_TRACE_FACILITY        0
43 #define configUSE_STATS_FORMATTING_FUNCTIONS    0
```

```
44
45      /* Co-routine related definitions. */
46      #define configUSE_CO_ROUTINES                   0
47      #define configMAX_CO_ROUTINE_PRIORITIES  2
48
49      /* Software timer related definitions. */
50      #define configUSE_TIMERS                1
51      #define configTIMER_TASK_PRIORITY               3
52      #define configTIMER_QUEUE_LENGTH                10
53      #define configTIMER_TASK_STACK_DEPTH            configMINIMAL_STACK_SIZE
54
55      /* Interrupt nesting behaviour configuration. */
56      #define configKERNEL_INTERRUPT_PRIORITY         255
57      #define configMAX_SYSCALL_INTERRUPT_PRIORITY    191
58
59
60
61
62      /* Set the following definitions to 1 to include the API function, or zero
63      to exclude the API function. */
64
65      #define INCLUDE_vTaskPrioritySet                1
66      #define INCLUDE_uxTaskPriorityGet               1
67      #define INCLUDE_vTaskDelete                     0
68      #define INCLUDE_vTaskCleanUpResources           1
69      #define INCLUDE_vTaskSuspend                    1
70      #define INCLUDE_vTaskDelayUntil                 1
71      #define INCLUDE_vTaskDelay                      1
72      #define INCLUDE_xResumeFromISR                  1
