

# Лекция 1

## Основные термины и определения

### *Встраиваемые вычислительные системы (Embedded systems)*

Встраиваемая система (встроенная система, англ. embedded system) – специализированная микропроцессорная система управления, концепция разработки которой заключается в том, что такая система будет работать, будучи встроенной непосредственно в устройство, которым она управляет.

### *Микроконтроллер (англ. Micro Controller Unit, MCU)*

Микросхема, предназначенная для управления электронными устройствами. Типичный микроконтроллер сочетает на одном кристалле функции процессора и периферийных устройств, содержит ОЗУ и (или) ПЗУ. По сути, это однокристалльный компьютер, способный выполнять простые задачи.

### *Контроллер*

1. Изделие для автоматизации и управления.
2. Микросхема или часть микросхемы реализующая отдельную функцию или задачу управления.

### *Отладочная, оценочная или демонстрационная плата*

Электронный модуль, как правило, в бескорпусном изготовлении, содержащий минимально необходимый набор микросхем для разработки ПО для МК.

### *Интегрированная среда разработки. IDE(англ. IDE, Integrated development environment)*

Система программных средств, используемая программистами для разработки программного обеспечения.

Обычно, среда разработки включает в себя:

- текстовый редактор,
- компилятор и/или интерпретатор,
- средства автоматизации сборки,
- отладчик.

### *SWD - Serial Wire Debug.*

Двухпроводной отладочный порт

### *Компилятор*

Программа выполняющая трансляцию исходного кода из предметно-ориентированного языка на машинно-ориентированный язык.

### *Компоновщик(Линковщик)*

Программа собирающая исходный код на машино-ориентированном языке и производящую сборку в исполняемый модуль

### *Стек*

Абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in – first out, «последним пришёл – первым вышел»).

Возможны три операции со стеком: добавление элемента (иначе проталкивание, push), удаление элемента (pop) и чтение головного элемента (peek). Мы будем использовать определение Стекa, в значении аппаратный стек

### *Аппаратный стек*

В микроконтроллере стек – это непрерывная область памяти, адресуемая специальными регистрами SP (указатель стека)

### *Регистр*

Сверхбыстрая память внутри процессора, предназначенная для хранения адресов и промежуточных результатов вычислений (регистр общего назначения/регистр данных) или данных, необходимых для работы самого процессора.

# Среда разработки программ для микроконтроллера

В учебных целях мы будем использовать интегрированную среду разработки IAR Workbench for ARM. Компания IAR бесплатно предлагает для ознакомления две версии своего продукта: версию evolution с полным функционалом и ограничением времени использования 30 дней и версию kickstart (в имени дистрибутива есть буквы KS) с ограничением на размер генерируемого исполняемого кода -32 кбайт), но без ограничения времени использования.

Еще каких-то 10-15 лет назад для создания простейших программ, минимально необходимым набором инструментального ПО являлись: текстовый редактор, транслятор ассемблерного кода и симуляторы для отладки. С развитием микропроцессоров, с ростом объема оперативной памяти и памяти программ и широчайшим распространением МК в различных областях техники, а также требований к надежности и качеству разрабатываемого программного обеспечения минимального набора стало не хватать.

Для создания качественных программ и повторного использования уже отлаженного кода, в виде библиотек, появились редакторы связей (линковщики, компоновщики), появились отладчики, и более совершенные трансляторы и, наконец, стало возможным и обоснованным применение компиляторов (примерно с середины 90-х прошлого века), появился диалект Embedded C/C++. И все эти средства для удобства использования стали объединять в один программный продукт - так появились интегрированные среды разработки (IDE) и целая отрасль разработки ПО. Одними из лидеров в этой области являются фирмы IAR Systems."

Для студенческих нужд размера кода в 32КБ более чем достаточно. В курсе мы будем использовать IAR Embedded Workbench for ARM ver 8.40. Состав этого инструмента показан на Рисунке [Процесс разработки с точки зрения IAR Workbench](#).

# Состав интеграционной среды разработки IAR Workbench

Процесс разработки программного обеспечения в общем случае ничем не отличается от процесса разработки приложения для обычных компьютеров, который включает в себя проектирование (Design), разработка кода (Develop), отладка (Debug)

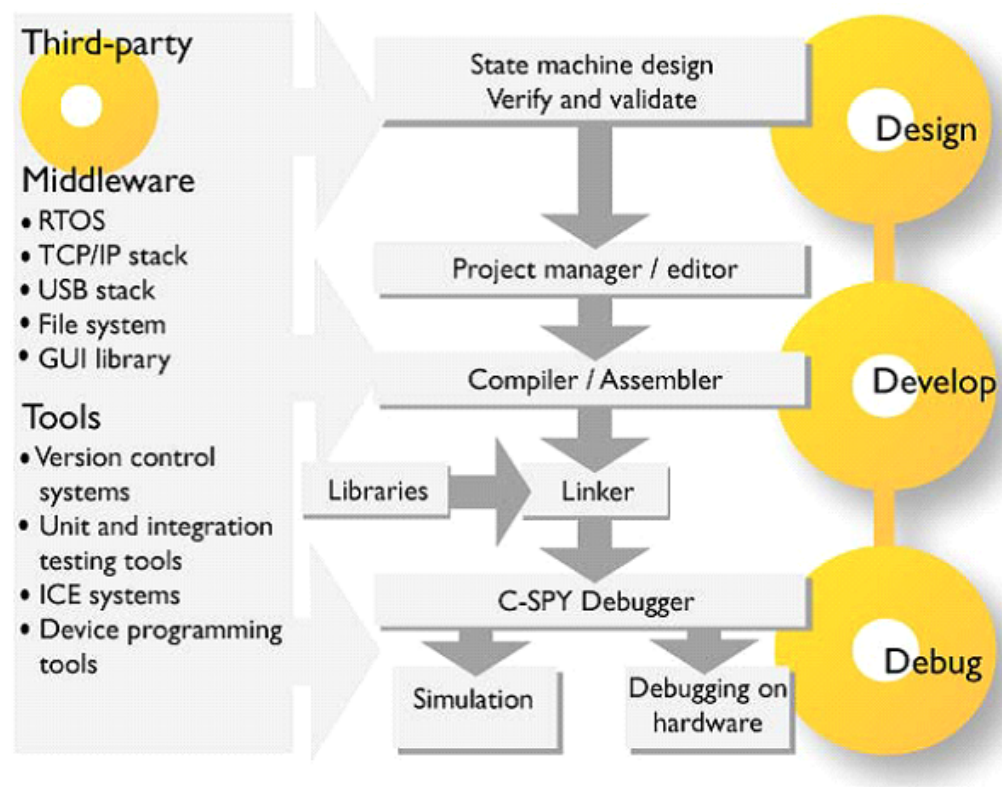


Рисунок 1. Процесс разработки с точки зрения IAR Workbench

## Процесс создания исполняемого образа

Процесс преобразования кода на языке программирования высокого уровня C++ в файл, содержащий образ исполняемой программы, готовый для прошивки в микроконтроллер можно разделить на два этапа:

- Трансляция кода в объектный файл
- Компоновка кода в исполняемый файл

## Трансляция кода

Трансляцию кода выполняет компилятор. Структурно процесс трансляции с помощью компилятора показан на рисунке [\[Схема Трансляции\]](#). После трансляции вы можете получить на выходе либо файлы библиотеки, которые впоследствии можно будет использовать в других проектах, либо объектные файлы.

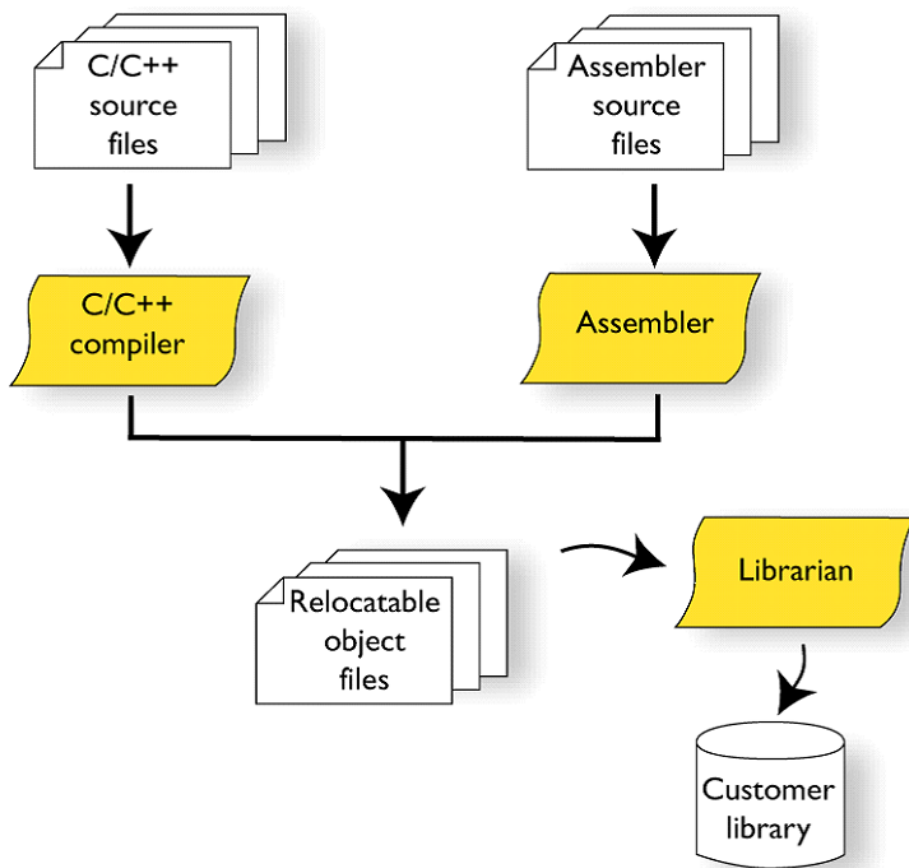


Рисунок 2. Схема процесса трансляции

## Компоновка кода

Компоновку кода выполняет линковщик. Структурно процесс компоновки с помощью линковщика показан на [\[Схема компоновки\]](#).

