Porting Zephyr OS to a SoC (Cortex-R5)

M Tamseel Shams
Associate Staff Engineer
Samsung Semiconductor India Research

Agenda

- About Zephyr OS
- Porting Components
- Hardware Support Implementation
- Debugging Tips
- Key tools and build commands
- Timers
- UART Driver
- Things for Future

About Zephyr OS

- Zephyr is a small real-time operating system (RTOS) for connected, resource-constrained and embedded devices
- Zephyr OS is based on small scale kernel to support simple embedded system
- It is a mini kernel which provides similar features like Linux kernel. Like memory management, multi threading, scheduler, interrupt handling
- Supports wide verity of CPU architectures
- Provides build system (west) which makes easy to compile multiple kernel modules along with user applications
- Provides SDK which has tool chains for different architectures at one place

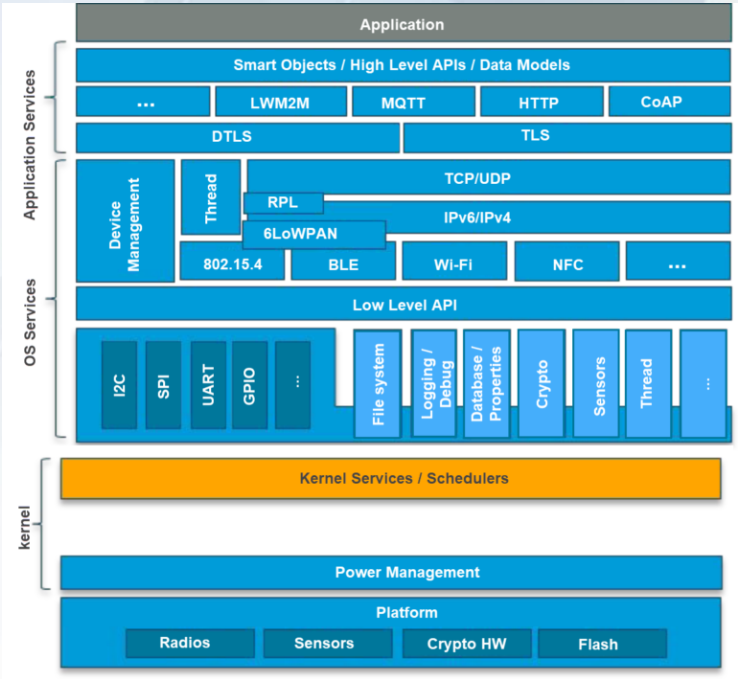


Fig. 1: Zephyr System Architecture

Image Reference: https://www.microcontrollertips.com/zephyr-open-source-rtos-iot-faq/

Porting Components

- In this session will discuss about key folder changes and adding new project to Zephyr
  - Adding a new SoC
  - Adding a new Board
  - Adding new Drivers
  - Adding Device Tre

