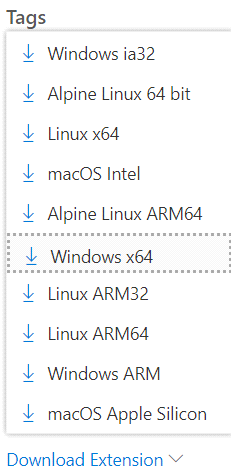
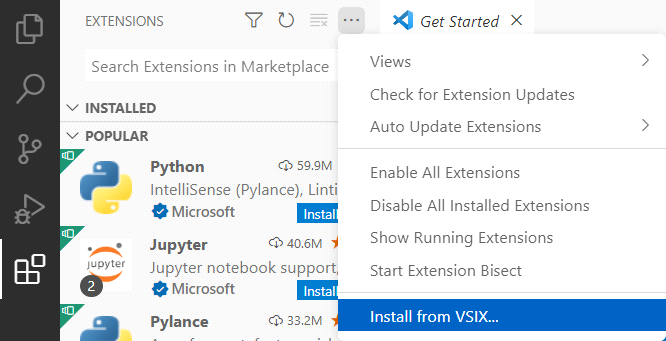
**1 Быстрый старт**. Далее приведена последовательность действий по установке и настройке инструментов разработки для компиляции простейшей программы на C++ под Windows.

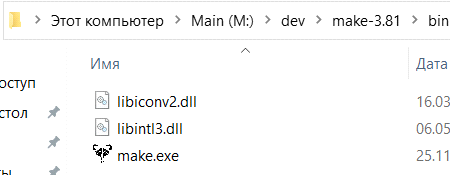
* Скачать Visual Studio Code с официального [сайта](https://code.visualstudio.com/) проекта.
* Скачать расширение для C++ с [marketplace](https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools) проекта Visual Studio для установки offline. Для этого выбрать «Download Extension» Windows x64.



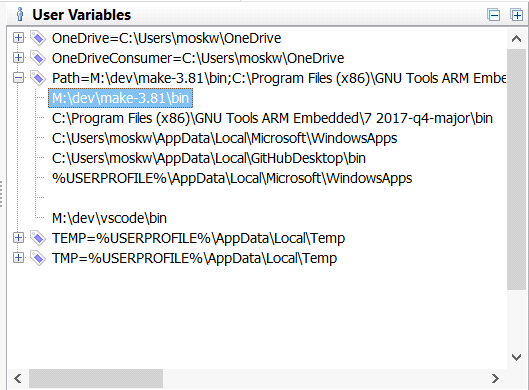
* Установить vscode (права администратора не требуются).
* Запустить vscode и установить расширение для C++. Для этого нужно на панели слева выбрать «Extensions», затем на открывшейся вкладке «EXTENSIONS» выбрать «…» «Install from VSIX…». Выбрать ранее скачанное расширение для C++.



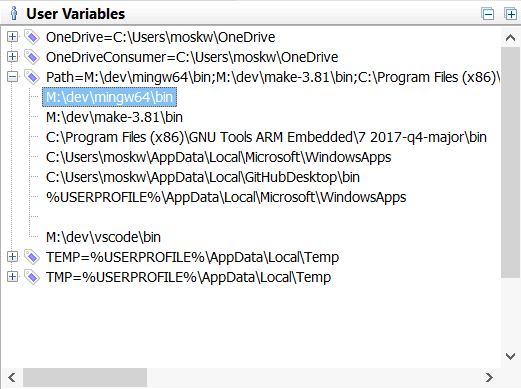
* Скачать программу управления компиляцией [make для Windows](http://gnuwin32.sourceforge.net/packages/make.htm), а также зависимости (Dependencies) для неё. Зависимости необходимо переместить в каталог с make.exe:



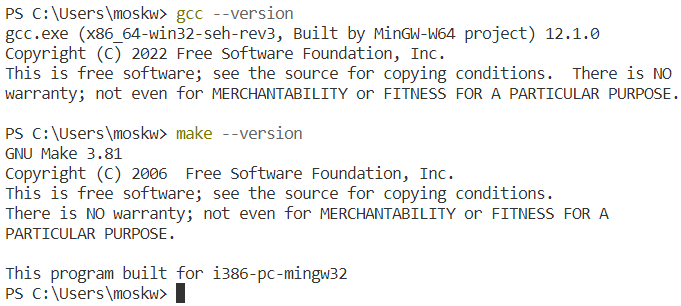
* Прописать путь к программе make в переменной PATH, например, с помощью редактора [rapidee](https://www.rapidee.com/en/download). Установка редактора не требует прав администратора. Запись в User Variables доступна без прав администратора.



* Скачать пакет инструментов для компиляции – [mingw](https://www.mingw-w64.org/downloads/).
* Аналогичным образом прописать путь к этим инструментам в переменной PATH.



* Перезапустить vscode чтобы процесс мог получить новое окружение.
* Открыть терминал через меню «View» «Terminal».
* Проверить факт корректной установки компилятора и программы make. Для этого ввести в терминале команды gcc --version, make --version. Должны увидеть примерно такой вывод:



* Всё готово для того, чтобы написать и скомпилировать нашу первую программу на C++ в среде разработки vscode. Сперва нужно создать каталог для проекта, например, «ex00-hellovscode». Этот каталог будет являться workspace в терминах среды разработки. Т.е. будет содержать дерево исходных кодов проекта, файлы описания проекта, скрипты для построения этих файлов описания проекта, а также служебные каталоги и файлы.
* Открыть каталог «ex00-hellovscode» в среде vscode. Для этого можно воспользоваться п. меню «File» «Open Folder…». Строка состояния vscode должна стать синего цвета. В диалоге «доверять ли авторам», разумеется, ответить «Yes».
* Добавить файл исходного кода «main.cpp» и ввести в него следующий код:

#include <iostream>

int main (void)

{

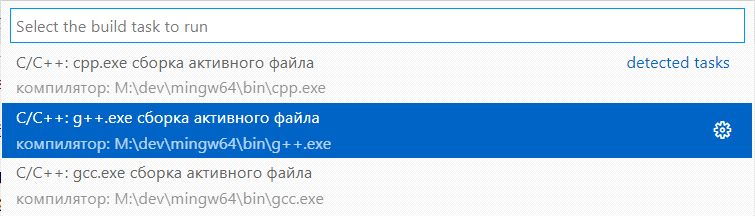
std::cout << "hello, fucking world!" << std::endl;

return 0;

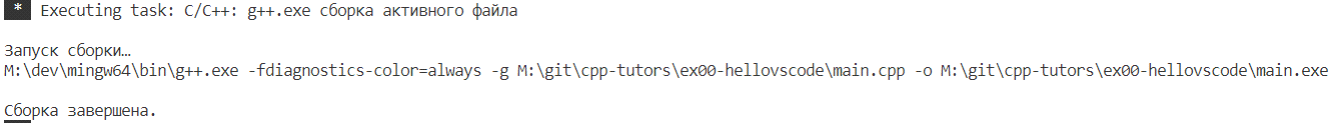
}

И сохранить его.

