Zephyr Project Documentation

发布 ***1.9.0***

many

8月 28, 2017

Contents

提示： 当前版本的中文版正在翻译当中，您可以在 这里 查看其它版本。

关于早期发布的其它版本的信息，请查阅 zephyr_release_notes。

Zephyr OS 的主体代码遵循开源协议 Apache 2.0 license ，您可以通过查看 GitHub 仓库 中的 LINCENSE 文件查看该协议的具体内容。除此之外，Zephyr OS 还导入/引用了一些软件包、脚本以及其它文件，这些文件遵循协议 Zephyr_Licensing。

Zephyr 项目的源代码维护在 GitHub 仓库 中。

CHAPTER 1

目录

Introducing Zephyr

The Zephyr OS is based on a small-footprint kernel designed for use on resource-constrained systems: from simple embedded environmental sensors and LED wearables to sophisticated smart watches and IoT wireless gateways.

The Zephyr kernel supports multiple architectures, including ARM Cortex-M, Intel x86, ARC, NIOS II, Tensilica Xtensa and RISC V. The full list of supported boards can be found here.

Licensing

Zephyr uses the Apache 2.0 license (as found in the LICENSE file in the project's GitHub repo). There are some imported or reused components of the Zephyr project that use other licensing, as described in Zephyr_Licensing.

Distinguishing Features

The Zephyr kernel offers a number of features that distinguish it from other small-footprint OSes:

1. **Single address-space**. Combines application-specific code with a custom kernel to create a monolithic image that gets loaded and executed on a system's hardware. Both the application code and kernel code execute in a single shared address space.
2. **Highly configurable**. Allows an application to incorporate *only* the capabilities it needs as it needs them, and to specify their quantity and size.
3. **Compile-time resource definition**. Allows system resources to be defined at compile-time, which reduces code size and increases performance.
4. **Minimal error checking**. Provides minimal run-time error checking to reduce code size and increase performance. An optional error-checking infrastructure is provided to assist in debugging during application development.
5. **Extensive suite of services**. Offers a number of familiar services for development:

- *Multi-threading Services* for priority-based, non-preemptive and preemptive threads with optional round robin time-slicing.
- *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.
- *Inter-thread Synchronization Services* for binary semaphores, counting semaphores, and mutex semaphores.
- *Inter-thread Data Passing Services* for basic message queues, enhanced message queues, and byte streams.
- *Power Management Services* such as tickless idle and an advanced idling infrastructure.

Community Support

The Zephyr Project Developer Community includes developers from member organizations and the general community all joining in the development of software within the Zephyr Project. Members contribute and discuss ideas, submit bugs and bug fixes, and provide training. They also help those in need through the community's forums such as mailing lists and IRC channels. Anyone can join the developer community and the community is always willing to help its members and the User Community to get the most out of the Zephyr Project.

Welcome to the Zephyr community!

Resources

Here's a quick summary of resources to find your way around the Zephyr Project support systems:

- **Zephyr Project Website**: The https://zephyrproject.org website is the central source of information about the Zephyr Project. On this site, you'll find background and current information about the project as well as all the relevant links to project material. For a quick start, refer to the Zephyr Introduction and Getting Started Guide.
- **Releases**: Source code for Zephyr kernel releases are available at https://zephyrproject.org/downloads. On this page, you'll find release information, and links to download or clone source code from our GitHub repository. You'll also find links for the Zephyr SDK, a moderated collection of tools and libraries used to develop your applications.
- **Source Code in GitHub**: Zephyr Project source code is maintained on a public GitHub repository at https://github.com/zephyrproject-rtos/zephyr. You'll find information about getting access to the repository and how to contribute to the project in this Contribution Guide document.
- **Samples Code**: In addition to the kernel source code, there are also many documented Sample and Demo Code Examples that can help show you how to use Zephyr services and subsystems.
- **Documentation**: Extensive Project technical documentation is developed along with the Zephyr kernel itself, and can be found at https://zephyrproject.org/doc. Additional documentation is maintained in the Zephyr GitHub wiki.
- **Issue Reporting and Tracking**: Requirements and Issue tracking is done in our JIRA system: https://jira.zephyrproject.org. You can browse through the reported issues and submit issues of your own.
- **Mailing List**: The Zephyr Mailing Lists are perhaps the most convenient way to track developer discussions and to ask your own support questions to the Zephyr project community. You can also read through message archives to follow past posts and discussions, a good thing to do to discover more about the Zephyr project.
- **IRC Chatting**: You can chat online with the Zephyr project developer community and other users in our IRC channel #zephyrproject on the freenode.net IRC server. You can use the http://webchat.freenode.net web client or use a client-side application such as pidgin.

Fundamental Terms and Concepts

See glossary

Getting Started Guide

Use this guide to get started with your *Zephyr* development.

Set Up the Development Environment

The Zephyr project supports these operating systems:

- Linux
- Mac OS
- Windows 8.1

Use the following procedures to create a new development environment.

Development Environment Setup on Linux

This section describes how to set up a Linux development system.

After completing these steps, you will be able to compile and run your Zephyr applications on the following Linux distributions:

- Ubuntu 16.04 LTS 64-bit
- Fedora 25 64-bit

Where needed, alternative instructions are listed for Ubuntu and Fedora.

Installing the Host's Operating System

Building the project's software components including the kernel has been tested on Ubuntu and Fedora systems. Instructions for installing these OSes are beyond the scope of this document.

Update Your Operating System

Before proceeding with the build, ensure your OS is up to date. On Ubuntu, you'll first need to update the local database list of available packages before upgrading:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

On Fedora:

```
$ sudo dnf upgrade
```

Note that having a newer version available for an installed package (and reported by `dnf check-update`) does not imply a subsequent `dnf upgrade` will install it, because it must also ensure dependencies and other restrictions are satisfied.

Installing Requirements and Dependencies

Install the following with either apt-get or dnf.

Install the required packages in a Ubuntu host system with:

```
$ sudo apt-get install git make gcc g++ ncurses-dev \
      doxygen dfu-util device-tree-compiler python3-ply python3-pip
```

Install the required packages in a Fedora host system with:

```
$ sudo dnf group install "Development Tools"
$ sudo dnf install git make gcc glibc-static \
      libstdc++-static ncurses-devel \
      doxygen dfu-util dtc python3-pip \
      python3-ply python3-yaml dfu-util dtc python3-pykwalify
```

Install additional packages required for development with Zephyr:

```
$ pip3 install --user -r scripts/requirements.txt
```

Installing the Zephyr Software Development Kit

Zephyr's SDK (Software Development Kit) contains all necessary tools and cross-compilers needed to build the kernel on all supported architectures. Additionally, it includes host tools such as a custom QEMU and a host compiler for building host tools if necessary. The SDK supports the following architectures:

- X86 (Intel Architecture 32 bits)
- X86 IAMCU ABI (Intel Architecture 32 bits IAMCU ABI)
- ARM (Advanced RISC Machines)
- ARC (Argonaut RISC Core)
- NIOS II
- XTENSA

Follow these steps to install the SDK on your Linux host system.

1. Download the latest SDK self-extractable binary.

   Visit the Zephyr SDK archive to find all available SDK versions, including the latest version.

   Alternatively, you can use the following command to download the desired version (*0.9.1* can be replaced with the version number you wish to download).

   ```
   $ wget https://github.com/zephyrproject-rtos/meta-zephyr-sdk/releases/download/0.
   ↪9.1/zephyr-sdk-0.9.1-setup.run
   ```

2. Run the installation binary, follow this example:

   重要: Make sure you have installed all required packages for your host distribution as described in the previous section *linux_required_software* otherwise the SDK installation will fail.

   ```
   $ chmod +x zephyr-sdk-<version>-setup.run
   $ ./zephyr-sdk-<version>-setup.run
   ```

There is no need to use `sudo` if the SDK is installed in the current user's home directory.

3. Follow the installation instructions on the screen. The toolchain's default installation location is `/opt/zephyr-sdk/`. To install in the default installation location, you will need to use sudo. It is recommended to install the SDK in your home directory and not in a system directory.

4. To use the Zephyr SDK, export the following environment variables and use the target location where SDK was installed, type:

```
$ export ZEPHYR_GCC_VARIANT=zephyr
$ export ZEPHYR_SDK_INSTALL_DIR=<sdk installation directory>
```

To use the same toolchain in new sessions in the future you can set the variables in the file `$HOME/.zephyrrc`, for example:

```
$ cat <<EOF > ~/.zephyrrc
export ZEPHYR_GCC_VARIANT=zephyr
export ZEPHYR_SDK_INSTALL_DIR=/opt/zephyr-sdk
EOF
```

Development Environment Setup on Mac OS

This section describes how to set up a Mac OS development system.

After completing these steps, you will be able to compile and run your Zephyr applications on the following Mac OS version:

- Mac OS X 10.11 (El Capitan)
- macOS Sierra 10.12

Developing for Zephyr on macOS generally requires you to build the toolchain yourself. However, if there is already an macOS toolchain for your target architecture you can use it directly.

Using a 3rd Party toolchain

If a toolchain is available for the architecture you plan to build for, then you can use it as explained in: *Using Custom and 3rd Party Cross Compilers*.

An example of an available 3rd party toolchain is GCC ARM Embedded for the Cortex-M family of cores.

Installing Requirements and Dependencies

To install the software components required to build the Zephyr kernel on a Mac, you will need to build a cross compiler for the target devices you wish to build for and install tools that the build system requires.

注解: Minor version updates of the listed required packages might also work.

Before proceeding with the build, ensure your OS is up to date.

First, install the **`Homebrew`** (The missing package manager for macOS). Homebrew is a free and open-source software package management system that simplifies the installation of software on Apple's macOS operating system.

To install **`Homebrew`**, visit the Homebrew site and follow the installation instructions on the site.

To complete the Homebrew installation, you might be prompted to install some missing dependency. If so, follow please follow the instructions provided.

After Homebrew was successfully installed, install the following tools using the brew command line.

Install tools to build Zephyr binaries:

```
$ brew install dfu-util doxygen qemu dtc python3
$ curl -O 'https://bootstrap.pypa.io/get-pip.py'
$ ./get-pip.py
$ rm get-pip.py
$ pip3 install --user -r scripts/requirements.txt
```

Install tools needed for building the toolchain (if needed):

```
$ brew install gettext help2man mpfr gmp coreutils wget
$ brew tap homebrew/dupes
$ brew install grep --with-default-names
```

To build the toolchain, you will need the latest version of crosstool-ng (1.23). This version was not available via brew when writing this documentation, you can however try and see if you get 1.23 installed:

```
$ brew install crosstool-ng
```

Alternatively you can install the latest version of **`crosstool-ng`** from source. Download the latest version from the crosstool-ng site. The latest version usually supports the latest released compilers.

```
$ wget http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-1.23.0.tar.bz2
$ tar xvf crosstool-ng-1.23.0.tar.bz2
$ cd crosstool-ng-1.23.0/
$ ./configure
$ make
$ make install
```

Setting Up the Toolchain

Creating a Case-sensitive File System

Building the compiler requires a case-sensitive file system. Therefore, use **`diskutil`** to create an 8 GB blank sparse image making sure you select case-sensitive file system (OS X Extended (Case-sensitive, Journaled) and mount it.

Alternatively you can use the script below to create the image:

```
#!/bin/bash
ImageName=CrossToolNG
ImageNameExt=${ImageName}.sparseimage
diskutil umount force /Volumes/${ImageName} && true
rm -f ${ImageNameExt} && true
hdiutil create ${ImageName} -volname ${ImageName} -type SPARSE -size 8g -fs HFSX
hdiutil mount ${ImageNameExt}
cd /Volumes/$ImageName
```

When mounted, the file system of the image will be available under `/Volumes`. Change to the mounted directory:

```
$ cd /Volumes/CrossToolNG
$ mkdir build
$ cd build
```

Setting the Toolchain Options

In the Zephyr kernel source tree we provide two configurations for both ARM and X86 that can be used to pre-select the options needed for building the toolchain. The configuration files can be found in `$ZEPHYR_BASE/scripts/cross_compiler/`.

Currently the following configurations are provided:

- i586.config: for standard ABI, for example for Galileo and qemu_x86
- iamcu.config: for IAMCU ABI, for example for the Arduino 101
- nios2.config: for Nios II boards

```
$ cp ${ZEPHYR_BASE}/scripts/cross_compiler/i586.config .config
```

You can create a toolchain configuration or customize an existing configuration yourself using the configuration menus:

```
$ export CT_PREFIX=/Volumes/CrossToolNG
$ ct-ng menuconfig
```

Verifying the Configuration of the Toolchain

Before building the toolchain it is advisable to perform a quick verification of the configuration set for the toolchain.

1. Open the generated `.config` file.
2. Verify the following lines are present, assuming the sparse image was mounted under `/Volumes/CrossToolNG`:

```
...
CT_LOCAL_TARBALLS_DIR="/Volumes/CrossToolNG/src"
# CT_SAVE_TARBALLS is not set
CT_WORK_DIR="${CT_TOP_DIR}/.build"
CT_PREFIX_DIR="/Volumes/CrossToolNG/x-tools/${CT_TARGET}"
CT_INSTALL_DIR="${CT_PREFIX_DIR}"
# Following options prevent link errors
CT_WANTS_STATIC_LINK=n
CT_CC_STATIC_LIBSTDCXX=n
...
```

Building the Toolchain

To build the toolchain, enter:

```
$ ct-ng build
```

The above process takes a while. When finished, the toolchain will be available under `/Volumes/CrossToolNG/x-tools`.

Repeat the step for all architectures you want to support in your environment.

To use the toolchain with Zephyr, export the following environment variables and use the target location where the toolchain was installed, type:

```
$ export ZEPHYR_GCC_VARIANT=xtools
$ export XTOOLS_TOOLCHAIN_PATH=/Volumes/CrossToolNG/x-tools
```

To use the same toolchain in new sessions in the future you can set the variables in the file `$HOME/.zephyrrc`, for example:

```
$ cat <<EOF > ~/.zephyrrc
export XTOOLS_TOOLCHAIN_PATH=/Volumes/CrossToolNG/x-tools
export ZEPHYR_GCC_VARIANT=xtools
EOF
```

Development Environment Setup on Windows

This section describes how to configure your development environment and to build Zephyr applications in a Microsoft Windows environment.

This guide was tested by building the Zephyr hello_world sample application on Windows versions 7, 8.1, and 10.