На входе линковщика могут быть, внешние библиотеки, полученные на этапе трансляции в других проектах и программах, объектные файлы полученные на предыдущем этапе, стандартные (встроенные) библиотеки C++, и конфигурационный файл, описывающий настройки по размещению кода и данных в адресном пространстве микроконтроллера. Компоновщик создает исполняемый файл, который можно запустить на микроконтроллере

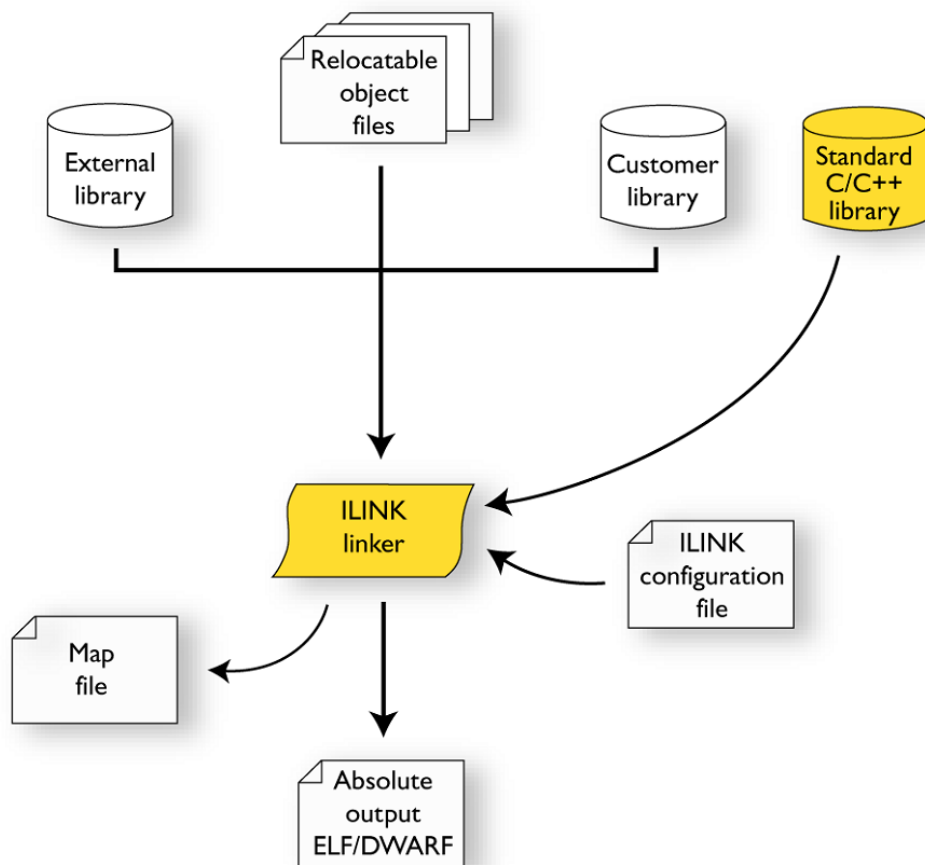


Рисунок 3. Схема процесса компоновки

## Запуск и отладка

Последний этап, показанный на рисунке [\[IAR Workbench\]](#) - отладка. Компоновщик IAR создает файл в формате ELF, который содержит исполняемый образ программы. Этот файл может быть использован для:

- Загрузки в систему отладки IAR-CSPY или в любой другой отладчик, например GDB, способный читать ELF формат
- Загрузки образа в ПЗУ микроконтроллера используя программатор.

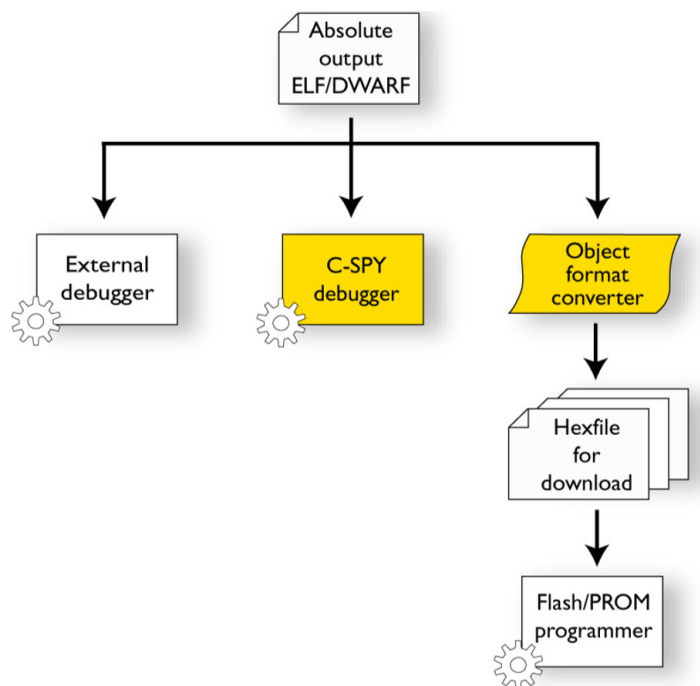


Рисунок 4. Возможные варианты использования выходного файла

## Запуск программного обеспечения

Функция `int main()` является точкой входа программы, для пользователя программа начинается с вызова этой функции и выполнения тела этой функции. Однако на самом деле, еще до функции `main()` микроконтроллер выполняет множество различных действий, например, инициализацию стека, глобальных переменных, констант.

## Инициализация стека

Сразу после подачи питания происходит инициализация указателя стека на конечный адрес стека.

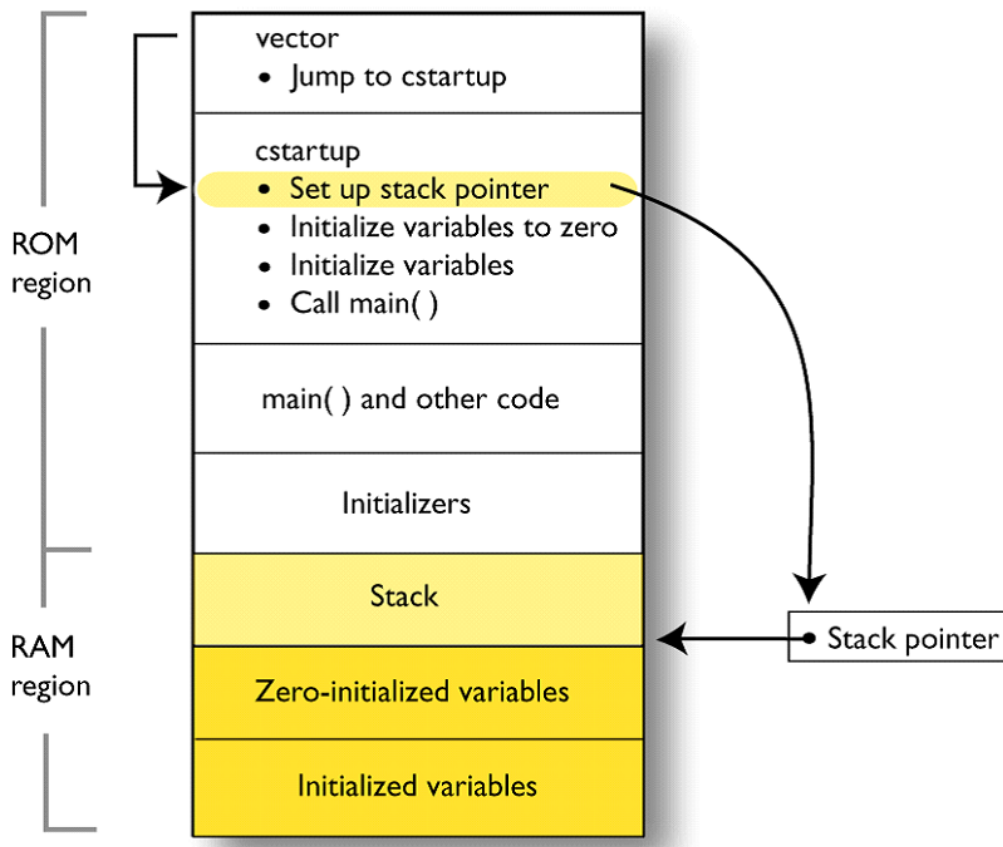


Рисунок 5. Стадия инициализации стека

## Инициализация переменных в нулевые значения

После подачи питания на микроконтроллер, регистр адреса команды указывает на 0 адрес, микроконтроллер начинает работу с адреса 0. По адресу 0, находится таблица векторов прерываний, по начальному вектору находится команда инициализации указателя стека на конечный адрес стека и далее перехода на функцию инициализации.

После подачи питания и инициализации стека, выполняется функция инициализации памяти нулями (данные указанные как zero-initialized data, непроинициализированные глобальные переменные, такие как `int i;`)



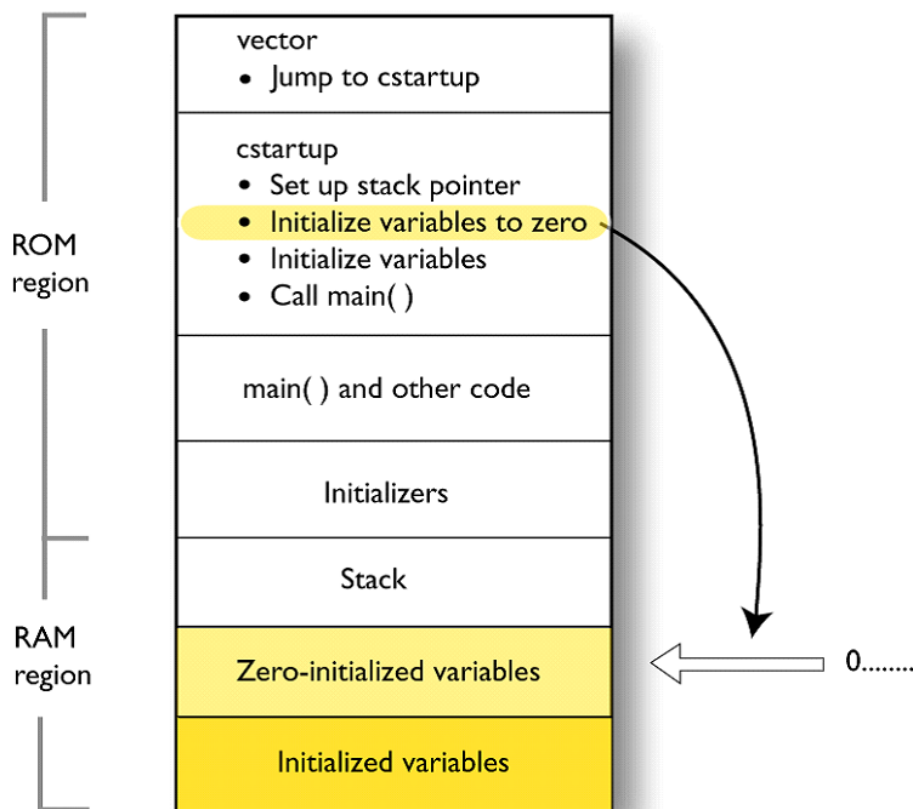


Рисунок 6. Стадия инициализации непроинициализированных переменных

## Инициализация переменных

Далее должна произойти инициализация данных определенных как initialized data, например `int i = 6`. Значения инициализации для каждой переменной будут скопированы из ПЗУ в ОЗУ.

