CMake Build System

Introduction to the CMake build system

V2.1, SOTR 22/23
Paulo Pedreiras

Introduction

- CMake is a tool to manage building of source code – **build system generator**.
- CMake was originally designed as a generator for various variants of Makefile, but nowadays CMake generates modern build systems such as Ninja as well as project files for IDEs such as Visual Studio.
- CMake is normally used for C/C++ languages, but it can be used to build source code of other languages too.

- **Basically, CMake takes plain text files that describe a project as input and produces project files or makefiles for use with a wide variety of native development tools.**

Introduction

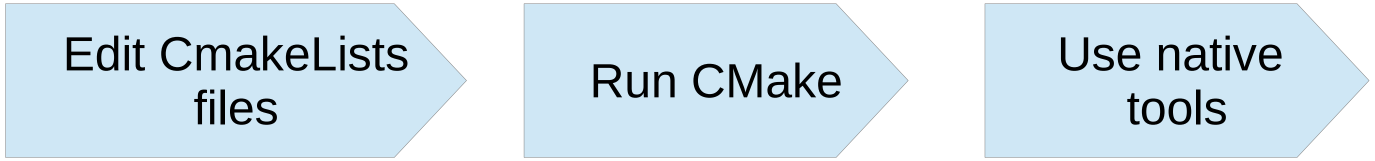
- Why CMake
  - Easy to use and works well
    - For large projects it is much simpler than typing Makefiles!
  - Cross-platform
    - Linus, Windows, OSx, …
    - Makefiles (Unix, Borland, ..), Ninja, MSBuild
    - IDEs: Code::Blocks, Eclipse CDT, Visual Studio, KDevelop, ...
  - Popular, both in scientific and industry communities
  - Allows building a directory tree outside of the source tree: “Out-of-source builds”
    - Source tree: project source code
    - Bin directory/Build directory: where CMake will put the resulting object files, libraries, and executables
    - CMake does not write any files to the source directory, only to the binary directory
  - Etc.

Installation

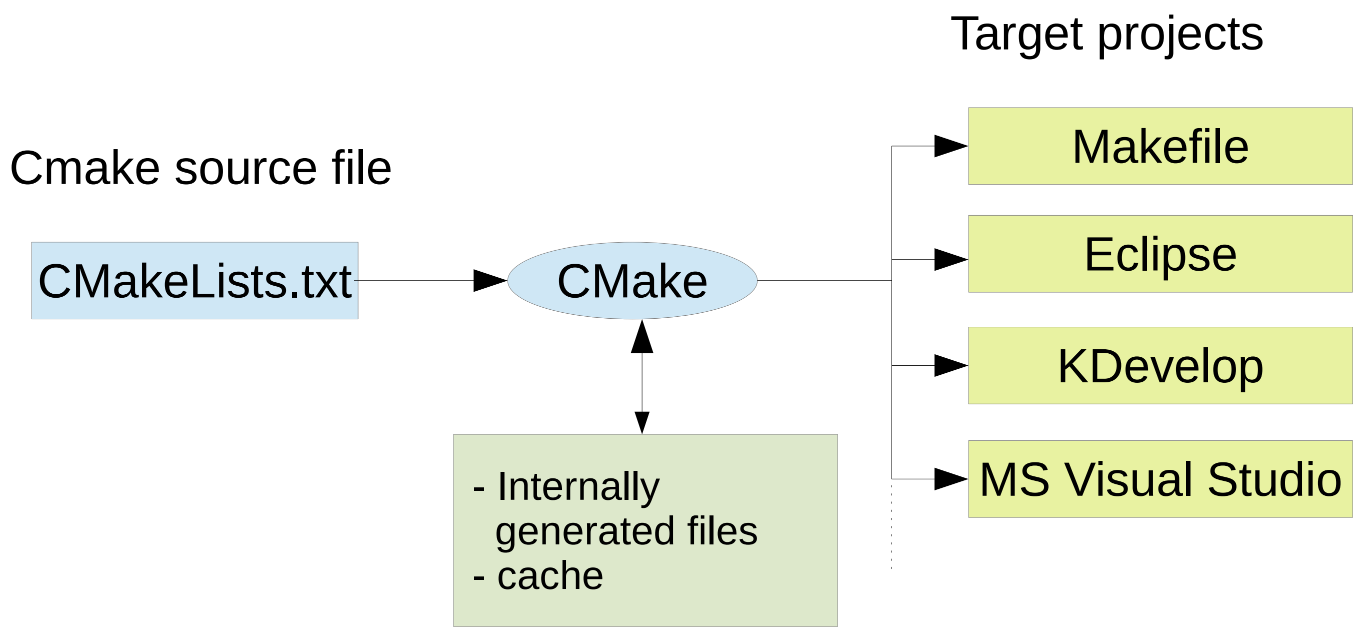
- Available at: https://cmake.org/download/
  - Source distributions for Unix/Linux and Windows
- Binary distributions for
  - Windows X64
  - Windows I386
  - Mac OS 10.10/10.13
  - Linux x86_64
- For Linux it is recommended to use the package manager of your distribution

Build process

- Workflow – 3 steps
  1) Define the project in one or more CMakeLists files
  2) Run cmake to generate/configure the native build system files
  3) Open the project files and use the native build tools



Build process



CmakeLists files

- CMakeLists files
  - Input plain text files that contain the project description in CMake’s Language.
  - Contain project parameters and describe the flow control of the entire build process
  - CMake’s language:
    - series of comments, commands and variables
  - Example: simplest possible “CMakeLists.txt” file

`cmake_minimum_required(VERSION 3.20)`	Command that allows projects to require a minimum CMake version
`# set the project name`	Lines starting with “#“ are (optional) comments
`project(HelloWorld)`	Command that sets the project name. May specify other options such as language or version. Causes CMake to generate a top-level Makefile or IDE project file
`# add the executable`	
`add_executable(Hello Hello.c)`	Command to generate executable (Hello) using a given source file (Hello.c)

Configuration and generation

- After having the source files and CmakeLists.txt set, it is time to configure and generate the build system
  - Via GUI
    - cmake-gui (requires QT)
      - Select the source code folder
      - Select the build directory (if it does not exist it is created automatically)
      - Then hit “Configure”, select the generator (e.g. Unix Makefiles) and the compiler/toolchain
      - Eventually adjust any variables
        - E.g. CMAKE_BUILD_TYPE can be “Debug”, “Release”, … (details latter on)
        - Configure again after changing a variable
      - Finally hit “Generate”
    - Go to the build dir and “make” the project

Configuration and generation

- Via curses interface
  - Most Linux/UNIX platforms support the curses library, enabling the use of ccmake.
    - Simple text-based interface
    - Go to the build dir
    - Run ccmake indicating as argument the directory where the sources are placed
    - Configure (“c”)
    - Edit the variables, if needed
    - Configure and generate (“c” then “g”)
    - “make” the project

Configuration and generation

- Via command line
  - cmake can be called form the command line to generate a project build system.
  - Best suited for projects with few or no options. For larger projects the use of ccmake or cmake-gui is recommended.
  - To build a project with cmake:
    - First create and change to the build dir
    - Run cmake specifying the path to the source tree and passing in any options using the -D flag.
      - Unlike ccmake, or the cmake-gui, the configure and generate steps are combined into one when using command line cmake.
      - E.g. 1: in build folder and assuming source is in the parent folder just type **`$cmake ..`**
      - E.g. 2: in the top level directory type **`$cmake -S . -B build`**

Configuration and generation

- A few important configuration options
  - Compiler
    - For Makefile-based generators, CMake tries a list of usual compilers until it finds a working one.
      - The desired compiler can be set by the user via environment variables
        - **CC** environment variable specifies the C compiler
        - **CXX** specifies the C++ compiler.
        - The user can specify the compiler directly on the command line by using “CMAKE_CXX_COMPILER”
          - E.g. “-DCMAKE_CXX_COMPILER=gcc”
      - Once cmake has been run and picked a compiler, if you wish to change the compiler, start over with an empty binary directory
    - The flags for the compiler and the linker can also be changed by setting environment variables:
      - Setting LDFLAGS initialize the cache values for link flags
      - CXXFLAGS and CFLAGS initialize CMAKE_CXX_FLAGS and CMAKE_C_FLAGS respectively.

Configuration and generation

- A few important configuration options (cont)
  - Build configurations: allow a project to be built in different ways.
  - CMake supports, by default:
    - **Debug** - basic debug flags turned on
    - **Release** - Release has the basic optimizations turned on
    - **MinSizeRel** - smallest object code
    - **RelWithDebInfo** - optimized build with debug information
  - For Makefile-based generators:
    - Only one configuration can be active at the time CMake
    - Configuration is specified with the CMAKE_BUILD_TYPE variable (can be empty)
    - Multiple configurations are managed by means of different build dirs
      - E.g. via command line: **`$cmake .. -DCMAKE_BUILD_TYPE=Debug`**

Writing CmakeLists files

- CMakeLists files are simple text files
  - Any editor can be used
  - Some do provide syntax highlight (e.g. vim, emacs, geany)
- CMakeLists files determine everything from which options to present to users, to which source files to compile.
- When the CMakeLists file is modified there are rules that automatically invoke CMake to update the generated files (e.g. Makefiles or project files)

Writing CmakeLists files

- The CMake language is composed of **comments**, **commands**, and **variables**.
- **Comments**:
  - start with # and run to the end of the line. Meaning and use are obvious ...
- **Variables**:
  - CMakeLists files use variables much like any programming language
  - Variable names are case sensitive and may only contain alphanumeric characters and underscores
  - There are several variables automatically defined by Cmake
    - These variables begin with “CMAKE_”. Don’t use this naming convention for variables specific to your project.
  - Variable contents are defined via the “set” command

Writing CmakeLists files

- **Variables (cont)**:
  - "set" command
    - The first argument to set is the name of the variable and the rest of the arguments are the values.
    - Multiple value arguments are packed into a semicolon-separated list and stored in the variable as a string.
    - Escape character is "\"

```
set(Foo "")         # 1 quoted arg -> value is ""
set(Foo a)          # 1 unquoted arg -> value is "a"
set(Foo "a b c")    # 1 quoted arg -> value is "a b c"
set(Foo a b c)      # 3 unquoted args -> value is "a;b;c"
set(VAR “a\\b\\c and \”embedded quotes\””)
message(${VAR})     # “a\b\c and ”embedded quotes””
```

Writing CmakeLists files

- **Variables (cont)**:
  - Referencing variables
    - Variables may be referenced in command arguments using syntax **${VAR}** where VAR is the variable name
    - Replacement is performed prior to the expansion of unquoted arguments

```
set(Foo a b c)    # 3 unquoted args -> value is "a;b;c"

command(${Foo})   # unquoted arg replaced by a;b;c

                  # and expands to three arguments

command("${Foo}") # quoted arg value is "a;b;c"
```

Writing CmakeLists files

- **Variables (cont)**:
  - Referencing variables (cont)
    - Environment variables can be accessed via the syntax **$ENV{VAR}**
  - Variable scope
    - When a variable is set it is visible:
      - In the current CMakeLists file or function and any subdirectory’s CMakeLists files,
      - Any functions or macros that are invoked, and
      - Any files that are included using the include command.
    - Any new variables created in the child scope, or changes made to existing variables, will not impact the parent scope, unless explicitly stated (PARENT_SCOPE argument of set command, example in a latter slide).