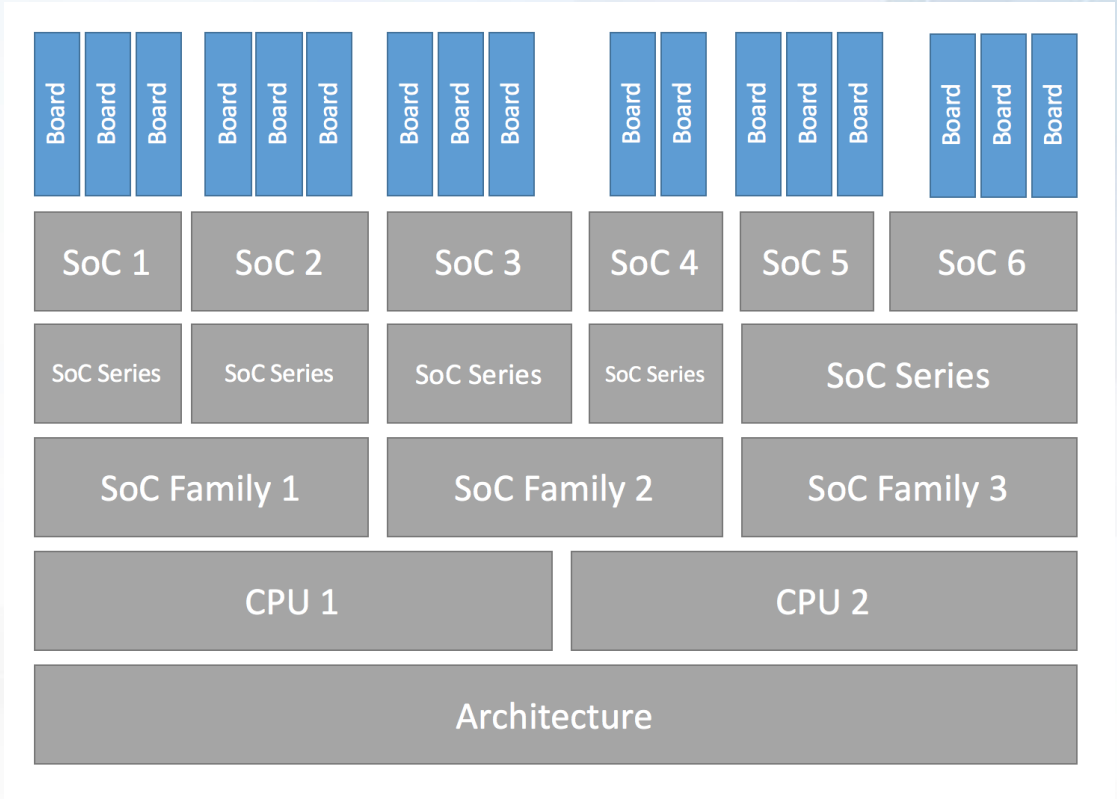


Fig. 2: Zephyr Configuration Hierarchy

Image Reference: https://docs.zephyrproject.org/3.1.0/hardware/porting/board_porting.html

Hardware Support Implementation

- Architecture: **arm**
- CPU core: **CORTEX_R5**
- SoC family: **<company_name>_<soc_name>**
  (e.g. samsung_xyz)
- SoC Series: **CR5**
- SoC: **CR5**
- Drivers: **Serial, Timer, Interrupt, etc.**
- Board: **Samsung Board**