73      #define INCLUDE_xTaskGetSchedulerState          1
74      #define INCLUDE_xTaskGetCurrentTaskHandle       0
75      #define INCLUDE_uxTaskGetStackHighWaterMark     0
76      #define INCLUDE_xTaskGetIdleTaskHandle          0
77      #define INCLUDE_xTimerGetTimerDaemonTaskHandle  0
78      #define INCLUDE_pcTaskGetTaskName               0
79      #define INCLUDE_eTaskGetState                   0
80      #define INCLUDE_xEventGroupSetBitFromISR        1
81      #define INCLUDE_xTimerPendFunctionCall          1
82
83
84
85
86      #define xPortSysTickHandler SysTick_Handler
```

```
87 #define xPortPendSVHandler PendSV_Handler
88 #define vPortSVCHandler SVC_Handler
89
90
   #endif /* FREERTOS_CONFIG_H */
```

Выбор модели памяти

Помните мы там копировали несколько файлов heap_*.c? Я еще писал, что они определяют управление памятью. Там у нас четыре возможные модели:

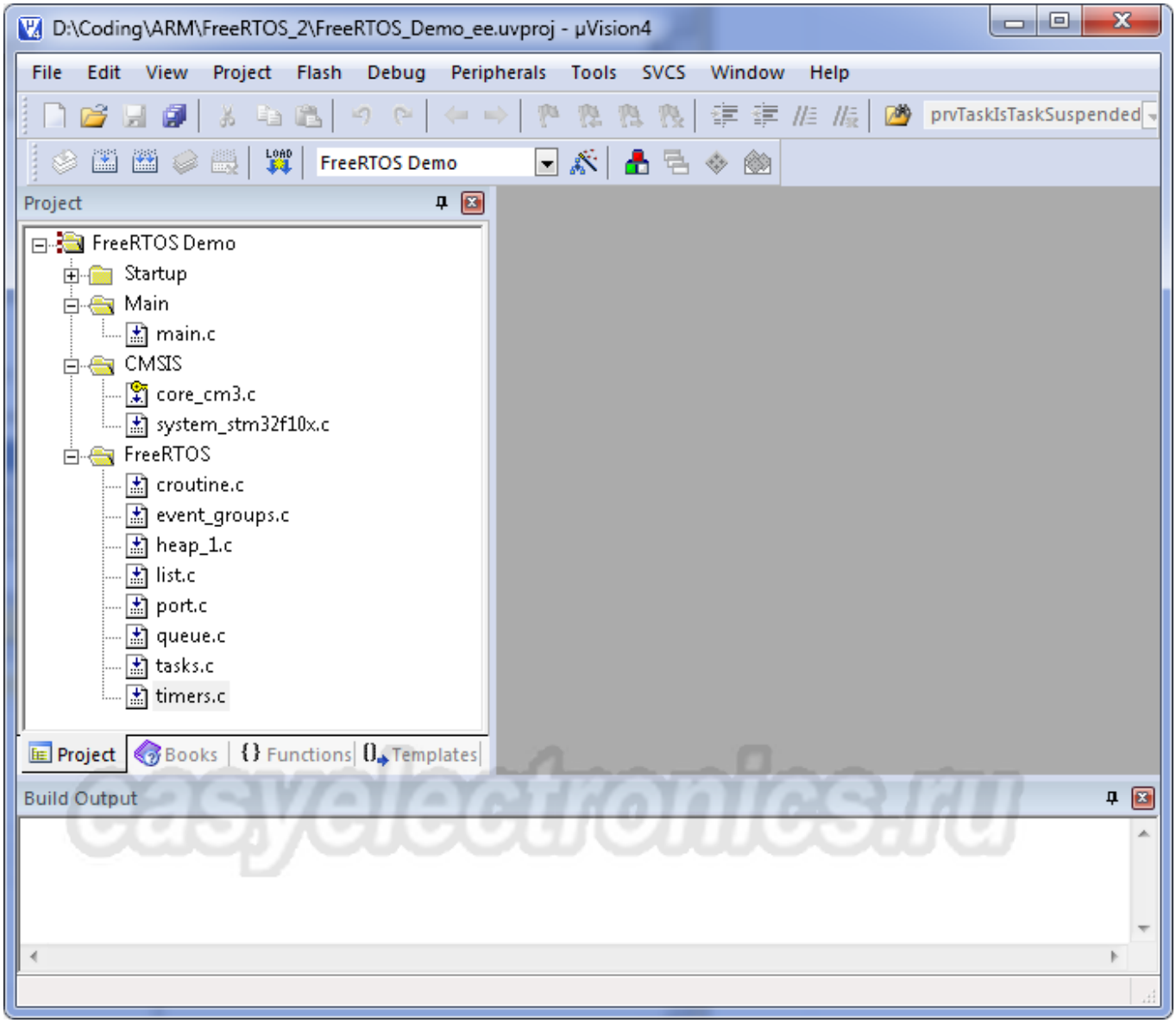
- heap_1.c — собственный хитрый Malloc без Free. Т.е. мы можем создавать задачи (семафоры, очереди и т.д.), но не можем удалять их, чтобы освободить память. В принципе, для не слишком замороченных программ его хватает чуть более чем всегда. Задачи/очереди/семафоры всякие обычно создаются единожды и редко уничтожаются.