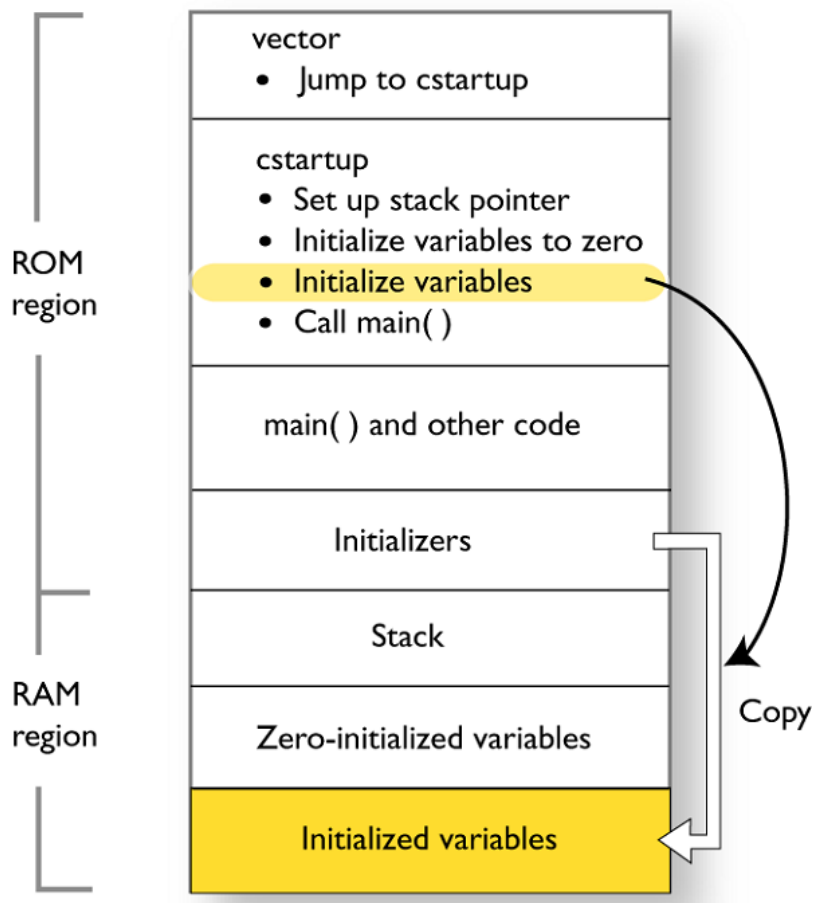


Рисунок 7. Стадия инициализации проинициализированных переменных

## Запуск функции main()

Завершающий этап – это вызов функции main().

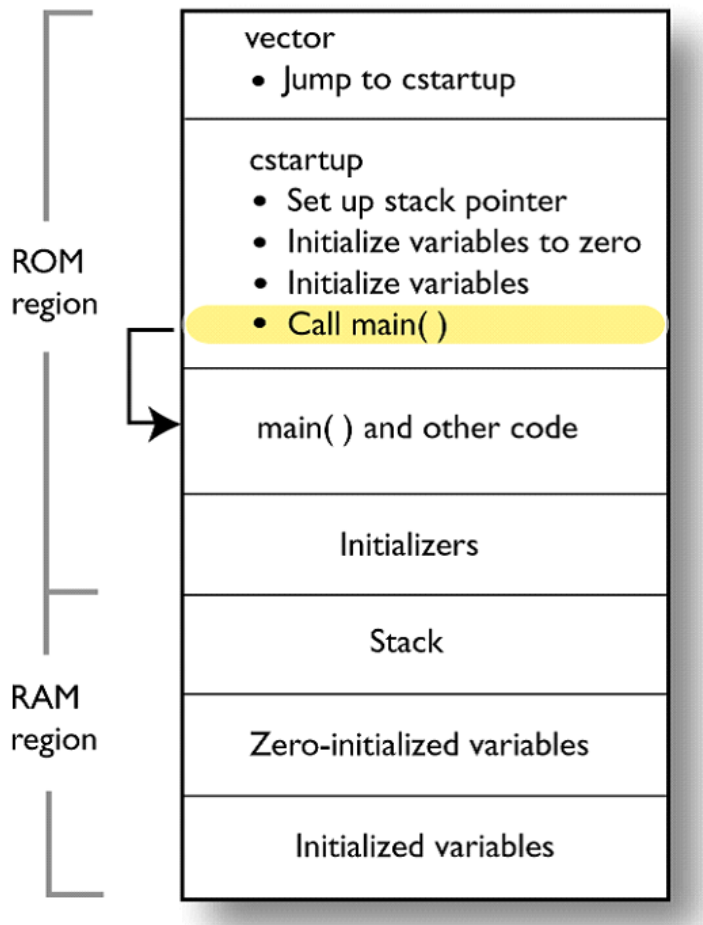


Рисунок 8. Стадия запуска функции *main()*

Как видно, перед тем как запуститься функция `main` необходимо выполнить инициализацию стека и переменных, кроме того, если в вашем проекте будут использоваться прерывания, то в таблицу векторов прерываний необходимо добавить переходы на адреса обработчиков ваших прерываний.

# Преимущества IAR Embedded Workbench

За последние время в среде разработки IAR Embedded был сделан огромный скачек с точки зрения удобства использования, так и с точки зрения поддержки современных стандартов. Так версия 8.X получила поддержку стандарта C14, а начиная с версии 8.40 и поддержку стандарт C17 и это является огромным плюсом для разработки надежного, понятного и качественно ПО. Свои мысли по этому поводу я озвучил в статье [\[Можно ли использовать C++ вместо Си для небольших проектов в микроконтроллерах:\]](#)

Некоторые характеристики среды вы можете получить из Таблицы - [\[Характеристики IAR Embedded Workbench\]](#), данные взяты из [\[IAR C/C++ Development Guide\]](#)

Таблица 1. Характеристики IAR Embedded Workbench

Характеристика	IAR Embedded Workbench
Языки	C/C++
Стандарты языка	C++ 17 начиная с версии 8.40
Оптимизация кода	Да, кроме condition_variable, future, mutex, shared_mutex, thread, поддержка atomic урезана и реализована только для типов для которых есть аппаратная поддержка atomic специальными командами в микроконтроллерах
Контроль размера стека	Да
Поддержка RTOS	Да
Статический анализатор кода с набором правил	Да - MISRAC++2008, SECURITY, CERT, STDCHECKS
Динамический анализ кода	C-RUN
Сертификация и проверка соответствию стандартам безопасности	Сертификация на безопасность по стандартам IEC 61508 и ISO 26262 экспертной организацией TUV SUD – SIL3 сертификат
Поддержка микроконтроллера STM32F411 RE	Полная

## Вопросы по разделу

1. Дайте определение понятию “Интегрированной среде разработки”

Ответ:

2. Что такое компилятор и чем он отличается от транслятора?

Ответ:

3. Что такое компоновщик и какие функции он выполняет?

Ответ:

4. Почему важен процесс проектирования ПО какие задачи входят в этот процесс?

Ответ:

5. Дорисуйте процесс разработки ПО, описанный на изображении [Процесс разработки с точки зрения IAR Workbench](#) с учетом итеративности связей в этом процессе

Ответ:

6. Зачем нужна отладка и в каких случаях она применяется? Для чего применяются точки останова?

Ответ:

7. Какие еще важные IAR workbench можно добавить в таблицу [\[Характеристики IAR\]](#)

Ответ:

# Запуск программного обеспечения

## Файл cstartup.cpp

Действия по инициализации прописываются в файле cstartup. Этот файл может быть написан как на ассемблере, на Си, так и на С+. Поскольку мы будем использовать С+, то и файл будем использовать cstartup.cpp, который будет выглядеть примерно так

```
extern "C" void __iar_program_start(void) ;

class InterruptHandler {
public:
    static void DummyHandler() { for(;;) {} }
};

using tIntFunc = void(*)();
using tIntVectItem = union {tIntFunc __fun; void * __ptr;};
#pragma segment = "CSTACK"
#pragma location = ".intvec"
const tIntVectItem __vector_table[] = {
    { .__ptr = __sfe( "CSTACK" ) }, //инициализация стека
    __iar_program_start, //переход на адрес функции __iar_program_start

    InterruptHandler::DummyHandler,
    ...
    InterruptHandler::DummyHandler,      ////TIM4
};

extern "C" void __cmain(void) ;
extern "C" __weak void __iar_init_core(void) ;
extern "C" __weak void __iar_init_vfp(void) ;

#pragma required = __vector_table
void __iar_program_start(void) {
    __iar_init_core() ;
    __iar_init_vfp() ;
    __cmain() ;
}
```

Немного проясним, что здесь написано, строка:

```
extern "C" void __iar_program_start( void );
```

Описывает прототип функции \_\_iar\_program\_start, которая будет отвечать за инициализацию переменных и запуск функции main(). Реализацию этой функции вы можете увидеть в самом конце файла cstartup.cpp

```
void __iar_program_start( void ) {
__iar_init_core();
__iar_init_vfp();
__cmain();
}
```

Далее идет определение класса с описание одного единственного метода handler(). Это метод и будет тем самым обработчиком прерывания который вызовется при срабатывании соответствующего прерывания. Реализация метода проста – бесконечный цикл, т.е. попав в прерывание программа “навсегда” останется в нем:

```
__weak void DummyModule::handler() { for(;;) {} };
```

Это сделано для того, что пока не планируем использовать никаких прерываний, и если все таки каким то образом прерывание сработало, значит, что-то было сделано не так. В дальнейшем в разделе [\[Прерывания\]](#) будет показано, как сделать нужный нам обработчик прерывания, но сейчас мы не будем на этом заострять внимание. Следующий две строки определяют новый тип, который будет использоваться для задания элементов таблицы векторов прерываний:

```
typedef void( *intfunc )( void );
typedef union { intfunc __fun; void * __ptr; } intvec_elem;
```

Как видно этот тип есть объединение двух типов, указателя на функцию типа void и указателя на void. Это необходимо для того, чтобы правильно интерпретировать элементы таблицы. Ведь начальный вектор прерывания не содержит никакого обработчика, а просто содержит конечный адрес стека, а последующие вектора содержат адреса обработчиков, именно поэтому первый элемент таблицы векторов должен иметь тип указателя на void, а последующие указателей на функцию типа void. Собственно далее идет и сама таблица лежащая в выделенном для неё сегменте .intvec, который задается в настройках линковщика #pragma location = ".intvec" Таблица начинается с адреса стека, который также задается сегментом CSTACK в настройке линковщика, а следующий элемент таблицы есть адрес функции инициализации переменных, а затем адреса обработчиков для конкретных прерываний.