Writing CmakeLists files

- **Variables (cont)**:
  - Variable scope: example

```
function(foo)
  message(${test}) # test is 1 here
  set(test 2)
  message(${test}) # test is 2 here, but only in this scope
endfunction()

set(test 1)
foo()
message(${test}) # test will still be 1 here
```

Writing CmakeLists files

- **Commands**:
  - A command consists of the command name, opening parenthesis, white space separated arguments, and a closing parenthesis: **cmd_name(arg1 arg2 ...)**
  - Each command is evaluated in the order that it appears in the CMakeLists file
  - Command names are not case-sensitive. Convention is using lowercase
  - Command arguments:
    - Are space separated and case sensitive
    - May be either quoted or unquoted.
    - A quoted argument starts and ends in a double quote (“) and always represents exactly one argument. Any double quotes contained inside the value must be escaped with a backslash.

Writing CmakeLists files

- **Commands (cont)**:
  - Example:

```
command("")          # 1 quoted argument
command("a b c")     # 1 quoted argument
command("a;b;c")     # 1 quoted argument
command("a" "b" "c") # 3 quoted arguments
command(a b c)       # 3 unquoted arguments
command(a;b;c)       # 1 unquoted argument expands to 3
```

Writing CmakeLists files

- Basic commands
  - **set** and **unset**: explicitly set or unset variables.
  - **string**, **list**, and **separate_arguments**: basic manipulation of strings and lists.
  - **add_executable** and **add_library**: main commands for defining the executables and libraries to build, and which source files comprise them
- **Flow Control**
  - Conditional statements, looping constructs and procedure definitions
  - **If/elseif**

```
if(MSVC80)
  # do something here
elseif(MSVC90)
  # do something else
elseif(APPLE)
  # do something else
endif()
```

Writing CmakeLists files

- **Flow Control (cont)**
  - **foreach** and **while**
    - These commands allow handling repetitive tasks that occur in sequence.
    - The break command breaks a loop before normal end.
    - The first argument of **foreach** is the name of the variable and the remaining arguments are the list of values over which to loop

```
foreach(tfile
        TestButterworthLowPass
        TestButterworthHighPass
        )
  add_test(${tfile}-image ${VTK_EXECUTABLE}
    ${VTK_SOURCE_DIR}/Tests/rtImageTest.tcl
    ….
    )
endforeach()
```

Writing CmakeLists files

- **Flow Control (cont)**
  - **foreach** and **while**
    - The **while** command provides looping based on a test condition.
    - The format for the test expression in the while command is the same as it is for the if command

```
#####################################################
# run paraview and ctest test dashboards for 6 hours
#
while(${CTEST_ELAPSED_TIME} LESS 36000)
  set(START_TIME ${CTEST_ELAPSED_TIME})
  ctest_run_script("dash1_ParaView_vs71continuous.cmake")
  ctest_run_script("dash1_cmake_vs71continuous.cmake")
endwhile()
```

Writing CmakeLists files

- **Procedure definitions**
  - Macro and function commands support repetitive tasks scattered throughout CMakeLists files.
  - Macro or function can be used by any CMakeLists files processed after its definition.
  - A Cmake **function** resembles a C/C++ function
  - Functions can have arguments that become variables within the function. Standard variables such as ARGC, ARGV, ARGN, and ARGV0, ARGV1 are also defined
  - Note the use of the PARENT_SCOPE keyword inside the function DetermineTime

```
function(DetermineTime _time)
  # pass the result up to whatever invoked this
  set(${_time} "1:23:45" PARENT_SCOPE)
endfunction()

# now use the function we just defined
DetermineTime(current_time)

if(DEFINED current_time)
  message(STATUS "The time is now: ${current_time}")
endif()
```

Writing CmakeLists files

- **Procedure definitions**
  - Macros are defined and called in the same manner as functions.
  - The main differences are that macros do not create a new variable scope and arguments to a macro are not treated as variables but as strings replaced prior to execution.
    - Very similar to the differences between a macro and a function in C or C++

```
# define a simple macro
macro(assert TEST COMMENT)
  if(NOT ${TEST})
    message("Assertion failed: ${COMMENT}")
  endif()
endmacro()

# use the macro
find_library(FOO_LIB foo /usr/local/lib)
assert(${FOO_LIB} "Unable to find library foo")
```

Cmake Cache

- The CMake cache may be thought of as a configuration file.
- The first time CMake is run on a project, it produces a CMakeCache.txt file in the top directory of the build tree.
- CMake uses this file to store a set of global cache variables, whose values persist across multiple runs within a project build tree.
- There are a few purposes of this cache:
  - **Store** user's **selections** and **choices**, so that if they should run CMake again they will not need to reenter that information.
    - For example, the option command creates a Boolean variable and stores it in the cache.
    - The user can set that variable from the user interface and its value will remain in case CMake is executed again in the future.

```
option(USE_JPEG "Do you want to use the jpeg library")
```

    - Variable in cache can also be created via the standard set command with the CACHE option

```
set(USE_JPEG ON CACHE BOOL "include jpeg support?")
```

Cmake Cache

- There are a few purposes of this cache (cont):
  - **Persistently store values** between CMake runs
    - These entries may not be visible or adjustable by the user. Typically, these values are system-dependent variables that require CMake to compile and run a program to determine their value.
    - Once these values have been determined, they are stored in the cache to avoid having to recompute them every time CMake is run.
    - **If you significantly change your computer (e.g. changing the OS or compiler) the cache file and likely all of your binary tree’s object files, libraries, and executables must be deleted.**
- Restricting a cache entry to a limited set of predefined options.
  - Can be done by setting the STRINGS property on the cache entry. When cmake-gui is run and the user selects the variable cache entry from a pulldown list

```
set(CRYPTOBACKEND "OpenSSL" CACHE STRING "Select a cryptography backend")
set_property(CACHE CRYPTOBACKEND PROPERTY STRINGS
             "OpenSSL" "LibTomCrypt" "LibDES")
```

Basic usage

- The most basic project is an executable built from source code files.
- For simple projects, a three line CMakeLists.txt file is all that is required.

```
cmake_minimum_required(VERSION 3.10)

# set the project name
project(SimpleProj)

# set the executable and source file(s)
add_executable(simple src/main.c)
```

Basic usage – Structuring SW

- Projects should be structured using modules/libraries
- Lets assume that each library is stored into a sub-directory.
  - Each of these sub-directories contains the corresponding “.h” and “.c” files
  - Each of these sub-directories must have its own CmakeLists.txt defining the library name and the source file.
    - E.g. add_library(MovAvgLib movavg.c)
- Then it must be added a **add_subdirectory** call in the top-level CMakeLists.txt file, so that the library is built
- It is also necessary to add the new library to the include directory so that the “.h” header file can be found.
- CmakeLists example in next slide.
  - Adding a simple MovAvg library (“.c”, “.h” and library “CmakeLists.txt” in sub-directory MovAvgLib)

Basic usage – adding a library

CMake
Cross-platform Make

Root CMakeLists.txt

```
...
# Add the MovAvgLib folder to the build. CMake will look for a
#   CMakeLists.txt in that folder and will process it
add_subdirectory(MovAvgLibFolder)

# Generate executable "libdemo" from source file "main.c"
add_executable(libdemo src/main.c)

# Generate variables with set of libs and include folders
# Dependencies are automatically set
list(APPEND EXTRA_LIBS MovAvgLib)
list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/MovAvgLibFolder")

# Specify libraries or flags to use when linking a given target and/or its dependents
target_link_libraries(libdemo PUBLIC ${EXTRA_LIBS})

# Specifies include directories to use when compiling a given target
target_include_directories(libdemo PUBLIC ${EXTRA_INCLUDES})
```

Library CMakeLists.txt

```
# Set the source files that make up the library
set(MOVAVG_SRCS
   movavg.h movavg.c
)

# Set the library type (can be static or dynamic )
SET (LIB_TYPE STATIC)

# Create the library
add_library (MovAvgLib ${LIB_TYPE} ${MOVAVG_SRCS})
```

Basic usage – Generating documentation via Doxygen

Root CMakeLists.txt

```
...
project(LibExample)

# Ask if the user wants to generate documentation
option(GENERATE_DOC "Generate Doxygen documentation for the project?")
if(GENERATE_DOC)
	# Generate documentation if Doxygen is found
	find_package(Doxygen REQUIRED dot OPTIONAL_COMPONENTS mscgen dia)
	if(Doxygen_FOUND)
		message(STATUS "Doxygen was found, will build docs")
		add_subdirectory(docs)
	else()
		message(STATUS "Doxygen not found, not building docs")
	endif()
Endif()
...
add_subdirectory(MovAvgLibFolder)
...
```

CmakeLists.txt in “docs” folder

```
set(DOXYGEN_EXTRACT_ALL YES)
set(DOXYGEN_BUILTIN_STL_SUPPORT YES)


# set the documentation target name and files to include
doxygen_add_docs(LibDemoDoc
    "${PROJECT_SOURCE_DIR}/mainpage.md"
    "${PROJECT_SOURCE_DIR}/MovAvgLibFolder" )
```

Notes:

- Documentation files are created inside the build directory
- GENERATE_DOC must be set. E.g.:
  - **$cmake .. -DGENERATE_DOC=ON**
- Documentation must be explicitly generated **$make LibDemoDoc** in folder “build”
- See “mainpage.md” in the example for more detailed instructions

Basic usage – Unity Unit Test

CMake Cross-platform Make

Directory structure\

```
root---|
       |-- src            // Module to test (.c and .h)
       |-- unity_src      // Unity source files
       |-- test           // Test .c file

Note 1: this files are just examples. Have been created to
be generic, requiring minimal modifications.
Note 2: make test executes the test file
```

Root CMakeLists.txt

```
# Add CTest, which is the CMake test driver program
include(CTest)

# Add directories with the source code to test,
#    Unity and test code
add_subdirectory(src)
add_subdirectory(unity_src)
add_subdirectory(test)
```

“src” CMakeLists.txt

```
file(GLOB SOURCES ./*.c)

add_library(src STATIC ${SOURCES})

target_include_directories(src PUBLIC .)
```

“test” CMakeLists.txt

```
add_executable(testMyVectorLibApp testMyVectorLib.c)

target_link_libraries(testMyVectorLibApp src unity)

add_test(MyVectorLibTest testMyVectorLibApp)
```

“unity_src” CMakeLists.txt

```
add_library(unity STATIC unity.c)

target_include_directories(unity PUBLIC .)
```

Finaly remarks and bibliography

- CMake is a complete, powerful, rich and complex tool!
- This presentation just scratches the surface. But it should allow you to understand and use CMake-based development frameworks such as ExpressIF (ESP32), Nordic nRF and several others, as well as to carry out some basic but common tasks
- There are many books and tutorials on CMake.
- The following are highly recommended (first to are part of the official documentation):
  - https://cmake.org/cmake/help/book/mastering-cmake/
  - https://cmake.org/cmake/help/book/mastering-cmake/cmake/Help/guide/tutorial/index.html#
  - https://cliutils.gitlab.io/modern-cmake/
  - https://github.com/toeb/moderncmake

A short introduction to DeviceTrees

V1.1, SOTR 22-23

Paulo Pedreiras
pbrp@ua.pt
UA/DETI/IT

What is a Device Tree

- From the Devicetree Specification Release v0.3-40-g7e1cc17:

  “A devicetree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent.

  A DTSpec-compliant devicetree describes device information in a system that cannot necessarily be dynamically detected by a client program”

  – **That is, a device tree describes hardware that <u>may or may not</u> be detectable by other means, e.g. probing**.

Where do I find DeviceTrees

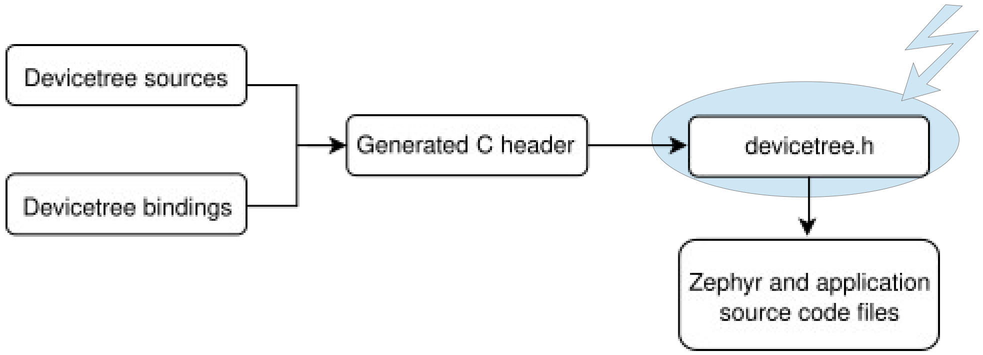
- Device Trees are used in many systems
  - Linux (embedded or not)
  - STM products and boards
  - The Zephyr Embedded OS
    - And consequently on the Nordic nrF Connect SDK
  - Etc.
- **The focus of this presentation is on Zephyr/Nordic**

Introduction

- A devicetree is a hierarchical data structure that describes hardware.
- The Devicetree specification defines its source and binary representations.
- **Zephyr uses devicetree to describe the hardware available on its Supported Boards, as well as that hardware's initial configuration.**
- There are two types of devicetree input files:
  - devicetree sources and devicetree bindings.
    - The sources contain the devicetree itself.
    - The bindings describe its contents, including data types.

Introduction

- The build system uses devicetree sources and bindings to produce a generated C header.
- The generated header’s contents are abstracted by the devicetree.h API, which you can use to get information from your devicetree.
- **All Zephyr and application source code files can include and use devicetree.h**. This includes device drivers, applications, tests, the kernel, etc.



Syntax and Structure

- As the name indicates, a devicetree is a (hierarchical) tree.
- The human-readable text format for this tree is called DTS (DeviceTree Source), and is defined in the Devicetree specification.
- Example DTS file:

```
/dts-v1/;
/ {
      a-node {
         subnode_label: a-sub-node {
                foo = <3>;
         };
      };
};
```

- /dts-v1/
  - file's contents are in version 1 of the DTS syntax
- The tree has 3 nodes
  - "/": root node
  - A node named "a-node" (child of root)
  - A node named "a-sub-node" (child of "a-node")
- Nodes can be given labels that can be used to refer to the labeled node elsewhere in the devicetree.
  - "a-sub-node" has label "subnode_label"

Syntax and Structure

- Devicetree nodes have **paths** identifying their locations in the tree and work like Unix file system paths
  - E.g. the full path to “a-sub-node” is “/a-node/a-sub-node”

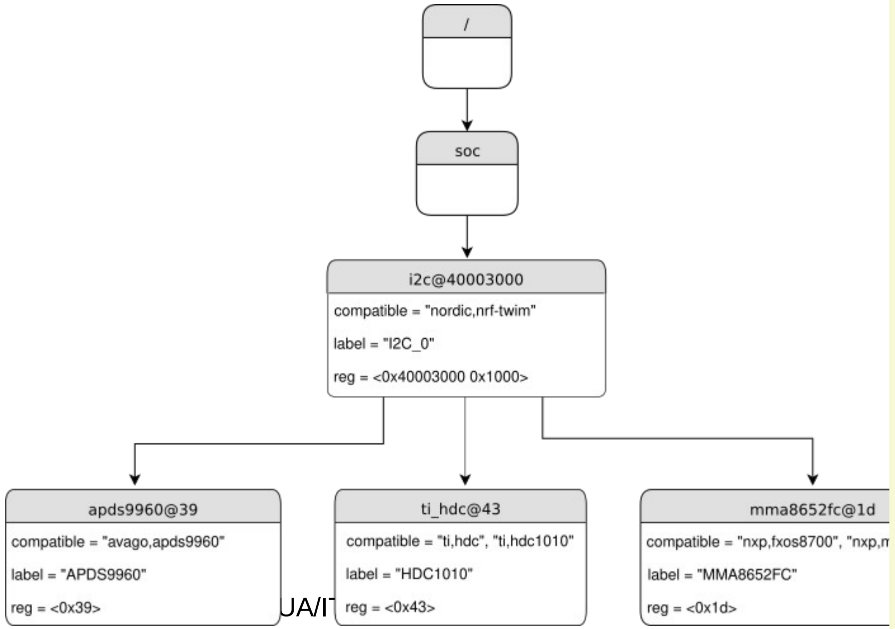
- Devicetree nodes can also have **properties**.
  - **Properties** are **name/value** pairs.
  - Values can be any sequence of bytes.
  - In some cases, the values are an array of what are called cells. A cell is just a 32-bit unsigned integer.

```
/dts-v1/;
/ {
        a-node {
           subnode_label: a-sub-node {
                   foo = <3>;
           };
        };
};
```

  - Node “a-sub-node” has a property named “foo”, whose value is a cell with value 3. The size and type of foo‘s value are implied by the enclosing angle brackets (< and >) in the DTS

Syntax and Structure

- In practice, devicetree nodes usually correspond to some hardware, and the node hierarchy reflects the hardware's physical layout. For example:



/
soc
i2c@40003000
compatible = "nordic,nrf-twim"
label = "I2C_0"
reg = <0x40003000 0x1000>
apds9960@39
compatible = "avago,apds9960"
label = "APDS9960"
reg = <0x39>
ti_hdc@43
compatible = "ti,hdc", "ti,hdc1010"
label = "HDC1010"
reg = <0x43>
mma8652fc@1d
compatible = "nxp,fxos8700", "nxp,m
label = "MMA8652FC"
reg = <0x1d>

```
/dts-v1/;

/ {
    soc {
        i2c@40003000 {
                compatible = "nordic,nrf-twim";
                label = "I2C_0";
                reg = <0x40003000 0x1000>;

                apds9960@39 {
                    compatible = "avago,apds9960";
                    label = "APDS9960";
                    reg = <0x39>;
                };
                ti_hdc@43 {
                    compatible = "ti,hdc", "ti,hdc1010";
                    label = "HDC1010";
                    reg = <0x43>;
                };
                mma8652fc@1d {
                    compatible = "nxp,fxos8700", "nxp,mma8652fc";
                    label = "MMA8652FC";
                    reg = <0x1d>;
                };
        };
    };
};
```

Syntax and Structure

- In addition to showing more realistic names and properties, the DTS example introduces a new devicetree concept - **unit addresses**.
  - Unit addresses are the parts of node names after an “at” sign (@), like 40003000 in i2c@40003000 or 39 in apds9960@39.
  - Unit addresses are optional: the soc node does not have one.

```
/dts-v1/;

/ {
    soc {
        i2c@40003000 {
                compatible = "nordic,nrf-twim";
                label = "I2C_0";
                reg = <0x40003000 0x1000>;

                apds9960@39 {
                    compatible = "avago,apds9960";
                    label = "APDS9960";
                    reg = <0x39>;
                };
                ti_hdc@43 {
                    compatible = "ti,hdc", "ti,hdc1010";
                    label = "HDC1010";
                    reg = <0x43>;
                };
                mma8652fc@1d {
                    compatible = "nxp,fxos8700", "nxp,mma8652fc";
                    label = "MMA8652FC";
                    reg = <0x1d>;
                };
        };
    };
};
```

Syntax and Structure

- Unit addresses examples
  - **Memory-mapped peripherals:**
    - The peripheral’s register map base address. For example, the node named i2c@40003000 represents an I2C controller whose register map base address is 0x40003000.
  - **I2C peripherals**
    - The peripheral’s address on the I2C bus. For example, the child node apds9960@39 of the I2C controller in the previous section has I2C address 0x39.
  - **SPI peripherals**
    - An index representing the peripheral’s chip select line number. If there is no chip select line, 0 is used.
  - **Memory**
    - The physical start address. For example, a node named memory@2000000 represents RAM starting at physical address 0x2000000.
  - There are others, e.g. flash ...