- heap_2.c — динамичная фрагментированная память. Т.е. память динамически выделяется, освобождается, но при этом участки памяти получаются фрагментированными, с дырками на месте освободившейся памяти. И мы не сможем выделить большой кусок памяти, пусть даже у нас будет куча маленьких свободных секторов суммарно нужного обьема. А если новая задача укладывается в какой-либо из этих просветов, то будет выбран наиболее подходящий по размеру свободный кусок и задача развернется там.
- heap_3.c — использован классический для Си Malloc/Free механизм с небольшой допилкой для невозожности запуска его из двух разных потоков.
- heap_4.c — Динамическое выделение памяти, может дефрагментировать память, чтобы получить нужный кусок. Разумеется за все приходится платить. Временем в первую очередь.

Подробней читайте у Курница и в официальной доке.

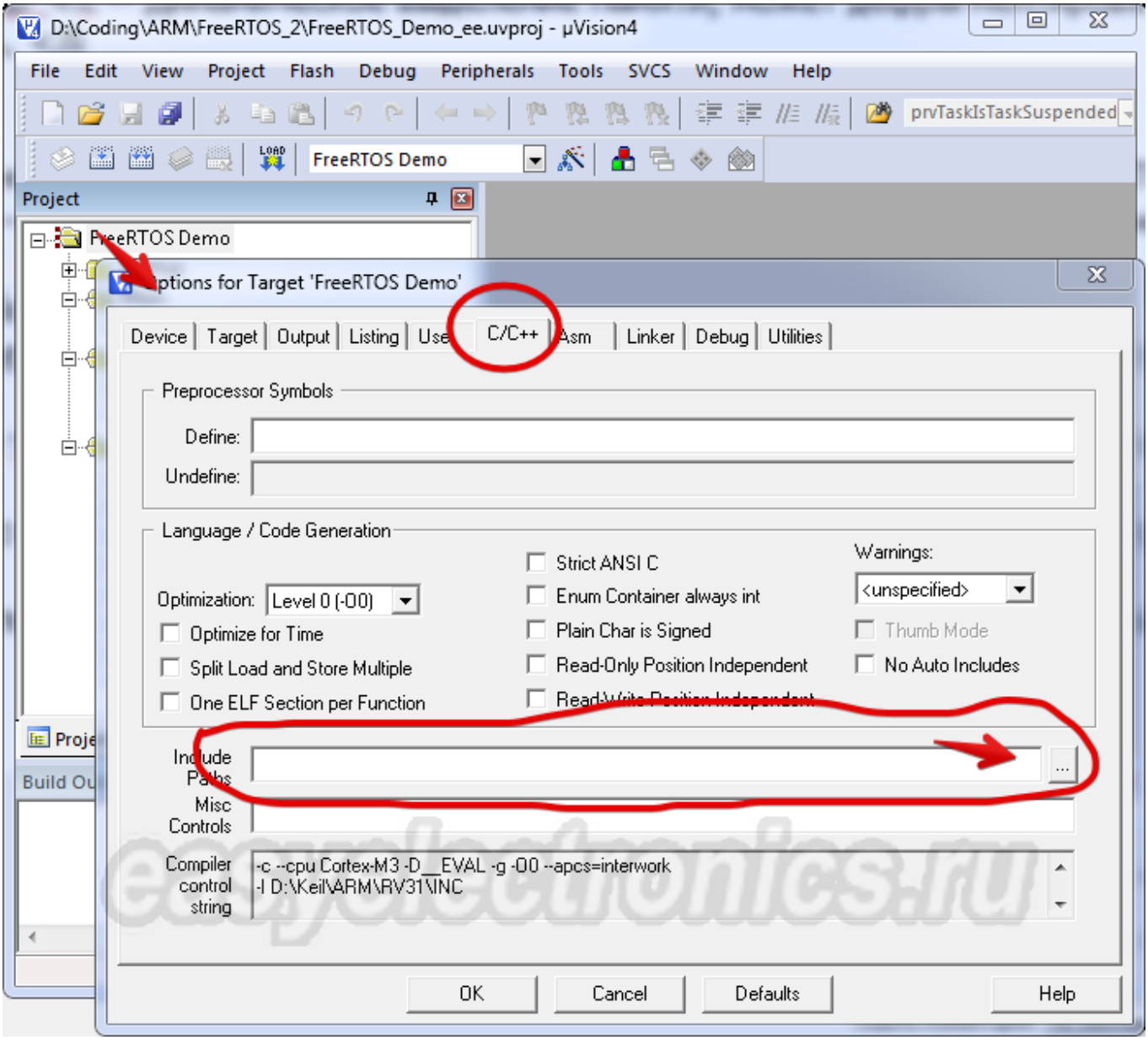
Сборка проекта

Конфигурация завершена. Теперь давайте соберем наш проект в компилируемый вид.

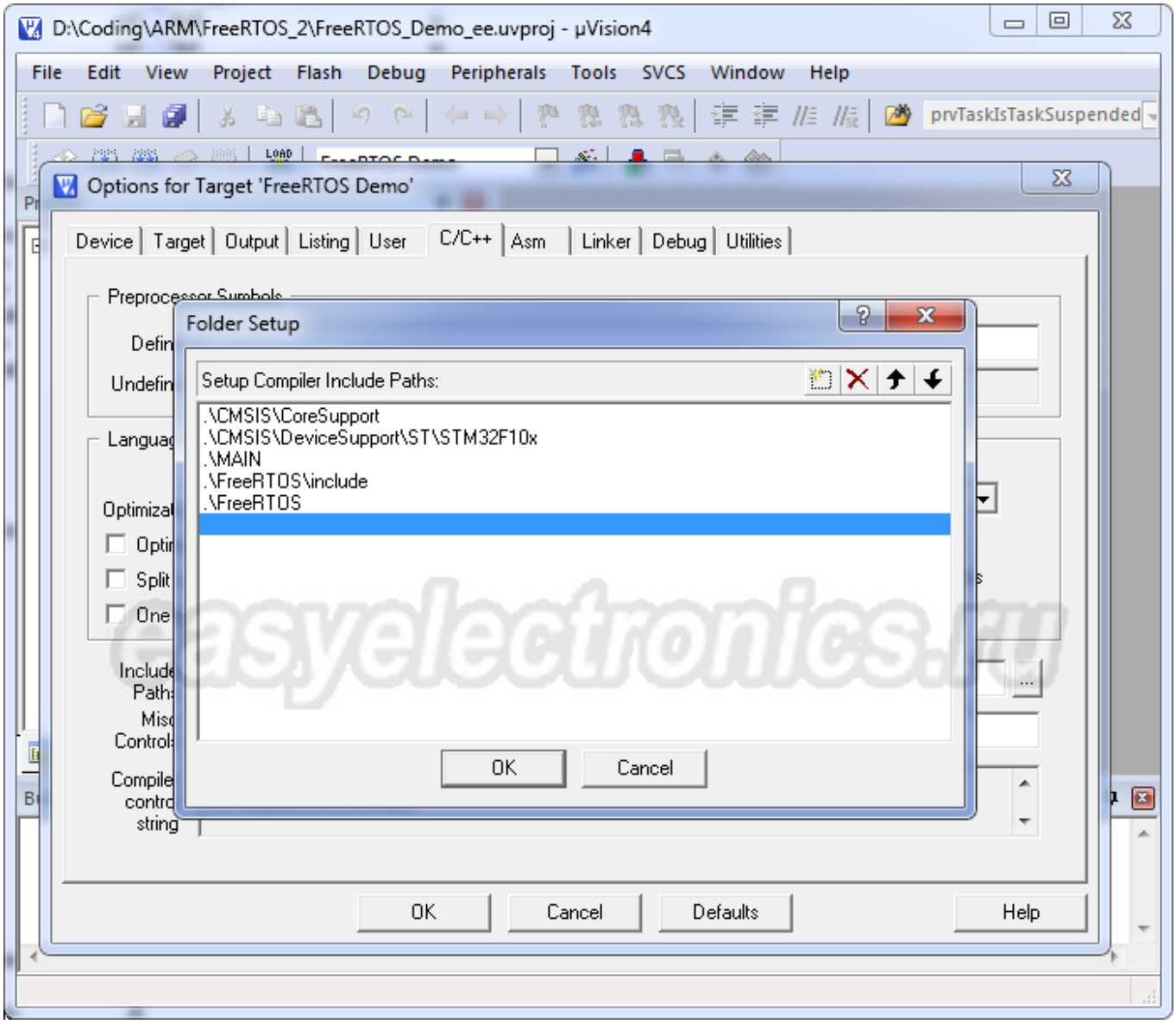
В кейле тыкаем на наши группы в дереве каталогов и добавляем все *.C файлы которые у нас есть. Получится должно примерно так:



Обратите внимание на то, что я добавил только один **heap_1.c** файл. Т.е. я, тем самым, выбрал 1-ю модель распределения памяти. Следующим шагом добавляем пути:



Это в опциях проекта.