# Использование C++

- Так же как когда-то Си пробивал себе дорогу в качестве стандарта для встроенного ПО, так и язык C++ уже вполне может заменить Си в этой области.

Язык программирования стандарта C++ и современные компиляторы имеют достаточно средств для того чтобы создавать компактный код и не уступать по эффективности коду, созданному на Си, а благодаря нововведениям быть понятнее и надежнее.. Начиная с версии IAR Workbench 8.40 компилятор поддерживает полезные нововведения стандарта C++17, такие, как например “структурные привязки”, “инициализация в ветвлениях”, “встроенные переменные”.

- C++ является строго типизированным языком, а значит программы написанные на нем более безопасны, чем программы написанные на Си и меньше вероятность того, что программист допустит ошибку.
- C++ является языком программирования полностью поддерживающий парадигму программирования ООП, которая отлично подходит для разработки программного обеспечения измерительных устройств.

Ведь нужно понимать, что для измерительного устройства нам нужно описать логику работы, интерфейс взаимодействия с пользователем, реализовать расчеты, а не помнить, что для того чтобы считать данные с АЦП, нужно вначале его выбрать с помощью сигнала CS, находящегося на порту GPIOA.3 и установить его в единицу. Этим должен заниматься разработчик драйверов.

Большинство драйверов для работы с аппаратурой уже реализованы производителями микроконтроллеров, например, в библиотеках CMSIS и CMSIS\_HAL, ими можно воспользоваться для обращения к функциям доступа к аппаратуре, упростить и ускорить разработку.

Замечаниями по этому поводу может служить то, что эти библиотеки довольно громоздкие и для использования в небольших приложениях вряд ли подойдут, кроме того, не всегда они имеют необходимые сертификаты надежности, а потому при разработке реальных измерительных устройств, применение которых планируется в местах с повышенной безопасностью промышленных предприятиях, вряд ли стоит пользоваться этими библиотеками.

Именно поэтому, мы будем использовать C++ от написания драйверов и уровня аппаратуры и до реализации логики работы с пользователем. Начнем же изучение с создания проекта, системы тактирования и небольшой программы мигания светодиода.



# Программа на C++

Как было сказано в разделе [Состав интеграционной среды разработки IAR Workbench](#) первоначально мы должны создать исходные файлы на языке программирования C. В C разделяют два типа файлов:

- Исходный файл (файл с расширением \*.cpp)
- Заголовочный файл (файл с расширением \*.h, \*.hpp)

Заголовочные файлы подключаются с помощью директивы `#include` и при трансляции просто вставляются в текст \*.cpp файла. Используются они для того, чтобы вынести общие определения, используемые в нескольких \*.cpp файлах в одно место.

*Вот так может выглядеть ваша программа:*

```
#include "gpioaregisters.hpp" //for Gpioa
#include "rccregisters.hpp"   //for RCC

int main()
{
    RCC::AHB1ENR::GPIOAEN::Enable::Set() ;
    GPIOA::MODER::MODER15::Output::Set() ;
    GPIOA::ODR::ODR15::Enable::Set() ;
    return 0 ;
}
```

# Создание проекта и работа в IAR Workbench

- Создать новый проект Project ⇒ Create New Project.

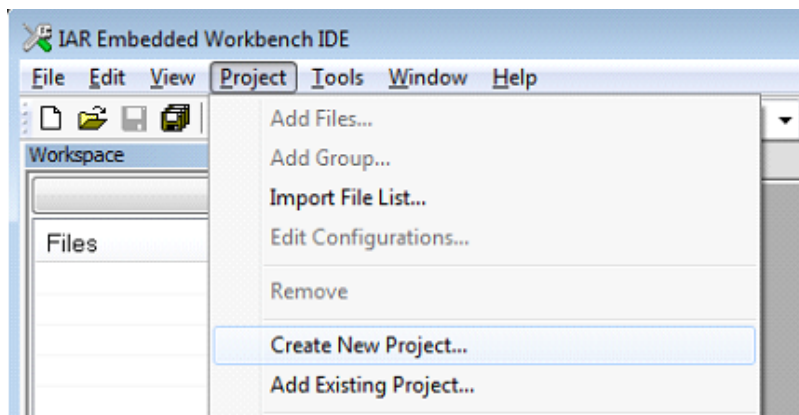


Рисунок 9. Создание нового проекта

## Выбор шаблона проекта

- Выбирать шаблон проекта( ProjectTemplates): C++ - main

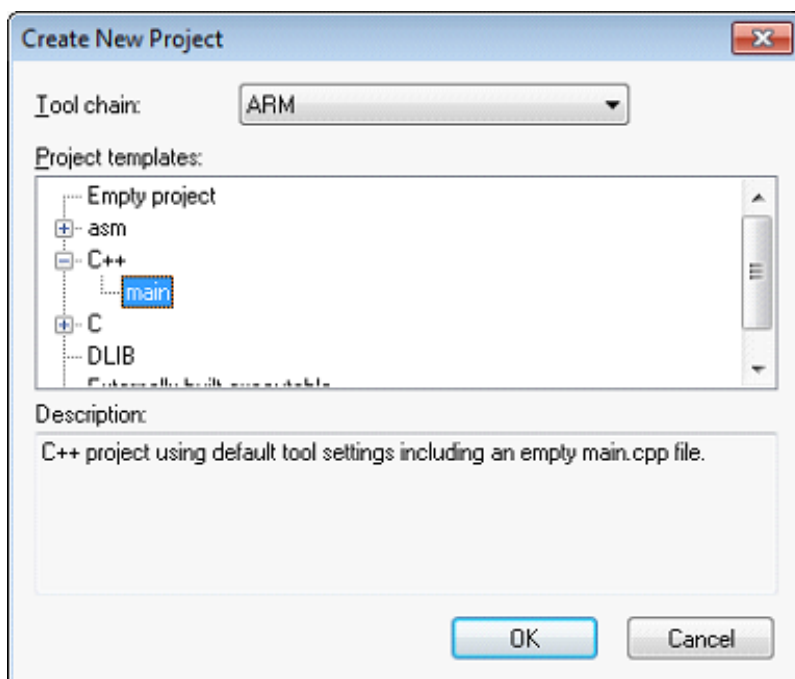


Рисунок 10. Выбор шаблона проекта

## Выбор микроконтроллера

- Сохранить проект под каким-либо именем
- В свойствах проекта выбрать модель микроконтроллера ST ⇒ STM32F4 ⇒ STM32F411 ⇒ STM32F411RE см. [Выбор микроконтроллера](#). Для этого правой кнопкой мыши щелкнуть

по проекту, выбирать Options и далее в категории General Option выбрать закладку Target.

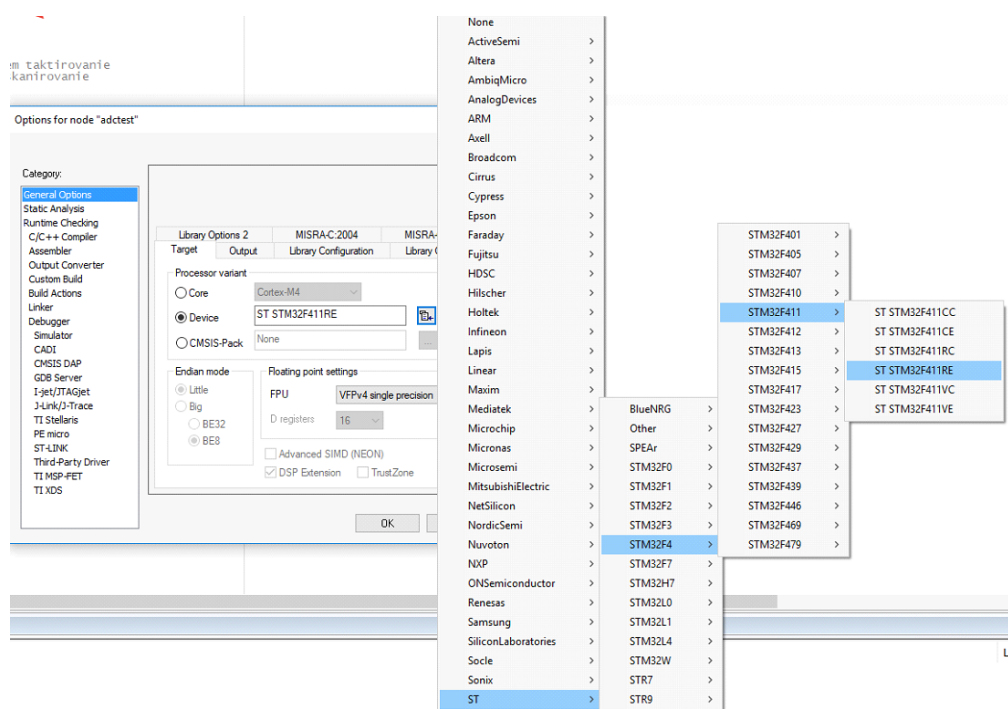


Рисунок 11. Выбор микроконтроллера

## Запуск в режиме отладки

После создания проекта необходимо сохранить так называемое рабочее пространство или (workspace).

В рабочее пространство можно загружать несколько проектов (например, проекты всех лабораторных работ) и переключаться между проектами по мере необходимости.

После того, как проект сделан, и имеет вид показанный на [\[Вид созданного проекта\]](#), можно попробовать собрать проект, нажав кнопку Ctrl-F7, а затем загрузить полученный бинарный файл в микропроцессор и запустить на отладку с помощью кнопки Ctrl-D.

Все тоже самое можно сделать и с помощью кнопок быстрого доступа на панели инструментов, через меню среды или контекстное меню проекта (загрузить которое можно нажав на правую клавишу мыши на проекте).