Syntax and Structure

- Some important properties
  - **compatible**
    - The name of the hardware device the node represents. The recommended format is "vendor,device", like "avago,apds9960", or a sequence of these, like "ti,hdc", "ti,hdc1010".
    - The vendor part is an abbreviated name of the vendor.
    - It is also sometimes a value like gpio-keys, mmio-sram, or fixed-clock when the hardware’s behavior is generic.
    - **The build system uses the compatible property to find the right bindings for the node. Device drivers use devicetree.h to find nodes with relevant compatibles, in order to determine the available hardware to manage.**
    - **Within Zephyr’s bindings syntax, this property has type string-array.**

Syntax and Structure

- Some important properties
  - **label**
    - The device's name according to Zephyr's Device Driver Model.
    - The value can be passed to *device_get_binding()* to retrieve the corresponding driver-level *struct device**.
    - This pointer can then be passed to the correct driver API by application code to interact with the device.
    - **For example, calling device_get_binding("I2C_0") would return a pointer to a device structure which could be passed to I2C API functions like i2c_transfer().**
    - The generated C header will also contain a macro which expands to this string.

Syntax and Structure

- Some important properties
  - **reg**
    - Information used to address the device. The value is specific to the device.
    - The reg property is a sequence of (address, length) pairs. Each pair is called a “register block”.
    - Some common patterns:
      - Devices accessed via memory-mapped I/O registers (like i2c@40003000)
        - Address is usually the base address of the I/O register space, and length is the number of bytes occupied by the registers.
      - I2C devices (like apds9960@39 and its siblings):
        - Address is a slave address on the I2C bus. There is no length value
      - SPI devices: address is a chip select line number; there is no length.
    - **There are some evident similarities between the reg property and common unit addresses described before. It is not a coincidence ad the reg property can be seen as a more detailed view of the addressable resources within a device than its unit address.**

Syntax and Structure

- Some important properties
  - **status**
    - A string which describes whether the node is enabled.
    - The devicetree specification allows this property to have values "okay", "disabled", "reserved", "fail", and "fail-sss". Only the values "okay" and "disabled" are currently relevant to Zephyr.
    - A node is considered enabled if its status property is either "okay" or not defined .
    - Nodes with status "disabled" are explicitly disabled.
    - **Devicetree nodes which correspond to physical devices must be enabled for the corresponding struct device in the Zephyr driver model to be allocated and initialized.**
  - **interrupts**
    - Information about interrupts generated by the device, encoded as an array of one or more interrupt specifiers. Each interrupt specifier has some number of cells.

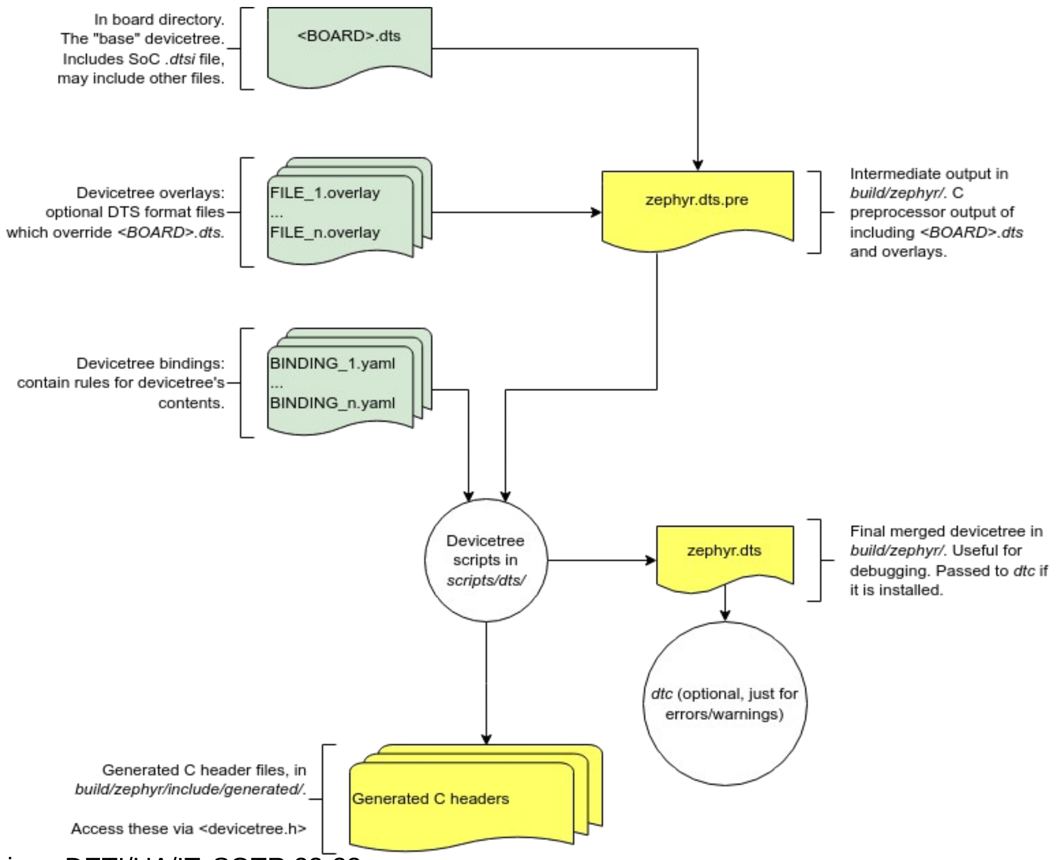
Syntax and Structure

- Aliases and chosen nodes
  - There are two additional ways beyond node labels to refer to a particular node without specifying its entire path:
    - by alias, or by chosen node.
  - Example devicetree which uses both:

```
/dts-v1/;

/ {
	chosen {
		zephyr,console = &uart0;
	};

	aliases {
		my-uart = &uart0;
	};

	soc {
		uart0: serial@12340000 {
			...
		};
	};
};
```

- The /aliases and /chosen nodes do not refer to an actual hardware device.
- Their purpose is to specify other nodes in the devicetree.
  - “my-uart” is an alias for the node with path /soc/serial@12340000
- Aliases are often used to override particular hardware. E.g. the “Blinky” demo uses it to adapt to LEDs on different ports

Syntax and Structure


In board directory. The "base" devicetree. Includes SoC .dtsi file, may include other files.
<BOARD>.dts
Devicetree overlays: optional DTS format files which override <BOARD>.dts.
FILE_1.overlay ... FILE_n.overlay
Devicetree bindings: contain rules for devicetree's contents.
BINDING_1.yaml ... BINDING_n.yaml
zephyr.dts.pre
Intermediate output in build/zephyr/. C preprocessor output of including <BOARD>.dts and overlays.
Devicetree scripts in scripts/dts/
zephyr.dts
Final merged devicetree in build/zephyr/. Useful for debugging. Passed to dtc if it is installed.
dtc (optional, just for errors/warnings)
Generated C header files, in build/zephyr/include/generated/.
Access these via <devicetree.h>
Generated C headers

- Devicetree input (green) and output (yellow) files
- Overlays are often used to extend/cutsomize the base hardware that comes e.g. with a devkit

Devicetree access from C

- There are many different ways of accessing the DT from C
  - E.g. it is possible to get a Node Identifier: By path, By node label, By alias, By instance number, By chosen node, ...
- To reduce complexity we will focus on one single method.
- Example: blink LED0, which is connected to P0.13

```
...
#define GPIO0_NID DT_NODELABEL(gpio0)
...
void main(void) {

    /* Local vars */
    const struct device *gpio0_dev;         /* Pointer to GPIO device structure */
    ...
    /* Bind to GPIO 0 */
    gpio0_dev = device_get_binding(DT_LABEL(GPIO0_NID));
    if (gpio0_dev == NULL) {
        printk("Failed to bind to GPIO0\n\r");
      return;
    }
    else {
        printk("Bind to GPIO0 successfull \n\r");
    }
```

Devicetree access from C

- Despite not strictly related with DTs, let us take the opportunity to show the libraries provided by Zephyr/Nordic

```
….
/* Configure PIN */
ret = gpio_pin_configure(gpio0_dev, BOARDLED1, GPIO_OUTPUT_ACTIVE);
if (ret < 0) {
    printk("gpio_pin_configure() failed with error %d\n\r", ret);
  return;
}

/* Blink loop */
while(1) {

    /* Toggle led status */
    if(ledstate == 0)
        ledstate = 1;
    else
        ledstate = 0;
    gpio_pin_set(gpio0_dev, BOARDLED1, ledstate);

    /* Pause  */
    k_msleep(BLINKPERIOD_MS);
}
```

Devicetree access from C

- The board DTS is available as part of the VS Code or emStudio project, so the programmer can analyze it to check for the available hardware, labels, etc.
  - Sample contents of zephyr.dts:

```
/dts-v1/;
/ {
        #address-cells = < 0x1 >;
        #size-cells = < 0x1 >;
        model = "Nordic nRF52840 DK NRF52840";
        compatible = "nordic,nrf52840-dk-nrf52840";
        chosen {
                zephyr,entropy = &cryptocell;
                zephyr,flash-controller = &flash_controller;
                zephyr,console = &uart0;
…

aliases {
                led0 = &led0;
                led1 = &led1;
                led2 = &led2;
...

gpio0: gpio@50000000 {
                        compatible = "nordic,nrf-gpio";
                        gpio-controller;
                        reg = < 0x50000000 0x200 0x50000500 0x300 >;
...
```

Devicetree access from C

- Please note that libraries must also be enabled in prj.conf
- E.g.

```
CONFIG_PRINTK=y
CONFIG_HEAP_MEM_POOL_SIZE=256
CONFIG_ASSERT=y
CONFIG_GPIO=y
CONFIG_PWM=y
CONFIG_ADC=y
CONFIG_TIMING_FUNCTIONS=y
CONFIG_USE_SEGGER_RTT=y
CONFIG_RTT_CONSOLE=n
CONFIG_UART_CONSOLE=y
```