Теперь открываем наш Main.c и копируем туда код простейшей моргалки, но уже на FreeRTOS

```
1  #define F_CPU 8000000UL
2  #include "stm32f10x.h"
3  #include "FreeRTOS.h"
4  #include "task.h"
5  #include "queue.h"
6
7  // Отдадочная затычка. Сюда можно вписать код обработки ошибок.
8  #define   ERROR_ACTION(CODE,POS)          do{}while(0)
9
10
11 // Задача моргалка. Просто так. Мигает диодиком на порту LED3
12 void vBlinker (void *pvParameters)
```

```
13  {
14   while(1)
15   {
16          GPIOB->BSRR = GPIO_BSRR_BR5;                // Сбросили бит.
17          vTaskDelay(600);                    // Выдержка 600мс
18          GPIOB->BSRR = GPIO_BSRR_BS5;                // Установили бит.
19          vTaskDelay(20);                             // Выдержка 20мс
20   }
21  }
22
23
24  int main(void)
25  {
26  // Инициализация всего. Ну почти всего, тут только всякие таймеры
27  стартуют, а всякую периферию я последнее время
28  // предочитаю инициализировать непосредственно в задачах где она
29  используется. При старте.
30  SystemInit();
31  RCC->APB2ENR      |= RCC_APB2ENR_IOPBEN;
32
33  // Конфигурируем CRL регистры.
34  GPIOB->CRL        &= ~GPIO_CRL_CNF5;        // Сбрасываем биты CNF для
35  бита 5. Режим 00 - Push-Pull
36  GPIOB->CRL        |= GPIO_CRL_MODE5_0;       // Выставляем бит MODE0 для
37  пятого пина. Режим MODE01 = Max Speed 10MHz
38
39
40
41
42  if(pdTRUE != xTaskCreate(vBlinker,"Blinker",
43  configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, NULL))
44  ERROR_ACTION(TASK_NOT_CREATE,0);
45  // Создаем задачи. Если при создании задачи возвращенный параметр не TRUE,
46  то обрабатываем ошибку.
47  // vBlinker                - это имя функции которая будет задачей.
48  // "Blinker"               - текстовая строка для отладки. Может быть любой,