Update Your Operating System

Before proceeding with the build, ensure that you are running your Windows system with the latest updates installed.

Installing Requirements and Dependencies

Using MSYS2

The Zephyr development environment on Windows relies on MSYS2, a modern UNIX environment for Windows. Follow the steps below to set it up:

1. Download and install **MSYS2**. Download the appropriate (32 or 64-bit) MSYS2 installer from the MSYS2 website and execute it. On the final installation screen, check the "Run MSYS2 now." box to start up an MSYS2 shell when installation is complete. Follow the rest of the installation instructions on the MSYS2 website to update the package database and core system packages. You may be advised to "terminate MSYS2 without returning to shell and check for updates again". If so, simply close the `MSYS2 MSYS Shell` desktop app and run it again to complete the update.)

2. Launch the `MSYS2 MSYS Shell` desktop app from your start menu (if it's not still open).

   注解: Make sure you start `MSYS2 MSYS Shell`, not `MSYS2 MinGW Shell`.

   注解: There are multiple `export` statements in this tutorial. You can avoid typing them every time by placing them at the bottom of your `~/.bash_profile` file.

3. If you're behind a corporate firewall, you'll likely need to specify a proxy to get access to internet resources:

   ```
   $ export http_proxy=http://proxy.mycompany.com:123
   $ export https_proxy=$http_proxy
   ```

4. Install the dependencies required to build Zephyr:

   ```
   $ pacman -S git make gcc dtc diffutils ncurses-devel python3
   ```

5. Install pip and the required Python modules:

```
$ curl -O 'https://bootstrap.pypa.io/get-pip.py'
$ ./get-pip.py
$ rm get-pip.py
$ pip install --user -r scripts/requirements.txt
```

6. The build system should now be ready to work with any toolchain installed in your system. In the next step you'll find instructions for installing toolchains for building both x86 and ARM applications.
7. Install cross compiler toolchain:
   - For x86, install the 2017 Windows host ISSM toolchain from the Intel Developer Zone: ISSM Toolchain. Use your web browser to download the toolchain's `tar.gz` file.

     You'll need the tar application to unpack this file. In an `MSYS2 MSYS` console, install `tar` and use it to extract the toolchain archive:

     ```
     $ pacman -S tar
     $ tar -zxvf /c/Users/myusername/Downloads/issm-toolchain-windows-2017-01-15.
     ↪tar.gz -C /c
     ```

     substituting the .tar.gz path name with the one you downloaded.

     注解: The ISSM toolset only supports development for Intel® Quark™ Microcontrollers, for example, the Arduino 101 board. (Check out the "Zephyr Development Environment Setup" in this Getting Started on Arduino 101 with ISSM document.) Additional setup is required to use the ISSM GUI for development.

   - For ARM, install GNU ARM Embedded from the ARM developer website: GNU ARM Embedded (install to `c:\gccarmemb`).
8. From within the MSYS2 MSYS Shell, clone a copy of the Zephyr source into your home directory using Git:

   ```
   $ cd ~
   $ git clone https://github.com/zephyrproject-rtos/zephyr.git
   ```

9. Also within the MSYS console, set up environment variables for the installed tools and for the Zephyr environment (using the provided shell script):

   For x86:

   ```
   $ export ZEPHYR_GCC_VARIANT=issm
   $ export ISSM_INSTALLATION_PATH=/c/issm0-toolchain-windows-2017-01-25
   ```

   Use the path where you extracted the ISSM toolchain.

   For ARM:

   ```
   $ export ZEPHYR_GCC_VARIANT=gccarmemb
   $ export GCCARMEMB_TOOLCHAIN_PATH=/c/gccarmemb
   ```

   And for either, run the provided script to set up zephyr project specific variables:

   ```
   $ unset ZEPHYR_SDK_INSTALL_DIR
   $ source ~/zephyr/zephyr-env.sh
   ```

10. Finally, you can try building the hello_world sample to check things out.

    To build for the Intel® Quark™ (x86-based) Arduino 101:

```
$ cd $ZEPHYR_BASE/samples/hello_world
$ make BOARD=arduino_101
```

To build for the ARM-based Nordic nRF52 Development Kit:

```
$ cd $ZEPHYR_BASE/samples/hello_world
$ make BOARD=nrf52_pca10040
```

This should check that all the tools and toolchain are set up correctly for your own Zephyr development.

Using Windows 10 WSL (Windows Subsystem for Linux)

If you are running a recent version of Windows 10 you can make use of the built-in functionality to natively run Ubuntu binaries directly on a standard command-prompt. This allows you to install the standard Zephyr SDK and build for all supported architectures without the need for a Virtual Machine.

1. Install Windows Subsystem for Linux (WSL) following the instructions on the official Microsoft website: WSL Installation

   注解： For the Zephyr SDK to function properly you will need Windows 10 build 15002 or greater. You can check which Windows 10 build you are running in the "About your PC" section of the System Settings. If you are running an older Windows 10 build you might need to install the Creator's Update.

2. Follow the instructions for Ubuntu detailed in the Zephyr Linux Getting Started Guide which can be found here: *Development Environment Setup on Linux*

Checking Out the Source Code Anonymously

The code is hosted in a GitHub repo that supports anonymous cloning via git.

To clone the repository anonymously, enter:

```
$ git clone https://github.com/zephyrproject-rtos/zephyr.git
```

You have successfully checked out a copy of the source code to your local machine.

Building and Running an Application

Using the 'Hello World' sample application as a base model, the following section will describe the pieces necessary for creating a Zephyr application.

The processes to build and run a Zephyr application are the same across operating systems. Nevertheless, the commands needed do differ from one OS to the next. The following sections contain the commands used in a Linux development environment. If you are using Mac OS please use the appropriate commands for your OS.

Building a Sample Application

To build an example application follow these steps:

1. Make sure your environment is setup by exporting the following environment variables. When using the Zephyr SDK on Linux for example, type:

```
$ export ZEPHYR_GCC_VARIANT=zephyr
$ export ZEPHYR_SDK_INSTALL_DIR=<sdk installation directory>
```

2. Navigate to the main project directory:

```
$ cd zephyr-project
```

3. Source the project environment file to set the project environment variables:

```
$ source zephyr-env.sh
```

4. Build the hello_world example project, enter:

```
$ cd $ZEPHYR_BASE/samples/hello_world
$ make
```

The above invocation of make will build the hello_world sample application using the default settings defined in the application's Makefile. You can build for a different board by defining the variable BOARD with one of the supported boards, for example:

```
$ make BOARD=arduino_101
```

For further information on the supported boards go see here. Alternatively, run the following command to obtain a list of the supported boards:

```
$ make help
```

Sample projects for different features of the project are available at at `$ZEPHYR_BASE/samples`. After building an application successfully, the results can be found in the `outdir` sub-directory under the application root directory, in a subdirectory that matches the BOARD string.

The ELF binaries generated by the build system are named by default `zephyr.elf`. This value can be overridden in the application configuration The build system generates different names for different use cases depending on the hardware and boards used.

Using Custom and 3rd Party Cross Compilers

The Zephyr SDK is provided for convenience and ease of use. It provides cross-compilers for all ports supported by the Zephyr OS and does not require any extra flags when building applications or running tests.

If you have a custom cross-compiler or if you wish to use a vendor provided SDK, follow the steps below to build with any custom or 3rd party cross-compilers:

1. To avoid any conflicts with the Zephyr SDK, enter the following commands.

```
$ unset ZEPHYR_GCC_VARIANT
$ unset ZEPHYR_SDK_INSTALL_DIR
```

2. We will use the GCC ARM Embedded compiler for this example, download the package suitable for your operating system from the GCC ARM Embedded website and extract it on your file system. This example assumes the compiler was extracted to: `~/gcc-arm-none-eabi-5_3-2016q1/`.

3. Navigate to the main project directory:

```
$ cd zephyr-project
```

4. Source the project environment file to set the project environment variables:

```
$ source zephyr-env.sh
```

5. Build the example hello_world project and make sure you supply the CROSS_COMPILE on the command line, enter:

```
$ export GCCARMEMB_TOOLCHAIN_PATH="~/gcc-arm-none-eabi-5_3-2016q1/"
$ export ZEPHYR_GCC_VARIANT=gccarmemb
$ cd $ZEPHYR_BASE/samples/hello_world
$ make CROSS_COMPILE=~/gcc-arm-none-eabi-5_3-2016q1/bin/arm-none-eabi-␣
↪BOARD=arduino_due
```

The above will build the sample using the toolchain downloaded from GCC ARM Embedded.

Alternatively, you can use the existing support for GCC ARM Embedded:

```
$ export GCCARMEMB_TOOLCHAIN_PATH="~/gcc-arm-none-eabi-5_3-2016q1/"
$ export ZEPHYR_GCC_VARIANT=gccarmemb
$ cd zephyr-project
$ source zephyr-env.sh
$ cd $ZEPHYR_BASE/samples/hello_world
$ make BOARD=arduino_due
```

Running a Sample Application in QEMU

To perform rapid testing of an application in the development environment you can use the QEMU emulation board configuration available for both X86 and ARM Cortex-M3 architectures. This can be easily accomplished by calling a special target when building an application that invokes QEMU once the build process is completed.

To run an application using the x86 emulation board configuration (qemu_x86), type:

```
$ make BOARD=qemu_x86 run
```

To run an application using the ARM qemu_cortex_m3 board configuration, type:

```
$ make BOARD=qemu_cortex_m3 run
```

QEMU is not supported on all boards and SoCs. When developing for a specific hardware target you should always test on the actual hardware and should not rely on testing in the QEMU emulation environment only.

Contributing to the Zephyr Project

As an open-source project, we welcome and encourage the community to submit patches for code, documentation, tests, and more, directly to the project.

Contribution Guidelines

As an open-source project, we welcome and encourage the community to submit patches directly to the project. In our collaborative open source environment, standards and methods for submitting changes help reduce the chaos that can result from an active development community.

This document explains how to participate in project conversations, log bugs and enhancement requests, and submit patches to the project so your patch will be accepted quickly in the codebase.

Licensing

Licensing is very important to open source projects. It helps ensure the software continues to be available under the terms that the author desired.

Zephyr uses the Apache 2.0 license (as found in the LICENSE file in the project's GitHub repo) to strike a balance between open contribution and allowing you to use the software however you would like to. There are some imported or reused components of the Zephyr project that use other licensing, as described in Zephyr Licensing.

The license tells you what rights you have as a developer, provided by the copyright holder. It is important that the contributor fully understands the licensing rights and agrees to them. Sometimes the copyright holder isn't the contributor, such as when the contributor is doing work on behalf of a company.

Developer Certification of Origin (DCO)

To make a good faith effort to ensure licensing criteria are met, the Zephyr project requires the Developer Certificate of Origin (DCO) process to be followed.

The DCO is an attestation attached to every contribution made by every developer. In the commit message of the contribution, (described more fully later in this document), the developer simply adds a `Signed-off-by` statement and thereby agrees to the DCO.

When a developer submits a patch, it is a commitment that the contributor has the right to submit the patch per the license. The DCO agreement is shown below and at http://developercertificate.org/.

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the
    best of my knowledge, is covered under an appropriate open
    source license and I have the right under that license to
    submit that work with modifications, whether created in whole
    or in part by me, under the same open source license (unless
    I am permitted to submit under a different license), as
    Indicated in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including
    all personal information I submit with it, including my
    sign-off) is maintained indefinitely and may be redistributed
    consistent with this project or the open source license(s)
    involved.
```

DCO Sign-Off Methods

The DCO requires a sign-off message in the following format appear on each commit in the pull request:

```
Signed-off-by: Zephyrus Zephyr <zephyrus@zephyrproject.org>
```

The DCO text can either be manually added to your commit body, or you can add either `-s` or `--signoff` to your usual Git commit commands. If you forget to add the sign-off you can also amend a previous commit with the sign-off by running `git commit --amend -s`. If you've pushed your changes to GitHub already you'll need to force push your branch after this with `git push -f`.

Prerequisites

As a contributor, you'll want to be familiar with the Zephyr project, how to configure, install, and use it as explained in the Zephyr Project website and how to set up your development environment as introduced in the Zephyr Getting Started Guide.

The examples below use a Linux host environment for Zephyr development. You should be familiar with common developer tools such as Git and Make, and platforms such as GitHub.

If you haven't already done so, you'll need to create a (free) GitHub account on http://github.com and have Git tools available on your development system.

Repository layout

To clone the main Zephyr Project repository use:

```
$ git clone https://github.com/zephyrproject-rtos/zephyr
```

The Zephyr project directory structure is described in Source Tree Structure documentation. In addition to the Zephyr kernel itself, you'll also find the sources for technical documentation, sample code, supported board configurations, and a collection of subsystem tests. All of these are available for developers to contribute to and enhance.

Pull Requests and Issues

Before starting on a patch, first check in our Jira Zephyr Project Issues system to see what's been reported on the issue you'd like to address. Have a conversation on the Zephyr-devel mailing list (or the #zephyrproject IRC channel on freenode.net) to see what others think of your issue (and proposed solution). You may find others that have encountered the issue you're finding, or that have similar ideas for changes or additions. Send a message to the Zephyr-devel mailing list to introduce and discuss your idea with the development community.

It's always a good practice to search for existing or related issues before submitting your own. When you submit an issue (bug or feature request), the triage team will review and comment on the submission, typically within a few business days.

You can find all open pull requests on GitHub and open Zephyr Project Issues in Jira.

Development Tools and Git Setup

Signed-off-by

The name in the commit message `Signed-off-by:` line and your email must match the change authorship information. Make sure your *.git/config* is set up correctly:

```
$ git config --global user.name "David Developer"
$ git config --global user.email "david.developer@company.com"
```

gitlint

When you submit a pull request to the project, a series of checks are performed to verify your commit messages meet the requirements. The same step done during the CI process can be performed locally using the the *gitlint* command.

Install gitlint and run it locally in your tree and branch where your patches have been committed:

```
$ sudo pip3 install gitlint
$ gitlint
```

Note, gitlint only checks HEAD (the most recent commit), so you should run it after each commit, or use the `--commits` option to specify a commit range covering all the development patches to be submitted.

sanitycheck

To verify that your changes did not break any tests or samples, please run the `sanitycheck` script locally before submitting your pull request to GitHub. To run the same tests the CI system runs, follow these steps from within your local Zephyr source working directory:

```
$ source zephyr-env.sh
$ make host-tools
$ export PREBUILT_HOST_TOOLS=${ZEPHYR_BASE}/bin
$ export USE_CCACHE=1
$ ./scripts/sanitycheck
```

The above will execute the basic sanitycheck script, which will run various kernel tests using the QEMU emulator. It will also do some build tests on various samples with advanced features that can't run in QEMU.

We highly recommend you run these tests locally to avoid any CI failures. Using CCACHE and pre-built host tools is optional, however it speeds up the execution time considerably.

Coding Style

Use these coding guidelines to ensure that your development complies with the project's style and naming conventions.

In general, follow the Linux kernel coding style, with the following exceptions:

- Add braces to every `if` and `else` body, even for single-line code blocks. Use the `--ignore BRACES` flag to make *checkpatch* stop complaining.
- Use spaces instead of tabs to align comments after declarations, as needed.
- Use C89-style single line comments, `/* */`. The C99-style single line comment, `//`, is not allowed.
- Use `/** */` for doxygen comments that need to appear in the documentation.

The Linux kernel GPL-licensed tool `checkpatch` is used to check coding style conformity. Checkpatch is available in the scripts directory. To invoke it when committing code, edit your *.git/hooks/pre-commit* file to contain:

```
#!/bin/sh
set -e exec
exec git diff --cached | ${ZEPHYR_BASE}/scripts/checkpatch.pl - || true
```

Contribution Workflow

One general practice we encourage, is to make small, controlled changes. This practice simplifies review, makes merging and rebasing easier, and keeps the change history clear and clean.

When contributing to the Zephyr Project, it is also important you provide as much information as you can about your change, update appropriate documentation, and test your changes thoroughly before submitting.

The general GitHub workflow used by Zephyr developers uses a combination of command line Git commands and browser interaction with GitHub. As it is with Git, there are multiple ways of getting a task done. We'll describe a typical workflow here:

1. Create a Fork of Zephyr to your personal account on GitHub. (Click on the fork button in the top right corner of the Zephyr project repo page in GitHub.)
2. On your development computer, clone the fork you just made:

   ```
   $ git clone https://github.com/<your github id>/zephyr
   ```

   This would be a good time to let Git know about the upstream repo too:

   ```
   $ git remote add upstream https://github.com/zephyrproject-rtos/zephyr.git
   ```

   and verify the remote repos:

   ```
   $ git remote -v
   ```

3. Create a topic branch (off of master) for your work (if you're addressing Jira issue, we suggest including the Jira issue number in the branch name):

   ```
   $ git checkout master
   $ git checkout -b fix_comment_typo
   ```

   Some Zephyr subsystems do development work on a separate branch from master so you may need to indicate this in your checkout:

   ```
   $ git checkout -b fix_out_of_date_patch origin/net
   ```

4. Make changes, test locally, change, test, test again, … (Check out the prior chapter on *sanitycheck* as well).
5. When things look good, start the pull request process by adding your changed files:

   ```
   $ git add [file(s) that changed, add -p if you want to be more specific]
   ```

   You can see files that are not yet staged using:

   ```
   $ git status
   ```

6. Verify changes to be committed look as you expected:

   ```
   $ git diff --cached
   ```

7. Commit your changes to your local repo:

   ```
   $ git commit -s
   ```

   The `-s` option automatically adds your `Signed-off-by:` to your commit message. Your commit will be rejected without this line that indicates your agreement with the *DCO*. See the *Commit Guidelines* section for specific guidelines for writing your commit messages.

8. Push your topic branch with your changes to your fork in your personal GitHub account:

   ```
   $ git push origin fix_comment_typo
   ```

9. In your web browser, go to your forked repo and click on the `Compare & pull request` button for the branch you just worked on and you want to open a pull request with.
10. Review the pull request changes, and verify that you are opening a pull request for the appropriate branch. The title and message from your commit message should appear as well.
11. If you're working on a subsystem branch that's not `master`, you may need to change the intended branch for the pull request here, for example, by changing the base branch from `master` to `net`.
12. GitHub will assign one or more suggested reviewers (based on the CODEOWNERS file in the repo). If you are a project member, you can select additional reviewers now too.
13. Click on the submit button and your pull request is sent and awaits review. Email will be sent as review comments are made, or you can check on your pull request at https://github.com/zephyrproject-rtos/zephyr/pulls.
14. While you're waiting for your pull request to be accepted and merged, you can create another branch to work on another issue. (Be sure to make your new branch off of master and not the previous branch.):

    ```
    $ git checkout master
    $ git checkout -b fix_another_issue
    ```

    and use the same process described above to work on this new topic branch.

15. If reviewers do request changes to your patch, you can interactively rebase commit(s) to fix review issues. In your development repo:

    ```
    $ git fetch --all
    $ git rebase --ignore-whitespace upstream/master
    ```

    The `--ignore-whitespace` option stops `git apply` (called by rebase) from changing any whitespace. Continuing:

    ```
    $ git rebase -i <offending-commit-id>^
    ```

    In the interactive rebase editor, replace `pick` with `edit` to select a specific commit (if there's more than one in your pull request), or remove the line to delete a commit entirely. Then edit files to fix the issues in the review.

    As before, inspect and test your changes. When ready, continue the patch submission:

    ```
    $ git add [file(s)]
    $ git rebase --continue
    ```

    Update commit comment if needed, and continue:

    ```
    $ git push --force origin fix_comment_typo
    ```

    By force pushing your update, your original pull request will be updated with your changes so you won't need to resubmit the pull request.

Commit Guidelines

Changes are submitted as Git commits. Each commit message must contain:

- A short and descriptive subject line that is less than 72 characters, followed by a blank line. The subject line must include a prefix that identifies the subsystem being changed, followed by a colon, and a short title, for example: `doc: update wiki references to new site`. (If you're updating an existing file, you can use `git log <filename>` to see what developers used as the prefix for previous patches of this file.)

- A change description with your logic or reasoning for the changes, followed by a blank line.
- A Signed-off-by line, `Signed-off-by: <name> <email>` typically added automatically by using `git commit -s`
- If the change address a Jira issue, include a line of the form:

  ```
  Jira: ZEP-xxx
  ```

All changes and topics sent to GitHub must be well-formed, as described above.

Commit Message Body

When editing the commit message, please briefly explain what your change does and why it's needed. A change summary of `"Fixes stuff"` will be rejected. An empty change summary is only acceptable for trivial changes fully described by the commit title (e.g., `doc: fix misspellings in CONTRIBUTING doc`)

The description body of the commit message must include:

- **what** the change does,
- **why** you chose that approach,
- **what** assumptions were made, and
- **how** you know it works – for example, which tests you ran.

For examples of accepted commit messages, you can refer to the Zephyr GitHub changelog.

Other Commit Expectations

- Commits must build cleanly when applied on top of each other, thus avoiding breaking bisectability.
- Commits must pass the *scripts/checkpatch.pl* requirements.
- Each commit must address a single identifiable issue and must be logically self-contained. Unrelated changes should be submitted as separate commits.
- You may submit pull request RFCs (requests for comments) to send work proposals, progress snapshots of your work, or to get early feedback on features or changes that will affect multiple areas in the code base.

Identifying Contribution Origin

When adding a new file to the tree, it is important to detail the source of origin on the file, provide attributions, and detail the intended usage. In cases where the file is an original to Zephyr, the commit message should include the following ("Original" is the assumption if no Origin tag is present):

```
Origin: Original
```

In cases where the file is imported from an external project, the commit message shall contain details regarding the original project, the location of the project, the SHA-id of the origin commit for the file, the intended purpose, and if the file will be maintained by the Zephyr project, (whether or not the Zephyr project will contain a localized branch or if it is a downstream copy).

For example, a copy of a locally maintained import:

```
Origin: Contiki OS
License: BSD 3-Clause
URL: http://www.contiki-os.org/
commit: 853207acfdc6549b10eb3e44504b1a75ae1ad63a
Purpose: Introduction of networking stack.
Maintained-by: Zephyr
```

For example, a copy of an externally maintained import:

```
Origin: Tiny Crypt
License: BSD 3-Clause
URL: https://github.com/01org/tinycrypt
commit: 08ded7f21529c39e5133688ffb93a9d0c94e5c6e
Purpose: Introduction of TinyCrypt
Maintained-by: External
```

Zephyr Kernel Primer

This document provides a general introduction of the Zephyr kernel's key capabilities and services. Additional details can be found by consulting the *API Documentation* and *Application Development Primer* documentation, and by examining the code in the Zephyr source tree.

Overview

The Zephyr kernel lies at the heart of every Zephyr application. It provides a low footprint, high performance, multi-threaded execution environment with a rich set of available features. The rest of the Zephyr ecosystem, including device drivers, networking stack, and application-specific code, uses the kernel's features to create a complete application.

The configurable nature of the kernel allows you to incorporate only those features needed by your application, making it ideal for systems with limited amounts of memory (as little as 2 KB!) or with simple multi-threading requirements (such as a set of interrupt handlers and a single background task). Examples of such systems include: embedded sensor hubs, environmental sensors, simple LED wearable, and store inventory tags.

Applications requiring more memory (50 to 900 KB), multiple communication devices (like WiFi and Bluetooth Low Energy), and complex multi-threading, can also be developed using the Zephyr kernel. Examples of such systems include: fitness wearables, smart watches, and IoT wireless gateways.

Source Tree Structure

Understanding the Zephyr source tree can be helpful in locating the code associated with a particular Zephyr feature.

The Zephyr source tree provides the following top-level directories, each of which may have one or more additional levels of subdirectories which are not described here.

arch Architecture-specific kernel and system-on-chip (SoC) code. Each supported architecture (for example, x86 and ARM) has its own subdirectory, which contains additional subdirectories for the following areas:

- architecture-specific kernel source files
- architecture-specific kernel include files for private APIs
- SoC-specific code

boards Board related code and configuration files.

`doc` Zephyr technical documentation source files and tools used to generate the http://zephyrproject.org/doc web content.

`drivers` Device driver code.

`dts` Device tree source (.dts) files used to describe non-discoverable board-specific hardware details previously hard coded in the OS source code.

`ext` Externally created code that has been integrated into Zephyr from other sources, such as hardware interface code supplied by manufacturers and cryptographic library code.

`include` Include files for all public APIs, except those defined under `lib`.

`kernel` Architecture-independent kernel code.

`lib` Library code, including the minimal standard C library.

`misc` Miscellaneous code that doesn't belong to any of the other top-level directories.

`samples` Sample applications that demonstrate the use of Zephyr features.

`scripts` Various programs and other files used to build and test Zephyr applications.

`subsys` Subsystems of Zephyr, including:

- USB device stack code.
- Networking code, including the Bluetooth stack and networking stacks.
- File system code.
- Bluetooth host and controller

`tests` Test code and benchmarks for Zephyr features.

Changes from Version 1 Kernel

The current Zephyr kernel incorporates numerous changes from kernels found in the 1.5 and earlier releases, and improves ease of use for developers.

Some of the benefits of these changes are:

- elimination of separate microkernel and nanokernel build types,
- elimination of the MDEF in microkernel-based applications,
- simplifying and streamlining the kernel API,
- easing restrictions on the use of kernel objects,
- reducing memory footprint by merging duplicated services, and
- improving performance by reducing context switching.

The most significant changes are discussed below.

Application Design

The earlier microkernel and nanokernel portions of Zephyr have been merged into a single entity, which is simply referred to as "the kernel". Consequently, there is now only a single way to design and build Zephyr applications.

The task and fiber context types have been merged into a single type, known as a "thread". Setting a thread's priority to a negative priority makes it a "cooperative thread", which operates in a fiber-like manner; setting it to a non-negative priority makes it a "preemptive thread", which operates in a task-like manner.

Kernel objects can now be used by both task-like threads and fiber-like threads. (The previous kernel did not permit fibers to use microkernel objects, and could result in undesirable busy-waiting when nanokernel objects were used by tasks.)

Kernel objects now typically allow multiple threads to wait on a given object. (The previous kernel restricted waiting on certain types of kernel object to a single thread.)

Kernel object APIs now always execute in the context of the invoking thread. (The previous kernel required microkernel object APIs to context switch the thread to the microkernel server fiber, followed by another context switch back to the invoking thread.)

The MDEF has been eliminated. Consequently, all kernel objects are now defined directly in code.

Kernel APIs

Most kernel APIs have been renamed or have had changes to their arguments (or both) to make them more intuitive, and to improve consistency. The **k_** and **K_** prefixes are now used by most kernel APIs.

A previous kernel operation that can be invoked from a task, a fiber, or an ISR using distinct APIs is now invoked from a thread or an ISR using a single common API.

Many kernel APIs now return 0 to indicate success and a non-zero error code to indicate the reason for failure. (The previous kernel supported only two error codes, rather than an unlimited number of them.)

Threads

A task-like thread can now make itself temporarily non-preemptible by locking the kernel's scheduler (rather than by locking interrupts).

It is now possible to pass up to 3 arguments to a thread's entry point. (The previous kernel allowed 2 arguments to be passed to a fiber and allowed no arguments to be passed to a task.)

It is now possible to delay the start of a statically-defined threads. (The previous kernel only permitted delaying of fibers spawned at run time.)

A task can no longer specify an "task abort handler" function that is invoked automatically when the task terminates or aborts.

An application can no longer use "task groups" to alter the operation of a set of related tasks by invoking a single kernel API. However, applications can provide their own APIs to achieve a similar effect.

The kernel now spawns both a "main thread" and an "idle thread" during startup. (The previous kernel spawned only a single thread.)

The kernel's main thread performs system initialization and then invokes `main()`. If no `main()` is defined by the application, the main thread terminates.

System initialization code can now perform blocking operations, during which time the kernel's idle thread executes.

Timing

Most kernel APIs now specify timeout intervals in milliseconds, rather than in system clock ticks. This change makes things more intuitive for most developers. However, the kernel still implements timeouts using the tick-based system clock.

The previous nanokernel timer and microkernel timer object types have been merged into a single type.

Memory Allocation

The microkernel memory map object has been renamed to "memory slab", to better reflect its management of equal-size memory blocks.

It is now possible to specify the alignment used by the memory blocks belonging to a memory slab or a memory pool.

It is now possible to define a memory pool directly in code.

It is now possible to allocate and free memory in a malloc()-like manner from a heap data pool.

Synchronization

The previous nanokernel semaphore and microkernel semaphore object types have been merged into a single type. The new type can now be used as a binary semaphore, as well as a counting semaphore.

An application can no longer use a "semaphore group" to allow a thread to wait on multiple semaphores simultaneously. Until the kernel incorporates a `select()` or `poll()` capability an application wishing to wait on multiple semaphores must either test them individually in a non-blocking manner or use an additional mechanism, such as an event object, to signal the application that one of the semaphores is available.

The previous microkernel event object type is renamed to "alert" and is now presented as a relative to Unix-style signaling. Due to improvements to the implementation of semaphores, alerts are now less efficient to use for basic synchronization than semaphores; consequently, alerts should now be reserved for scenarios requiring the use of a callback function.

Data Passing

The previous microkernel FIFO object type has been renamed to "message queue", to avoid confusion with the nanokernel FIFO object type.

It is now possible to specify the alignment used by the data items stored in a message queue (aka microkernel FIFO).

The previous microkernel mailbox object type no longer supports the explicit message priority concept. Messages are now implicitly ordered based on the priority of the sending thread.

The mailbox object type now supports sending asynchronous messages using a message buffer. (The previous kernel only supported asynchronous messages using a message block.)

It is now possible to specify the alignment used by a pipe object's buffer.

Threads

This section describes kernel services for creating, scheduling, and deleting independently executable threads of instructions.

Lifecycle

A *thread* is a kernel object that is used for application processing that is too lengthy or too complex to be performed by an ISR.

Concepts

Any number of threads can be defined by an application. Each thread is referenced by a *thread id* that is assigned when the thread is spawned.

A thread has the following key properties:

- A **stack area**, which is a region of memory used for the thread's stack. The **size** of the stack area can be tailored to conform to the actual needs of the thread's processing. Special macros exist to create and work with stack memory regions.
- A **thread control block** for private kernel bookkeeping of the thread's metadata. This is an instance of type `struct k_thread`.
- An **entry point function**, which is invoked when the thread is started. Up to 3 **argument values** can be passed to this function.
- A **scheduling priority**, which instructs the kernel's scheduler how to allocate CPU time to the thread. (See *Scheduling*.)
- A set of **thread options**, which allow the thread to receive special treatment by the kernel under specific circumstances. (See *Thread Options*.)
- A **start delay**, which specifies how long the kernel should wait before starting the thread.

Thread Creation

A thread must be created before it can be used. The kernel initializes the thread control block as well as one end of the stack portion. The remainder of the thread's stack is typically left uninitialized.

Specifying a start delay of `K_NO_WAIT` instructs the kernel to start thread execution immediately. Alternatively, the kernel can be instructed to delay execution of the thread by specifying a timeout value – for example, to allow device hardware used by the thread to become available.

The kernel allows a delayed start to be canceled before the thread begins executing. A cancellation request has no effect if the thread has already started. A thread whose delayed start was successfully canceled must be re-spawned before it can be used.

Thread Termination

Once a thread is started it typically executes forever. However, a thread may synchronously end its execution by returning from its entry point function. This is known as **termination**.

A thread that terminates is responsible for releasing any shared resources it may own (such as mutexes and dynamically allocated memory) prior to returning, since the kernel does *not* reclaim them automatically.

注解: The kernel does not currently make any claims regarding an application's ability to respawn a thread that terminates.

Thread Aborting

A thread may asynchronously end its execution by **aborting**. The kernel automatically aborts a thread if the thread triggers a fatal error condition, such as dereferencing a null pointer.

A thread can also be aborted by another thread (or by itself) by calling `k_thread_abort()`. However, it is typically preferable to signal a thread to terminate itself gracefully, rather than aborting it.

As with thread termination, the kernel does not reclaim shared resources owned by an aborted thread.

注解: The kernel does not currently make any claims regarding an application's ability to respawn a thread that aborts.

Thread Suspension

A thread can be prevented from executing for an indefinite period of time if it becomes **suspended**. The function `k_thread_suspend()` can be used to suspend any thread, including the calling thread. Suspending a thread that is already suspended has no additional effect.

Once suspended, a thread cannot be scheduled until another thread calls `k_thread_resume()` to remove the suspension.

注解: A thread can prevent itself from executing for a specified period of time using `k_sleep()`. However, this is different from suspending a thread since a sleeping thread becomes executable automatically when the time limit is reached.

Thread Options

The kernel supports a small set of *thread options* that allow a thread to receive special treatment under specific circumstances. The set of options associated with a thread are specified when the thread is spawned.

A thread that does not require any thread option has an option value of zero. A thread that requires a thread option specifies it by name, using the `|` character as a separator if multiple options are needed (i.e. combine options using the bitwise OR operator).

The following thread options are supported.

K_ESSENTIAL This option tags the thread as an *essential thread*. This instructs the kernel to treat the termination or aborting of the thread as a fatal system error.

By default, the thread is not considered to be an essential thread.

K_FP_REGS and K_SSE_REGS These x86-specific options indicate that the thread uses the CPU's floating point registers and SSE registers, respectively. This instructs the kernel to take additional steps to save and restore the contents of these registers when scheduling the thread. (For more information see *Floating Point Services*.)

By default, the kernel does not attempt to save and restore the contents of these registers when scheduling the thread.

Implementation

Spawning a Thread

A thread is spawned by defining its stack area and its thread control block, and then calling `k_thread_create()`. The stack area must be defined using `K_THREAD_STACK_DEFINE` to ensure it is properly set up in memory.

The thread spawning function returns its thread id, which can be used to reference the thread.

The following code spawns a thread that starts immediately.