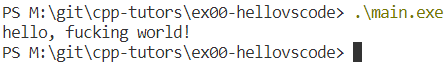
* Скомпилировать полученный файл, вызвав задачу построения по-умолчанию. Для этого выбрать п. меню «Terminal» «Run build task…», или сочетание клавиш CTRL-SHIFT-B. vscode сам попытается определить как собирать полученный исходный файл и предложит наиболее подходящий вариант. Нам нужен g++:



Далее последует процесс компиляции, и мы увидим в консоли терминала нечто подобное:



* Попробуем запустить скомпилированную программу. Для этого вызовем её в терминале:

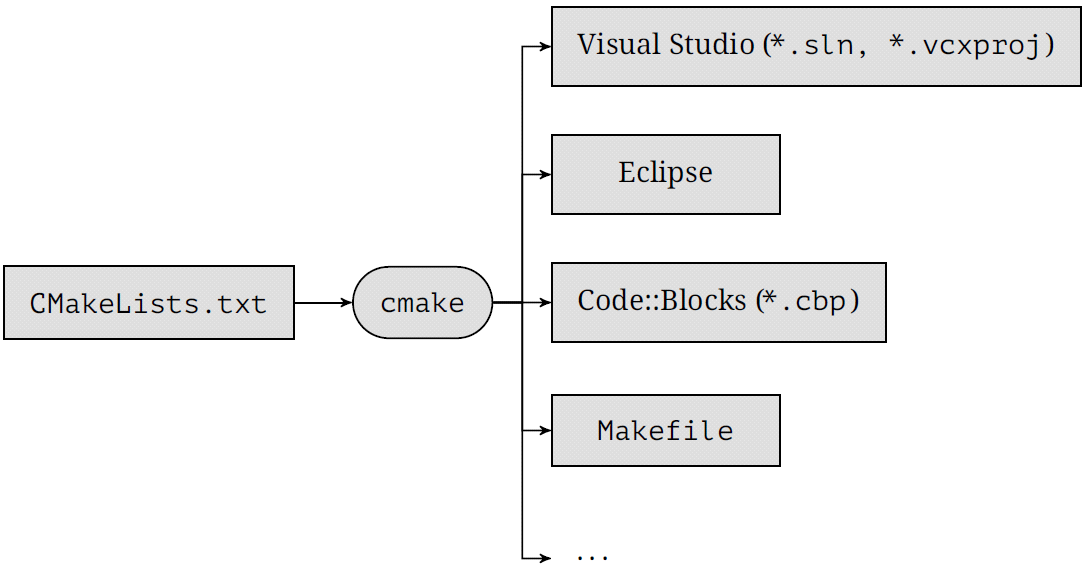


Мои поздравления! Вы написали, скомпилировали и запустили вашу первую программу на C++ в среде разработки vscode практически out-of-box. Мы ничего не делали в плане конфигурации проекта, по сути, мы просто использовали vscode как текстовый редактор с возможностями подсветки синтаксиса и автодополнения (благодаря расширению для С++). Вы можете заметить, что в вашем каталоге проекта не появилось ни одного лишнего файла, даже шаблона-затычки для более комплексных задач.

Однако для сколь-угодно сложных проектов, со сложной иерархией дерева исходных кодов, внешними зависимостями и специфичными для целевой платформы настройками, требуется способ описания проекта, а также настройка среды для сборки и отладки целей этого проекта.

**2 Система построения проектов CMake**.

CMake – это кроссплатформенная система построения проектов. Она создает файлы проекта, пригодные для использования конечными инструментами сборки, такими как make, ninja, Visual Studio, eclipse и др. Такой посредник позволяет использовать единообразный подход к описанию проекта, безотносительно конечных инструментов сборки, и использует для этого интерпретируемый язык CMake. Синтаксис языка CMake является более высокоуровневым по сравнению, например, с языком описания проектов для системы сборки make, тем самым, значительно упрощая задачу управления компиляцией. Исходными файлами для системы CMake являются файлы CMakeLists.txt, а также подключаемые в них файлы с расширением \*.cmake.



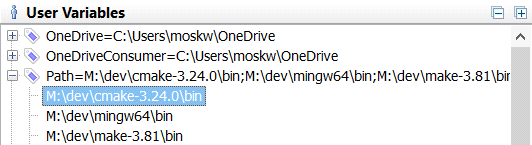
Тогда мы разработаем описание проекта CMakeLists.txt, чтобы на его основе система cmake сгенерировала описание проекта Makefile для системы сборки make, установленной на предыдущем шаге. Позже мы можем заменить систему сборки make на любую другую и нам совершенно не нужно будет вникать в специфичный для неё способ описания проекта. Такая универсальность даёт возможность менять системы сборки очень легко и быстро, всего лишь путем изменения значения ключа при вызове CMake.

Создадим первое описание для нашего первого проекта на языке CMake. Для этого нам понадобится скачать расширение CMake для vscode, установить инструменты CMake, создать описание проекта в файле CMakeLists.txt и затем вызвать cmake с соответствующими аргументами в терминале для построения файлов проекта.

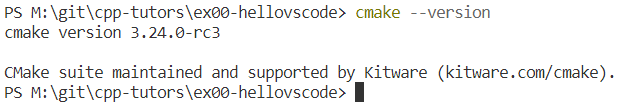
* Скачать расширение vscode – CMake language c [marketplace](https://marketplace.visualstudio.com/items?itemName=twxs.cmake) проекта Visual Studio. Данное расширение необходимо для поддержки подсветки синтаксиса и автодополнения языка CMake.
* Установить расширение CMake language в vscode. Для этого нужно на панели слева выбрать «Extensions», затем на открывшейся вкладке «EXTENSIONS» выбрать «…» «Install from VSIX…». Выбрать ранее скачанное расширение CMake language.

NB: можно заметить, что на marketplace имеется еще одно расширение для поддержки CMake – CMake Tools. Оно позволяет автоматически конфигурировать проект, и получить на выходе все необходимые файлы для построения. Однако на начальных этапах, для лучшего понимания принципов работы CMake, это расширение мы устанавливать пока не будем.

* Скачать инструменты CMake с [сайта](https://cmake.org/download/) проекта CMake.
* Прописать путь к этим инструментам в переменной PATH.



* Перезапустить vscode, чтобы процесс получил новое окружение.
* Проверить в терминале vscode факт корректной установки cmake. Для этого выполнить команду cmake --version. Должны увидеть приблизительно такой вывод:



Все готово для того, чтобы написать наш первый скрипт CMake, который сгенерирует Makefile для сборки нашего первого приложения.

В первую очередь, нам необходимо написать файл с описанием используемых инструментов сборки для cmake. Этот файл необходимо будет указать в специальной переменной CMAKE\_TOOLCHAIN\_FILE, которая используется при кросс-компиляции. Переменная CMAKE\_TOOLCHAIN\_FILE содержит путь к файлу, который считывается на раннем этапе при запуске CMake, и в котором прописаны пути к компилятору, различным утилитам и другая зависимая от платформы информация.