Last notes

- This a complex topic, with many details
- It is necessary to acquire at least a basic understanding to be able to “navigate” on the framework, identifying the peripherals, attributes, etc.
- Must resort heavily to:
  - Documentation (see Bibliography)
  - Examples
    - The toolchain brings many examples that illustrate the use of many of the peripherals

Bibliography

- Nordic nrF Connect SDK documentation
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/guides/dts/index.html
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/reference/devicetree/index.html
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/guides/dts/bindings.html#dt-bindings
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/guides/dts/api-usage.html#dt-from-c

- Devicetree Specification, Release v0.3-40-g7e1cc17
  - https://www.devicetree.org/specifications/

- Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support
  - https://elinux.org/images/f/f9/Petazzoni-device-tree-dummies_0.pdf

Embedded and Real-Time Systems course

Zephyr Project Introduction

SOTR 2022/2023
Paulo Pedreiras
DETI/UA/IT

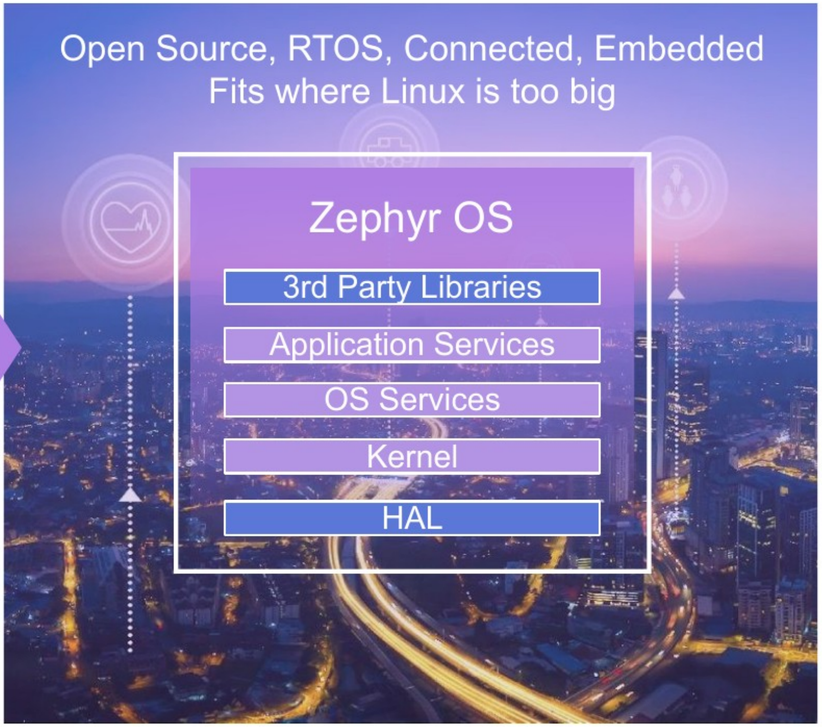
Vision

Quoted from the Linux Foundation documentation:

“The Zephyr Project strives to deliver the best-in-class RTOS for connected resource-constrained devices, built to be secure and safe.”

Zephyr Project Overview

- **Open source real time operating system**
- Vibrant **Community** participation
- Built with **safety** and **security** in mind
- **Cross-architecture** with broad SoC and development board support.
- **Vendor Neutral governance**
- **Permissively licensed** - Apache 2.0
- **Complete, fully integrated, highly configurable and modular (for flexibility)**
- Product development ready using **LTS** includes **security updates**
- **Auditable codebase,** for certification



Some products using Zephyr

Grush Gaming Toothbrush

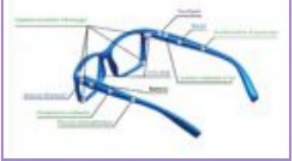
Proglove

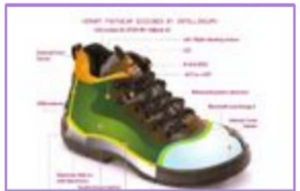
Rigado IoT Gateway

Adero Tracking Devices

Distancer

OB-4


Ellcie-Healthy Smart Connected Eyewear


Intellinium Safety Shoes

GNARBOX 2.0 SSD

HereO Core Box

Safety Pod

Oticon More

hereO Smartwatch

Point Home Alarm

RUUVI Node

Anicare Reindeer Tracker

Sentrius

See.Sense AIR

Zephyr vs other RTOS

- There are many other RTOS
- Selecting the "best one" depends on the criteria used (i.e. the user, the application, the context, ...).
- Among the most popular "competitors"
  - FreeRTOS (one of my favorites)
    - Provides essentially scheduling services and IPC
    - Much lower abstraction level regarding peripherals and comm stacks when compared with Zephyr
    - Backed by Amazon ...
  - Mbed OS
    - Also aims at IoT, complete ecosystem, etc.,
    - But it is owned by ARM and supports only ARM Cortex-M hardware
  - Azure RTOS (ThreadX)
    - Also aims at IoT, complete ecosystem, etc.,
    - But backed/controlled by Microsoft

Supported Boards

- Wide variety of supported architectures and boards
  - x86
  - ARM/ARM64
    - Hundreds of boards from NXP, ST, Nordic, BBC, TI, Microchip, Renesas, Digilent, Adafruit, Arduino, …
  - MIPS
  - NIOS II
  - RISC-V
  - XTENSA
  - SPARC
- It can even run as a native Linux application

ARC Synopsys

arm
Cortex-M, Cortex-R & Cortex-A

intel
X86 & x86_64

Nios II Processor

RISC-V
32 & 64 bit

tensilica
Xtensa

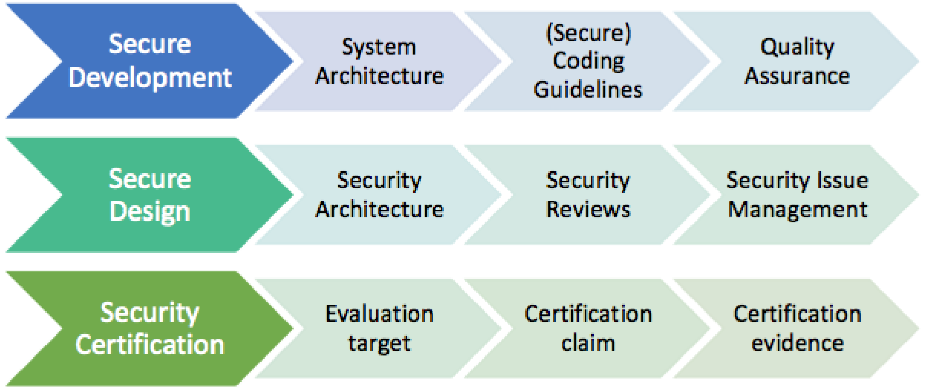
Build and configuration system

- The build and configuration system of Zephyr is unique in comparison to other RTOSs
  - The build system is Cmake, which is common in the (embedded) Linux world
  - The configuration system is Kconfig, which is common in the Linux world as well (it is the Kernel’s configuration system).

- Advantages
  - Based on solid, well established and widely disseminated tools.
    - Requires a significant effort to learn, but the knowledge acquired can be useful in many contexts and is long-lasting.
  - Embedded Linux developers will feel at home when working with the Zephyr RTOS, or, people that learn Zephyr will feel at home when moving to embedded Linux ...
  - Eases setup and maintenance of software products

Security

- Security is becoming a critical aspect of embedded and real-time applications
- The Zephyr project maintains a Security Overview which summarizes the measures taken to make the code base as secure as practically possible.



- There are Secure Coding Guidelines
  - https://docs.zephyrproject.org/latest/security/secure-coding.html

Safety

- Many embedded/real-time applications are subject to **safety requirements**
  - That is the case e.g. when a **malfunction** device can cause **significant economical impact or endanger human lives**
  - Common requirements in areas such as medical, industrial, automotive, ...
- Zephyr aims to be the first safety-certified free, open source RTOS

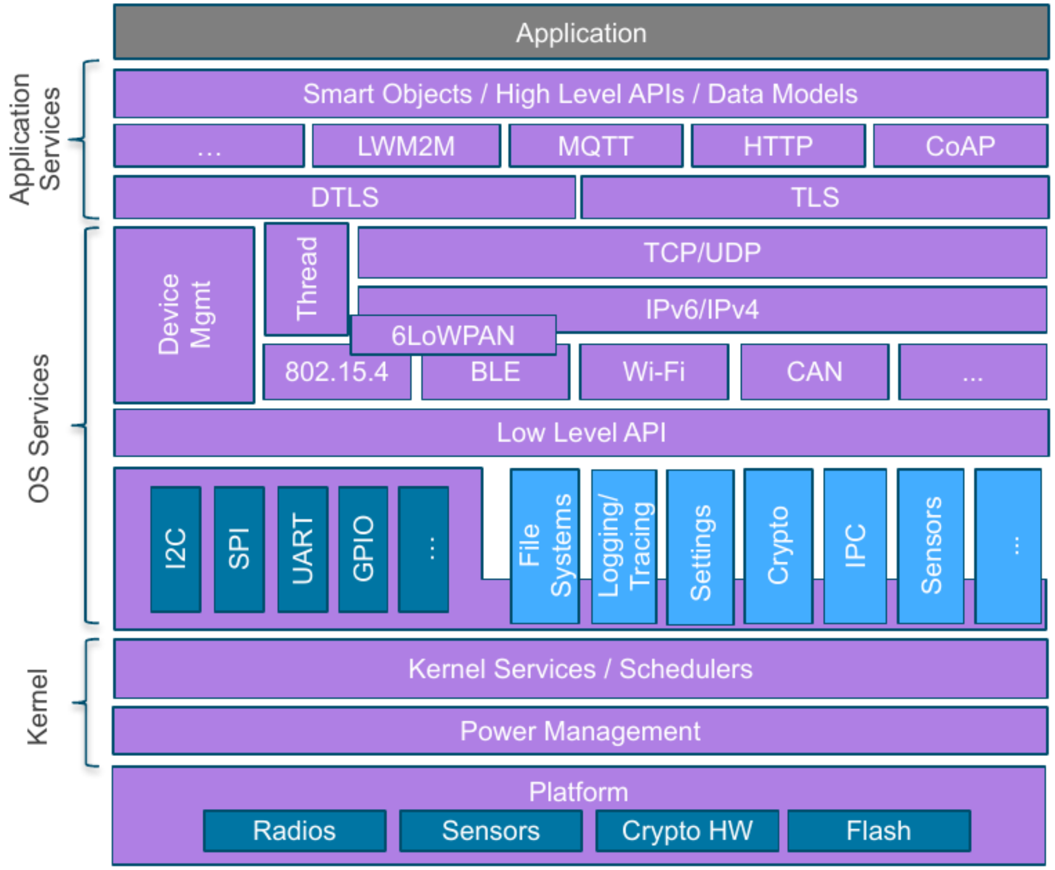
https://www.zephyrproject.org/zephyr-project-rtos-first-functional-safety-certification-submission-for-an-open-source-real-time-operating-system/