49  но длина ограничина в конфиге ОС. Ну и память она тоже ест немного.
50  // configMINIMAL_STACK_SIZE    - размер стека для задачи. Определяется
51  опытным путем. В данном случае стоит системный минимум ,т.к.
52  //                      Задачи используют очень мало памяти. Должно хватить.
53  // NULL                 - передаваемый pvParameters в задачу. В данном случае не
54  нужен, потому NULL. Но можно задаче при создании передевать
55  //                      Разные данные, для инициализации или чтобы различать
```

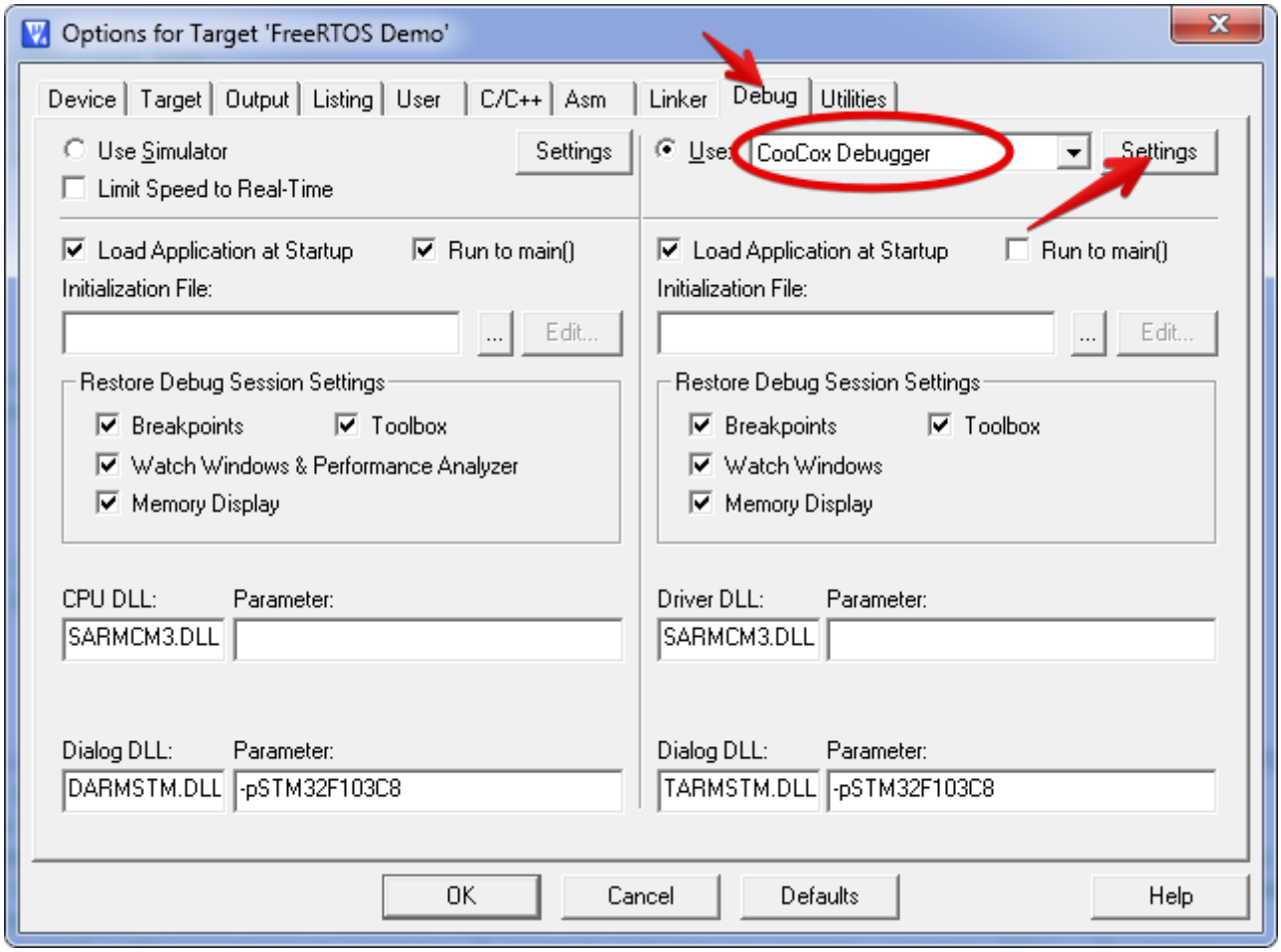
```
56 две копии задачи между собой.
57 // tskIDLE_PRIORITY + 1         - приоритет задачи. В данный момент выбран минимально возможный. На 1 пункт выше чем IDLE
// NULL                - тут вместо NULL можно было вписать адрес переменной типа xTaskHandle в которую бы упал дескриптор задачи.
//                      Например так: xTaskCreate(vBlinker,"Blinker",configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, &xBlinkerTask);
//                      где xBlinkerTask это глобальная переменная обьявленная как xTaskHandle xBlinkerTask; глобальная - чтобы ее везде
//                      было видно. Отовсюду она была доступна. Но можно и как static обьявить или еще каким образом передать хэндл.
//                      И зная эту переменную и то что там дескриптор этой задачи мы могли бы из другой задачи ее грохнуть, поменять
//                      приоритет или засуспендить. В общем, управлять задачей. Но в моем примере это не требуется.
//                      Остальные аналогично.

// Запускаем диспетчер и понеслась.
vTaskStartScheduler();
}
```

После чего все можно скомпилировать и запустить в отладчике-симуляторе. Сами увидите как оно все крутится и вертится. Но это не интересно, лучше прогнать в железе.