* В каталоге проекта создать новый файл «mingw64.cmake» и записать в него следующие строчки:

set(CMAKE\_SYSTEM\_NAME Windows)

set(CMAKE\_SYSTEM\_PROCESSOR AMD64)

set(CMAKE\_C\_COMPILER "gcc.exe")

set(CMAKE\_ASM\_COMPILER "as.exe")

set(CMAKE\_CXX\_COMPILER "g++.exe")

set(CMAKE\_OBJCOPY "objcopy.exe" CACHE INTERNAL "objcopy tool")

set(CMAKE\_SIZE\_UTIL "size.exe" CACHE INTERNAL "size tool")

* В каталоге проекта создать новый файл CMakeLists.txt и записать в него следующие строки:

cmake\_minimum\_required(VERSION 3.15.3)

project(hello)

set(CMAKE\_VERBOSE\_MAKEFILE ON)

set(CMAKE\_TOOLCHAIN\_FILE ${CMAKE\_SOURCE\_DIR}/mingw64.cmake)

# Ключи компиляции

add\_compile\_options(-std=c++17 -Wall -g)

# Сборка исполняемого файла

add\_executable(${PROJECT\_NAME} main.cpp)

* Создать подкаталог build/, в котором будут разворачиваться файлы проекта, а также будет происходить сборка проекта.
* Зайти в каталог build/ и запустить cmake командой

cmake -DCMAKE\_MAKE\_PROGRAM=make.exe -G "MinGW Makefiles" ..

После выполнения данной команды в каталоге build/ должен появиться сгенерированный Makefile, пригодный для передачи его программе управления компиляцией make.

* Выполнить сборку проекта с помощью команды

cmake --build .

* Проверить успешность построения, запустив приложение hello:



Мои поздравления! В этом разделе у нас получилось сконфигурировать проект таким образом, чтобы он стал пригоден для сборки с помощью программы make и компилятора MinGW.

**3 Автоматизированная сборка/очистка с помощью vscode**.

Вместо ручного консольного ввода команд сборки проекта, мы можем задать конфигурацию для задачи сборки по-умолчанию в vscode. Тогда мы сможем запускать процедуру сборки из меню «Terminal» 🡪 «Run Build Task...», или сочетанием клавиш CTRL-SHIFT-B, вместо того, чтобы печатать соответствующие консольные команды.

Привязку различных инструментов к среде vscode обеспечивают задачи (tasks), записанные в формате JSON.

* Создать JSON файл для описания задач сборки проекта. Для этого выбрать п. меню "Terminal" --> "Configure Tasks...". В появившемся списке выбрать "Create tasks.json from template", и далее "Others Example to run an arbitrary external command".

NB: нужного шаблона еще нет в vscode, т.к. мы всё еще не установили расширение CMake Tools.

В каталоге проекта появится подкаталог .vscode, в котором находится только что созданный нами шаблон задачи tasks.json.

* Записать задачу сборки проекта:

"type": "shell",

"label": "Build",

"command": "cmake --build .",

"options": {

"cwd": "${workspaceFolder}/build"

},

"group": {

"kind": "build",

"isDefault": true

},

"problemMatcher": {

"base": "$gcc",

"fileLocation": ["relative", "${workspaceFolder}/build"]

}

* Записать задачу очистки проекта:

"type": "shell",

"label": "Clean",

"command": "cmake --build . --target clean",

"options": {

"cwd": "${workspaceFolder}/build"

},

"group": {

"kind": "build",

"isDefault": false

}

NB: задачи записываются в фигурных скобках, через запятую { }, { }, ...

* Выполнить сборку программы, вызвав задачу "Build" сборки по умолчанию через п. меню «Terminal» 🡪 «Run Build Task...», либо через сочетание клавиш CTRL-SHIFT-B. В каталоге build/ должен появиться исполняемый файл "hello.exe".
* Выполнить очистку проекта, вызвав задачу «Clean», для этого выбрать п. меню «Terminal» 🡪 «Run Task...», в появившемся списке ввести «Clean» и выбрать его. Все выходные файлы должны быть удалены.

Таким образом, теперь мы можем собирать наш проект с помощью непосредственно среды vscode, вместо того, чтобы вбивать соответствующие команды в консоли вручную.

NB: процесс заполнения значений свойств задач в файле tasks.json называется их конфигурированием.

NB: задачу также можно вызвать через п. меню «Go» 🡪 «Go To File…», или нажатием CTRL-P. Для этого требуется ввести в строчку выбора «task », обратите внимание на пробел в конце.

**4 Автоматизированный запуск/отладка с помощью vscode**.

Для возможности запуска и отладки приложений с помощью отладчика gdb, нам необходимо записать задачу запуска в формате JSON в файл launch.json.

В целях проверки возможности отладки усложним немного наше тестовое приложение. Разработаем класс, выполняющий конвертацию объявленных им целочисленных кодов ошибок в осмысленное сообщение, пригодное для вывода в поток. Напишем тестовую функцию для этого класса.

* Записать в main.cpp строчки исходного кода:

#include <iostream>

class err\_c

{

static const std::size\_t msgcnt = 32;

static const char \*messages[msgcnt];

int \_err;

public:

enum {

OK, NULL\_POINTER, OUT\_OF\_RANGE, INVALID\_ARG, UNKNOWN\_COMMAND,

};

err\_c(int err) : \_err(err) { }

const char \*operator()() { return messages[\_err]; }

};

const char \*err\_c::messages[err\_c::msgcnt] = {

"OK", "null pointer", "out of range", "invalid arg", "unknown command"

};

int foo(void \*p) {

if (!p) {

// can not work. Return

return err\_c::NULL\_POINTER;

} else {

// do any work here

// ...

return 0;

}

}

int main (void)

{

static int x;

std::cout << "foo result is: " << err\_c(foo(&x))() << std::endl;

std::cout << "foo result is: " << err\_c(foo(nullptr))() << std::endl;

return 0;

}

err\_c представляет собой функтор, возвращающий сообщение по коду ошибки. Он совершенно не выполняет никакой функциональной логики и по сути просто является удобным способом организации объявления кодов ошибок – он ограничивает область видимости констант, обеспечивает централизованное хранение строк сообщений об ошибках. Использование этого класса вместо строковых литералов, может существенно сэкономить расход flash-памяти.

* Выбрать п. меню «Run» 🡪 «Add Configuration…». В появившемся списке выбрать «C++ (GDB/LLDB)».

В подкаталоге проекта появится новый файл конфигурации – launch.json.

* Записать задачу запуска отладки:

{

"version": "0.2.0",

"configurations": [

{

"cwd": "${workspaceRoot}",

"name": "Debug",

"type": "cppdbg",

"request": "launch",

"program": "${workspaceRoot}/build/hello.exe",

"stopAtEntry": false,

"MIMode": "gdb",

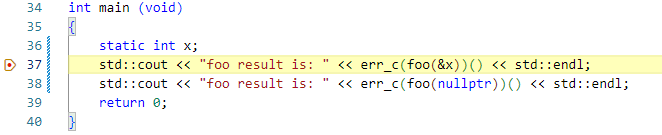
}

]

}

* Установить точку останова на первую строчку вызова foo() и запустить отладку с помощью п. меню «Run» 🡪 «Start Debugging» или клавиши F5.

После запуска отладки исполнение приложения должно быть остановлено на выбранной точке останова:



В верхней части vscode должны появиться элементы управления ходом отладки:



* Выполнять поочередно шаг с заходом с помощью соответствующей кнопки панели отладки или клавиши F11, проследить поочередный заход в функцию foo(), затем в конструктор err\_c. На панели слева отображается стек вызовов, локальные и наблюдаемые переменные.