```
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

extern void my_entry_point(void *, void *, void *);

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);
struct k_thread my_thread_data;

k_tid_t my_tid = k_thread_create(&my_thread_data, my_stack_area,
                                 K_THREAD_STACK_SIZEOF(my_stack_area),
                                 my_entry_point,
                                 NULL, NULL, NULL,
                                 MY_PRIORITY, 0, K_NO_WAIT);
```

Alternatively, a thread can be spawned at compile time by calling `K_THREAD_DEFINE`. Observe that the macro defines the stack area, control block, and thread id variables automatically.

The following code has the same effect as the code segment above.

```
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

extern void my_entry_point(void *, void *, void *);

K_THREAD_DEFINE(my_tid, MY_STACK_SIZE,
                my_entry_point, NULL, NULL, NULL,
                MY_PRIORITY, 0, K_NO_WAIT);
```

Terminating a Thread

A thread terminates itself by returning from its entry point function.

The following code illustrates the ways a thread can terminate.

```
void my_entry_point(int unused1, int unused2, int unused3)
{
    while (1) {
        ...
        if (<some condition>) {
            return; /* thread terminates from mid-entry point function */
        }
        ...
    }

    /* thread terminates at end of entry point function */
}
```

Suggested Uses

Use threads to handle processing that cannot be handled in an ISR.

Use separate threads to handle logically distinct processing operations that can execute in parallel.

Configuration Options

Related configuration options:

- None.

APIs

The following thread APIs are provided by `kernel.h`:

- `K_THREAD_DEFINE`
- `k_thread_create()`
- `k_thread_cancel()`
- `k_thread_abort()`
- `k_thread_suspend()`
- `k_thread_resume()`
- `K_THREAD_STACK_DEFINE`
- `K_THREAD_STACK_ARRAY_DEFINE`
- `K_THREAD_STACK_MEMBER`
- `K_THREAD_STACK_SIZEOF`
- `K_THREAD_STACK_BUFFER`

Scheduling

The kernel's priority-based scheduler allows an application's threads to share the CPU.

Concepts

The scheduler determines which thread is allowed to execute at any point in time; this thread is known as the **current thread**.

Whenever the scheduler changes the identity of the current thread, or when execution of the current thread is supplanted by an ISR, the kernel first saves the current thread's CPU register values. These register values get restored when the thread later resumes execution.

Thread States

A thread that has no factors that prevent its execution is deemed to be **ready**, and is eligible to be selected as the current thread.

A thread that has one or more factors that prevent its execution is deemed to be **unready**, and cannot be selected as the current thread.

The following factors make a thread unready:

- The thread has not been started.
- The thread is waiting on for a kernel object to complete an operation. (For example, the thread is taking a semaphore that is unavailable.)
- The thread is waiting for a timeout to occur.
- The thread has been suspended.
- The thread has terminated or aborted.

Thread Priorities

A thread's priority is an integer value, and can be either negative or non-negative. Numerically lower priorities takes precedence over numerically higher values. For example, the scheduler gives thread A of priority 4 *higher* priority over thread B of priority 7; likewise thread C of priority -2 has higher priority than both thread A and thread B.

The scheduler distinguishes between two classes of threads, based on each thread's 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 unready.
- A *preemptible thread* has a non-negative priority value. Once it becomes the current thread, a preemptible thread may be supplanted 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 up or down after the thread has been started. Thus it possible for a preemptible thread to become a cooperative thread, and vice versa, by changing its priority.

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 of thread, resulting the following usable priority ranges:

- cooperative threads: (-`CONFIG_NUM_COOP_PRIORITIES`) to -1
- preemptive threads: 0 to (`CONFIG_NUM_PREEMPT_PRIORITIES` - 1)

For example, configuring 5 cooperative priorities and 10 preemptive priorities results in the ranges -5 to -1 and 0 to 9, respectively.

Scheduling Algorithm

The kernel's scheduler selects the highest priority ready thread to be the current thread. When multiple ready threads of the same priority exist, the scheduler chooses the one that has been waiting longest.

注解: Execution of ISRs takes precedence over thread execution, so the execution of the current thread may be supplanted by an ISR at any time unless interrupts have been masked. This applies to both cooperative threads and preemptive threads.

Cooperative Time Slicing

Once a cooperative thread becomes the current thread, it remains the current thread until it performs an action that makes it unready. Consequently, if a cooperative thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of higher priority and equal priority.

To overcome such problems, a cooperative thread can voluntarily relinquish the CPU from time to time to permit other threads to execute. A thread can relinquish the CPU in two ways:

- Calling `k_yield()` puts the thread at the back of the scheduler's prioritized list of ready threads, and then invokes the scheduler. All ready threads whose priority is higher or equal to that of the yielding thread are then allowed to execute before the yielding thread is rescheduled. If no such ready threads exist, the scheduler immediately reschedules the yielding thread without context switching.
- Calling `k_sleep()` makes the thread unready for a specified time period. Ready threads of *all* priorities are then allowed to execute; however, there is no guarantee that threads whose priority is lower than that of the sleeping thread will actually be scheduled before the sleeping thread becomes ready once again.

Preemptive Time Slicing

Once a preemptive thread becomes the current thread, it remains the current thread until a higher priority thread becomes ready, or until the thread performs an action that makes it unready. Consequently, if a preemptive thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of equal priority.

To overcome such problems, a preemptive thread can perform cooperative time slicing (as described above), or the scheduler's time slicing capability can be used to allow other threads of the same priority to execute.

The scheduler divides time into a series of **time slices**, where slices are measured in system clock ticks. The time slice size is configurable, but this size can be changed while the application is running.

At the end of every time slice, the scheduler checks to see if the current thread is preemptible and, if so, implicitly invokes `k_yield()` on behalf of the thread. This gives other ready threads of the same priority the opportunity to execute before the current thread is scheduled again. If no threads of equal priority are ready, the current thread remains the current thread.

Threads with a priority higher than specified limit are exempt from preemptive time slicing, and are never preempted by a thread of equal priority. This allows an application to use preemptive time slicing only when dealing with lower priority threads that are less time-sensitive.

注解: The kernel's time slicing algorithm does *not* ensure that a set of equal-priority threads receive an equitable amount of CPU time, since it does not measure the amount of time a thread actually gets to execute. For example, a thread may become the current thread just before the end of a time slice and then immediately have to yield the CPU. However, the algorithm *does* ensure that a thread never executes for longer than a single time slice without being required to yield.

Scheduler Locking

A preemptible thread that does not wish to be preempted while performing a critical operation can instruct the scheduler to temporarily treat it as a cooperative thread by calling `k_sched_lock()`. This prevents other threads from interfering while the critical operation is being performed.

Once the critical operation is complete the preemptible thread must call `k_sched_unlock()` to restore its normal, preemptible status.

If a thread calls `k_sched_lock()` and subsequently performs an action that makes it unready, the scheduler will switch the locking thread out and allow other threads to execute. When the locking thread again becomes the current thread, its non-preemptible status is maintained.

Thread Sleeping

A thread can call `k_sleep()` to delay its processing for a specified time period. During the time the thread is sleeping the CPU is relinquished to allow other ready threads to execute. Once the specified delay has elapsed the thread becomes ready and is eligible to be scheduled once again.

A sleeping thread can be woken up prematurely by another thread using `k_wakeup()`. This technique can sometimes be used to permit the secondary thread to signal the sleeping thread that something has occurred *without* requiring the threads to define a kernel synchronization object, such as a semaphore. Waking up a thread that is not sleeping is allowed, but has no effect.

Busy Waiting

A thread can call `k_busy_wait()` to perform a `busy wait` that delays its processing for a specified time period *without* relinquishing the CPU to another ready thread.

A busy wait is typically used instead of thread sleeping when the required delay is too short to warrant having the scheduler context switch from the current thread to another thread and then back again.

Suggested Uses

Use cooperative threads for device drivers and other performance-critical work.

Use cooperative threads to implement mutually exclusion without the need for a kernel object, such as a mutex.

Use preemptive threads to give priority to time-sensitive processing over less time-sensitive processing.

Configuration Options

Related configuration options:

- `CONFIG_NUM_COOP_PRIORITIES`
- `CONFIG_NUM_PREEMPT_PRIORITIES`
- `CONFIG_TIMESLICING`
- `CONFIG_TIMESLICE_SIZE`
- `CONFIG_TIMESLICE_PRIORITY`

APIs

The following thread scheduling-related APIs are provided by `kernel.h`:

- `k_current_get()`
- `k_sched_lock()`
- `k_sched_unlock()`
- `k_yield()`
- `k_sleep()`
- `k_wakeup()`
- `k_busy_wait()`
- `k_sched_time_slice_set()`

Custom Data

A thread's *custom data* is a 32-bit, thread-specific value that may be used by an application for any purpose.

Concepts

Every thread has a 32-bit custom data area. The custom data is accessible only by the thread itself, and may be used by the application for any purpose it chooses. The default custom data for a thread is zero.

注解: Custom data support is not available to ISRs because they operate within a single shared kernel interrupt handling context.

Implementation

Using Custom Data

By default, thread custom data support is disabled. The configuration option `CONFIG_THREAD_CUSTOM_DATA` can be used to enable support.

The `k_thread_custom_data_set()` and `k_thread_custom_data_get()` functions are used to write and read a thread's custom data, respectively. A thread can only access its own custom data, and not that of another thread.

The following code uses the custom data feature to record the number of times each thread calls a specific routine.

注解: Obviously, only a single routine can use this technique, since it monopolizes the use of the custom data feature.