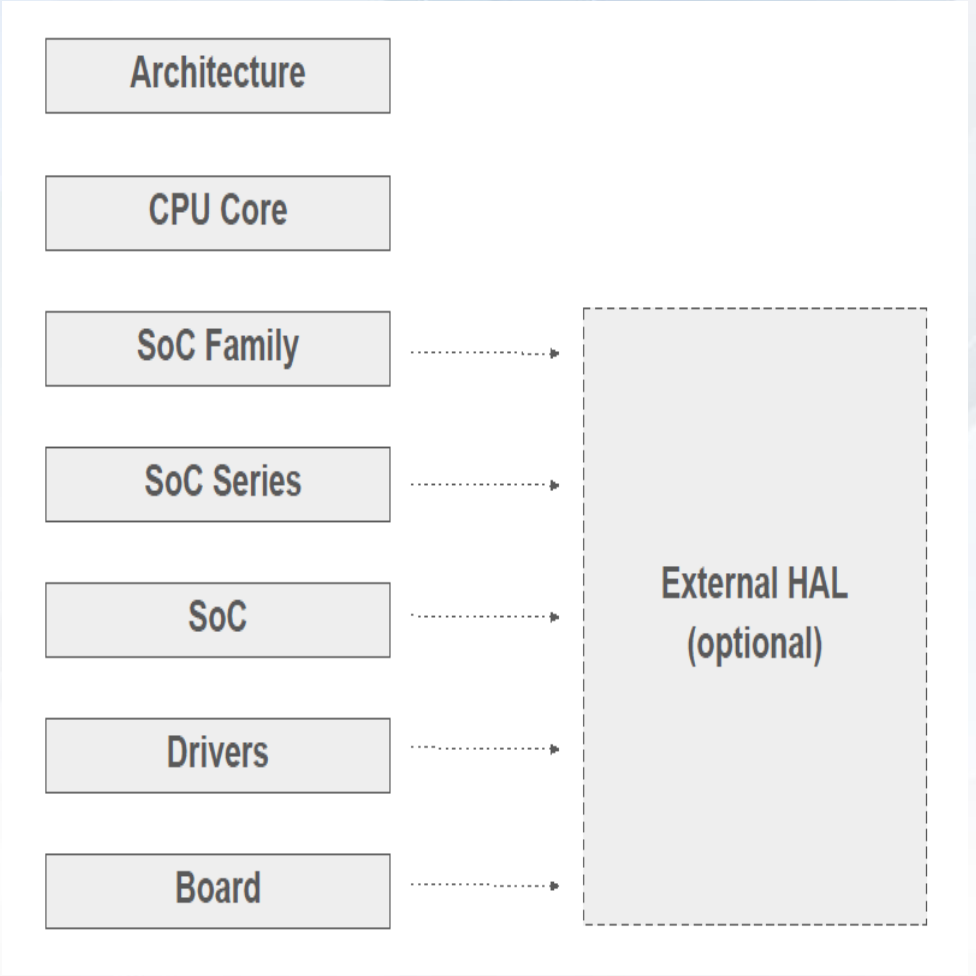


Fig. 3

Hardware Support Implementation

Hardware Configuration Hierarchy:

- Architecture - arc, arm, nios2, posix, riscv32, x86, xtensa, etc.
- CPU Core - Implements early boot sequence, interrupt and exception handling, thread context switching, thread creation and termination, CPU idling/power management, fault management, linker scripts and toolchains. E.g. ARCV2, CORTEX_M0PLUS, CORTEX_R5, etc.
- SoC Family - Represents a single SoC type that can have more than one variations in terms of peripherals and features. E.g. KINETIS, IMX, NRF, EXX32, STM32, etc.

Hardware Support Implementation

- SoC Series - Represents the specific peripherals and features for the SoC family variations. E.g. NRF51X, NRF52X, etc.
- SoC - The actual SoC that is “soldered” in the hardware platform and its configuration. E.g. MKL25Z32VFM4, MCIMX7D5EVM10SC, etc.
- Drivers - Include device model responsible for configuring and initialize drivers. Each driver follows a device model API and a specific driver type API. E.g. interrupt controller, timer, serial communications, etc.
- Board - Includes a SoC and it’s associated peripherals and features including external components and devices. E.g. NRF51_BLENANO, NUCLEO_F103RB, etc.

Hardware Support Implementation

- Top level hardware configurations are defined via Kconfigs and the final processing results located in the files:

  build/<board>/zephyr/.config

  build/<board>/zephyr/include/generated/autoconf.h

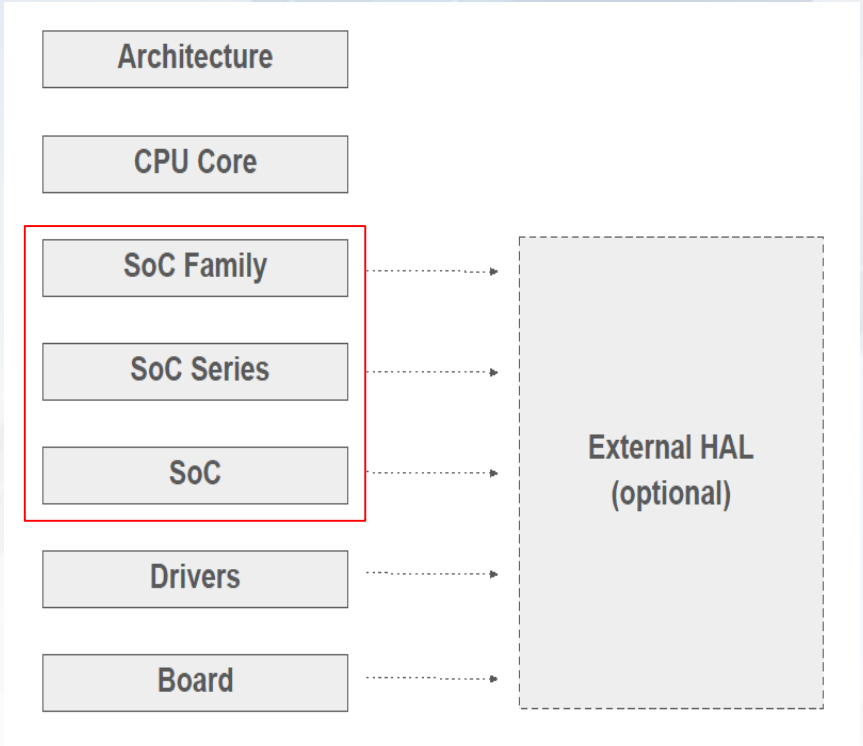
- Low level hardware specific configurations are defined via device tree and the final processing results located in the files:

  build/<board>/zephyr/include/generated/generated_dts_board.conf

  build/<board>/zephyr/include/generated/generated_dts_board.h

Adding a new SoC

- Defines the SOC_FAMILY, SOC_SERIES, SOC and SOC_PART_NUMBER configs
- Located at: soc/<architecture>/<soc_family>/<soc_series>/ (soc/arm/samsung_xyz/cr5/)
- Each SoC folder holds all it's basic information about processor, IRQ controller, flash details, linkers, soc related init configurations
- Called in the system initialization process with priority 0
- Provides a soc.h header which will be included by the board and drivers sources
- Contains a set of Kconfig files, linker definitions, and device tree fixups



Adding a new SoC

- Default Architecture, SOC_FAMILY, SOC_SERIES configs are selected in

  boards/<architecture>/<board_name>/<board_name>_defconfig

  (boards/arm/samsung_xyz/samsung_cr5_defconfig)

- Has a dtsi defining peripherals and features properties presented in the SoC and is located at

  dts/<architecture>/<vendor>/<vendor>_<soc_name>.dtsi

  (dts/arm/samsung_xyz/samsung_cr5.dtsi)

soc/arm/samsung_xyz/ → SOC

Family

- CMakeLists.txt
- Kconfig
- Kconfig.defconfig
- Kconfig.soc
- Kconfig.zephyrver
- cr5 → SOC variant
  - CMakeLists.txt
  - Kconfig.defconfig.series
  - Kconfig.series
  - Kconfig.soc
  - linker.ld
  - soc.c
  - soc.h

Adding DTS file

- Device tree include files which contains main SoC details, and IP nodes like IRQ controller, CPU information, etc.
- These dtsi files will be included in actual board dts files
- Dtsi files are reusable there is no strict rule to create new dtsi file every soc
- Location is dts/<architecture>/<vendor>/<vendor>_<soc_name>.dtsi

  (dts/arm/samsung_xyz/samsung_cr5.dtsi)
- Instead of requiring additional runtime memory to store the DTB blob, the DTS information is used during compile time
- The compiled DTS files are then cross checked using rules specified in YAML bindings. The extracted information is placed in a header file that is used by the rest of the code as the project is compiled