**5 Подключение библиотеки Boost**.

Библиотека boost поставляется в заголовочных файлах и для использования большей части её возможностей компиляция библиотеки не требуется. Всё что нам необходимо сделать – скачать её с официального [сайта](https://www.boost.org/) проекта и указать компилятору где находятся её заголовочные файлы. Библиотека нам понадобится далее для просмотра выводимых типов.

* Скачать библиотеку с официального [сайта](https://www.boost.org/), на момент написания статьи последняя версия библиотеки – 1.79.0.
* Распаковать библиотеку в выбранном вами каталоге, например, M:\boost\_1\_79\_0.
* В скрипте CMakeLists.txt добавить команду подключения заголовочных файлов:

cmake\_minimum\_required(VERSION 3.15.3)

project(hello)

set(CMAKE\_VERBOSE\_MAKEFILE ON)

set(CMAKE\_TOOLCHAIN\_FILE ${CMAKE\_SOURCE\_DIR}/mingw64.cmake)

# Ключи компиляции

add\_compile\_options(-std=c++17 -Wall -g)

# Заголовочные файлы

include\_directories("M:\\boost\_1\_79\_0")

# Сборка исполняемого файла

add\_executable(${PROJECT\_NAME} main.cpp)

* Перезапустить cmake, чтобы новая информация о путях к заголовочным файлам стала известна компилятору. Для этого зайти в каталог build/ и еще раз выполнить команду:

cmake -DCMAKE\_MAKE\_PROGRAM=make.exe -G "MinGW Makefiles" ..

* Теперь с помощью директивы #include мы можем добавить заголовочный файл boost/type\_index.hpp.

NB: поскольку мы не используем (пока) расширение CMake Tools, intelliSense не сможет ничего узнать о новых путях поиска заголовочных файлов или директивах препроцессора, записанных в CMakeLists.txt. Поэтому временно придется смириться с дискомфортом в виде подчеркнутого «неизвестного» заголовочного файла и отсутствием автодополнения при наборе кода.

* Добавить строчки кода в main.cpp:

auto x1 = {0};

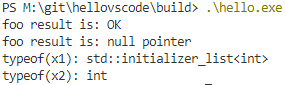
auto x2 = 0;

std::cout << "typeof(x1): " << type\_id\_with\_cvr<decltype(x1)>().pretty\_name() << std::endl;

std::cout << "typeof(x2): " << type\_id\_with\_cvr<decltype(x2)>().pretty\_name() << std::endl;

* Скомпилируем приложение с помощью ранее созданной задачи сборки по умолчанию, нажав сочетание клавиш CTRL-SHIFT-B.
* Запускаем на исполнение любым удобным способом – из консоли терминала vscode или с помощью ранее созданной задачи запуска отладки по клавише F5.

Должны увидеть следующий вывод программы hello.exe:



Мои поздравления! Мы не только успешно подключили библиотеку boost, но еще и узнали, что компилятор по-разному выводит тип переменной, в зависимости от вида её инициализатора. Можете пофантазировать, и попробовать повыводить еще разные типы, например, добавьте следующие строчки кода в main.cpp:

auto arg = x2;

std::cout << "typeof(arg): " << type\_id\_with\_cvr<decltype(arg)>().pretty\_name() << std::endl;

std::cout << "typeof((arg)): " << type\_id\_with\_cvr<decltype((arg))>().pretty\_name() << std::endl;

Выполните сборку и запустите программу. Должны увидеть следующий вывод:



В первом случае вы имеете lvalue, а во втором ссылку на lvalue. Вот это поворот!

**6 Конфигурация проекта с помощью CMake Tools**.

В этом разделе мы установим расширение CMake Tools, которое избавит нас от ручного консольного ввода команды cmake, будет отслеживать все изменения скриптов CMakeLists.txt и конфигурировать проект при необходимости автоматически. Это расширение также установит необходимую связь между описанием проекта в CMakeLists.txt и службой intelliSense среды vscode. Взаимодействие расширения CMake Tools и CMake основано на т.н. cmake-file-api.

Конфигурация проекта CMake подразумевает:

1. Выбор активного кита (комплекта инструментов для компиляции и компоновки).
2. Выбор CMake генератора.
3. Заполнение кэша переменных CMake.
4. Вызов CMake и развертывание файлов проекта.

Кроме этого, CMake Tools способен также выполнять конечную сборку целей и выполнять процедуру очистки.

* Удалить всё из каталога сборки проекта build/.
* Скачать и установить расширение [CMake Tools](https://marketplace.visualstudio.com/items?itemName=ms-vscode.cmake-tools) как мы это уже проделывали дважды ранее с расширениями для C++ и подсветки синтаксиса CMake.
* Выполнить сканирование комплектов с помощью команды «CMake: Scan for kits» из палитры команд. CMake Tools попытается найти все установленные на вашем компьютере комплекты компиляторов. При этом поиск производится по каталогам, указанным в переменной $PATH, и путям, в которые обычно устанавливаются компиляторы.

Все найденные комплекты CMake Tools записывает в JSON файл cmake-tools-kits.json, расположенный в каталоге C:\Users\<User>\AppData\Local\CMakeTools\. На моей машине этот файл имеет вид:

[

{

"name": "GCC 12.1.0 x86\_64-w64-mingw32",

"compilers": {

"C": "M:\\apps\\mingw64\\bin\\gcc.exe",

"CXX": "M:\\apps\\mingw64\\bin\\g++.exe"

},

"preferredGenerator": {

"name": "MinGW Makefiles"

},

"environmentVariables": {

"CMT\_MINGW\_PATH": "M:\\apps\\mingw64\\bin"

}

}

]

Из этого файла видно, что найден единственный комплект с компилятором GCC 12.1.0 и это единственный компилятор, который сейчас установлен на моём домашнем компьютере.

Обратите внимание на атрибут preferredGenerator, который учитывается (но не в первую очередь) при выборе генератора для вызова CMake.

Следует иметь ввиду, что, если нужный компилятор не нашелся, можно добавить его в этот файл вручную. Тогда эта запись будет иметь глобальную область видимости для всех проектов пользователя <User>.

NB: на самом деле сканирование комплектов будет запущено автоматически после установки расширения CMake Tools, о чем свидетельствует сообщение в консоли:

[kit] Successfully loaded 1 kits from C:\Users\moskw\AppData\Local\CMakeTools\cmake-tools-kits.json

Выбранный комплект отображается в строке состояния, в нижней части редактора vscode:



Мы можем щелкнуть по нему и выбрать другой комплект при необходимости.

Если внимательно посмотреть на файл cmake-tools-kits.json, то можно понять, что он содержит те же сведения, что и toolchain-файл (NB: mingw64.cmake), созданный нами ранее на шаге 2. Следовательно, мы можем попытаться отказаться от него. Кроме этого, здесь также определена опция предпочитаемого генератора CMake – “MinGW Makefiles”, которую мы указывали ранее явно вручную при вводе команды cmake, на шаге 2.

* Удалить toolchain-файл mingw64.cmake и убрать строчку его подключения из CMakeLists.txt. Он нам больше не нужен, так как информация об используемом toolchain теперь будет браться из JSON файла.
* Выполнить конфигурацию проекта с помощью команды «CMake: Configure» из палитры команд. При этом начнется процесс заполнения кэша переменных CMake и генерация файлов проекта.