- There are certified RTOS, but they are not FOSS
  - E.g. FreeRTOS has a certified version SAFERTOS, which is not FOSS

Architecture

Main characteristics

- Highly Configurable and Modular
- Cooperative and Preemptive Threading
- Memory and Resources are typically statically allocated
- Integrated device driver interface
- Memory Protection: Stack overflow protection, Kernel object and device driver permission tracking, Thread isolation
- Bluetooth® Low Energy (BLE 5.1) with both controller and host, BLE Mesh
- 802.15.4 OpenThread
- Native, fully featured and optimized networking stack



Focusing on Kernel and scheduling

Kernel services for development:

- **Multi-threading Services** for cooperative, priority-based, non-preemptive, and preemptive threads with optional round robin time-slicing. Includes POSIX pthreads compatible API support.
- **Interrupt Services** for compile-time registration of interrupt handlers.
- **Memory Allocation Services** for dynamic allocation and freeing of fixed-size or variable-size memory blocks.
  - Note that for **Real-Time static memory allocation is preferred**
- **Inter-thread Synchronization Services** include binary semaphores, counting semaphores, and mutex semaphores.
- **Inter-thread Data Passing Services** include basic message queues, enhanced message queues, and byte streams.
- **Power Management Services** such as tickless idle and an advanced idling infrastructure.

Focusing on Kernel and scheduling

Zephyr provides a comprehensive set of thread scheduling choices:

- Cooperative and Preemptive Scheduling
- Sort of “Earliest Deadline First” (EDF)
  - When enabled, EDF is applied to jobs that have the same fixed priority. Works like a second-level scheduler.
- Meta IRQ scheduling implementing “interrupt bottom half” or “tasklet” behavior
- Timeslicing:
  - Enables time slicing between preemptible threads of equal priority
- Multiple ready-queue management strategies:
  - Simple linked-list ready queue: unsorted, small # of tasks
  - Red/black tree ready queue (self-balanced binary tree. Some memory and overhead. Use for 20+ tasks and scales well)
  - Traditional multi-queue ready queue (array of lists, one per priority)

Threads/Tasks

- **A short note in terminology**
  - The terms “**task**” and “**thread**” are often used **interchangeably**
  - Both refer to an object that has certain properties (e.g. periodicity, deadline, priority, state, stack) and executes some work (a C function) that consumes some time to complete
  - Usually tasks/threads share the same addressing space (unlike processes)
- **Threads/tasks are on the base of the real-time model**
  - A **real-time application** is structured as a **set of (potentially) cooperating tasks** that:
    - Carry out a specific processing (well defined inputs and/or outputs)
    - Are activated at appropriate instants (periodically, in response to an event)
    - Are scheduled for execution according to rules that assure that the requirements are met (e.g. response time, precedence, etc.)
      - E.g. when a user presses an emergency button a machine stops in no more than 100 ms

Threads/Tasks

- **In Zephyr any number of threads can be defined by an application**
  - Limit is the available RAM.
  - Each thread is referenced by a thread id that is assigned when the thread is created
- **Main properties of a thread**
  - A (private) stack area. Size should be adapted according to the space required by thread local variables
  - A thread control block for private kernel bookkeeping of the thread's metadata (e.g. the current state)
  - An entry point (a C function), that corresponds to the work to be carried out
  - A scheduling priority
  - A set of thread options (depend on architecture, e.g. Floating Point Unit)
  - A start delay (how long the kernel should wait before starting the thread)
  - Execution mode (supervisor or user mode).
    - By default threads run in supervisor mode (access to privileged CPU instructions, the entire memory address space, and peripherals). User mode threads have a reduced set of privileges.
      Support to User Mode threads is optional (CONFIG_USERSPACE)

Threads/Tasks

- **Thread lifecycle and states**
  - A thread must be created before use
  - Threads typically execute forever, but they can also be terminated
    - A thread terminates if it returns from its C function
    - In such case it must release all owned shared resources (e.g. mutexes, dynamically allocated memory), as the the kernel does not reclaim them automatically.
    - A thread can be aborted if triggers a fatal condition (e.g. dereferencing a null pointer) or by an explicit call to k_thread_abort() (by itself or other thread)

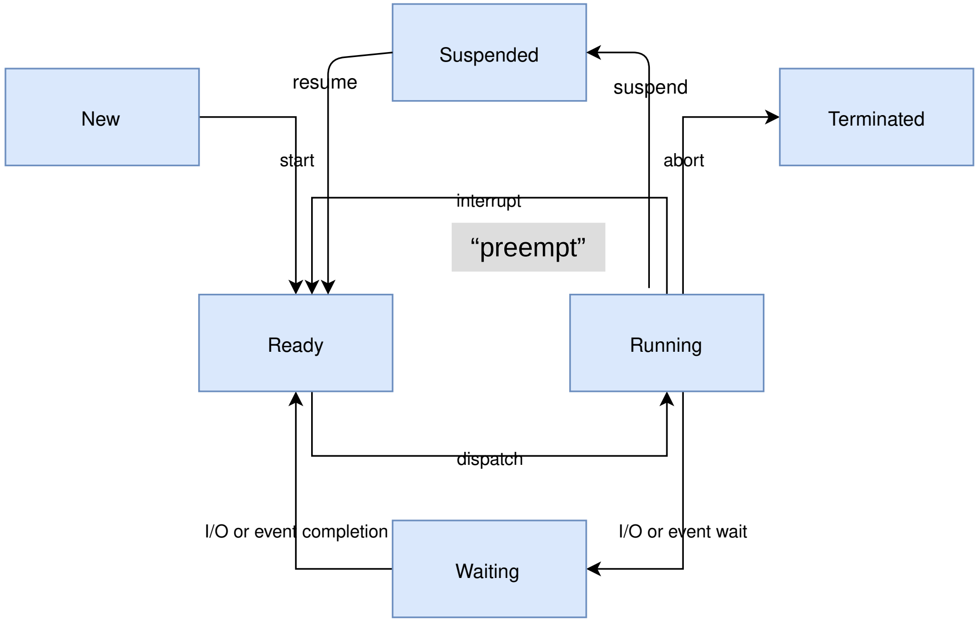


Image from:
https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/reference/kernel/threads/index.html

Threads/Tasks

- **Thread lifecycle and states**
  - A task is **Ready** if it is eligible for execution
    - The scheduler decides which ready tasks are granted with the CPU
    - Preemption can occur
  - A **Suspended** task is prevented from executing for an indefinite time.
    - Calls to k_thread_suspend() and k_thread_resume() mange the Suspended state
  - A task is on the **Waiting** state if is waiting for event, e.g. a semaphore that is unavailable or a timeout to occur

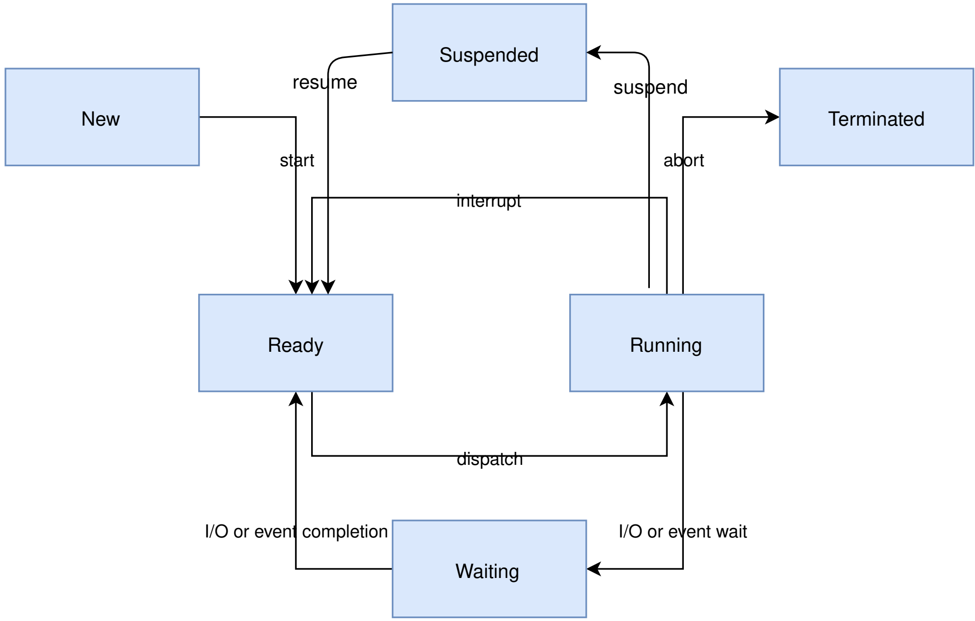


Image from:
https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/reference/kernel/threads/index.html

Threads/Tasks

Thread Stack objects

- Each thread requires its own stack buffer for the CPU to push context. Depending on configuration, there are several constraints that must be met:
  - Additional memory may be reserved for memory management structures
  - If guard-based stack overflow detection is enabled, a small write-protected memory management region must immediately precede the stack buffer to catch overflows.
  - If userspace is enabled, a separate fixed-size privilege elevation stack must be reserved to serve as a private kernel stack for handling system calls.
  - If userspace is enabled, the thread's stack buffer must be appropriately sized and aligned such that a memory protection region may be programmed to exactly fit.
- Moreover alignment constraints can be quite restrictive, for example some MPUs require their regions to be of a power of two in size and word-aligned.
- Because of this, **portable code can't simply pass an arbitrary character buffer to k_thread_create()**. Special macros exist to instantiate stacks, prefixed with K_KERNEL_STACK and K_THREAD_STACK.

