Zephyr build system

The Zephyr build system is highly sophisticated and built upon a few key components: **CMake**, **Kconfig**, and **Devicetree**. It's designed to be flexible, allowing for extensive configuration and targeting a wide range of hardware.

Here's a breakdown of the process of generating build systems on Zephyr:

Core Components:

- **CMake:** This is the primary build system generator. It takes `CMakeLists.txt` files (which define the project structure, dependencies, and build rules) and generates native build scripts for your host platform. Zephyr officially supports **Ninja** and **Make** as the native build tools.
- **Kconfig:** This is a powerful configuration system (similar to the Linux kernel's Kconfig). It allows you to select and configure various features, drivers, and modules for your Zephyr application and kernel. Configuration options are typically set in `prj.conf` files.
- **Devicetree:** This is a hardware description language used to describe the hardware components of your target board (SoC, peripherals, memory layout, etc.). `.dts` and `.dtsi` files define the devicetree, and overlays (`.overlay` files) can be used to customize it for your specific application.

The Two Main Phases:

The Zephyr build process is conceptually divided into two main phases:

1. Configuration Phase (Driven by CMake)

This phase is where the build system is "generated" and tailored to your specific application and hardware.

- **CMakeLists.txt Processing:** CMake starts by processing the `CMakeLists.txt` file in your application directory. This file, in turn, references Zephyr's top-level `CMakeLists.txt` and other CMake files throughout the Zephyr source tree. These scripts define how the application and kernel should be built together.
- **Kconfig Configuration:**
    - Zephyr collects `Kconfig` files from the application, board, SoC, and various subsystems.
    - It merges all configuration fragments (e.g., your `prj.conf`, board-specific `defconfig` files, etc.).
    - Based on these configurations, Kconfig generates:

        - `.config`: A file listing all selected Kconfig options and their values.
        - `autoconf.h`: A header file containing C preprocessor macros corresponding to the selected Kconfig options, making them accessible in your source code.
- **Devicetree Processing:**
    - `.dts` and `.dtsi` files are collected from various directories (architecture, SoC, board, application).
    - The C preprocessor merges `.dtsi` files and applies any devicetree `.overlay` files, expanding macros.
    - A Python script (`gen_defines.py`) parses the preprocessed devicetree sources to generate:
        - `devicetree_generated.h`: A header file with preprocessor macros derived from the devicetree, allowing your source code to access hardware information.
        - `zephyr.dts`: The final devicetree file, useful for debugging.
    - (Optional) If installed, the `dtc` (devicetree compiler) is run on `zephyr.dts` to catch any additional warnings or errors.
- **Build Script Generation:** Based on all the gathered information (CMake rules, Kconfig selections, Devicetree descriptions), CMake generates the native build scripts (e.g., `build.ninja` for Ninja or `Makefile` for Make) in your designated build directory. These scripts contain all the instructions needed to compile and link your application and the Zephyr kernel.

2. Build Phase (Driven by Ninja or Make)

This phase uses the generated build scripts to actually compile the source code and produce the final executable.

- **Pre-build:** Tasks like preparing directories, copying files, and generating necessary headers (e.g., from Kconfig and Devicetree) occur before source files are compiled.
- **Source Compilation:** Source files (C, C++, assembly) from your application, the Zephyr kernel, and enabled modules are compiled into object files (`.o` files) and then archived into libraries (`.a` files).
- **Linking:** The compiled object files and libraries are linked together to create intermediate binaries.
- **Post-processing:** Additional scripts are executed on the intermediate binaries. This might involve:
    - Scanning for kernel objects and generating perfect hash functions for object validation (when user mode is enabled).
    - Processing devicetree data for device relationships.

- **Final Binary Generation:** The ultimate output is a complete Zephyr application image in a format suitable for flashing onto your target board (e.g., `zephyr.elf`, `zephyr.hex`, `zephyr.bin`).

How you typically initiate this process:

While you can directly invoke `cmake` and then `ninja` (or `make`), the most common and recommended way to build Zephyr applications is using the `west` meta-tool:

Bash

```
west build -b <board> <your_application_directory>
```

- `west build`: This command acts as a wrapper, invoking CMake and the underlying build tool (Ninja by default).
- `-b <board>`: Specifies the target board for which you are building. This is crucial as it informs Zephyr about the hardware configuration.
- `<your_application_directory>`: The path to your application's root directory, which contains your `CMakeLists.txt`, `prj.conf`, and source files.

Example:

If you have a "hello_world" application in `~/my_zephyr_apps/hello_world` and you want to build it for the `nrf52840dk_nrf52840` board, you would typically run:

Bash

```
cd ~/my_zephyr_apps/hello_world
west build -b nrf52840dk_nrf52840
```

This command will:

1. Create a `build` directory (or use an existing one).
2. Run CMake to configure the build system within that `build` directory.
3. Execute Ninja (or Make) to compile and link your application and the Zephyr kernel, producing the final firmware image.

Sysbuild (System Build):

For more complex scenarios, Zephyr offers **Sysbuild**. This is a higher-level build system that can combine multiple other build systems. For example, you might use Sysbuild to build a Zephyr application along with a bootloader like MCUboot, ensuring they are correctly linked and flashed. Sysbuild also leverages CMake and can be invoked via `west build --sysbuild` or directly through CMake.

Understanding these components and phases is key to effectively developing and debugging applications with Zephyr.