Если мы посмотрим на консоль вывода, то увидим там вызов программы cmake, аналогичный тому, что мы вбивали вручную на шаге 2:

[main] Configuring folder: hellovscode

[proc] Executing command: M:\apps\cmake-3.24.0\bin\cmake.EXE --no-warn-unused-cli -DCMAKE\_EXPORT\_COMPILE\_COMMANDS:BOOL=TRUE -DCMAKE\_BUILD\_TYPE:STRING=Debug -DCMAKE\_C\_COMPILER:FILEPATH=M:\apps\mingw64\bin\gcc.exe -DCMAKE\_CXX\_COMPILER:FILEPATH=M:\apps\mingw64\bin\g++.exe -Sm:/git/hellovscode -Bm:/git/hellovscode/build -G "MinGW Makefiles".

Таким образом, команда CMake: Configure избавляет нас от необходимости вручную заполнять кэш переменных и вызывать программу cmake в терминале vscode. Кроме того, нам больше не нужен toolchain-файл .cmake.

В результате работы программы cmake в каталоге build/ должен появиться Makefile для конечной сборки целей. Теперь мы можем собрать проект, вызвав задачу сборки по умолчанию, которую мы описали вручную на шаге 3. Однако с помощью расширения CMake Tools такая задача может быть создана автоматически.

* Удалить задачу сборки по умолчанию из файла tasks.json, прописанную нами ранее на шаге 2.
* Создать новую задачу сборки по умолчанию для расширения CMake Tools. Для этого выбрать п. меню «Terminal» 🡪 «Configure Tasks…» 🡪 «CMake: build».

Мы увидим, что в файл tasks.json добавится новая задача. Её можно кастомизировать по своему усмотрению, изменяя или добавляя значения полей. Изменим свойство «group», чтобы назначить эту задачу как задачу по умолчанию. Тогда задача сборки для CMake Tools должна иметь вид:

{

"type": "cmake",

"label": "CMake: build",

"command": "build",

"targets": [

"hello"

],

"group": {

"kind": "build",

"isDefault": true

},

"problemMatcher": [],

"detail": "CMake template build task"

}

* Вызвать задачу сборки по умолчанию с помощью п. меню «Terminal» 🡪 «Run Build Task…» или сочетания клавиш CTRL-SHIFT-B.
* В каталоге build/ должен появиться исполняемый файл нашей программы. Запустить его любым удобным способом и убедиться в корректности его работы.

Мои поздравления! У нас получилось построить приложение с помощью расширения CMake Tools. Благодаря автоматизированному поиску комплектов и конфигурации проекта нам больше не требуется вручную писать задачи построения и очистки, а также вручную описывать toolchain. Все что мы должны сделать – это просто описать цели построения.

**7 Установка комплекта xPack arm-none-eabi**.

К этому шагу мы уже обладаем достаточным набором знаний о vscode, CMake, CMake Tools чтобы начать разрабатывать и собирать программы под другие платформы, кроме Windows. В последующих шагах будет рассмотрена процедура выбора и описания целей, разработка скриптов CMakeLists.txt для генерации правил сборки ОСРВ FreeRTOS в статическую библиотеку librtos.a, предназначенную для исполнения на процессоре ARM Cortex-M3.

* Начнем с установки компилятора arm-none-eabi-gcc. Для всех платформ бинарные файлы xPack GNU Arm Embedded GCC выпущены как portable архивы, которые могут быть установлены в любое место. Архивы могут быть загружены со страницы [выпусков на GitHub](https://github.com/xpack-dev-tools/arm-none-eabi-gcc-xpack/releases/).

Windows-версия xPack GNU Arm Embedded GCC упакована в ZIP-архивы. Загрузите последнюю версию с именем, похожим на:

xpack-arm-none-eabi-gcc-11.2.1-1.2-win32-x64.zip

* Для установки вручную xPack GNU Arm Embedded GCC распакуйте архив и скопируйте папку с версией в каталог %USERPROFILE%\AppData\Roaming\xPacks\arm-none-eabi-gcc\. В соответствии с рекомендациями Microsoft, AppData\Roaming – это рекомендуемое местоположение для установки пользовательских пакетов. Вы можете сократить имя последней папки чтобы оставить только версию.
* Добавить распакованный toolchain в файл C:\Users\<name>\AppData\Local\CMakeTools\cmake-tools-kits.json, т.к. CMake Tools не сможет найти этот комплект самостоятельно:

[

{

“name”: “GCC 12.1.0 x86\_64-w64-mingw32”,

“compilers”: {

“C”: “M:\\apps\\mingw64\\bin\\gcc.exe”,

“CXX”: “M:\\apps\\mingw64\\bin\\g++.exe”

},

“preferredGenerator”: {

“name”: “MinGW Makefiles”

},

“environmentVariables”: {

“CMT\_MINGW\_PATH”: “M:\\apps\\mingw64\\bin”

}

},

{

“name”: “GCC 11.2.1 xpack-arm-none-eabi”,

“compilers” : {

“C”: “C:\\Users\\moskw\\AppData\\Roaming\\xPacks\\arm-none-eabi-gcc\\

11.2.1-1.2\\bin\\arm-none-eabi-gcc.exe”,

“CXX”: “C:\\Users\\moskw\\AppData\\Roaming\\xPacks\\

arm-none-eabi-gcc\\11.2.1-1.2\\bin\\arm-none-eabi-g++.exe”

},

“preferredGenerator”:{

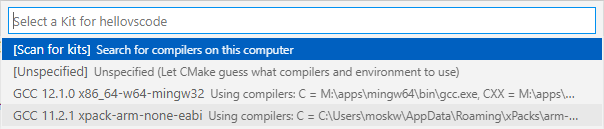
“name”: “Unix Makefiles”

}

}

]

* Теперь при выборе активного комплекта в строке состояния vscode у нас должен отобразиться новый комплект – «GCC 11.1.1 xpack-arm-none-eabi»:



* На этом мы заканчиваем работу с учебным проектом и переходим к «боевому». Закрыть hellovscode с помощью п. меню «File» 🡪 «Close Folder».

**8 Простейший проект для 1986ВЕ92У1 на базе ARM Cortex-M3**.

Прежде всего нам нужно создать базовую структуру проекта:

* Создать и открыть в vscode каталог приложения – *helloarm/*, создать в нем подкаталоги:

*docs/* – различная документация.

*src/* – дерево иерархии исходных кодов.

*include/* – заголовочные файлы проекта.

*build/* – файлы проекта (Makefile) и сборка проекта.

*ldscripts/* – скрипты компоновщика.

*CMSIS/* – исходные файлы библиотеки CMSIS.

*utils/* – различные вспомогательные утилиты и лайфхаки.

Такой базовой структуры каталогов предлагается придерживаться во всех проектах.

Далее нам необходимо добавить некоторые 3rd parties – библиотеку CMSIS для поддержки доступа к регистрам ядра и устройства, библиотеку Standard Peripheral Driver, а также скрипт компоновщика, который непосредственно управляет размещением кода и данных в памяти микроконтроллера. Для демонстрационного примера мы будем использовать микроконтроллер 1986ВЕ92У1. Скачать необходимую поддержку можно с сайта производителя микроконтроллера по [ссылке](https://ic.milandr.ru/soft/) (NB: или взять у меня).