```
int call_tracking_routine(void)
{
    u32_t call_count;

    if (k_is_in_isr()) {
        /* ignore any call made by an ISR */
    } else {
        call_count = (u32_t)k_thread_custom_data_get();
        call_count++;
        k_thread_custom_data_set((void *)call_count);
    }

    /* do rest of routine's processing */
    ...
}
```

Suggested Uses

Use thread custom data to allow a routine to access thread-specific information, by using the custom data as a pointer to a data structure owned by the thread.

Configuration Options

Related configuration options:

- `CONFIG_THREAD_CUSTOM_DATA`

APIs

The following thread custom data APIs are provided by `kernel.h`:

- `k_thread_custom_data_set()`
- `k_thread_custom_data_get()`

System Threads

A *system thread* is a thread that the kernel spawns automatically during system initialization.

Concepts

The kernel spawns the following system threads.

Main thread This thread performs kernel initialization, then calls the application's `main()` function (if one is defined).

By default, the main thread uses the highest configured preemptible thread priority (i.e. 0). If the kernel is not configured to support preemptible threads, the main thread uses the lowest configured cooperative thread priority (i.e. -1).

The main thread is an essential thread while it is performing kernel initialization or executing the application's `main()` function; this means a fatal system error is raised if the thread aborts. If `main()` is not defined, or if it executes and then does a normal return, the main thread terminates normally and no error is raised.

Idle thread This thread executes when there is no other work for the system to do. If possible, the idle thread activates the board's power management support to save power; otherwise, the idle thread simply performs a "do nothing" loop. The idle thread remains in existence as long as the system is running and never terminates.

The idle thread always uses the lowest configured thread priority. If this makes it a cooperative thread, the idle thread repeatedly yields the CPU to allow the application's other threads to run when they need to.

The idle thread is an essential thread, which means a fatal system error is raised if the thread aborts.

Additional system threads may also be spawned, depending on the kernel and board configuration options specified by the application. For example, enabling the system workqueue spawns a system thread that services the work items submitted to it. (See *Workqueue Threads*.)

Implementation

Writing a main() function

An application-supplied `main()` function begins executing once kernel initialization is complete. The kernel does not pass any arguments to the function.

The following code outlines a trivial `main()` function. The function used by a real application can be as complex as needed.

```
void main(void)
{
    /* initialize a semaphore */
    ...

    /* register an ISR that gives the semaphore */
    ...

    /* monitor the semaphore forever */
    while (1) {
        /* wait for the semaphore to be given by the ISR */
        ...
        /* do whatever processing is now needed */
        ...
    }
}
```

Suggested Uses

Use the main thread to perform thread-based processing in an application that only requires a single thread, rather than defining an additional application-specific thread.

Configuration Options

Related configuration options:

- `CONFIG_MAIN_THREAD_PRIORITY`
- `CONFIG_MAIN_STACK_SIZE`
- `CONFIG_IDLE_STACK_SIZE`

APIs

None.

Workqueue Threads

A *workqueue* is a kernel object that uses a dedicated thread to process work items in a first in, first out manner. Each work item is processed by calling the function specified by the work item. A workqueue is typically used by an ISR or a high-priority thread to offload non-urgent processing to a lower-priority thread so it does not impact time-sensitive processing.

Concepts

Any number of workqueues can be defined. Each workqueue is referenced by its memory address.

A workqueue has the following key properties:

- A **queue** of work items that have been added, but not yet processed.
- A **thread** that processes the work items in the queue. The priority of the thread is configurable, allowing it to be either cooperative or preemptive as required.

A workqueue must be initialized before it can be used. This sets its queue to empty and spawns the workqueue's thread.

Work Item Lifecycle

Any number of **work items** can be defined. Each work item is referenced by its memory address.

A work item has the following key properties:

- A **handler function**, which is the function executed by the workqueue's thread when the work item is processed. This function accepts a single argument, which is the address of the work item itself.
- A **pending flag**, which is used by the kernel to signify that the work item is currently a member of a workqueue's queue.
- A **queue link**, which is used by the kernel to link a pending work item to the next pending work item in a workqueue's queue.

A work item must be initialized before it can be used. This records the work item's handler function and marks it as not pending.

A work item may be **submitted** to a workqueue by an ISR or a thread. Submitting a work item appends the work item to the workqueue's queue. Once the workqueue's thread has processed all of the preceding work items in its queue the thread will remove a pending work item from its queue and invoke the work item's handler function. Depending on the scheduling priority of the workqueue's thread, and the work required by other items in the queue, a pending work item may be processed quickly or it may remain in the queue for an extended period of time.

A handler function can utilize any kernel API available to threads. However, operations that are potentially blocking (e.g. taking a semaphore) must be used with care, since the workqueue cannot process subsequent work items in its queue until the handler function finishes executing.

The single argument that is passed to a handler function can be ignored if it is not required. If the handler function requires additional information about the work it is to perform, the work item can be embedded in a larger data structure. The handler function can then use the argument value to compute the address of the enclosing data structure, and thereby obtain access to the additional information it needs.

A work item is typically initialized once and then submitted to a specific workqueue whenever work needs to be performed. If an ISR or a thread attempts to submit a work item that is already pending, the work item is not affected; the work item remains in its current place in the workqueue's queue, and the work is only performed once.

A handler function is permitted to re-submit its work item argument to the workqueue, since the work item is no longer pending at that time. This allows the handler to execute work in stages, without unduly delaying the processing of other work items in the workqueue's queue.

重要: A pending work item *must not* be altered until the item has been processed by the workqueue thread. This means a work item must not be re-initialized while it is pending. Furthermore, any additional information the work item's handler function needs to perform its work must not be altered until the handler function has finished executing.

Delayed Work

An ISR or a thread may need to schedule a work item that is to be processed only after a specified period of time, rather than immediately. This can be done by submitting a **delayed work item** to a workqueue, rather than a standard work item.

A delayed work item is a standard work item that has the following added properties:

- A **delay** specifying the time interval to wait before the work item is actually submitted to a workqueue's queue.
- A **workqueue indicator** that identifies the workqueue the work item is to be submitted to.

A delayed work item is initialized and submitted to a workqueue in a similar manner to a standard work item, although different kernel APIs are used. When the submit request is made the kernel initiates a timeout mechanism that is triggered after the specified delay has elapsed. Once the timeout occurs the kernel submits the delayed work item to the specified workqueue, where it remains pending until it is processed in the standard manner.

An ISR or a thread may **cancel** a delayed work item it has submitted, providing the work item's timeout is still counting down. The work item's timeout is aborted and the specified work is not performed.

Attempting to cancel a delayed work item once its timeout has expired has no effect on the work item; the work item remains pending in the workqueue's queue, unless the work item has already been removed and processed by the workqueue's thread. Consequently, once a work item's timeout has expired the work item is always processed by the workqueue and cannot be canceled.

System Workqueue

The kernel defines a workqueue known as the *system workqueue*, which is available to any application or kernel code that requires workqueue support. The system workqueue is optional, and only exists if the application makes use of it.

重要: Additional workqueues should only be defined when it is not possible to submit new work items to the system workqueue, since each new workqueue incurs a significant cost in memory footprint. A new workqueue can be justified if it is not possible for its work items to co-exist with existing system workqueue work items without an unacceptable impact; for example, if the new work items perform blocking operations that would delay other system workqueue processing to an unacceptable degree.

Implementation

Defining a Workqueue

A workqueue is defined using a variable of type `struct k_work_q`. The workqueue is initialized by defining the stack area used by its thread and then calling `k_work_q_start()`. The stack area must be defined using `K_THREAD_STACK_DEFINE` to ensure it is properly set up in memory.

The following code defines and initializes a workqueue.

```
#define MY_STACK_SIZE 512
#define MY_PRIORITY 5

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);

struct k_work_q my_work_q;

k_work_q_start(&my_work_q, my_stack_area,
               K_THREAD_STACK_SIZEOF(my_stack_area), MY_PRIORITY);
```

Submitting a Work Item

A work item is defined using a variable of type `struct k_work`. It must then be initialized by calling `k_work_init()`.

An initialized work item can be submitted to the system workqueue by calling `k_work_submit()`, or to a specified workqueue by calling `k_work_submit_to_queue()`.

The following code demonstrates how an ISR can offload the printing of error messages to the system workqueue. Note that if the ISR attempts to resubmit the work item while it is still pending, the work item is left unchanged and the associated error message will not be printed.

```
struct device_info {
    struct k_work work;
    char name[16]
} my_device;

void my_isr(void *arg)
{
    ...
    if (error detected) {
        k_work_submit(&my_device.work);
    }
    ...
}

void print_error(struct k_work *item)
{
    struct device_info *the_device =
        CONTAINER_OF(item, struct device_info, work);
    printk("Got error on device %s\n", the_device->name);
}

/* initialize name info for a device */
strcpy(my_device.name, "FOO_dev");

/* initialize work item for printing device's error messages */
k_work_init(&my_device.work, print_error);

/* install my_isr() as interrupt handler for the device (not shown) */
...
```

Submitting a Delayed Work Item

A delayed work item is defined using a variable of type `struct k_delayed_work`. It must then be initialized by calling `k_delayed_work_init()`.

An initialized delayed work item can be submitted to the system workqueue by calling `k_delayed_work_submit()`, or to a specified workqueue by calling `k_delayed_work_submit_to_queue()`. A delayed work item that has been submitted but not yet consumed by its workqueue can be canceled by calling `k_delayed_work_cancel()`.

Suggested Uses

Use the system workqueue to defer complex interrupt-related processing from an ISR to a cooperative thread. This allows the interrupt-related processing to be done promptly without compromising the system's ability to respond to subsequent interrupts, and does not require the application to define an additional thread to do the processing.

Configuration Options

Related configuration options:

- `CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE`
- `CONFIG_SYSTEM_WORKQUEUE_PRIORITY`

APIs

- `k_work_q_start()`
- `k_work_init()`
- `k_work_submit()`
- `k_work_submit_to_queue()`
- `k_delayed_work_init()`
- `k_delayed_work_submit()`
- `k_delayed_work_submit_to_queue()`
- `k_delayed_work_cancel()`
- `k_work_pending()`

Timing

This section describes the kernel's time-based services, such as specifying time delays or for measuring the passage of time.

Kernel Clocks

The kernel's clocks are the foundation for all of its time-based services.

Concepts

The kernel supports two distinct clocks.

- The 32-bit **hardware clock** is a high precision counter that tracks time in unspecified units called **cycles**. The duration of a cycle is determined by the board hardware used by the kernel, and is typically measured in nanoseconds.
- The 64-bit **system clock** is a counter that tracks the number of **ticks** that have elapsed since the kernel was initialized. The duration of a tick is is configurable, and typically ranges from 1 millisecond to 100 milliseconds.

The kernel also provides a number of variables that can be used to convert the time units used by the clocks into standard time units (e.g. seconds, milliseconds, nanoseconds, etc), and to convert between the two types of clock time units.

The system clock is used by most of the kernel's time-based services, including kernel timer objects and the timeouts supported by other kernel object types. For convenience, the kernel's APIs allow time durations to be specified in milliseconds, and automatically converts them to the corresponding number of ticks.

The hardware clock can be used to measure time with higher precision than that provided by kernel services based on the system clock.

Clock Limitations

The system clock's tick count is derived from the hardware clock's cycle count. The kernel determines how many clock cycles correspond to the desired tick frequency, then programs the hardware clock to generate an interrupt after that many cycles; each interrupt corresponds to a single tick.

注解： Configuring a smaller tick duration permits finer-grained timing, but also increases the amount of work the kernel has to do to process tick interrupts since they occur more frequently. Setting the tick duration to zero disables *both* kernel clocks, as well as their associated services.

Any millisecond-based time interval specified using a kernel API represents the **minimum** delay that will occur, and may actually take longer than the amount of time requested.

For example, specifying a timeout delay of 100 ms when attempting to take a semaphore means that the kernel will never terminate the operation and report failure before at least 100 ms have elapsed. However, it is possible that the operation may take longer than 100 ms to complete, and may either complete successfully during the additional time or fail at the end of the added time.

The amount of added time that occurs during a kernel object operation depends on the following factors.

- The added time introduced by rounding up the specified time interval when converting from milliseconds to ticks. For example, if a tick duration of 10 ms is being used, a specified delay of 25 ms will be rounded up to 30 ms.
- The added time introduced by having to wait for the next tick interrupt before a delay can be properly tracked. For example, if a tick duration of 10 ms is being used, a specified delay of 20 ms requires the kernel to wait for 3 ticks to occur (rather than only 2), since the first tick can occur at any time from the next fraction of a millisecond to just slightly less than 10 ms; only after the first tick has occurred does the kernel know the next 2 ticks will take 20 ms.

Implementation

Measuring Time with Normal Precision

This code uses the system clock to determine how much time has elapsed between two points in time.

```
s64_t time_stamp;
s64_t milliseconds_spent;

/* capture initial time stamp */
time_stamp = k_uptime_get();

/* do work for some (extended) period of time */
...

/* compute how long the work took (also updates the time stamp) */
milliseconds_spent = k_uptime_delta(&time_stamp);
```

Measuring Time with High Precision

This code uses the hardware clock to determine how much time has elapsed between two points in time.

```
u32_t start_time;
u32_t stop_time;
u32_t cycles_spent;
u32_t nanoseconds_spent;

/* capture initial time stamp */
start_time = k_cycle_get_32();
```

```
/* do work for some (short) period of time */
...