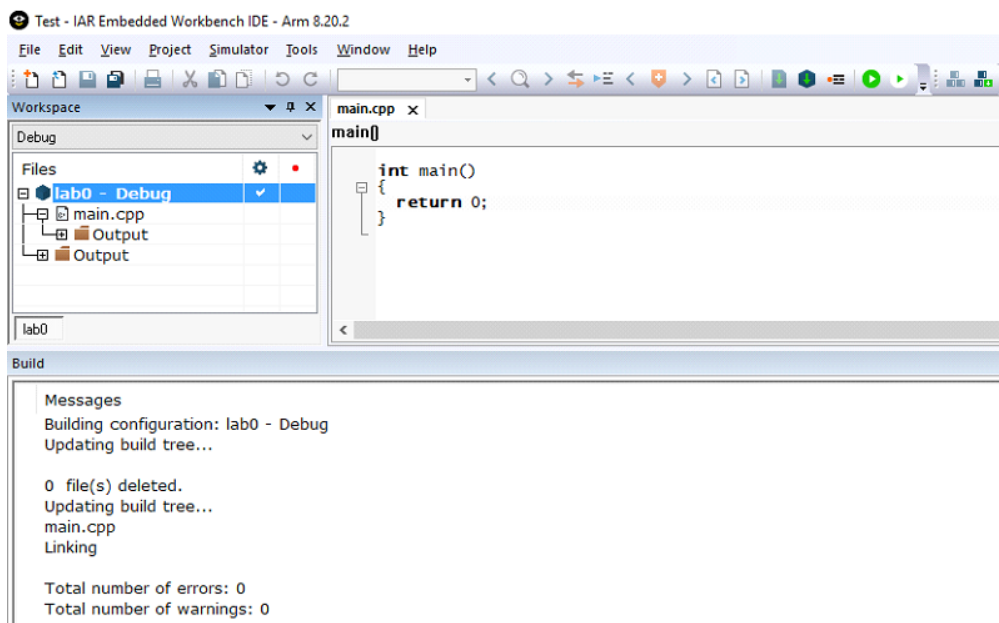


Рисунок 12. Вид созданного проекта

## Запуск проекта в режим симуляции

По умолчанию загрузка и отладка бинарного файла осуществляется в симулятор выбранного микроконтроллера. Поэтому, если вы выполнили все верно, то должно получиться нечто похожее, показанное на [\[Проект в режиме отладки\]](#).

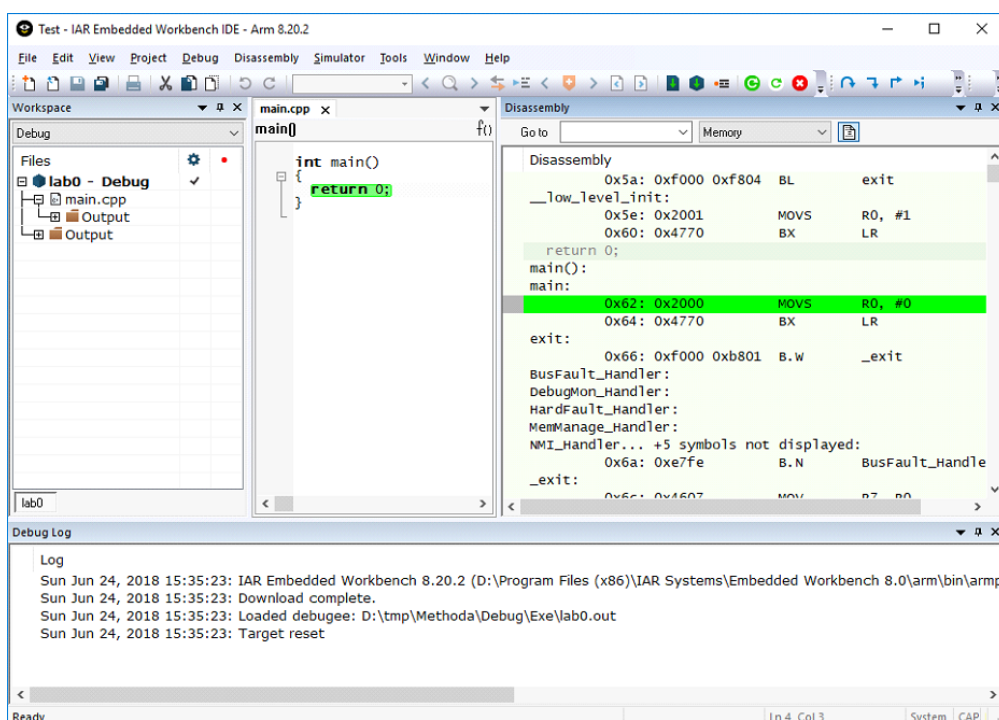


Рисунок 13. Проект в режиме отладки

На этом рисунке вы можете видеть, как сам ваш код написанный на C++, так и окно дизассемблера, показывающее как компилятор преобразовал ваш код в команды ассемблера. Зеленая строка показывает текущую исполняющую строку вашего кода и команду ассемблера.

Для того чтобы остановить отладку и выйти в режим разработки необходимо нажать кнопки Ctrl-Shift-D.

## Выбор внутрисхемного отладчика

Чтобы загрузить программу в микроконтроллер необходимо вместо симулятора выбрать внутрисхемный отладчик, которым вы пользуетесь. Это можно сделать, встав на проект и нажать на правую кнопку мыши, далее выбрать пункт меню Options⇒Debugger⇒Driver и выбрать в нем нужный вам внутрисхемный отладчик, см [Выбор внутрисхемного отладчика](#). Мы будем использовать отладчик ST-Link.

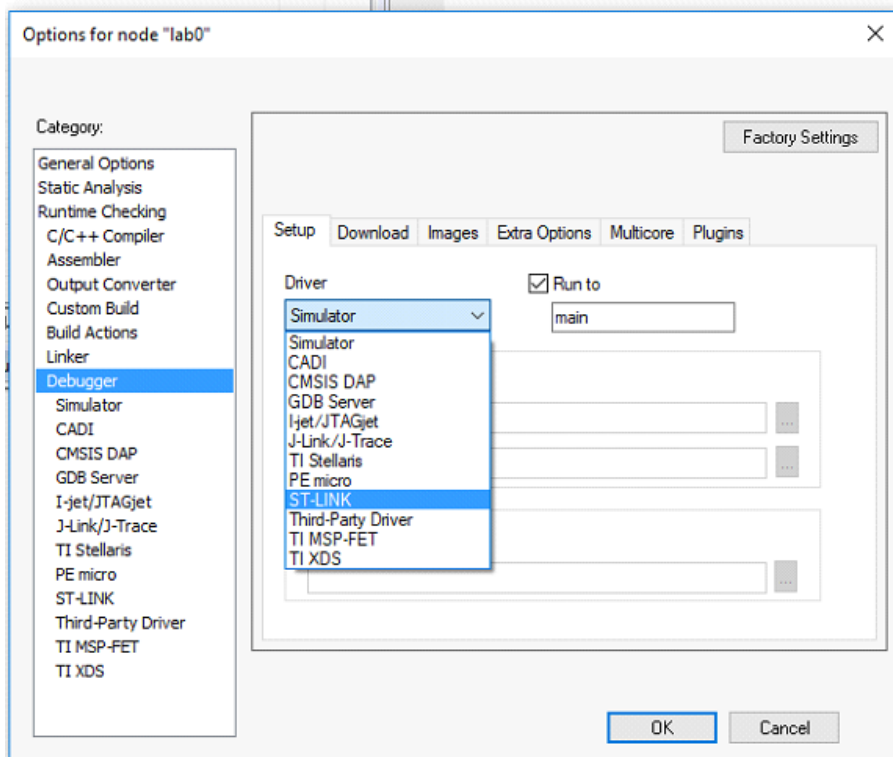


Рисунок 14. Выбор внутрисхемного отладчика

Теперь, если вы нажмете Ctrl-D, ваша программа загрузиться в микроконтроллер и отладка будет осуществляться непосредственно на ядре микроконтроллера. И так вы смогли сделать проект, откомпилировать пустую программу и загрузить её в симулятор и микроконтроллер, но всех этих действий недостаточно, для того, чтобы начать разрабатывать программное обеспечение. Рассмотрим, что же еще необходимо сделать для того, чтобы наш проект был полностью готов.

# Структура проекта

Для того, чтобы разработка была быстрой и качественной, необходимо структурировать паку проекта.

- Не нужно писать весь код в одном файле. Лучше каждый класс описывать в отдельном файле
- Файлы с классами, ответственные за один компонент, лучше держать в папках с именем этого компонента
- Не превращаем проект в мусорку

## Добавление файла (cstartup.cpp) в проект

В папку где вы сохранили проекта, необходимо скопировать файл cstartup.cpp. и добавить его к проекту: Для этого нужно нажать правую кнопку мыши на проекте и выбрав пункт Add⇒Add Files... как показано на [\[Добавление нового файла в проект\]](#), а затем выбрать файл startup\_stm32F411.cpp.

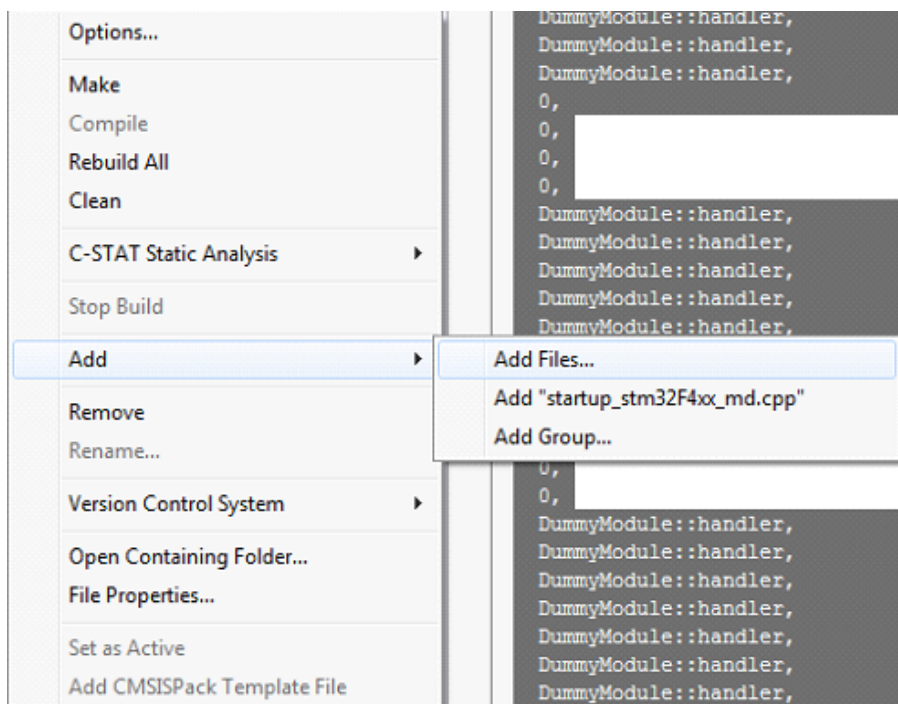


Рисунок 15. Добавление нового файла в проект

Как было сказано выше, в файле cstartup.cpp описывается таблица векторов прерываний и начальная инициализация. Поэтому первым делом нужно подключить файл cstartuo.cpp в проект. Тут следует иметь ввиду, что таблица векторов прерываний для разных микроконтроллеров разная, и соответственно файлы cstartup должен быть различных для разных микроконтроллеров. Чтобы не перепутать свой файлы для разных микроконтроллеров, назовем его startup\_stm32F411.cpp и подключим к проекту, нажав правую кнопку мыши на проекте и выбрав пункт Add⇒Add Files... (см. Рисунок 21 ), а затем выбрав файл startup\_stm32F411.cpp.