* Копируем к себе в проект следующие файлы от производителя:

|  |  |  |
| --- | --- | --- |
| *Файл* | *Каталог назначения* | *Описание* |
| MDR32F9Qx.ld | ldscripts/ | Скрипт компоновщика |
| core\_cm3.h  core\_cmFunc.h  core\_cmInstr.h | CMSIS/Core/ | Поддержка доступа к процессору |
| MDR32Fx.h | CMSIS/Device/MDR32F9Qx/inc/ | Поддержка доступа к регистрам устройства |
| startup\_MDR32F9Qx.s, system\_MDR32F9Qx.c, system\_MDR32F9Qx.h | CMSIS/Device/MDR32F9Qx/startup/ | Вектор сброса, стартовый код, символы таблицы прерываний, SystemInit(), SystemCoreClockUpdate(). |
| MDR32F9Qx\_config.h | include/ | Подключает MDR32Fx.h, задает константы тактирования, прочие системные опции |
| src/\*.c, inc/\*.h | MDR32F9Qx\_StdPeriph\_Driver/ | Стандартная библиотека периферии |

Далее нам необходимо создать local-project файл описания комплекта ***.vscode/cmake-kits.json***. Он нужен, так как в общем случае, нам скорее всего для каждого проекта будет нужен свой toolchain-файл как минимум для того, чтобы указать в нём команду статической компоновки. Иначе в процессе конфигурации мы не сможем пройти этап проверки компилятора.

* Создать файл ***.vscode/cmake-kits.json*** и добавить туда следующие строчки:

[

{

"name": "GCC 11.2.1 xpack-arm-none-eabi",

"compilers" : {

"C": "<your-path-to>\\arm-none-eabi-gcc.exe",

"CXX": "<your-path-to>\\arm-none-eabi-g++.exe"

},

"preferredGenerator" : {

"name": "Unix Makefiles"

},

"toolchainFile": "arm-none-eabi-gcc.cmake"

}

]

* Создать в «/» toolchain-файл ***arm-none-eabi-gcc.cmake*** и добавить туда следующие строчки:

set(CMAKE\_SYSTEM\_NAME Generic)

set(CMAKE\_SYSTEM\_PROCESSOR ARM)

# Additional toolchain utilities

set(CMAKE\_OBJCOPY "<your path to>\\arm-none-eabi-objcopy.exe" CACHE INTERNAL "")

set(CMAKE\_SIZE "<your path to>\\arm-none-eabi-size.exe" CACHE INTERNAL "")

# Perform compiler test with static library

set(CMAKE\_TRY\_COMPILE\_TARGET\_TYPE STATIC\_LIBRARY)

#---------------------------------------------------------------------------------------

# Set compiler/linker flags

#---------------------------------------------------------------------------------------

set(OBJECT\_GEN\_FLAGS "\

-Og \

-ggdb \

-O0 \

-mthumb \

-fsigned-char \

-Wall \

-Wno-comment \

-Wno-unused-function \

-Wno-unused-variable \

-ffunction-sections \

-fdata-sections \

-fomit-frame-pointer \

-mabi=aapcs \

-fno-builtin \

")

set(CMAKE\_C\_FLAGS "${OBJECT\_GEN\_FLAGS} -std=gnu99 " CACHE INTERNAL "C Compiler options")

set(CMAKE\_CXX\_FLAGS "${OBJECT\_GEN\_FLAGS} -std=c++14 " CACHE INTERNAL "C++ Compiler options")

set(CMAKE\_ASM\_FLAGS "${OBJECT\_GEN\_FLAGS} -x assembler-with-cpp " CACHE INTERNAL "ASM Compiler options")

set(CMAKE\_EXE\_LINKER\_FLAGS "\

-Wl,--gc-sections \

-Wl,--print-memory-usage \

-mthumb \

-mabi=aapcs \

-Wl,-Map=${CMAKE\_PROJECT\_NAME}.map \

--specs=nano.specs \

--specs=nosys.specs \

" CACHE INTERNAL "Linker options")

# Set executable suffix

set(CMAKE\_EXECUTABLE\_SUFFIX\_ASM ".elf")

set(CMAKE\_EXECUTABLE\_SUFFIX\_C ".elf")

set(CMAKE\_EXECUTABLE\_SUFFIX\_CXX ".elf")

Здесь мы установили некоторые базовые ключи компиляции, ключи компоновки, пути к специальным утилитам, а также указали статический способ связывания тестовых программ. Предполагается, что в toolchain файл нам больше не придется заглядывать, на протяжении всего проекта.

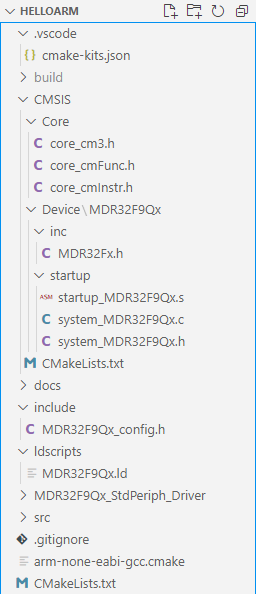
Обрати внимание, что при перечислении ключей не должен использоваться разделитель ‘\n’. Для этого его следует экранировать специальным символом косой черты ‘\’.

Таким образом, эти два файла полностью определяют все необходимые инструменты для сборки проекта и (почти все) опции для этих инструментов. Однако некоторые отдельные опции нам все же нужно будет указать непосредственно в командах CMakeLists.txt проекта, например, используемый процессор. Это имеет смысл, т.к. для большинства проектов базовые опции останутся неизменными, и toolchain файл, вероятно, можно будет использовать повторно в других проектах.

Рекомендуется придерживаться приблизительно такого же содержания и структуры для этих файлов во всех проектах.

По мере дальнейшего изучения возможностей среды vscode мы будем добавлять новые подкаталоги, файлы исходного кода и файлы конфигурации .json.

* После всех проделанных выше шагов дерево проекта должно выглядеть так:



Подобно тому смыслу, который имеет место быть при выборе способа организации дерева исходных файлов проекта, мы можем указать основные цели проекта как соответствующие подкаталоги. Если мы обратимся к справочной документации CMake, то увидим там, что цели отождествляются с директориями.

В данном примере у нас уже есть несколько явно выраженных целей – это библиотека CMSIS, библиотека SPD и главный исполняемый файл приложения. Для каждой цели мы должны написать скрипт генерации правил их построения в файлах CMakeLists.txt, расположенных в соответствующих каталогах. Чтобы собрать главную цель, нам также понадобится CMakeLists.txt верхнего уровня.

Каждый скрипт CMakeLists.txt для построения конкретной цели должен (обычно) содержать:

* Команду добавления цели add\_library() или add\_executable() и список исходников для построения этой цели.
* Команду задания директорий для поиска заголовочных файлов для цели target\_include\_directories() и соответствующий список директорий.
* Команду задания опций компиляции для цели target\_compile\_options().
* Команду задания определений препроцессора для цели target\_compile\_definitions().