/* capture final time stamp */
stop_time = k_cycle_get_32();

/* compute how long the work took (assumes no counter rollover) */
cycles_spent = stop_time - start_time;
nanoseconds_spent = SYS_CLOCK_HW_CYCLES_TO_NS(cycles_spent);
```

Suggested Uses

Use services based on the system clock for time-based processing that does not require high precision, such as *timer objects* or *Thread Sleeping*.

Use services based on the hardware clock for time-based processing that requires higher precision than the system clock can provide, such as *Busy Waiting* or fine-grained time measurements.

注解: The high frequency of the hardware clock, combined with its 32-bit size, means that counter rollover must be taken into account when taking high-precision measurements over an extended period of time.

Configuration

Related configuration options:

- `CONFIG_SYS_CLOCK_TICKS_PER_SEC`

APIs

The following kernel clock APIs are provided by `kernel.h`:

- `k_uptime_get()`
- `k_uptime_get_32()`
- `k_uptime_delta()`
- `k_uptime_delta_32()`
- `k_cycle_get_32()`
- `SYS_CLOCK_HW_CYCLES_TO_NS`
- `K_NO_WAIT`
- `K_MSEC`
- `K_SECONDS`
- `K_MINUTES`
- `K_HOURS`
- `K_FOREVER`

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 it can simply record the expiration and wait for the application to read its status.

Concepts

Any number of timers can be defined. Each timer is referenced by its memory address.

A timer has the following key properties:

- A *duration* specifying the time interval before the timer expires for the first time, measured in milliseconds. It must be greater than zero.
- A *period* specifying the time interval between all timer expirations after the first one, measured in milliseconds. It must be non-negative. A period of zero means that the timer is a one shot timer that stops after a single expiration. (For example then, if a timer is started with a duration of 200 and a period of 75, it will first expire after 200ms and then every 75ms after that.)
- An *expiry function* that is executed each time the timer expires. The function is executed by the system clock interrupt handler. If no expiry function is required a `NULL` function can be specified.
- A *stop function* that is executed if the timer is stopped prematurely while running. The function is executed by the thread that stops the timer. If no stop function is required a `NULL` function can be specified.
- A *status* value that indicates how many times the timer has expired since the status value was last read.

A timer must be initialized before it can be used. This specifies its expiry function and stop function values, sets the timer's status to zero, and puts the timer into the **stopped** state.

A timer is **started** by specifying a duration and a period. The timer's status is reset to zero, then the timer enters the **running** state and begins counting down towards expiry.

When a running timer expires its status is incremented and the timer executes its expiry function, if one exists; If a thread is waiting on the timer, it is unblocked. If the timer's period is zero the timer enters the stopped state; otherwise the timer restarts with a new duration equal to its period.

A running timer can be stopped in mid-countdown, if desired. The timer's status is left unchanged, then the timer enters the stopped state and executes its stop function, if one exists. If a thread is waiting on the timer, it is unblocked. Attempting to stop a non-running timer is permitted, but has no effect on the timer since it is already stopped.

A running timer can be restarted in mid-countdown, if desired. The timer's status is reset to zero, then the timer begins counting down using the new duration and period values specified by the caller. If a thread is waiting on the timer, it continues waiting.

A timer's status can be read directly at any time to determine how many times the timer has expired since its status was last read. Reading a timer's status resets its value to zero. The amount of time remaining before the timer expires can also be read; a value of zero indicates that the timer is stopped.

A thread may read a timer's status indirectly by **synchronizing** with the timer. This blocks the thread until the timer's status is non-zero (indicating that it has expired at least once) or the timer is stopped; if the timer status is already non-zero or the timer is already stopped the thread continues without waiting. The synchronization operation returns the timer's status and resets it to zero.

注解: Only a single user should examine the status of any given timer, since reading the status (directly or indirectly) changes its value. Similarly, only a single thread at a time should synchronize with a given timer. ISRs are not permitted to synchronize with timers, since ISRs are not allowed to block.

Timer Limitations

Since timers are based on the system clock, the delay values specified when using a timer are **minimum** values. (See *Clock Limitations*.)

Implementation

Defining a Timer

A timer is defined using a variable of type `struct k_timer`. It must then be initialized by calling `k_timer_init()`.

The following code defines and initializes a timer.

```
struct k_timer my_timer;
extern void my_expiry_function(struct k_timer *timer_id);

k_timer_init(&my_timer, my_expiry_function, NULL);
```

Alternatively, a timer can be defined and initialized at compile time by calling `K_TIMER_DEFINE`.

The following code has the same effect as the code segment above.

```
K_TIMER_DEFINE(my_timer, my_expiry_function, NULL);
```

Using a Timer Expiry Function

The following code uses a timer to perform a non-trivial action on a periodic basis. Since the required work cannot be done at interrupt level, the timer's expiry function submits a work item to the *system workqueue*, whose thread performs the work.

```
void my_work_handler(struct k_work *work)
{
    /* do the processing that needs to be done periodically */
    ...
}

K_WORK_DEFINE(my_work, my_work_handler);

void my_timer_handler(struct k_timer *dummy)
{
    k_work_submit(&my_work);
}

K_TIMER_DEFINE(my_timer, my_timer_handler, NULL);

...

/* start periodic timer that expires once every second */
k_timer_start(&my_timer, K_SECONDS(1), K_SECONDS(1));
```

Reading Timer Status

The following code reads a timer's status directly to determine if the timer has expired on not.

```
K_TIMER_DEFINE(my_status_timer, NULL, NULL);

...

/* start one shot timer that expires after 200 ms */
k_timer_start(&my_status_timer, K_MSEC(200), 0);

/* do work */
...

/* check timer status */
if (k_timer_status_get(&my_status_timer) > 0) {
    /* timer has expired */
} else if (k_timer_remaining_get(&my_status_timer) == 0) {
    /* timer was stopped (by someone else) before expiring */
} else {
    /* timer is still running */
}
```

Using Timer Status Synchronization

The following code performs timer status synchronization to allow a thread to do useful work while ensuring that a pair of protocol operations are separated by the specified time interval.

```
K_TIMER_DEFINE(my_sync_timer, NULL, NULL);

...

/* do first protocol operation */
...

/* start one shot timer that expires after 500 ms */
k_timer_start(&my_sync_timer, K_MSEC(500), 0);
```

```
/* do other work */
...

/* ensure timer has expired (waiting for expiry, if necessary) */
k_timer_status_sync(&my_sync_timer);

/* do second protocol operation */
...
```

注解: If the thread had no other work to do it could simply sleep between the two protocol operations, without using a timer.

Suggested Uses

Use a timer to initiate an asynchronous operation after a specified amount of time.

Use a timer to determine whether or not a specified amount of time has elapsed.

Use a timer to perform other work while carrying out operations involving time limits.

注解: If a thread has no other work to perform while waiting for time to pass it should call `k_sleep()`. If a thread needs to measure the time required to perform an operation it can read the *system clock or the hardware clock* directly, rather than using a timer.

Configuration Options

Related configuration options:

- None.

APIs

The following timer APIs are provided by `kernel.h`:

- `K_TIMER_DEFINE`
- `k_timer_init()`
- `k_timer_start()`
- `k_timer_stop()`
- `k_timer_status_get()`
- `k_timer_status_sync()`
- `k_timer_remaining_get()`

Memory Allocation

This section describes kernel services that allow threads to dynamically allocate memory.

Memory Slabs

A *memory slab* is a kernel object that allows memory blocks to be dynamically allocated from a designated memory region. All memory blocks in a memory slab have a single fixed size, allowing them to be allocated and released efficiently and avoiding memory fragmentation concerns.

Concepts

Any number of memory slabs can be defined. Each memory slab is referenced by its memory address.

A memory slab has the following key properties:

- The **block size** of each block, measured in bytes. It must be at least 4N bytes long, where N is greater than 0.
- The **number of blocks** available for allocation. It must be greater than zero.
- A **buffer** that provides the memory for the memory slab's blocks. It must be at least "block size" times "number of blocks" bytes long.

The memory slab's buffer must be aligned to an N-byte boundary, where N is a power of 2 larger than 2 (i.e. 4, 8, 16, ...). To ensure that all memory blocks in the buffer are similarly aligned to this boundary, the block size must also be a multiple of N.

A memory slab must be initialized before it can be used. This marks all of its blocks as unused.

A thread that needs to use a memory block simply allocates it from a memory slab. When the thread finishes with a memory block, it must release the block back to the memory slab so the block can be reused.

If all the blocks are currently in use, a thread can optionally wait for one to become available. Any number of threads may wait on an empty memory slab simultaneously; when a memory block becomes available, it is given to the highest-priority thread that has waited the longest.

Unlike a heap, more than one memory slab can be defined, if needed. This allows for a memory slab with smaller blocks and others with larger-sized blocks. Alternatively, a memory pool object may be used.

Internal Operation

A memory slab's buffer is an array of fixed-size blocks, with no wasted space between the blocks.

The memory slab keeps track of unallocated blocks using a linked list; the first 4 bytes of each unused block provide the necessary linkage.

Implementation

Defining a Memory Slab

A memory slab is defined using a variable of type `struct k_mem_slab`. It must then be initialized by calling `k_mem_slab_init()`.

The following code defines and initializes a memory slab that has 6 blocks that are 400 bytes long, each of which is aligned to a 4-byte boundary..

```
struct k_mem_slab my_slab;
char __aligned(4) my_slab_buffer[6 * 400];

k_mem_slab_init(&my_slab, my_slab_buffer, 400, 6);
```

Alternatively, a memory slab can be defined and initialized at compile time by calling `K_MEM_SLAB_DEFINE`.

The following code has the same effect as the code segment above. Observe that the macro defines both the memory slab and its buffer.

```
K_MEM_SLAB_DEFINE(my_slab, 400, 6, 4);
```

Allocating a Memory Block

A memory block is allocated by calling `k_mem_slab_alloc()`.

The following code builds on the example above, and waits up to 100 milliseconds for a memory block to become available, then fills it with zeroes. A warning is printed if a suitable block is not obtained.

```
char *block_ptr;

if (k_mem_slab_alloc(&my_slab, &block_ptr, 100) == 0)) {
    memset(block_ptr, 0, 400);
    ...
} else {
    printf("Memory allocation time-out");
}
```

Releasing a Memory Block

A memory block is released by calling `k_mem_slab_free()`.

The following code builds on the example above, and allocates a memory block, then releases it once it is no longer needed.

```
char *block_ptr;

k_mem_slab_alloc(&my_slab, &block_ptr, K_FOREVER);
... /* use memory block pointed at by block_ptr */
k_mem_slab_free(&my_slab, &block_ptr);
```

Suggested Uses

Use a memory slab to allocate and free memory in fixed-size blocks.

Use memory slab blocks when sending large amounts of data from one thread to another, to avoid unnecessary copying of the data.

Configuration Options

Related configuration options:

- None.

APIs

The following memory slab APIs are provided by `kernel.h`:

- `K_MEM_SLAB_DEFINE`
- `k_mem_slab_init()`
- `k_mem_slab_alloc()`
- `k_mem_slab_free()`
- `k_mem_slab_num_used_get()`
- `k_mem_slab_num_free_get()`

Memory Pools

A *memory pool* is a kernel object that allows memory blocks to be dynamically allocated from a designated memory region. The memory blocks in a memory pool can be of any size, thereby reducing the amount of wasted memory when an application needs to allocate storage for data structures of different sizes. The memory pool uses a "buddy memory allocation" algorithm to efficiently partition larger blocks into smaller ones, allowing blocks of different sizes to be allocated and released efficiently while limiting memory fragmentation concerns.

Concepts

Any number of memory pools can be defined. Each memory pool is referenced by its memory address.

A memory pool has the following key properties:

- A **minimum block size**, measured in bytes. It must be at least 4X bytes long, where X is greater than 0.
- A **maximum block size**, measured in bytes. This should be a power of 4 times larger than the minimum block size. That is, "maximum block size" must equal "minimum block size" times 4^Y, where Y is greater than or equal to zero.
- The **number of maximum-size blocks** initially available. This must be greater than zero.
- A **buffer** that provides the memory for the memory pool's blocks. This must be at least "maximum block size" times "number of maximum-size blocks" bytes long.

The memory pool's buffer must be aligned to an N-byte boundary, where N is a power of 2 larger than 2 (i.e. 4, 8, 16, ...). To ensure that all memory blocks in the buffer are similarly aligned to this boundary, the minimum block size must also be a multiple of N.

A thread that needs to use a memory block simply allocates it from a memory pool. Following a successful allocation, the `data` field of the block descriptor supplied by the thread indicates the starting address of the memory block. When the thread is finished with a memory block, it must release the block back to the memory pool so the block can be reused.

If a block of the desired size is unavailable, a thread can optionally wait for one to become available. Any number of threads may wait on a memory pool simultaneously; when a suitable memory block becomes available, it is given to the highest-priority thread that has waited the longest.

Unlike a heap, more than one memory pool can be defined, if needed. For example, different applications can utilize different memory pools; this can help prevent one application from hijacking resources to allocate all of the available blocks.

Internal Operation

A memory pool's buffer is an array of maximum-size blocks, with no wasted space between the blocks. Each of these "level 0" blocks is a *quad-block* that can be partitioned into four smaller "level 1" blocks of equal size, if needed. Likewise, each level 1 block is itself a quad-block that can be partitioned into four smaller "level 2" blocks in a similar way, and so on. Thus, memory pool blocks can be recursively partitioned into quarters until blocks of the minimum size are obtained, at which point no further partitioning can occur.

A memory pool keeps track of how its buffer space has been partitioned using an array of *block set* data structures. There is one block set for each partitioning level supported by the pool, or (to put it another way) for each block size. A block set keeps track of all free blocks of its associated size using an array of *quad-block status* data structures.

When an application issues a request for a memory block, the memory pool first determines the size of the smallest block that will satisfy the request, and examines the corresponding block set. If the block set contains a free block, the block is marked as used and the allocation process is complete. If the block set does not contain a free block, the memory pool attempts to create one automatically by splitting a free block of a larger size or by merging free blocks of smaller sizes; if a suitable block can't be created, the allocation request fails.

The memory pool's merging algorithm cannot combine adjacent free blocks of different sizes, nor can it merge adjacent free blocks of the same size if they belong to different parent quad-blocks. As a consequence, memory fragmentation issues can still be encountered when using a memory pool.

When an application releases a previously allocated memory block it is combined synchronously with its three "partner" blocks if possible, and recursively so up through the levels. This is done in constant time, and quickly, so no manual "defragmentation" management is needed.

Implementation

Defining a Memory Pool

A memory pool is defined using a variable of type `struct k_mem_pool`. However, since a memory pool also requires a number of variable-size data structures to represent its block sets and the status of its quad-blocks, the kernel does not support the run-time definition of a memory pool. A memory pool can only be defined and initialized at compile time by calling `K_MEM_POOL_DEFINE`.

The following code defines and initializes a memory pool that has 3 blocks of 4096 bytes each, which can be partitioned into blocks as small as 64 bytes and is aligned to a 4-byte boundary. (That is, the memory pool supports block sizes of 4096, 1024, 256, and 64 bytes.) Observe that the macro defines all of the memory pool data structures, as well as its buffer.