Adding DTS file

dts/arm/samsung_xyz/

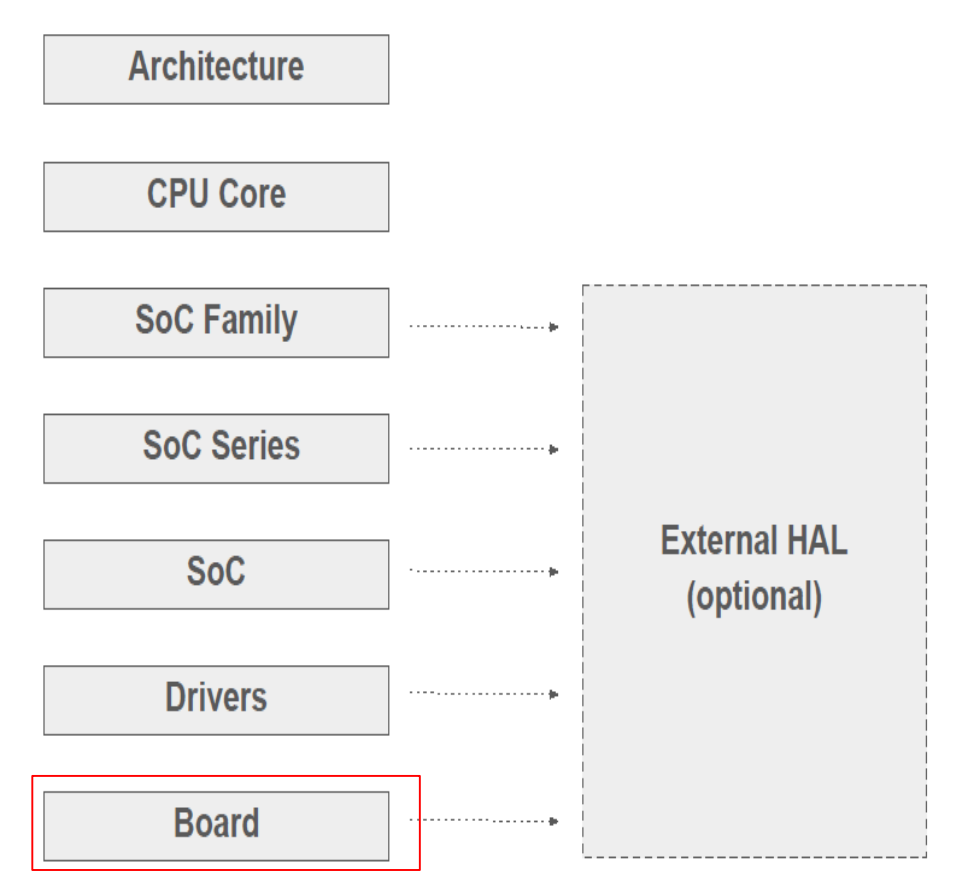
├── samsung_cr5.dtsi

├── samsung_m0.dtsi

```
cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            device_type = "cpu";
            compatible = "arm,cortex-r5";
            reg = <0>;
        };
    };
soc {
        interrupt-parent = <&gic>;
        gic: interrupt-controller@8000000 {
            compatible = "arm,gic";
            reg = <0x8000000 0x10000>,
                <0x8010000 0x10000>;
            interrupt-controller;
            #interrupt-cells = <4>;
            label = "GIC";
            status = "okay";
        };
};
```

Adding Board

- This folder represent actual application hardware
- Located at boards/<architecture>/<board_name>/ (boards/arm/samsung_xyz/)
- Board specific init information will be defined here
- Contains a <board_name>_defconfig file to select which SoC and basic features and interfaces included