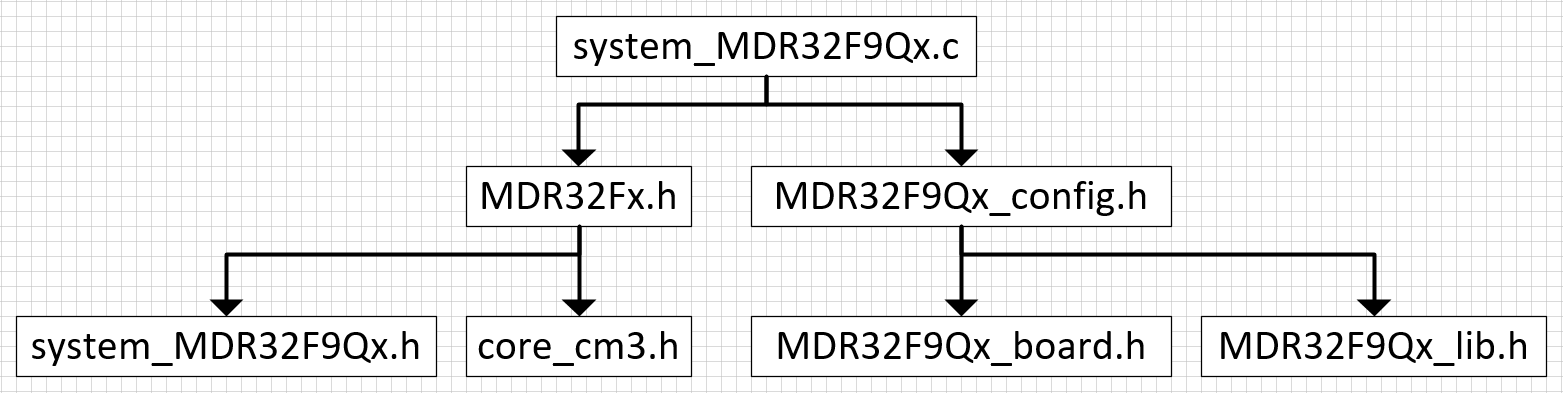
Вот собственно и все. Оказалось, все проще, чем могло показаться на первый взгляд. По крайней мере, это намного проще чем аналогичная процедура описания проекта на языке make.

Создадим поочередно все необходимые скрипты CMakeLists.txt и при создании мы будем руководствоваться следующими правилами code-style:

* Имена файлов исходного кода записывать в «» и относительными путями, на каждой строке. Кавычки требуются чтобы избежать неправильного синтаксического разбора службой intelliSense строковых литералов.
* Не использовать команду file( GLOB SRCS … ) для рекурсивного поиска всех файлов с указанными расширениями - \*.c, \*.cpp, \*.s. Использование данной команды приведет к тому, что CMake Tools не сможет отслеживать изменения в списках исходников, а сами вы можете забыть выполнить повторную конфигурацию проекта.
* Имена директорий заголовочных файлов записывать в «» и абсолютными путями, на каждой строке. При этом можно использовать встроенные переменные: ${CMAKE\_SOURCE\_DIR} – корневой каталог дерева исходных кодов, ${CMAKE\_CURRENT\_SOURCE\_DIR} – текущий каталог, в котором находится данный CMakeLists.txt.
* Команда target\_include\_directories() должна передавать список директорий поиска заголовочных файлов для данной цели с областью видимости PUBLIC. Это позволит всем зависимым целям от данной получить доступ к этому списку. NB: однако это не работает для объектных целей.
* Опции компиляции указывать в «», в одной строке, либо на каждой строке, заканчивающейся символом экранирования ‘\’ символа новой строки ‘\n’.
* Опции компоновки указывать в «», в одной строке, либо на каждой строке, заканчивающейся символом экранирования ‘\’ символа новой строки ‘\n’.

**Сборка библиотеки CMSIS**.

После выбора цели необходимо построить его дерево зависимости заголовочных файлов, затем подключить соответствующие директории к цели. Для библиотеки CMSIS дерево зависимости имеет следующий вид:



NB: директиву подключения заголовочного файла MDR32F9Qx\_board.h можно закомментировать, т.к. в нем нет ничего полезного и мы не добавляли его в проект. Все что он делает – объявляет символ USE\_MDR32F9Q3\_Rev1, через который затем в файле MDR32F9Qx\_config.h объявляется другой символ – USE\_MDR1986VE9x. Вместо этого, мы можем объявить тот же символ явно, в CMakeLists.txt.

Теперь у нас есть все исходные данные, и мы можем описать цель CMSIS в файле CMakeLists.txt.

* Создать файл ***CMSIS/CMakeLists.txt*** и записать в него следующие строчки:

set(CMSIS\_TARGET\_NAME cmsis\_o)

add\_library(${CMSIS\_TARGET\_NAME} OBJECT

"Device/MDR32F9Qx/startup/startup\_MDR32F9Qx.s"

"Device/MDR32F9Qx/startup/system\_MDR32F9Qx.c"

)

target\_include\_directories(${CMSIS\_TARGET\_NAME} PUBLIC

"${CMAKE\_CURRENT\_SOURCE\_DIR}/Core/" # core\_cm3.h

"${CMAKE\_CURRENT\_SOURCE\_DIR}/Device/MDR32F9Qx/startup/" # system\_MDR32F9Qx.h

"${CMAKE\_CURRENT\_SOURCE\_DIR}/Device/MDR32F9Qx/inc/" # MDR32Fx.h

"${CMAKE\_SOURCE\_DIR}/include/" # MDR32F9Qx\_board.h, MDR32F9Qx\_config.h

"${CMAKE\_SOURCE\_DIR}/MDR32F9Qx\_StdPeriph\_Driver/inc/" # MDR32F9Qx\_lib.h

)

target\_compile\_options(${CMSIS\_TARGET\_NAME} PRIVATE ${CPU})

target\_compile\_definitions(${CMSIS\_TARGET\_NAME} PRIVATE -DUSE\_MDR1986VE9x)

**Сборка библиотеки периферийных устройств**.

Аналогичным образом, как и для CMSIS, нам нужно создать CMakeLists.txt с описанием правил построения цели для стандартной периферийной библиотеки микроконтроллера.

* Создать файл ***MDR32F9Qx\_StdPeriph\_Driver/ CMakeLists.txt*** и записать в него следующие строчки:

set(SPD\_TARGET\_NAME spd\_o)

add\_library(${SPD\_TARGET\_NAME} OBJECT

#"src/MDR32F9Qx\_adc.c"

#"src/MDR32F9Qx\_arinc429R.c"

#"src/MDR32F9Qx\_arinc429t.c"

#"src/MDR32F9Qx\_audio.c"

#"src/MDR32F9Qx\_bkp.c"

#"src/MDR32F9Qx\_can.c"

#"src/MDR32F9Qx\_comp.c"

#"src/MDR32F9Qx\_dac.c"

"src/MDR32F9Qx\_dma.c"

#"src/MDR32F9Qx\_ebc.c"

"src/MDR32F9Qx\_eeprom.c"

#"src/MDR32F9Qx\_eth.c"

"src/MDR32F9Qx\_i2c.c"

#"src/MDR32F9Qx\_iwdg.c"

#"src/MDR32F9Qx\_keypad.c"

#"src/MDR32F9Qx\_led.c"

#"src/MDR32F9Qx\_mil\_std\_1553.c"

"src/MDR32F9Qx\_port.c"

"src/MDR32F9Qx\_power.c"

"src/MDR32F9Qx\_rst\_clk.c"