```
K_MEM_POOL_DEFINE(my_pool, 64, 4096, 3, 4);
```

Allocating a Memory Block

A memory block is allocated by calling `k_mem_pool_alloc()`.

The following code builds on the example above, and waits up to 100 milliseconds for a 200 byte memory block to become available, then fills it with zeroes. A warning is issued if a suitable block is not obtained.

Note that the application will actually receive a 256 byte memory block, since that is the closest matching size supported by the memory pool.

```
struct k_mem_block block;

if (k_mem_pool_alloc(&my_pool, &block, 200, 100) == 0)) {
    memset(block.data, 0, 200);
    ...
} else {
    printf("Memory allocation time-out");
}
```

Releasing a Memory Block

A memory block is released by calling `k_mem_pool_free()`.

The following code builds on the example above, and allocates a 75 byte memory block, then releases it once it is no longer needed. (A 256 byte memory block is actually used to satisfy the request.)

```
struct k_mem_block block;

k_mem_pool_alloc(&my_pool, &block, 75, K_FOREVER);
... /* use memory block */
k_mem_pool_free(&block);
```

Suggested Uses

Use a memory pool to allocate memory in variable-size blocks.

Use memory pool blocks when sending large amounts of data from one thread to another, to avoid unnecessary copying of the data.

APIs

The following memory pool APIs are provided by `kernel.h`:

- `K_MEM_POOL_DEFINE`
- `k_mem_pool_alloc()`
- `k_mem_pool_free()`

Heap Memory Pool

The *heap memory pool* is a predefined memory pool object that allows threads to dynamically allocate memory from a common memory region in a `malloc()`-like manner.

Concepts

Only a single heap memory pool can be defined. Unlike other memory pools, the heap memory pool cannot be directly referenced using its memory address.

The size of the heap memory pool is configurable. The following sizes are supported: 256 bytes, 1024 bytes, 4096 bytes, and 16384 bytes.

A thread can dynamically allocate a chunk of heap memory by calling `k_malloc()`. The address of the allocated chunk is guaranteed to be aligned on a multiple of 4 bytes. If a suitable chunk of heap memory cannot be found `NULL` is returned.

When the thread is finished with a chunk of heap memory it can release the chunk back to the heap memory pool by calling `k_free()`.

Internal Operation

The heap memory pool defines a single maximum size block that contains the entire heap; that is, a single block of 256, 1024, 4096, or 16384 bytes. The heap memory pool also defines a minimum block size of 64 bytes. Consequently, the maximum number of blocks of each size that the heap memory pool can support is shown in the following table.

heap size	64 byte blocks	256 byte blocks	1024 byte blocks	4096 byte blocks	16384 byte blocks
256	4	1	0	0	0
1024	16	4	1	0	0
4096	64	16	4	1	0
16384	256	64	16	4	1

注解: The number of blocks of a given size that can be allocated simultaneously is typically smaller than the value shown in the table. For example, each allocation of a 256 byte block from a 1024 byte heap reduces the number of 64 byte blocks available for allocation by 4. Fragmentation of the memory pool's buffer can also further reduce the availability of blocks.

The kernel uses the first 16 bytes of any memory block allocated from the heap memory pool to save the block descriptor information it needs to later free the block. Consequently, an application's request for an N byte chunk of heap memory requires a block that is at least (N+16) bytes long.

Implementation

Defining the Heap Memory Pool

The size of the heap memory pool is specified using the `CONFIG_HEAP_MEM_POOL_SIZE` configuration option.

By default, the heap memory pool size is zero bytes. This value instructs the kernel not to define the heap memory pool object.

Allocating Memory

A chunk of heap memory is allocated by calling `k_malloc()`.

The following code allocates a 200 byte chunk of heap memory, then fills it with zeros. A warning is issued if a suitable chunk is not obtained.

Note that the application will actually allocate a 256 byte memory block, since that is the closest matching size supported by the heap memory pool.

```
char *mem_ptr;

mem_ptr = k_malloc(200);
if (mem_ptr != NULL)) {
    memset(mem_ptr, 0, 200);
    ...
} else {
    printf("Memory not allocated");
}
```

Releasing Memory

A chunk of heap memory is released by calling `k_free()`.

The following code allocates a 75 byte chunk of memory, then releases it once it is no longer needed. (A 256 byte memory block from the heap memory pool is actually used to satisfy the request.)

```
char *mem_ptr;

mem_ptr = k_malloc(75);
... /* use memory block */
k_free(mem_ptr);
```

Suggested Uses

Use the heap memory pool to dynamically allocate memory in a `malloc()`-like manner.

Configuration Options

Related configuration options:

- `CONFIG_HEAP_MEM_POOL_SIZE`

APIs

The following heap memory pool APIs are provided by `kernel.h`:

- `k_malloc()`
- `k_free()`

Synchronization

This section describes kernel services for synchronizing the operation of different threads, or the operation of an ISR and a thread.

Semaphores

A *semaphore* is a kernel object that implements a traditional counting semaphore.

Concepts

Any number of semaphores can be defined. Each semaphore is referenced by its memory address.

A semaphore has the following key properties:

- A **count** that indicates the number of times the semaphore can be taken. A count of zero indicates that the semaphore is unavailable.
- A **limit** that indicates the maximum value the semaphore's count can reach.

A semaphore must be initialized before it can be used. Its count must be set to a non-negative value that is less than or equal to its limit.

A semaphore may be **given** by a thread or an ISR. Giving the semaphore increments its count, unless the count is already equal to the limit.

A semaphore may be **taken** by a thread. Taking the semaphore decrements its count, unless the semaphore is unavailable (i.e. at zero). When a semaphore is unavailable a thread may choose to wait for it to be given. Any number of threads may wait on an unavailable semaphore simultaneously. When the semaphore is given, it is taken by the highest priority thread that has waited longest.

注解: The kernel does allow an ISR to take a semaphore, however the ISR must not attempt to wait if the semaphore is unavailable.

Implementation

Defining a Semaphore

A semaphore is defined using a variable of type `struct k_sem`. It must then be initialized by calling `k_sem_init()`.

The following code defines a semaphore, then configures it as a binary semaphore by setting its count to 0 and its limit to 1.

```
struct k_sem my_sem;

k_sem_init(&my_sem, 0, 1);
```

Alternatively, a semaphore can be defined and initialized at compile time by calling `K_SEM_DEFINE`.

The following code has the same effect as the code segment above.

```
K_SEM_DEFINE(my_sem, 0, 1);
```

Giving a Semaphore

A semaphore is given by calling `k_sem_give()`.

The following code builds on the example above, and gives the semaphore to indicate that a unit of data is available for processing by a consumer thread.

```
void input_data_interrupt_handler(void *arg)
{
    /* notify thread that data is available */
    k_sem_give(&my_sem);

    ...
}
```

Taking a Semaphore

A semaphore is taken by calling `k_sem_take()`.

The following code builds on the example above, and waits up to 50 milliseconds for the semaphore to be given. A warning is issued if the semaphore is not obtained in time.

```
void consumer_thread(void)
{
    ...

    if (k_sem_take(&my_sem, K_MSEC(50)) != 0) {
        printk("Input data not available!");
    } else {
        /* fetch available data */
        ...
    }
    ...
}
```

Suggested Uses

Use a semaphore to control access to a set of resources by multiple threads.

Use a semaphore to synchronize processing between a producing and consuming threads or ISRs.

Configuration Options

Related configuration options:

- None.

APIs

The following semaphore APIs are provided by `kernel.h`:

- `K_SEM_DEFINE`
- `k_sem_init()`
- `k_sem_give()`
- `k_sem_take()`
- `k_sem_reset()`
- `k_sem_count_get()`

Mutexes

A *mutex* is a kernel object that implements a traditional reentrant mutex. A mutex allows multiple threads to safely share an associated hardware or software resource by ensuring mutually exclusive access to the resource.

Concepts

Any number of mutexes can be defined. Each mutex is referenced by its memory address.

A mutex has the following key properties:

- A **lock count** that indicates the number of times the mutex has be locked by the thread that has locked it. A count of zero indicates that the mutex is unlocked.
- An **owning thread** that identifies the thread that has locked the mutex, when it is locked.

A mutex must be initialized before it can be used. This sets its lock count to zero.

A thread that needs to use a shared resource must first gain exclusive rights to access it by **locking** the associated mutex. If the mutex is already locked by another thread, the requesting thread may choose to wait for the mutex to be unlocked.

After locking a mutex, the thread may safely use the associated resource for as long as needed; however, it is considered good practice to hold the lock for as short a time as possible to avoid negatively impacting other threads that want to use the resource. When the thread no longer needs the resource it must **unlock** the mutex to allow other threads to use the resource.

Any number of threads may wait on a locked mutex simultaneously. When the mutex becomes unlocked it is then locked by the highest-priority thread that has waited the longest.

注解: Mutex objects are *not* designed for use by ISRs.

Reentrant Locking

A thread is permitted to lock a mutex it has already locked. This allows the thread to access the associated resource at a point in its execution when the mutex may or may not already be locked.

A mutex that is repeatedly locked by a thread must be unlocked an equal number of times before the mutex becomes fully unlocked so it can be claimed by another thread.

Priority Inheritance

The thread that has locked a mutex is eligible for *priority inheritance*. This means the kernel will *temporarily* elevate the thread's priority if a higher priority thread begins waiting on the mutex. This allows the owning thread to complete its work and release the mutex more rapidly by executing at the same priority as the waiting thread. Once the mutex has been unlocked, the unlocking thread resets its priority to the level it had before locking that mutex.

注解： The `CONFIG_PRIORITY_CEILING` configuration option limits how high the kernel can raise a thread's priority due to priority inheritance. The default value of 0 permits unlimited elevation.

When two or more threads wait on a mutex held by a lower priority thread, the kernel adjusts the owning thread's priority each time a thread begins waiting (or gives up waiting). When the mutex is eventually unlocked, the unlocking thread's priority correctly reverts to its original non-elevated priority.

The kernel does *not* fully support priority inheritance when a thread holds two or more mutexes simultaneously. This situation can result in the thread's priority not reverting to its original non-elevated priority when all mutexes have been released. It is recommended that a thread lock only a single mutex at a time when multiple mutexes are shared between threads of different priorities.

Implementation

Defining a Mutex

A mutex is defined using a variable of type `struct k_mutex`. It must then be initialized by calling `k_mutex_init()`.

The following code defines and initializes a mutex.

```
struct k_mutex my_mutex;

k_mutex_init(&my_mutex);
```

Alternatively, a mutex can be defined and initialized at compile time by calling `K_MUTEX_DEFINE`.

The following code has the same effect as the code segment above.

```
K_MUTEX_DEFINE(my_mutex);
```

Locking a Mutex

A mutex is locked by calling `k_mutex_lock()`.

The following code builds on the example above, and waits indefinitely for the mutex to become available if it is already locked by another thread.

```
k_mutex_lock(&my_mutex, K_FOREVER);
```

The following code waits up to 100 milliseconds for the mutex to become available, and gives a warning if the mutex does not become available.

```
if (k_mutex_lock(&my_mutex, K_MSEC(100)) == 0) {
    /* mutex successfully locked */
} else {
    printf("Cannot lock XYZ display\n");
}
```

Unlocking a Mutex

A mutex is unlocked by calling `k_mutex_unlock()`.

The following code builds on the example above, and unlocks the mutex that was previously locked by the thread.

```
k_mutex_unlock(&my_mutex);
```

Suggested Uses

Use a mutex to provide exclusive access to a resource, such as a physical device.

Configuration Options

Related configuration options:

- `CONFIG_PRIORITY_CEILING`

APIs

The following mutex APIs are provided by `kernel.h`:

- `K_MUTEX_DEFINE`
- `k_mutex_init()`
- `k_mutex_lock()`
- `k_mutex_unlock()`

Alerts

An *alert* is a kernel object that allows an application to perform asynchronous signaling when a condition of interest occurs.

Concepts

Any number of alerts can be defined. Each alert is referenced by its memory address.

An alert has the following key properties:

- An **alert handler**, which specifies the action to be taken when the alert is signaled. The action may instruct the system workqueue to execute a function to process the alert, mark the alert as pending so it can be processed later by a thread, or ignore the alert.
- An **pending count**, which records the number of pending alerts that have yet to be received.
- An **count limit**, which specifies the maximum number of pending alerts that will be recorded.

An alert must be initialized before it can be used. This establishes its alert handler and sets the pending count to zero.

Alert Lifecycle

An ISR or a thread signals an alert by **sending** the alert when a condition of interest occurs that cannot be handled by the detector.

Each time an alert is sent, the kernel examines its alert handler to determine what action to take.

- `K_ALERT_IGNORE` causes the alert to be ignored.
- `K_ALERT_DEFAULT` causes the pending count to be incremented, unless this would exceed the count limit.
- Any other value is assumed to be the address of an alert handler function, and is invoked by the system workqueue thread. If the function returns zero, the signal is deemed to have been consumed; otherwise the pending count is incremented, unless this would exceed the count limit.

  The kernel ensures that the alert handler function is executed once for each time an alert is sent, even if the alert is sent multiple times in rapid succession.

A thread accepts a pending alert by **receiving** the alert. This decrements the pending count. If the pending count is currently zero, the thread may choose to wait for the alert to become pending. Any number of threads may wait for a pending alert simultaneously; when the alert is pended it is accepted by the highest priority thread that has waited longest.

注解: A thread must processes pending alerts one at a time. The thread cannot receive multiple pending alerts in a single operation.

Comparison to Unix-style Signals

Zephyr alerts are somewhat akin to Unix-style signals, but have a number of significant differences. The most notable of these are:

- A Zephyr alert cannot be blocked; it is always delivered to its alert handler immediately.
- A Zephyr alert pends *after* it has been delivered to its alert handler, and only if an alert handler function does not consume the alert.
- Zephyr has no predefined alerts or actions. All alerts are application defined, and all have a default action that pends the alert.

Implementation

Defining an Alert

An alert is defined using a variable of type `struct k_alert`. It must then be initialized by calling `k_alert_init()`.

The following code defines and initializes an alert. The alert allows up to 10 unreceived alert signals to pend before it begins to ignore new pending alerts.