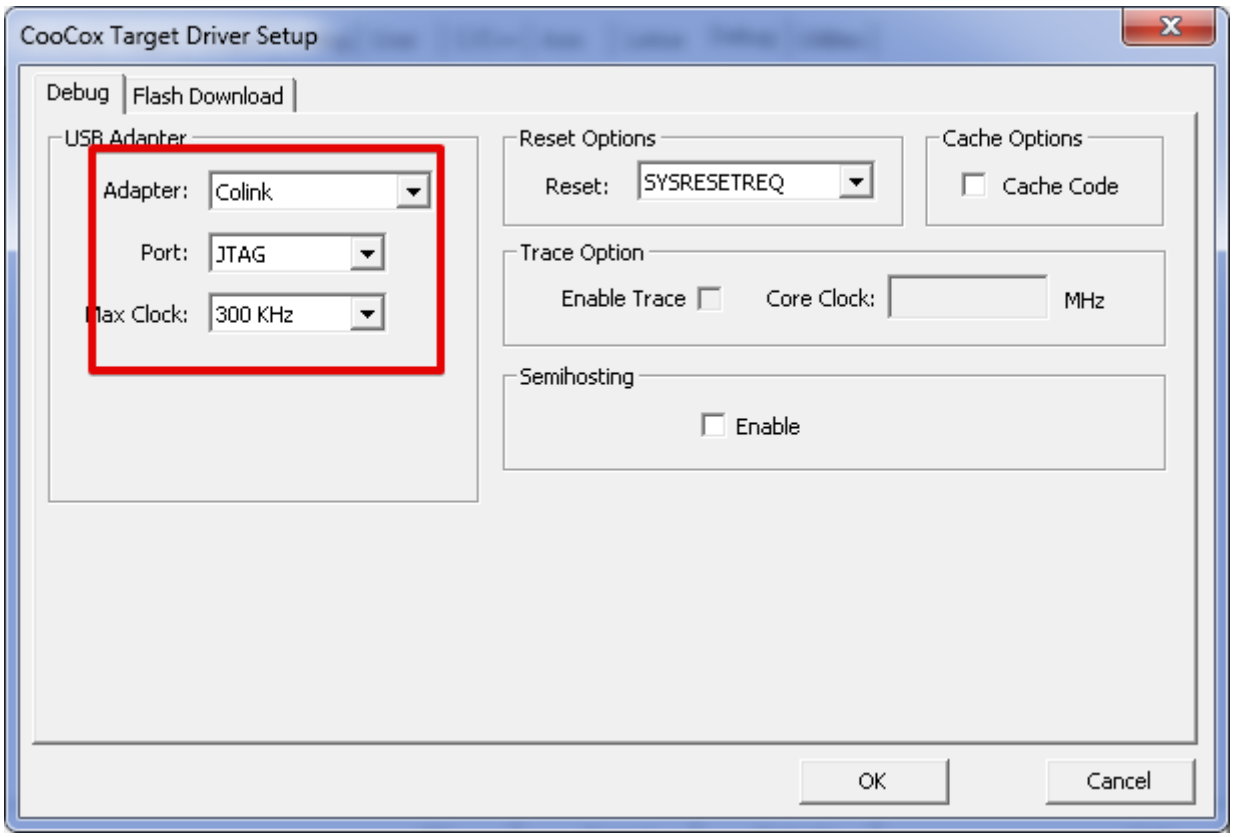
Конфигурируем отладчик

Осталась самая малость — сконфигурировать JTAG отладчик, который у вас поставляется вместе с Pinboard II + STM32. Если забыли как как это делается, то я напомню:

Открываете свойства проекта. Идете в раздел Debug:

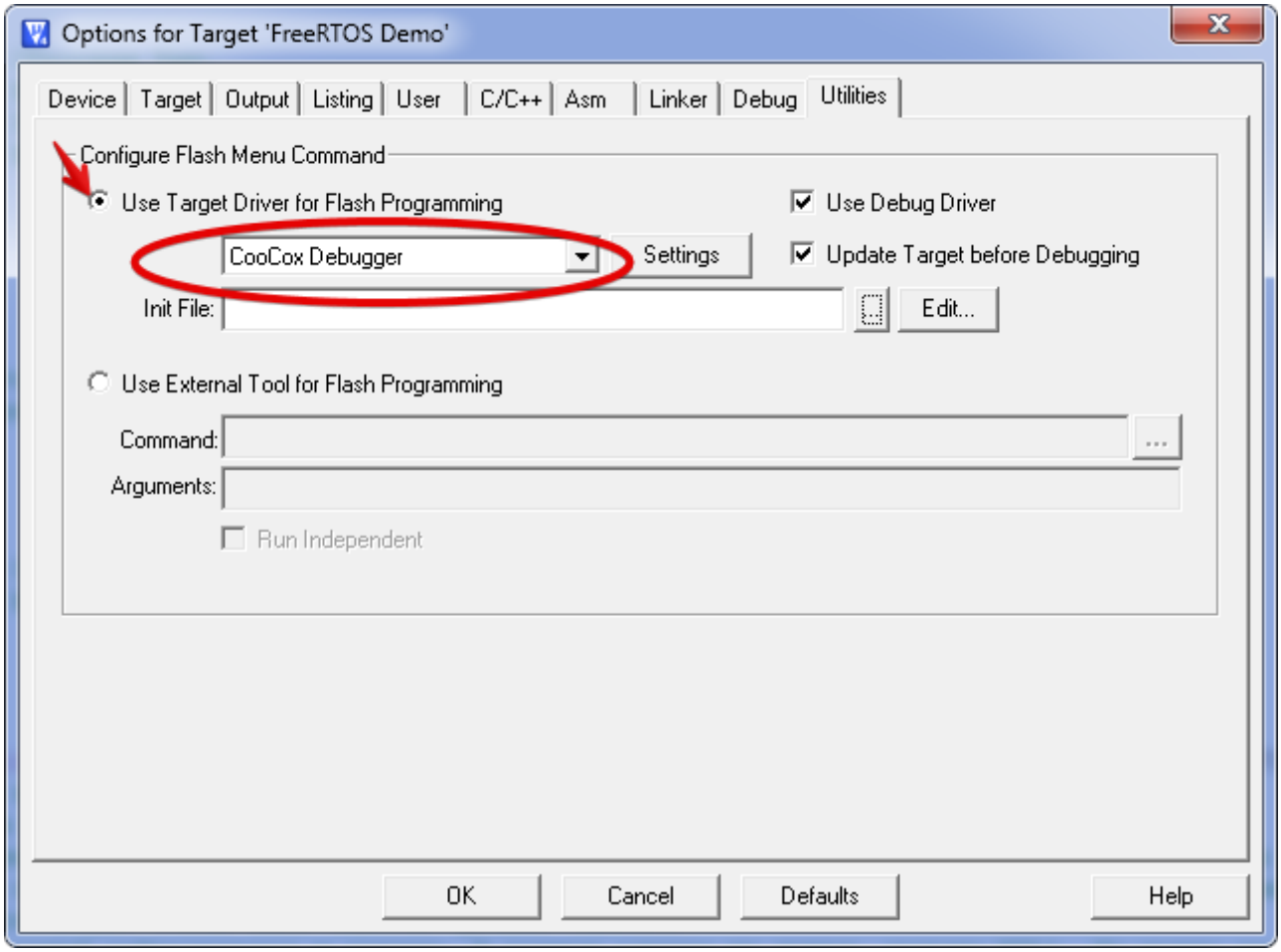


Там выбираете CooCox Debugger и жмете Settings:

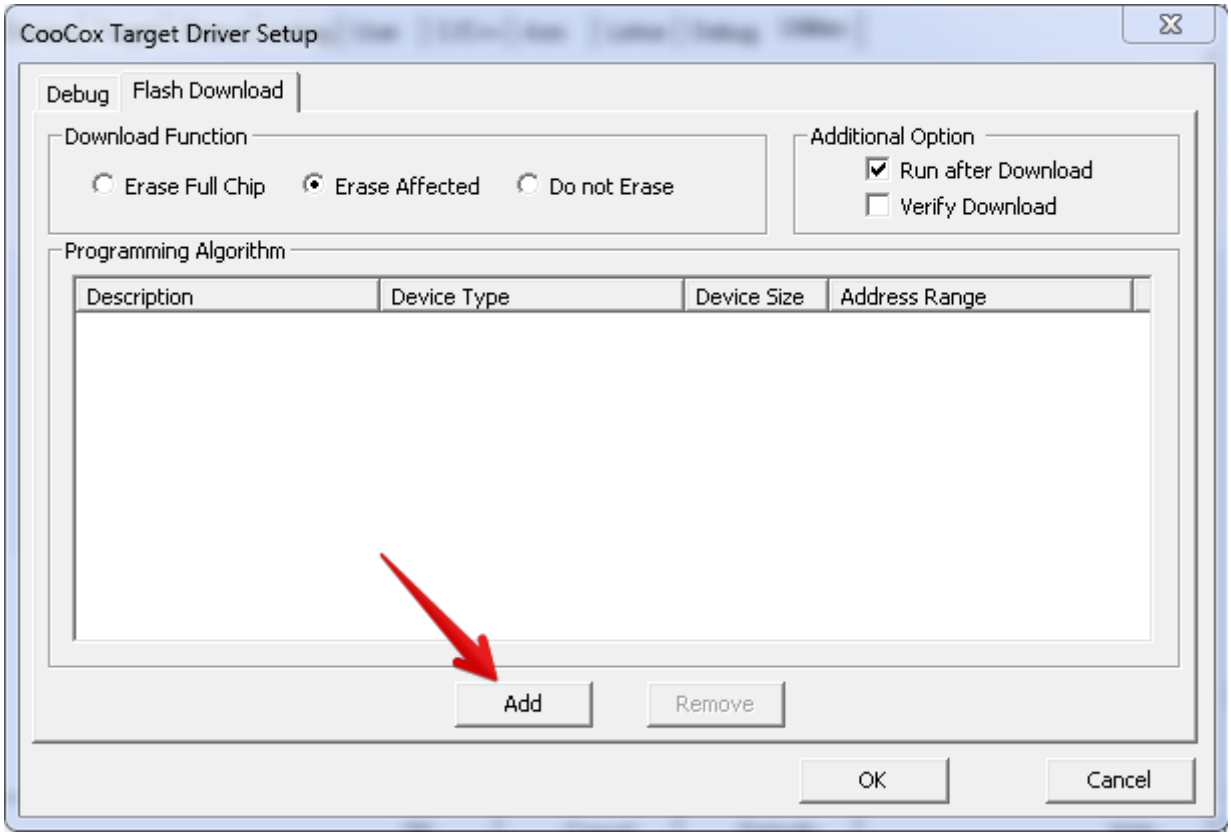


Выставляете адаптер CoLink (не путать с CoLinkEX).