## Начальная структура проекта

Добавив файл в проект у вас должно получиться следующая структура в среде IAR Workbench:

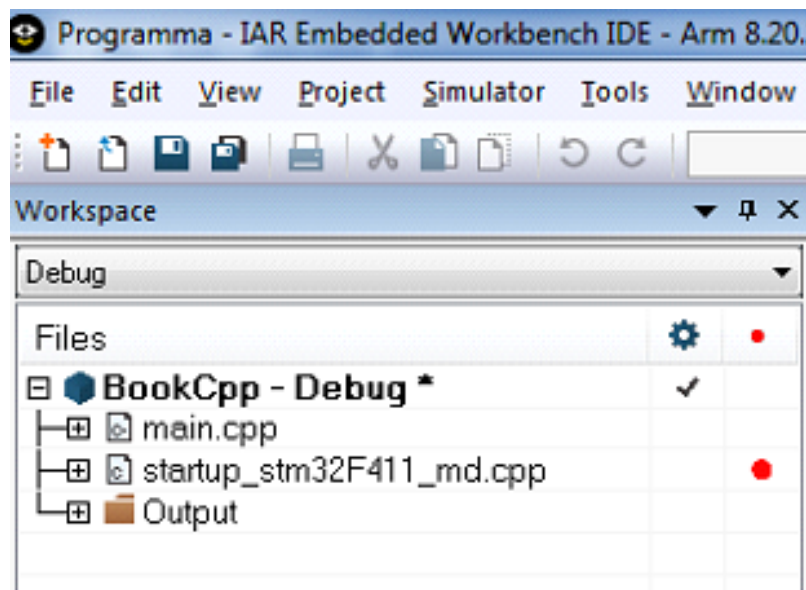


Рисунок 16. Начальная структура проекта

## Доступ к папке проекта

Теперь нужно разобраться с тем как будет организован наш проект на диске и в системе контроля версий. Если мы нажмем правой мышкой на проекте и выберем пункт Open Containing Folder см. [\[Открытие папки проекта\]](#), то мы попадем в папку нашего проекта.



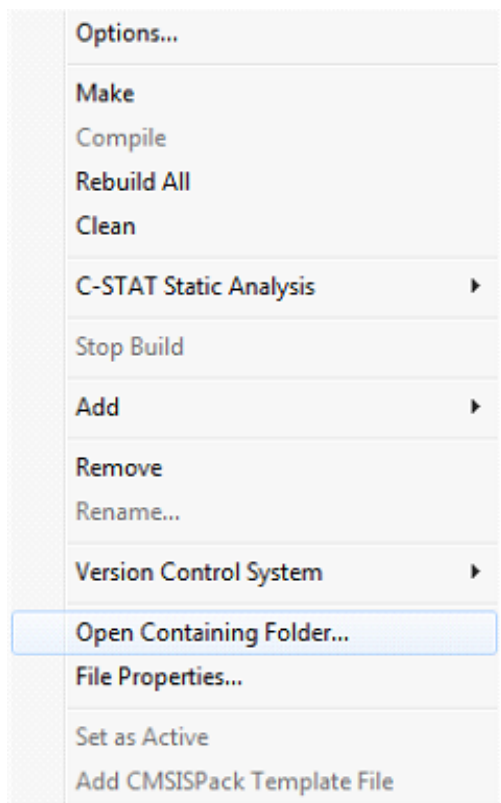


Рисунок 17. Открытие папки проекта

Мы увидим что у нас есть папки Debug и Settings, а также созданные нами файлы проекта, файлы рабочей области, main.cpp и startup\_stm32F411.cpp. В паке Debug хранятся объектные файлы, двоичные файлы для прошивки, листинг программы, созданные в режиме Debug (т.е. в режиме, когда в программу добавляется некая служебная информация и функциональность для того, чтобы можно было поддерживать внутрисхемную отладку. Существует также режим Release, когда двоичный файл содержит только код программы).

В папке Settings хранятся настройки проекта и рабочей области.

## Структура папки проекта

□ Нам нужна будет папка AbstractHardware/Registers. В которой находятся файлы с описанием полей регистров. Можно скопировать ее путем клонирования папки проекта преподавателя, набрав в командной строке:

```
git clone https://github.com/lamer0k/CortexLib.git
```

Вы можете скопировать папку преподавателя через Git, используя PowerShell. Для этого, нужно нажав на вашу папку правой кнопкой мыши, удерживая Shift, выбрать меню "Открыть окно PowerShell здесь".

□ В папке AbstractHardware будут содержаться файлы для работы с регистрами, аппаратурой и периферией.

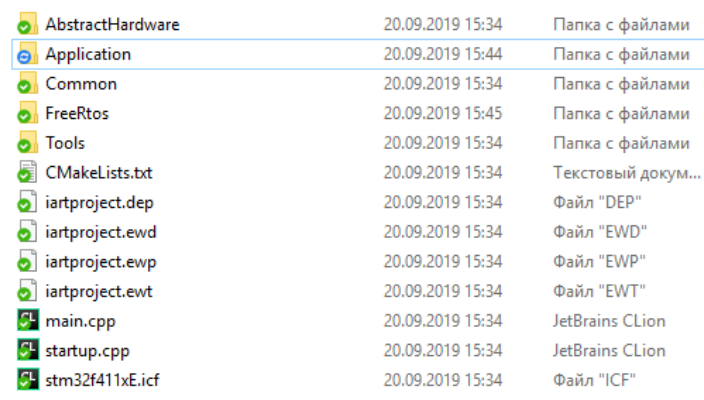
Папка AbstractHardware содержит зависимую от микроконтроллера часть.



□ Дополнительно создадим еще папку Application, в которой в дальнейшем будут содержаться файлы классов для работы с логикой программы.

Папка Application будет содержать полностью независимую часть, которую можно будет перенести на любую другую платформу и микроконтроллер. А папка AbstractHardware будет содержать модули зависящие от конкретного микроконтроллера.

□ В завершение добавим папку FreeRtos – она пригодится нам при работе с ОСРВ.



AbstractHardware	20.09.2019 15:34	Папка с файлами
Application	20.09.2019 15:44	Папка с файлами
Common	20.09.2019 15:34	Папка с файлами
FreeRtos	20.09.2019 15:45	Папка с файлами
Tools	20.09.2019 15:34	Папка с файлами
CMakeLists.txt	20.09.2019 15:34	Текстовый докум...
iartproject.dep	20.09.2019 15:34	Файл "DEP"
iartproject.ewd	20.09.2019 15:34	Файл "EWD"
iartproject.ewp	20.09.2019 15:34	Файл "EWP"
iartproject.ewt	20.09.2019 15:34	Файл "EWT"
main.cpp	20.09.2019 15:34	JetBrains CLion
startup.cpp	20.09.2019 15:34	JetBrains CLion
stm32f411xE.icf	20.09.2019 15:34	Файл "ICF"

Рисунок 18. Финальное содержимое папки проекта

## Изменение структуры проекта

Теперь необходимо создать точно такую же структуру в проекте IAR Workbench, как и структура папок. Для этого необходимо нажать правой мышкой на проект, и выбрать меню Add⇒Ggroup и создать группы Abstract\_Hardware, Application, Common, FreeRtos.

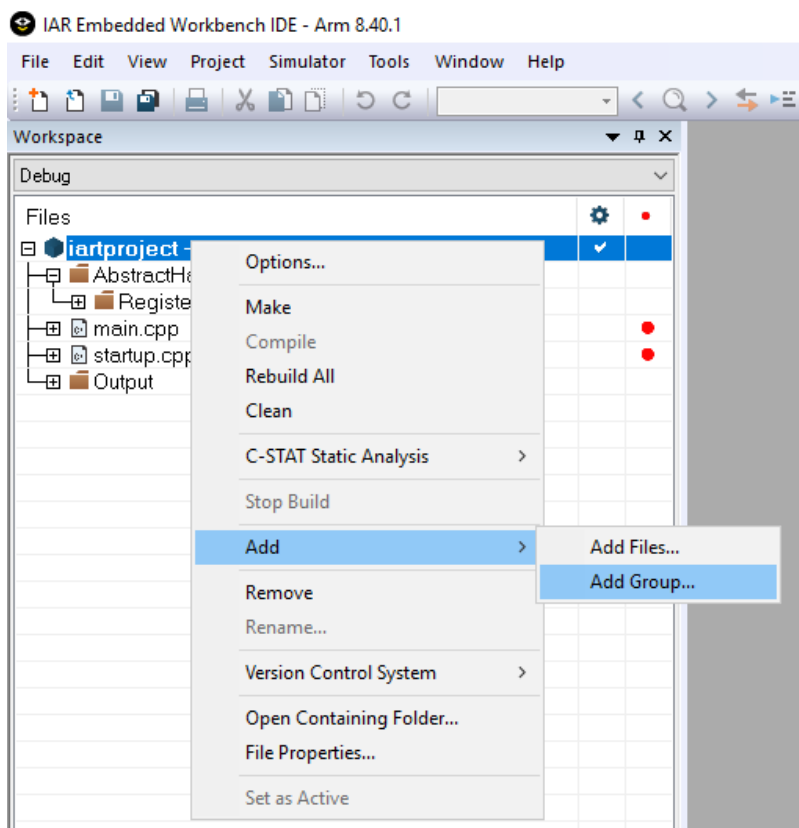


Рисунок 19. Изменение структуры проекта

## Финальная структура проекта

В конечном итоге у вас должна появиться вот такая структура:

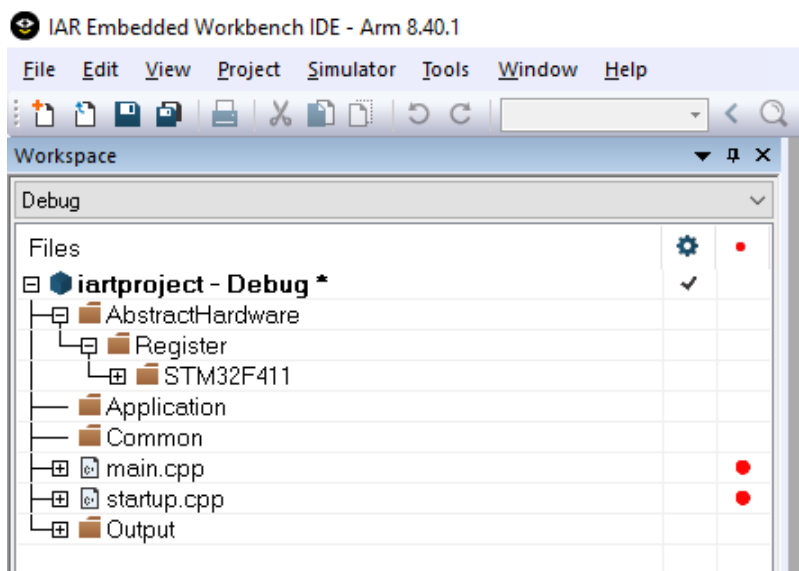


Рисунок 20. Финальная структура проекта

# Окончательная настройка проекта

Для окончательной настройки проекта, нам понадобится настроить компоновщик и установить размер сегментов памяти, стека и кучи.