#"src/MDR32F9Qx\_ssp.c"

"src/MDR32F9Qx\_timer.c"

#"src/MDR32F9Qx\_uart.c"

#"src/MDR32F9Qx\_usb.c"

#"src/MDR32F9Qx\_wwdg.c"

)

target\_include\_directories(${SPD\_TARGET\_NAME} PUBLIC

"${CMAKE\_CURRENT\_SOURCE\_DIR}/inc/"

"${CMAKE\_SOURCE\_DIR}/CMSIS/Core/"

"${CMAKE\_SOURCE\_DIR}/CMSIS/Device/MDR32F9Qx/inc/"

"${CMAKE\_SOURCE\_DIR}/CMSIS/Device/MDR32F9Qx/startup/"

"${CMAKE\_SOURCE\_DIR}/include/"

)

target\_compile\_options(${SPD\_TARGET\_NAME} PRIVATE ${CPU})

target\_compile\_definitions(${SPD\_TARGET\_NAME} PRIVATE -DUSE\_MDR1986VE9x)

Обрати внимание, что я закомментировал некоторые файлы исходного кода, которые, предположительно, не будут использоваться в проекте. Это сократит время построения цели.

**Сборка исполняемой цели**.

* Создать файл CMakeLists.txt верхнего уровня и записать в него следующие строчки:

cmake\_minimum\_required(VERSION 3.23.2)

project(helloarm C CXX ASM)

set (EXECUTABLE ${PROJECT\_NAME})

set(CPU -mcpu=cortex-m3)

set (LD "${CMAKE\_SOURCE\_DIR}/ldscripts/MDR32F9Qx.ld")

add\_subdirectory(CMSIS)

add\_subdirectory(MDR32F9Qx\_StdPeriph\_Driver)

add\_executable(${EXECUTABLE}

"src/main.cpp"

)

target\_include\_directories(${EXECUTABLE} PRIVATE

"include/"

"CMSIS/Device/MDR32F9Qx/inc/"

"CMSIS/Device/MDR32F9Qx/startup/"

"CMSIS/Core/"

"MDR32F9Qx\_StdPeriph\_Driver/inc/"

)

target\_link\_libraries(${EXECUTABLE}

$<TARGET\_OBJECTS:cmsis\_o>

$<TARGET\_OBJECTS:spd\_o>

)

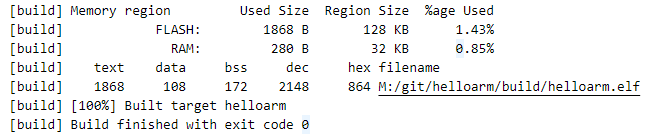
target\_compile\_options(${EXECUTABLE} PRIVATE ${CPU})

target\_compile\_definitions(${EXECUTABLE} PRIVATE USE\_MDR1986VE9x)

target\_link\_options(${EXECUTABLE} PRIVATE ${CPU} -T ${LD})

Настало время протестировать на собираемость наш проект.

* Создать файл ***src/main.cpp*** и записать в него пустую функцию main().
* Запустить конфигурацию проекта, выбрав команду «CMake: Configure» из палитры команд.
* Запустить сборку проекта, выбрав команду «CMake: Build» из палитры команд.
* Проконтролировать успешность построения проекта. Должны увидеть в консоли OUTPUT примерно следующий вывод:



**9 Сборка FreeRTOS**.

Сборка FreeRTOS в статическую библиотеку имеет смысл для конкретного проекта с целью облегчить инкрементальную сборку – при выполнении полной очистки проекта нам не придется пересобирать заново ядро ОС.

**Краткая справка по использованию дистрибутива ОСРВ FreeRTOS**.

FreeRTOS приложение стартует и выполняется также как не-RTOS приложение, до тех пор, пока не будет вызван vTaskStartScheduler(). Эта функция обычно вызывается из функции main() приложения. RTOS управляет последовательностью исполнения только после того, как будет вызван vTaskStartScheduler().

**Файлы исходного кода.**

FreeRTOS представлен стандартными файлами исходного кода на C, которые собираются вместе с другими C-файлами в вашем проекте. Файлы исходного кода FreeRTOS распространяются в zip архиве. Страница организации исходного кода FreeRTOS описывает структуру файлов в zip архиве. Как минимум, следующие файлы исходного кода должны быть включены в ваш проект:

* FreeRTOS/Source/tasks.c
* FreeRTOS/Source/queue.c
* FreeRTOS/Source/list.c
* FreeRTOS/Source/portable/[compiler]/[architecture]/port.c.
* FreeRTOS/Source/portable/MemMang/heap\_x.c where 'x' is 1, 2, 3 or 4.

Если директория, которая содержит port.c файл также содержит файл исходного кода на ассемблере, то этот файл также должен быть использован.

Если вам требуется функциональность программного таймера, то включите FreeRTOS/Source/timers.c.

Если вам требуется функциональность сопрограммы, то включите файл FreeRTOS/Source/croutine.c.

**Заголовочные файлы.**

Как минимум, следующие директории должны быть добавлены в пути поиска заголовков компилятора (компилятору необходимо сообщить, чтобы он искал заголовочные файлы в этих директориях):

* FreeRTOS/Source/include
* FreeRTOS/Source/portable/[compiler]/[architecture].

В зависимости от порта, может потребоваться, чтобы эти же директории были прописаны в путях ассемблеров.

**Конфигурационный файл.**

Каждый проект также требует так называемый конфигурационный файл - FreeRTOSConfig.h. Этот файл адаптирует ядро RTOS для собираемого приложения. Таким образом, он специфичен для приложения, не для RTOS, и должен быть размещен в директории приложения, но ни в одном из каталогов исходного кода ядра RTOS.

Если heap\_1, heap\_2 или heap\_4 включены в ваш проект, то определение в файле FreeRTOSConfig.h configTOTAL\_HEAP\_SIZE будет определять размер кучи FreeRTOS. Ваше приложение не будет линковаться, если размер кучи установлен слишком большим.

Определение в файле FreeRTOSConfig.h configMINIMAL\_STACK\_SIZE устанавливает размер стека, используемого холостой задачей. Если это значение установлено слишком низким, то холостая задача произведет переполнение стека. Рекомендуется скопировать значение минимального размера стека из официального демо проекта FreeRTOS, предоставляемого для той же архитектуры микроконтроллера. Демо проекты FreeRTOS хранятся в подкаталогах директории FreeRTOS/Demo. Заметьте, что некоторые демо проекты устарели, и не содержат всех доступных опций конфигурации.

**Векторы прерываний.**

Cortex-M пользователям: информация об установке обработчиков прерываний представлена в разделе FAQ **"**[**The application I created compiles, but does not run**](http://www.freertos.org/FAQHelp.html)**".**

Каждый RTOS порт использует таймер для генерации периодических тик-прерываний. Множество портов используют дополнительные прерывания для управления переключением контекста. Прерывания, требуемые RTOS портом, обслуживаются файлами исходного кода RTOS.

Метод, используемый для установки обработчиков прерываний, обеспечиваемый портом RTOS, зависит от порта и используемого компилятора. Обратитесь к (и если необходимо скопируйте) предоставленным официальным демо-проектам для используемого порта. Также обратитесь к странице документации, на которой представлены официальные демо-проекты.