Threads/Tasks

Thread Stack objects

- **Kernel-only Stacks**
  - If it is known that a thread will never run in user mode, or the stack is being used for special contexts like handling interrupts, it should be defined via K_KERNEL_STACK macros
  - These stacks minimize memory use (e.g. the kernel doesn’t need need to reserve additional room for the privilege elevation stack)
  - Attempts from user mode to use stacks declared in this way will result in a fatal error for the caller.
- **Thread stacks**
  - If it is known that a stack will need to host user threads, or if this cannot be determined, define the stack with K_THREAD_STACK macros. This may use more memory but the stack object is suitable for hosting user threads.

If CONFIG_USERSPACE is not enabled, K_THREAD_STACK and K_KERNEL_STACK macros become equivalent.

Threads/Tasks

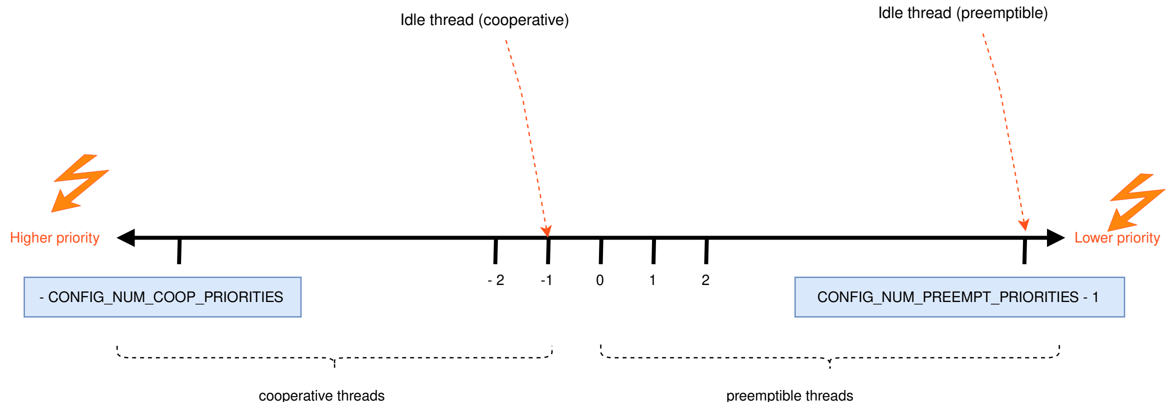
Thread Priorities

- Thread **priority** is an **integer value** (negative or non-negative).
  - Lower priority numbers correspond to higher priority levels
- The scheduler allows **two classes of threads**, based on priority:
  - A **cooperative** thread has a **negative priority value**.
    - Once it becomes the current thread, a cooperative thread remains the current thread until it performs an action that makes it not ready (Suspended or Waiting).
    - **Not well suited for real-time**
  - A **preemptible** thread has a **non-negative priority value**.
    - Once it becomes the current thread, a preemptible thread may be preempted at any time if a cooperative thread, or a preemptible thread of higher or equal priority, becomes ready.
- A thread's initial priority value can be altered dynamically. In particular, a preemptible thread may become a cooperative thread, and vice versa, by changing its priority.

Threads/Tasks

Thread Priorities

- The kernel supports a virtually unlimited number of thread priority levels.
- The configuration options **CONFIG_NUM_COOP_PRIORITIES** and **CONFIG_NUM_PREEMPT_PRIORITIES** specify the number of priority levels for each class, resulting in the following usable priority ranges:
  - Cooperative threads: (-CONFIG_NUM_COOP_PRIORITIES) to -1
  - Preemptive threads: 0 to (CONFIG_NUM_PREEMPT_PRIORITIES - 1)


Idle thread (cooperative)
Idle thread (preemptible)
Higher priority
Lower priority
- 2
-1
0
1
2
- CONFIG_NUM_COOP_PRIORITIES
CONFIG_NUM_PREEMPT_PRIORITIES - 1
cooperative threads
preemptible threads

Threads/Tasks

Implementing tasks/threads

```
#define TASK1_STACK_SIZE 500
#define TASK1_PRIORITY 5
...
void Task1_Code(void *arg1, void *arg2, void *arg3)
{
     /* Task code here */
}
...

K_THREAD_STACK_DEFINE(Task1_stack_area, TASK1_STACK_SIZE);
struct k_thread Task1_thread_data;
k_tid_t Task1_tid;
...

Task1_tid = k_thread_create(&Task1_thread_data, Task1_stack_area,
                          K_THREAD_STACK_SIZEOF(Task1_stack_area),
                          Task1_Code,         /* Pointer to code, i.e. the function name */
                          NULL, NULL, NULL,  /* Three optional arguments */
                          Task1_PRIORITY,
                          0,                 /* Thread options. Arch dependent */
                          K_NO_WAIT);        /* or delay in milliseconds */
```

Threads/Tasks

Implementing tasks/threads

- **Typical structure of a task/thread**

```
void Task1_Code(void *arg1, void *arg2, void *arg3)
{
      /* Initializations (executed only once) */
      ...

      /* Task body – usually never ends */
      while (1) {
            ...
            Do Computations();
            ...
            Some form of Suspend or Wait(); /* Tasks usually don’t run all the time */

      }

    /* Thread terminates if execution reaches this point */
}
```

Threads/Tasks

Implementing tasks/threads

- **Periodic tasks** are common in RT systems
- Simple (but poor, why?) method:

```
void Task1_Code(void *arg1, void *arg2, void *arg3)
{
      /* Initializations (executed only once) */
      ...

      /* Task body – usually never ends */
      while (1) {
            ...
            Do Computations();
            ...
            k_msleep(TASK1_PERIOD);
      }

    /* Thread terminates if execution reaches this point */
}
```

Threads/Tasks

Implementing tasks/threads

- **Sporadic tasks** are also common in RT systems
  - Usually there is a primitive that puts the thread in Waiting/Suspend state

```
void Task1_Code(void *arg1, void *arg2, void *arg3)
{
     /* Initializations (executed only once) */
     ...

     /* Task body – usually never ends */
     while (1) {
          ...
          Do Computations();
          ...
          Blocking call to a semaphore/mutex/event/condition_variable/...
     }

    /* Thread terminates if execution reaches this point */
}
```

Threads/Tasks

Thread runtime statistics

- Runtime statistics can be gathered and retrieved if CONFIG_THREAD_RUNTIME_STATS is enabled
- Example of statistics is the total number of execution cycles of a thread.
- By default runtime statistics are gathered using the default kernel timer. Some platforms have higher resolution timers. The use of these timers can be enabled via CONFIG_THREAD_RUNTIME_STATS_USE_TIMING_FUNCTIONS.
- Example:

```
k_thread_runtime_stats_t rt_stats_thread;

...

k_thread_runtime_stats_get(k_current_get(), &rt_stats_thread);

printk("Cycles: %llu\n", rt_stats_thread.execution_cycles);

...
```

Bibliography

- Kate Stewart, “Zephyr Project: Unlocking Innovation with an Open Source RTOS”, The Linux Foundation, 2021.
- Nordic documentation for kernel services
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/index.html#zephyr-project-documentation
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/introduction/index.html#introduction
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/services/scheduling/index.html#scheduling
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/services/threads/index.html#threads
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/timeutil.html#time-utilities
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/kernel/timeutil.html#time-utilities

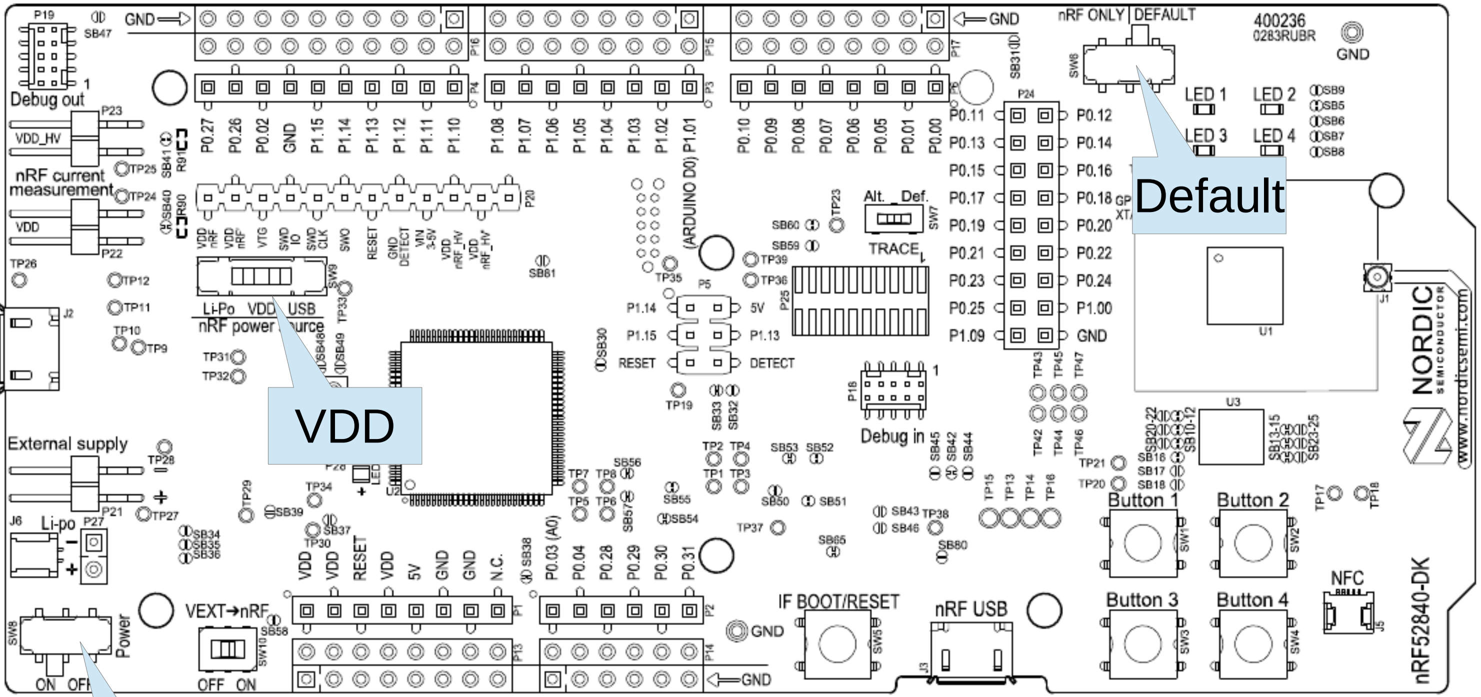
Creating an application with the nRF SDK