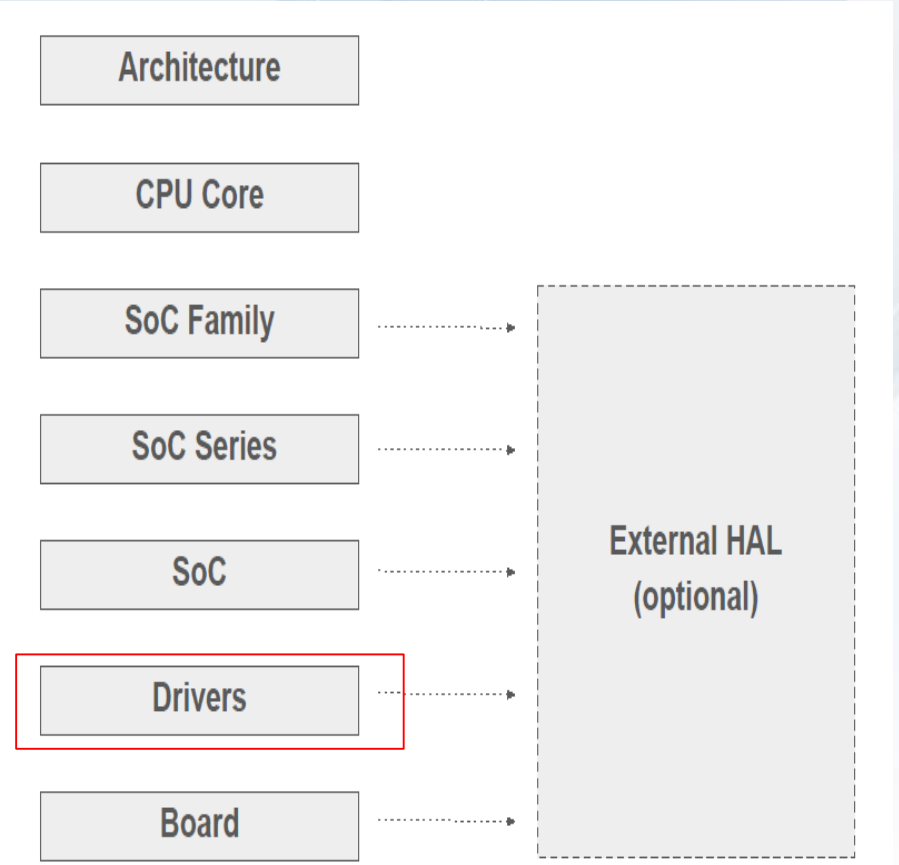
Adding Board

- While compiling zephyr image we always refer board name, which indirectly includes all soc, dtsi related information
- Contains a board.h to be used by the drivers and applications
- May provide a board.cmake to instruct how to flash/debug
- Includes a <board_name>.yaml file to list the board properties: e.g. flash and ram sizes and toolchain used, etc.

```
boards/arm/samsung/
├── board.c
├── CMakeLists.txt
├── Kconfig.board
├── Kconfig.defconfig
├──
samsung_cr5_defconfig
├── samsung_cr5.dts
└── samsung_cr5.yaml
```

Adding Drivers

- Located at drivers/<driver_type>/
- New drivers are added under drivers folder, and if the driver related to any specific protocol like serial, dma, usb, etc. should add under that sub folder
- Selection and configuration done via Kconfigs and device tree
- Initialization performed during the kernel boot
- Yaml file to describe the device tree nodes and properties
- Device tree file to define driver properties and configurations



Adding Drivers

- Steps to add driver to compilation:
  - Apart from driver files, Kconfig files need to be created or need to give driver entry in main Kconfig file.
  - Add respective driver entry in CMakeList of that driver folder.

```
/drivers/serial/
├── CMakeLists.txt
├── Kconfig
|── Kconfig.cr5
```

- Steps to add driver to init sequence:
  - #define DT_DRV_COMPAT cr5_uart (this same as device tree compatible name)
  - DEVICE_AND_API_INIT this macro has different variants with different arguments but all it do add drivers to init sequence.
  - Hook driver APIs to zephyr frame work.

Debugging Tips

- Look at other source code reference to understand what needs to be done to initialize the SoC
- Try to print to UART (accessing the registers directly) in the SoC initialization to guarantee that the core is up and running
- Implement the UART driver first, printk is life
- Turn on the System Logging or Logger
- Turn on asserts (CONFIG_ASSERT) to try to catch errors
- Use a on-chip debugger (UltraSoC, J-Link, ULINK etc.)

Key tools and build commands

- Zephyr build system use these important tools
  - Cmake -> Holds all source code build steps
  - Yaml -> these files provide information, used mostly for documentation, on help commands
  - West -> this tool provide build commands which internally use cmakelist which are defined in each sub directory
- Zephyr uses west commands to build
  - west init (To init zephyr build)
  - west board (To list available boards)
  - west build -b samsung_cr5 samples/hello_world/ -p

Timers

- A timer is a kernel object that measures the passage of time using the kernel's system clock
- When a timer's specified time limit is reached it can perform an application-defined action, or wait
- Any number of timers can be defined. Each timer is referenced by its memory address
- A timer has the following key properties: *duration*, *period*, *expiry function*, *stop function*
- Implementation available for using timers:
  - Defining a timer - A timer is defined using a variable of type k_timer. It must then be initialized by calling k_timer_init()
  - Using a Timer Expiry Function
  - Reading Timer Status
  - Using Timer Status Synchronization

UART Driver

- This sample demonstrates how to use the UART serial driver with a simple echo bot
- It reads data from the console and echoes the characters back after an end of line (return key) is received
- The polling API is used for sending data and the interrupt-driven API for receiving
- By default, the UART peripheral that is normally used for the Zephyr shell is used, so that almost every board should be supported
- Building and running

  west build -b nrf52840dk_nrf52840 samples/drivers/uart/echo_bot

  west flash

- Sample Output

  Hello! I\'m your echo bot.

  Tell me something and press enter:

  # Type e.g. "Hi there!" and hit enter!

  Echo: Hi there!

Things for Future

- Zephyr has frame work for most of generic drivers but how to add individual drivers to it need to be explored.
- Device tree can hold most of IP hardware details like clocks, IRQ line, address, register set, etc. and all these entries can be read from device driver itself with defined macros which need to be explored.
- All pre defined linker scripts, dtsi files can be modified and used in build. This areas need to be explored if any booting issues faced

THANK YOU