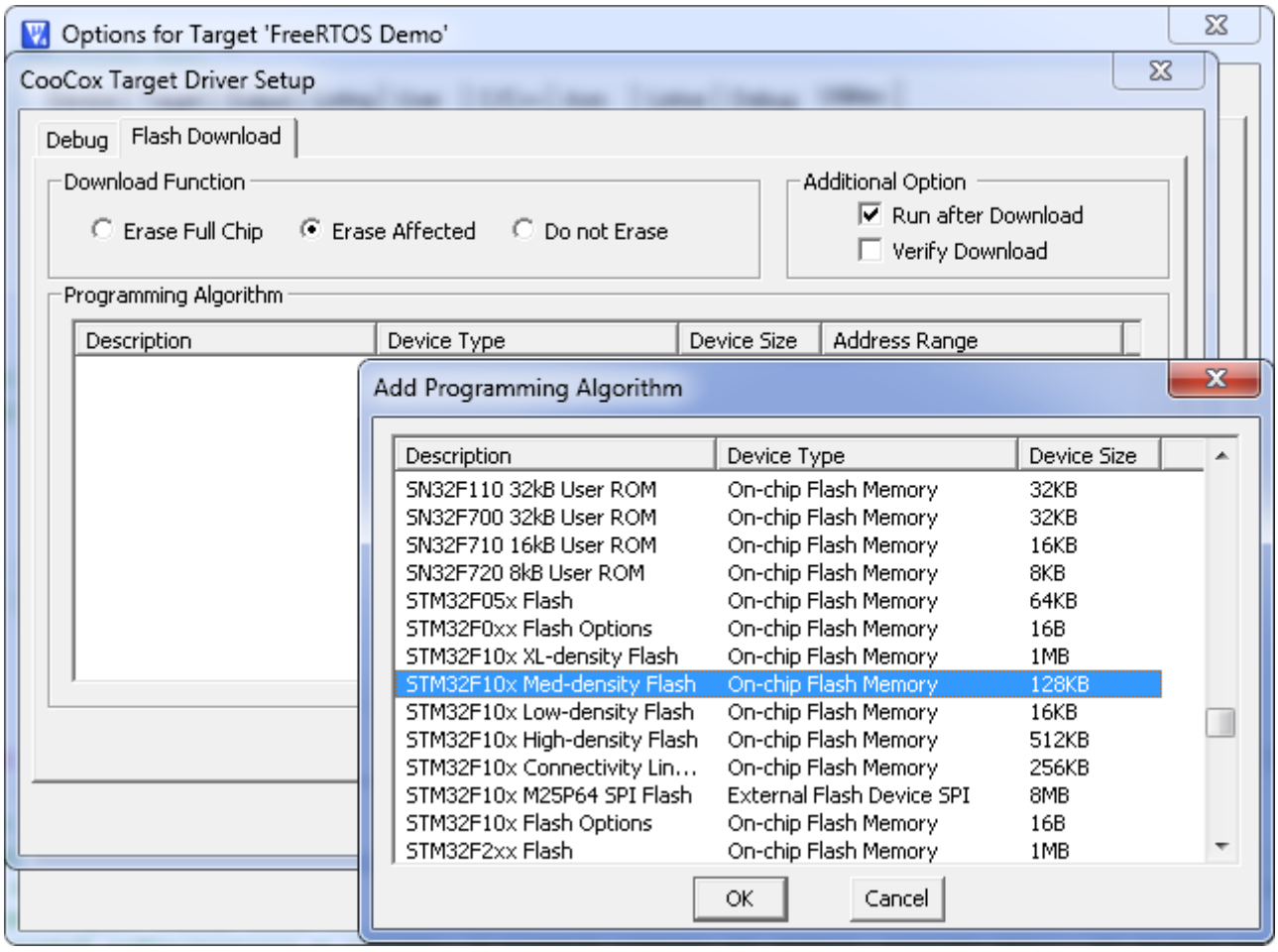
Следующим шагом надо указать программатор. Для этого идем в Utilites и выставляем CooCox Debugger и там. Если галочка будет серой, типо неактивная, то ничего страшного. Надо по ней кликнуть и ее разблокирует. Это глюк.



Следующим этапом нужно зайти в Settings и настроить там алгоритм прошивки.



Добавив нужную модель памяти контроллера из списка:



После чего жмем заливку и наблюдаем результат:

- **Архив с проектом из данной статьи**

В следующей статье я приведу гораздо более сложный пример. С несколькими задачами, очередями, мутексами. Ну и наглядно покажу ряд моментов по распределению времени в RTOS на вытесняющей многозадачности.

ARM CortexM3 FreeRTOS JTAG Keil PinBoard II RTOS

49 thoughts on “Установка и конфигурация FreeRTOS”

__bl__

25 Июнь 2014 в 23:23

А зачем Keil’у FreeRTOS? У него RTX есть.

★ DI HALT

25 Июнь 2014 в 23:55

Так она денег стоит. А тут Free

__bl__

26 Июнь 2014 в 10:30

Я думаю, что готовый проект с FreeRTOS и USB стёком уже не влезет в бесплатные 32 кБ от Keil.

★ DI HALT

26 Июнь 2014 в 10:56

Мне пока хватало этих 32кб под все. А в случае чего можно и на GCC перескочить. Просто в Keil инструментарий удобный для работы.

becopt

26 Июнь 2014 в 12:59