Перед тем как производить их настройку разберемся, что такое сегменты памяти, стек и куча и для чего они нужны.

## Организация памяти

Существует несколько признанных архитектур микропроцессоров \* Архитектура ФонНеймана \* Гарвардская архитектура В традиционных микропроцессорах используется архитектура Фон Неймана (названную так в честь американского математика Джона Фон Неймана), см. [\[Архитектуры микропроцессоров\]](#) А.

Эта архитектура состоит из единого блока памяти, в котором хранятся и команды, и данные, и общей шины для передачи данных и команд в ЦПУ и от него. При такой архитектуре перемножение двух чисел требует по меньшей мере трех циклов: двух циклов для передачи двух чисел в ЦПУ, и одного – для передачи команды. Данная архитектура приемлема в том случае, когда все действия могут выполняться последовательно. По сути говоря, в большинстве компьютеров общего назначения используется сегодня такая архитектура.

Однако для быстрой обработки сигналов больше подходит гарвардская архитектура, см [\[Архитектуры микропроцессоров\]](#) В. Данная архитектура получила свое название в связи с работами, проведенными в Гарвардском университете под руководством Ховарда Айкена. Данные и код программы хранятся в различных блоках памяти и доступ к ним осуществляется через разные шины, как показано на схеме. Т.к. шины работают независимо, выбор команд программы и данных может осуществляться одновременно, повышая таким образом скорость по сравнению со случаем использования одной шины в архитектуре Фон Неймана.

На [\[Архитектуры микропроцессоров\]](#) С, представлена модифицированная гарвардская архитектура, где и команды, и данные могут храниться в памяти программ.

ARM является модифицированной гарвардской архитектурой.



Рисунок 21. Архитектуры микропроцессоров

Доступ к памяти осуществляется по одной шине, а уже устройство управления памятью обеспечивает разделение шин при помощи управляющих сигналов: чтения, записи или выбора области памяти.

Данные и код могут находиться в одной и той же области памяти. В этом едином адресном пространстве может находиться и ПЗУ и ОЗУ и периферия. А это означает, что собственно и код и данные могут попасть хоть куда(в ОЗУ или в ПЗУ) и это зависит только от компилятора и линкера.

## Настройка области памяти в компошивке

Поэтому чтобы различить области памяти для ПЗУ(ROM) и ОЗУ их обычно указывают в настройках линкера.

В настройках линкера IAR 8.40.1 это выглядит вот так:

```
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0807FFFF;
define symbol __ICFEDIT_region_RAM_start__  = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__    = 0x2001FFFF;
define region ROM_region    = mem:[from __ICFEDIT_region_ROM_start__ to
__ICFEDIT_region_ROM_end__];
define region RAM_region    = mem:[from __ICFEDIT_region_RAM_start__ to
__ICFEDIT_region_RAM_end__];
```

В данном микроконтроллере диапазон адресом для памяти слудующий:

- ОЗУ(RAM) 0x20000000...0x2001FFF,
- ПЗУ(ROM) с 0x08000000...0x0807FFFF.

Вы легко можете поменять начальный адрес ROM\_start на адрес ОЗУ, скажем RAM\_start и

конечный адрес `ROM_end` на адрес `RAM_end` и ваша программа будет полностью расположена в ОЗУ.

Вы даже можете сделать наоборот и указать ОЗУ в области памяти ROM, и ваша программа успешно соберется и прошьется, правда работать не будет :)

Некоторые микроконтроллеры, такие как, AVR изначально имеют отдельное адресное пространство для памяти программ, памяти данных и периферии и потому там такие фокусы не пройдут, а программа по умолчанию записывается в ROM память.

*Важно*

Все адресное пространство в CortexM единое, и код и данные могут размещаться где угодно. С помощью настроек линкера можно задать регион для адресов ПЗУ(ROM) и ОЗУ(RAM) памяти. IAR располагает сегмент кода `.text` в регионе ROM памяти.

## Объектный файл и сегменты

Выше я упомянул про сегмент кода, давайте разберемся, что это такое.

На каждый компилируемый модуль создается отдельный объектный файл, который содержит следующую информацию:

- Сегменты кода и данных
- Отладочную информацию в формате DWARF
- Таблицу символов

Нас интересуют сегменты кода и данных.

Сегмент это такой элемент, содержащий часть кода или данных, который должен быть помещен по физическому адресу в памяти. Сегмент может содержать несколько фрагментов, обычно один фрагмент на каждую переменную или функцию. Сегмент может быть помещен как в ПЗУ(ROM) так и ОЗУ(RAM).

В общем и целом, сегмент это наименьший линкуемый блок.

## Атрибуты сегментов

Каждый сегмент имеет имя и атрибут, который определяет его содержимое. Атрибут используется для определения сегмента в конфигурации для линкера. Например, атрибуты могут быть: `* code` — исполняемый код `* readonly` — константные переменные `* readwrite` — инициализируемые переменные `* zeroinit` — инициализируемые нулем переменные

Конечно есть и другие типы сегментов, например сегменты, содержащие отладочную информацию, но нас будут интересовать только те, которые содержат код или данные нашего приложения.

Повторюсь, сегмент это наименьший линкуемый блок. Однако при необходимости линкеру можно указать и еще более мелкие блоки(фрагменты). Этот вариант рассматривать не будем, остановимся на сегментах.

## Предопределенные имена сегментов в IAR Workbench

Во время компиляции данные и функции размещаются в различные сегменты. А во время линковки, линкер назначает им реальные физические адреса. В компиляторе IAR есть предопределенные имена сегментов, некоторые из них приведены ниже:

- `.bss` — Содержит статические и глобальные переменные инициализируемые 0
- `.CSTACK` — Содержит стек используемый программой
- `.data` — Содержит статические и глобальные инициализируемые переменные
- `.data_init` — Содержит начальные значения для данных в `.data` секции, если используется директива инициализации для линкера
- `HEAP` — Содержит кучу, используемую для размещения динамических данных
- `.intvec` — Содержит таблицу векторов прерываний
- `.rodata` — Содержит константные данные
- `.text` — Содержит код программы

На практике это означает, что если вы определили переменную `int val = 3`, то сама переменная будет расположена компилятором в сегмент `.data` и помечена атрибутом `readwrite`, а число 3 может быть помещено либо в сегмент `.text`, либо в сегмент `.rodata` или, если применена специальная директива для линкера в `.data_init` и также помечается им как `readonly`.

Сегмент `.rodata` содержит константные данные и включает в себя константные переменные, строки, агрегатные литералы и так далее. И этот сегмент может быть размещен где угодно в памяти.

## Файл настройки компоновщика

Файл линкера имеет расширение `*.icf`. В нашем проекте этот файл называется `stm32f411xE.icf`. Давайте теперь поймем, что же прописано в настройках линкера и почему.

```

define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0807FFFF;
define symbol __ICFEDIT_region_RAM_start__  = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__    = 0x2001FFFF;
define region ROM_region  = mem:[from __ICFEDIT_region_ROM_start__ to
__ICFEDIT_region_ROM_end__];
define region RAM_region  = mem:[from __ICFEDIT_region_RAM_start__ to
__ICFEDIT_region_RAM_end__];

// Разместить сегменты .rodata и .data_init (константы и инициализаторы) в
(ПЗУ)ROM:

place in ROM_region { readonly };

// Разместить сегменты .data, .bss, .noinit, STACK и HEAP в (ОЗУ)RAM

place in RAM_region { readwrite, block STACK , block HEAP };

```

# Настройка стека

## Стек

Для начала определение из Википедии:

Стек (англ. Stack - стопка; читается стэк) - абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

В стек можно положить данные, и можно данные забрать, причем те данные которые были положены в стек последним, забираем из стека первым

Стек – это организация памяти, выполненная компоновщиком. На уровне микроконтроллера для работы со стеком есть специальные ассемблерные команды (например PUSH – положить регистры в стек, и POP – взять из стека). Так же для сохранения и считывания данных из стека могут использоваться инструкции STR и LDR

Обычно в стеке сохраняются регистры когда вы вызываете подпрограмму, или проваливаетесь в прерывание, для того, чтобы когда вернуться обратно в вашу программу восстановить весь контекст и все переменные. Кроме того, если в вашей функции передается слишком много переменных и под все не хватит регистров, то компилятор расположит их также на стеке. Локальные переменные функции также создаются на стеке.

В традиционной реализации память для всех локальных переменных функции выделяется сразу, одним "кадром стека" в начале работы функции. Внутри этого кадра стека компилятор еще на стадии компиляции разработает некую фиксированную карту расположения локальных переменных. При этом он может (и будет) располагать локальные переменные в этой карте совершенно произвольным образом, руководствуясь оптимизационными соображениями выравнивания, экономии памяти и т.д. и т.п.

## Правила задания размера стека

В большинстве "традиционных" платформ стек растет сверху-вниз: от старших адресов к младшим. Поэтому прежде всего нужно верно указать размер или вершину стека. Для того, чтобы сделать это есть пара правил:

1. Всегда считаем, что все локальные переменные создаются на стеке (Хотя часть из них могут быть созданы и на регистрах)
2. У нас 16 регистров + регистры блока с плавающей точкой. Которые должны быть сохранены на стеке
3. Каждая вложенная подпрограмма должна сохранить на стеке все данные из пункта 1 и 2. Т.е. если вложенность будет 2, то и сохранять придется примерно в два раза больше данных
4. Каждое прерывание должно сохранить данные из пункта 1 и 2.



# Установка размера стека

Обычно размер стека вычисляется эмпирически и задается с небольшим запасом.