Paulo Pedreiras
SOTR
2022/10

Preliminaries

- Connect the nRF DK to the PC via the USB cable
- Check that the switches are on the correct position
  - Power – ON (left)
  - NRF Power Source – VDD (middle)
  - NRF Only – Default (right)
- If using the VM, pass the USB to it
  - On the VM, Devices → USB → Segger J-Link

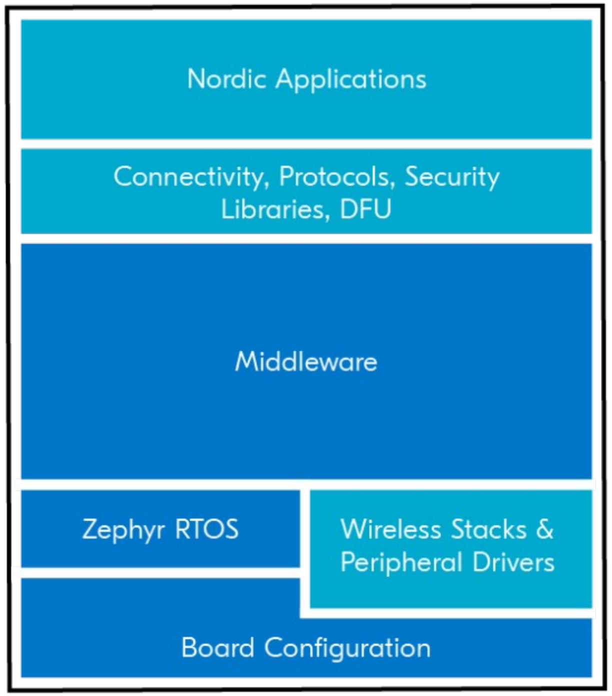
Preliminaries



The nRF Connect SDK structure

- nRF Connect SDK is an unified software development kit for building applications for Nordic Semiconductor nRF52, nRF53, and nRF91 Series
- It **integrates the Zephyr RTOS** and a wide range of complete applications, samples, and protocol stacks
  - E.g. BLE, Bluetooth mesh, Matter, Thread/Zigbee, LTE-M, NB-IoT, TCP/IP.
- It also includes middleware
  - E.g. CoAP, MQTT, LwM2M
- It also features various libraries, hardware drivers, Trusted Firmware-M for security, and a secure bootloader (MCUBoot).

The nRF Connect SDK structure


Nordic Applications
Connectivity, Protocols, Security Libraries, DFU
Middleware
Zephyr RTOS
Wireless Stacks & Peripheral Drivers
Board Configuration

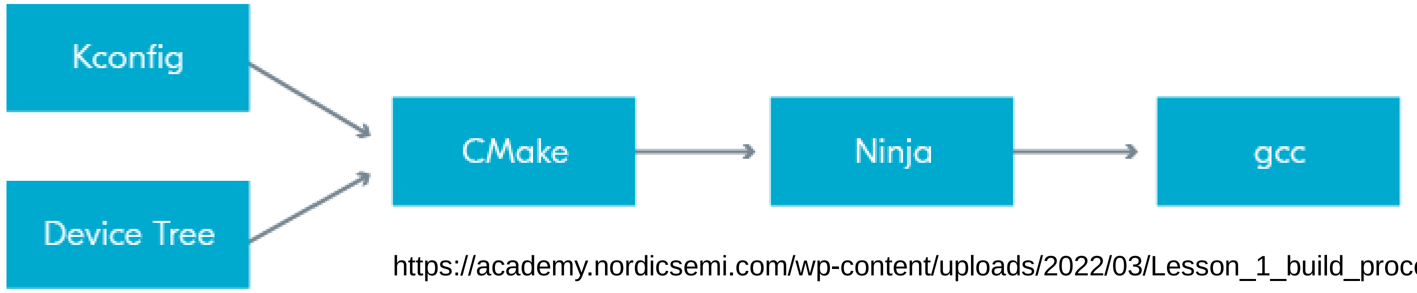
https://academy.nordicsemi.com/wp-content/uploads/2022/03/nRF_connect_sdk_2-01.png

The nRF Connect SDK structure

- Internally, the nRF Connect SDK code is organized into four main repositories:

– Nrf:	Applications, samples, connectivity protocols (Nordic)
– Nrfxlib:	Common libraries and stacks (Nordic)
– Zephyr:	RTOS & Board configurations (open source)
– MCUBoot:	Secure Bootloader (open source)

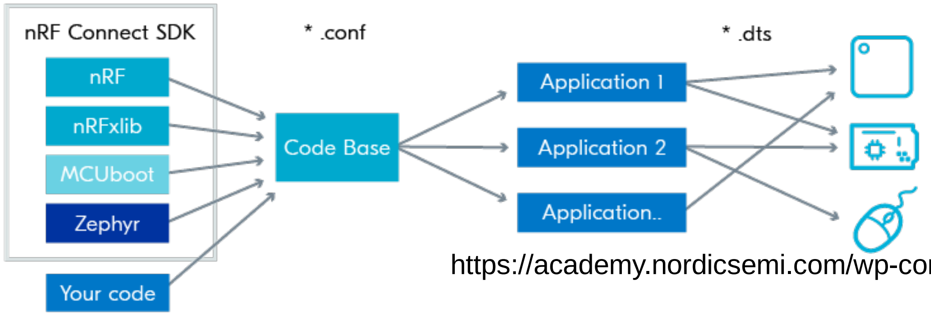
Toolchain structure



https://academy.nordicsemi.com/wp-content/uploads/2022/03/Lesson_1_build_process-04.png

- Kconfig: sets definitions that configure the whole system (e.g. which if GPIO is used, if UART is used, which wireless protocol to use …)
- Devicetree: describes in an abstract way the hardware.
- Cmake: uses the information from Kconfig and the devicetree to generate build files
- Ninja (compares to make) : takes the build files generated by Cmake to build the program
- GCC: the compiler toolchain used to create the executables

Toolchain structure

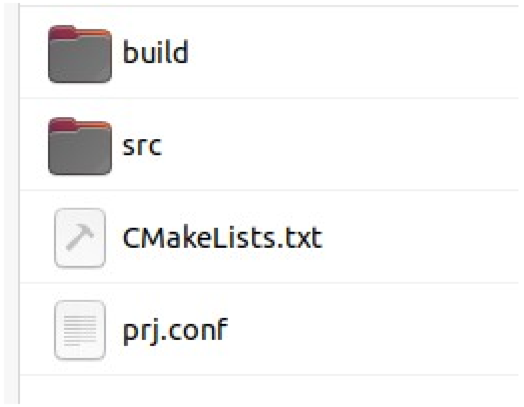


https://academy.nordicsemi.com/wp-content/uploads/2022/03/Application_build_process-05.png

- The high decoupling between source code and the HW platform, enabled by the use of Kconfig and devicetrees allows using the same application source code on different hardware and with different configurations with minimal changes.
- Huge impact on portability and maintainability

Structure of a project

- A basic project includes at least the following folders and files
  - build
    - Files generated automatically by the build system
  - src
    - Folder where the programmer should put the source files. i.e., the application code
  - CMakeLists.txt
    - Contains a set of directives and instructions describing the project's source files and targets
  - Prj.conf
    - Project configuration options. E.g. if the moiles uses the ADC, GPIOs, etc.



Structure of a project

- Example of CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(basic_01_blinkDemo)
target_sources(app PRIVATE src/main.c)
```

- Example of prj.conf

```
CONFIG_GPIO=y
CONFIG_PWM=y
CONFIG_STDOUT_CONSOLE=y
CONFIG_PRINTK=y
CONFIG_LOG=y
CONFIG_LOG_PRINTK=y
CONFIG_LOG_MODE_IMMEDIATE=y
CONFIG_PWM_LOG_LEVEL_DBG=y
```

Starting from a Nordic Sample

- Welcome → Create a new application
- Check the SDK, toolchain, …
  - Everything should be fine, if the isnbtallation is OK
  - Ignore the “gn not found” message. Only necessary for Matter aplications
- Choose a location
- In the templates select a suitable template
  - E.g. “Hello World”
- Assign a name
- Hit Cretae Applicaion button

Starting from an existing application

- Copy project folder to the intended path. It must include
  - CmakeLists.txt
  - Prj.conf
  - Eventually DTS Overlay files
  - Source code (e.g. in ./src folder)
  - Remove any build folder, e.g. “build”, or “build_nrf52840dk_nrf52840”. This step is not strictly necessary but can save you from trouble
- Go to VScode
  - Switch to nRF Connect view
  - Application → “+” / Add folder as application
  - Select the folder that has the project files

Add a build configuration

- The build configuration identifies the target HW and allows setting several parameters
  - Applications → Action button after application entry
  - Select “nrf52840dk_nrf52840” (in our case)
  - Confirm the name of the “prj.conf” file
  - Set any Cmake arguments
  - Set the build folder name (if not pretend to use the default one)
  - Click on “Build Configuration”

Linking the app and the HW and building + flashing

- On the Applications window, open the “App_Name” drop-down menu to show the “build” entry
  - Hit the button “Link Build Configuration and Device”
  - Any other apps linked must be removed
- Then you can hit button “Flash all linked devices” icon after “Applications” entry
  - There are several other options
    - E.g. Actions → Flash

Editing the files and configs and utilities

- The active application has a matching window just below the “Applications” window that allows to navigate through the files
  - Source files (e.g. user code)
  - Input files (e.g. CmakeLists.txt, prj.conf, …)
- There are a few windows below the code that are useful
  - Problems: report issues in user code, config files, …
  - Terminal (compilation results)
  - NRF Terminal – an integrated UART terminal, useful for debugging

The UART interface

- Default terminal configuration
  - 115200 bn1 rtscts:off
- Can use the internal terminal or any other one
  - E.g. minicom, putty, ...