Чтобы задать размер стека, нужно нажав на правую кнопку мыши на проекте, выбрать Option⇒Linker и нажать кнопку Edit, далее выбрать закладку Stack/Heap Size, см. [\[Установка размера стека и кучи\]](#)

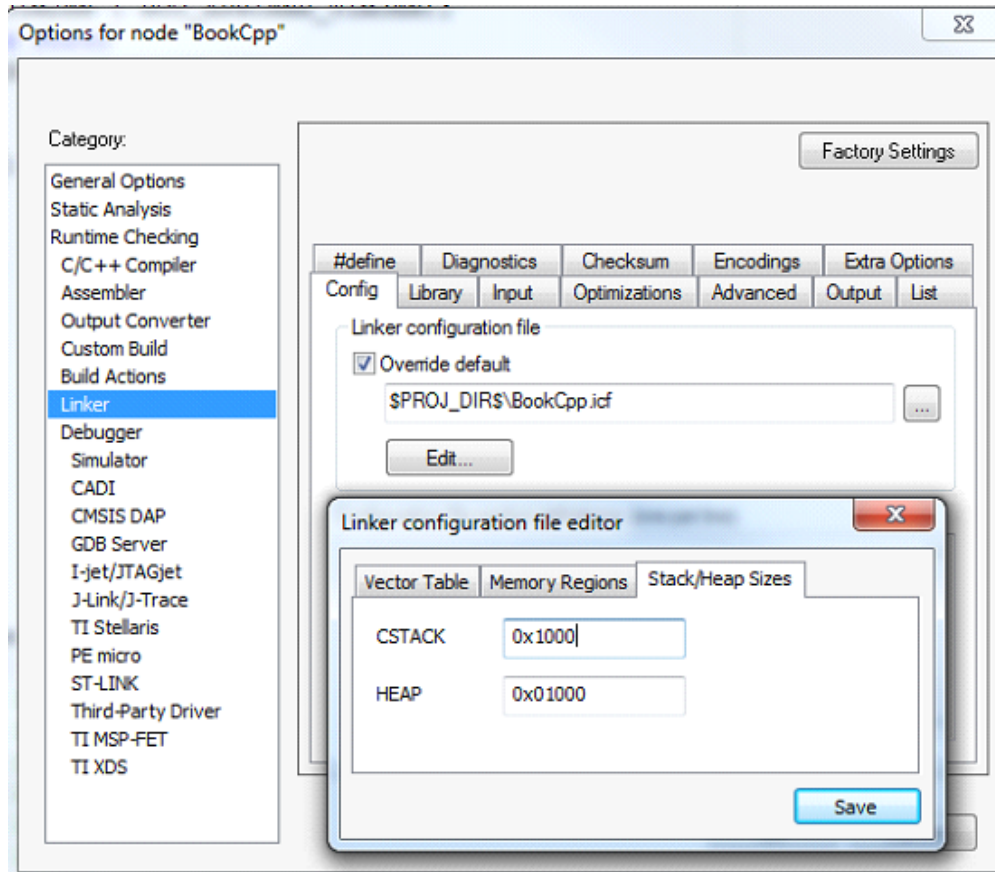


Рисунок 22. Установка размера стека и кучи

Тоже самое можно сделать руками в файле `stm32f411xE.icf`, поменяв значение символа `__ICFEDIT_size_cstack`

## Контроль за размером стеком

IAR Workbench имеет встроенные средства для контроля стека на этапе сборки он может указать максимально возможный размер стека для вашего приложения для самой глубокой цепочки вызова функций.

Это значение можно использовать как ориентир при установке максимального значения стека. Однако следует помнить, что во-первых, в вашей программе возможно никогда не будет самой глубокой цепочки вложенности, а во вторых не всегда компоновщик сможет определить верно размер, например, при использовании ОСРВ, указатель стека постоянно изменяется и стек выделяется под каждую задачу отдельно, в итоге вся программа может работать вообще без единого стека и его размер можно минимальным. Зато придется указывать размер стека для каждой задачи при её создании. В любом случае, очень полезно знать об этой особенности и как её задействовать.

Для включения достаточно поставить галочку в меню Option⇒Linker⇒Advanced⇒Enable stack usage analysis см. [\[Опция анализа глубины стека\]](#)

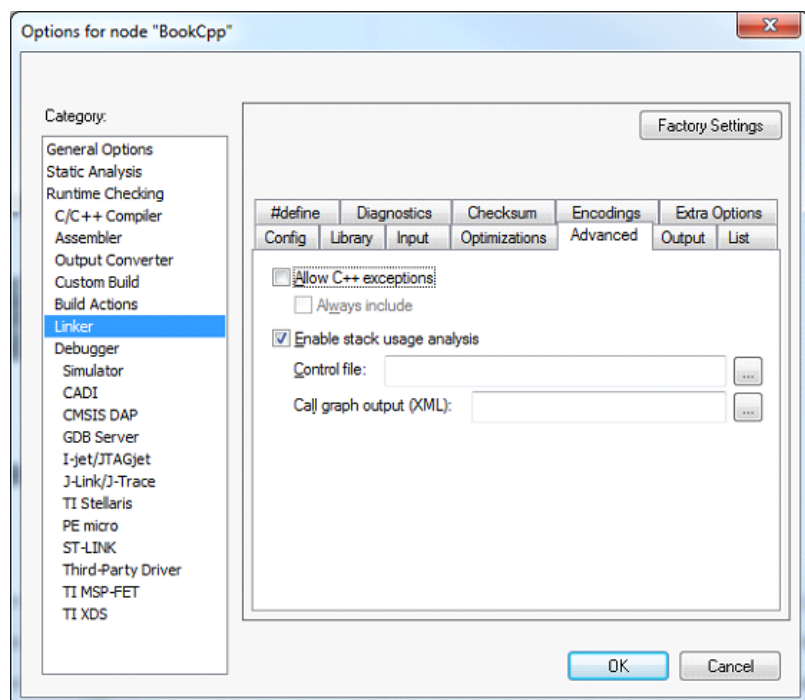


Рисунок 23. Опция анализа глубины стека

## Доступ к данным по анализу размеру стека

После установки этой опции на выходе компоновщика в файле с расширением \*.map можно будет увидеть результат анализа, например, такой:

Call Graph Root Category	Max Use	Total Use
Program entry	896	896
Uncalled function	0	0

Program entry  
 "\_\_iar\_program\_start": 0x08005291

Maximum call chain 896 bytes

"__iar_program_start"	8
"__cmain"	0
"main"	88
"std::ostream::operator <<(float)"	80
"std::num_punct<char>::grouping() const"	8
"std::num_punct<char>::do_grouping() const"	8
"std::string::basic_string(char const *)"	16
"std::string::assign(char const *)"	16
"std::string::assign(char const *, unsigned int)"	16
"std::string::assign(const std::string&, unsigned int, unsigned int)"	32
"std::string::_Grow(unsigned int, bool)"	16
"std::string::_Copy(unsigned int, unsigned int)"	32

В данном случае анализ стека показывает, что размер стека при максимальной цепочке вложенности может быть 896 байт.

## Куча

Куча (англ. heap) - структура данных с помощью которой организуется динамическое распределение памяти приложения. Размер кучи — размер памяти, выделенной операционной системой (ОС) для хранения кучи (под кучу).

Компоновщик выделяет раздел памяти под кучу в соответствии с заданным размером кучи, а при запуске программы происходит инициализация кучи, в ходе которой память, выделенная под кучу, отмечается как свободная.

Куча используется только при динамически выделяемой памяти, для нас это означает, что все объекты созданные с помощью оператора new будут расположены в куче.

Механизм выделения памяти описывать не будем, просто нужно запомнить, что если объект создан с помощью оператора new, то все его содержимое хранится в куче.

Я не советую использовать динамическое создание объектов. Так как динамическое выделение памяти не рекомендуется для использования в надежном ПО. Лучше делать все объекты статическими.

# Определение размера кучи

Как определить размер кучи, необходимой под кучу. Можно вооружиться несколькими правилами:

- Чтобы узнать размер объекта в куче, можно воспользоваться оператором `sizeof`, который может вернуть вам размер в байтах типа объекта (собственно, он будет равен размеру объекта расположенному в куче). Таким образом узнав размер всех объектов, можно приблизительно вычислить необходимый размер кучи
- Поскольку на кучи объекты могут как создаваться так и удаляться из неё, то куча может получаться неаргументированной, т.е. между объектами может быть пустая, незаполненная память. Поэтому если вы постоянно создаете и удаляете объекты, нужно учитывать этот факт и брать размер кучи с запасом.
- Размер кучи зависит от алгоритма работы вашей программы, если вы будете создавать и удалять последовательно объекты 100 раз, то нет никакого резона создавать кучу на 100 объектов, вполне разумно, создать кучу под 1-2 объекта с запасом на дефрагментацию – скажем 20% и все.

Как вы поняли использование кучу несет ряд трудностей с расчетом её размера, помимо этого использование кучи может тормозить выполнение программы., см, например, [\[Обзор одной российской RTOS\]](#). Поэтому во встроенном ПО использование кучи не приветствуется, по возможности её надо избегать, однако некоторые архитектурные приемы невозможны без использования динамических объектов (например для позднего связывания, или факта того, что мы не хотим использовать глобальные объекты), поэтому использовать в курсовых вы можете, но с одним условием, в нашем программном обеспечении созданные динамические объекты никогда не должны удаляться. Таким образом мы избежим дефрагментации кучи, а также слежением за памятью.

Для задачи размера кучи, нужно сделать те же действия что для задания размера стека, см. [Установка размера стека](#)

# Задания

3 Задания, кто не успеет в лабораторной, завершить дома.

## Задание 1

1. Создать проект C++ с main.cpp
2. Подключить к проекту файл cstartup.cpp
3. Создать папки AbstractHardware/Registers/FiledValues, Common, Application, FreeRtos
4. Создать структуру проекта в соответствии со структурой папок
5. Настроить STACK, HEAP
6. Скопировать содержимое папки Registers и Common с проекта преподавателя в свою папку
7. Написать программу в main.cpp

```
#include "gpioregisters.hpp" //for GPIOC
#include "rccregisters.hpp"  //for RCC

int main()
{
    RCC::AHB1ENR::GPIOCEN::Enable::Set() ;
    GPIOC::MODER::MODER5::Output::Set() ;
    GPIOC::ODR::ODR5::Enable::Set() ;
    GPIOC::ODR::ODR5::Disable::Set() ;
    return 0 ;
}
```

1. Посмотреть видео: <https://youtu.be/uC0jJGfDxtM>

## Задание 2

1. Откомпилировать и отлинковать программу
2. Загрузить программу в симуляторе
3. Сделать пошаговую отладку
4. Настроить Debugger на отладку через StLink
5. Подключить плату к компьютеру
6. Загрузить программу в плату
7. Выполнить пошаговую отладку
8. Описать полученный результат
9. Посмотреть видео: <https://youtu.be/c7CasTJKw7o>

## Задание 3

1. Запустить анализатор стека. Узнать рекомендуемый размер стека.
2. Изменить в проекте размер стека на рекомендуемый
3. Создать tar файл
4. Описать что написано в tar файле
5. Поставить размер кучи HEAP в 0. Объяснить почему так можно сделать. И почему STACK нельзя
6. Добавить проект в Git и сделать синхронизацию с GitHub
7. Сделать отчет по каждому пункту каждого задания в файле .adoc. Выложить файл в GitHub
8. Прислать ссылку на GitHub преподавателю для проверки
9. Посмотреть видео: <https://youtu.be/TajLTcjBgIg>