```
extern int my_alert_handler(struct k_alert *alert);

struct k_alert my_alert;

k_alert_init(&my_alert, my_alert_handler, 10);
```

Alternatively, an alert can be defined and initialized at compile time by calling `K_ALERT_DEFINE`.

The following code has the same effect as the code segment above.

```
extern int my_alert_handler(struct k_alert *alert);

K_ALERT_DEFINE(my_alert, my_alert_handler, 10);
```

Signaling an Alert

An alert is signaled by calling `k_alert_send()`.

The following code illustrates how an ISR can signal an alert to indicate that a key press has occurred.

```
extern int my_alert_handler(struct k_alert *alert);

K_ALERT_DEFINE(my_alert, my_alert_handler);

void keypress_interrupt_handler(void *arg)
{
    ...
    k_alert_send(&my_alert);
    ...
}
```

Handling an Alert

An alert handler function is used when a signaled alert should not be ignored or immediately pended. It has the following form:

```
int <function_name>(struct k_alert *alert)
{
    /* catch the alert signal; return zero if the signal is consumed, */
    /* or non-zero to let the alert pend                              */
    ...
}
```

The following code illustrates an alert handler function that processes key presses detected by an ISR (as shown in the previous section).

```
int my_alert_handler(struct k_alert *alert_id_is_unused)
{
    /* determine what key was pressed */
    char c = get_keypress();

    /* do complex processing of the keystroke */
    ...

    /* signaled alert has been consumed */
    return 0;
}
```

Accepting an Alert

A pending alert is accepted by a thread by calling `k_alert_recv()`.

The following code is an alternative to the code in the previous section. It uses a dedicated thread to do very complex processing of key presses that would otherwise monopolize the system workqueue. The alert handler function is now used only to filter out unwanted key press alerts, allowing the dedicated thread to wake up and process key press alerts only when a numeric key is pressed.

```
int my_alert_handler(struct k_alert *alert_id_is_unused)
{
    /* determine what key was pressed */
    char c = get_keypress();

    /* signal thread only if key pressed was a digit */
    if ((c >= '0') && (c <= '9')) {
        /* save key press information */
        ...
        /* signaled alert should be pended */
        return 1;
    } else {
        /* signaled alert has been consumed */
        return 0;
    }
}

void keypress_thread(void *unused1, void *unused2, void *unused3)
{
    /* consume numeric key presses */
    while (1) {

        /* wait for a key press alert to pend */
        k_alert_recv(&my_alert, K_FOREVER);
        /* process saved key press, which must be a digit */
        ...
    }
}
```

Suggested Uses

Use an alert to minimize ISR processing by deferring interrupt-related work to a thread to reduce the amount of time interrupts are locked.

Use an alert to allow the kernel's system workqueue to handle an alert, rather than defining an application thread to handle it.

Use an alert to allow the kernel's system workqueue to preprocess an alert, prior to letting an application thread handle it.

Configuration Options

Related configuration options:

- None.

APIs

The following alert APIs are provided by `kernel.h`:

- `K_ALERT_DEFINE`
- `k_alert_init()`
- `k_alert_send()`
- `k_alert_recv()`

Data Passing

This section describes kernel services for passing data between different threads, or between an ISR and a thread.

Fifos

A *fifo* is a kernel object that implements a traditional first in, first out (FIFO) queue, allowing threads and ISRs to add and remove data items of any size.

Concepts

Any number of fifos can be defined. Each fifo is referenced by its memory address.

A fifo has the following key properties:

- A **queue** of data items that have been added but not yet removed. The queue is implemented as a simple linked list.

A fifo must be initialized before it can be used. This sets its queue to empty.

Fifo data items must be aligned on a 4-byte boundary, as the kernel reserves the first 32 bits of an item for use as a pointer to the next data item in the queue. Consequently, a data item that holds N bytes of application data requires N+4 bytes of memory.

A data item may be **added** to a fifo by a thread or an ISR. The item is given directly to a waiting thread, if one exists; otherwise the item is added to the fifo's queue. There is no limit to the number of items that may be queued.

A data item may be **removed** from a fifo by a thread. If the fifo's queue is empty a thread may choose to wait for a data item to be given. Any number of threads may wait on an empty fifo simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

注解: The kernel does allow an ISR to remove an item from a fifo, however the ISR must not attempt to wait if the fifo is empty.

If desired, **multiple data items** can be added to a fifo in a single operation if they are chained together into a singly-linked list. This capability can be useful if multiple writers are adding sets of related data items to the fifo, as it ensures the data items in each set are not interleaved with other data items. Adding multiple data items to a fifo is also more efficient than adding them one at a time, and can be used to guarantee that anyone who removes the first data item in a set will be able to remove the remaining data items without waiting.

Implementation

Defining a Fifo

A fifo is defined using a variable of type `struct k_fifo`. It must then be initialized by calling `k_fifo_init()`.

The following code defines and initializes an empty fifo.

```
struct k_fifo my_fifo;

k_fifo_init(&my_fifo);
```

Alternatively, an empty fifo can be defined and initialized at compile time by calling `K_FIFO_DEFINE`.

The following code has the same effect as the code segment above.

```
K_FIFO_DEFINE(my_fifo);
```

Writing to a Fifo

A data item is added to a fifo by calling `k_fifo_put()`.

The following code builds on the example above, and uses the fifo to send data to one or more consumer threads.

```
struct data_item_t {
    void *fifo_reserved;    /* 1st word reserved for use by fifo */
    ...
};

struct data_item_t tx_data;

void producer_thread(int unused1, int unused2, int unused3)
{
    while (1) {
        /* create data item to send */
        tx_data = ...

        /* send data to consumers */
        k_fifo_put(&my_fifo, &tx_data);

        ...
    }
}
```

Additionally, a singly-linked list of data items can be added to a fifo by calling `k_fifo_put_list()` or `k_fifo_put_slist()`.

Reading from a Fifo

A data item is removed from a fifo by calling `k_fifo_get()`.

The following code builds on the example above, and uses the fifo to obtain data items from a producer thread, which are then processed in some manner.

```
void consumer_thread(int unused1, int unused2, int unused3)
{
    struct data_item_t  *rx_data;

    while (1) {
        rx_data = k_fifo_get(&my_fifo, K_FOREVER);

        /* process fifo data item */
        ...
    }
}
```

Suggested Uses

Use a fifo to asynchronously transfer data items of arbitrary size in a "first in, first out" manner.

Configuration Options

Related configuration options:

- None.

APIs

The following fifo APIs are provided by `kernel.h`:

- `K_FIFO_DEFINE`
- `k_fifo_init()`
- `k_fifo_put()`
- `k_fifo_put_list()`
- `k_fifo_put_slist()`
- `k_fifo_get()`

Lifos

A *lifo* is a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove data items of any size.

Concepts

Any number of lifos can be defined. Each lifo is referenced by its memory address.

A lifo has the following key properties:

- A **queue** of data items that have been added but not yet removed. The queue is implemented as a simple linked list.

A lifo must be initialized before it can be used. This sets its queue to empty.

Lifo data items must be aligned on a 4-byte boundary, as the kernel reserves the first 32 bits of an item for use as a pointer to the next data item in the queue. Consequently, a data item that holds N bytes of application data requires N+4 bytes of memory.

A data item may be **added** to a lifo by a thread or an ISR. The item is given directly to a waiting thread, if one exists; otherwise the item is added to the lifo's queue. There is no limit to the number of items that may be queued.

A data item may be **removed** from a lifo by a thread. If the lifo's queue is empty a thread may choose to wait for a data item to be given. Any number of threads may wait on an empty lifo simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

注解: The kernel does allow an ISR to remove an item from a lifo, however the ISR must not attempt to wait if the lifo is empty.

Implementation

Defining a Lifo

A lifo is defined using a variable of type `struct k_lifo`. It must then be initialized by calling `k_lifo_init()`.

The following defines and initializes an empty lifo.

```
struct k_lifo my_lifo;

k_lifo_init(&my_lifo);
```

Alternatively, an empty lifo can be defined and initialized at compile time by calling `K_LIFO_DEFINE`.

The following code has the same effect as the code segment above.

```
K_LIFO_DEFINE(my_lifo);
```

Writing to a Lifo

A data item is added to a lifo by calling `k_lifo_put()`.

The following code builds on the example above, and uses the lifo to send data to one or more consumer threads.

```
struct data_item_t {
    void *lifo_reserved;    /* 1st word reserved for use by lifo */
    ...
};

struct data_item_t tx data;

void producer_thread(int unused1, int unused2, int unused3)
{
    while (1) {
        /* create data item to send */
        tx_data = ...

        /* send data to consumers */
        k_lifo_put(&my_lifo, &tx_data);

        ...
    }
}
```

Reading from a Lifo

A data item is removed from a lifo by calling `k_lifo_get()`.

The following code builds on the example above, and uses the lifo to obtain data items from a producer thread, which are then processed in some manner.

```
void consumer_thread(int unused1, int unused2, int unused3)
{
    struct data_item_t  *rx_data;

    while (1) {
        rx_data = k_lifo_get(&my_lifo, K_FOREVER);

        /* process lifo data item */
        ...
    }
}
```

Suggested Uses

Use a lifo to asynchronously transfer data items of arbitrary size in a "last in, first out" manner.

Configuration Options

Related configuration options:

- None.

APIs

The following lifo APIs are provided by `kernel.h`:

- `K_LIFO_DEFINE`
- `k_lifo_init()`
- `k_lifo_put()`
- `k_lifo_get()`

Stacks

A *stack* is a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove a limited number of 32-bit data values.

Concepts

Any number of stacks can be defined. Each stack is referenced by its memory address.

A stack has the following key properties:

- A **queue** of 32-bit data values that have been added but not yet removed. The queue is implemented using an array of 32-bit integers, and must be aligned on a 4-byte boundary.
- A **maximum quantity** of data values that can be queued in the array.

A stack must be initialized before it can be used. This sets its queue to empty.

A data value can be **added** to a stack by a thread or an ISR. The value is given directly to a waiting thread, if one exists; otherwise the value is added to the lifo's queue. The kernel does *not* detect attempts to add a data value to a stack that has already reached its maximum quantity of queued values.

注解: Adding a data value to a stack that is already full will result in array overflow, and lead to unpredictable behavior.

A data value may be **removed** from a stack by a thread. If the stack's queue is empty a thread may choose to wait for it to be given. Any number of threads may wait on an empty stack simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

注解: The kernel does allow an ISR to remove an item from a stack, however the ISR must not attempt to wait if the stack is empty.

Implementation

Defining a Stack

A stack is defined using a variable of type `struct k_stack`. It must then be initialized by calling `k_stack_init()`.

The following code defines and initializes an empty stack capable of holding up to ten 32-bit data values.

```
#define MAX_ITEMS 10

u32_t my_stack_array[MAX_ITEMS];
struct k_stack my_stack;

k_stack_init(&my_stack, my_stack_array, MAX_ITEMS);
```

Alternatively, a stack can be defined and initialized at compile time by calling `K_STACK_DEFINE`.

The following code has the same effect as the code segment above. Observe that the macro defines both the stack and its array of data values.

```
K_STACK_DEFINE(my_stack, MAX_ITEMS);
```

Pushing to a Stack

A data item is added to a stack by calling `k_stack_push()`.

The following code builds on the example above, and shows how a thread can create a pool of data structures by saving their memory addresses in a stack.

```
/* define array of data structures */
struct my_buffer_type {
    int field1;
    ...
    };
struct my_buffer_type my_buffers[MAX_ITEMS];

/* save address of each data structure in a stack */
for (int i = 0; i < MAX_ITEMS; i++) {
    k_stack_push(&my_stack, (u32_t)&my_buffers[i]);
}
```

Popping from a Stack

A data item is taken from a stack by calling `k_stack_pop()`.

The following code builds on the example above, and shows how a thread can dynamically allocate an unused data structure. When the data structure is no longer required, the thread must push its address back on the stack to allow the data structure to be reused.

```
struct my_buffer_type *new_buffer;

k_stack_pop(&buffer_stack, (u32_t *)&new_buffer, K_FOREVER);
new_buffer->field1 = ...
```

Suggested Uses

Use a stack to store and retrieve 32-bit data values in a "last in, first out" manner, when the maximum number of stored items is known.

Configuration Options

Related configuration options:

- None.

APIs

The following stack APIs are provided by `kernel.h`:

- `K_STACK_DEFINE`
- `k_stack_init()`
- `k_stack_push()`
- `k_stack_pop()`

Message Queues

A *message queue* is a kernel object that implements a simple message queue, allowing threads and ISRs to asynchronously send and receive fixed-size data items.

Concepts

Any number of message queues can be defined. Each message queue is referenced by its memory address.

A message queue has the following key properties:

- A **ring buffer** of data items that have been sent but not yet received.
- A **data item size**, measured in bytes.
- A **maximum quantity** of data items that can be queued in the ring buffer.

The message queue's ring buffer must be aligned to an N-byte boundary, where N is a power of 2 (i.e. 1, 2, 4, 8, ...). To ensure that the messages stored in the ring buffer are similarly aligned to this boundary, the data item size must also be a multiple of N.

A message queue must be initialized before it can be used. This sets its ring buffer to empty.

A data item can be **sent** to a message queue by a thread or an ISR. The data item a pointed at by the sending thread is copied to a waiting thread, if one exists; otherwise the item is copied to the message queue's ring buffer, if space is available. In either case, the size of the data area being sent *must* equal the message queue's data item size.

If a thread attempts to send a data item when the ring buffer is full, the sending thread may choose to wait for space to become available. Any number of sending threads may wait simultaneously when the ring buffer is full; when space becomes available it is given to the highest priority sending thread that has waited the longest.

A data item can be **received** from a message queue by a thread. The data item is copied to the area specified by the receiving thread; the size of the receiving area *must* equal the message queue's data item size.

If a thread attempts to receive a data item when the ring buffer is empty, the receiving thread may choose to wait for a data item to be sent. Any number of receiving threads may wait simultaneously when the ring buffer is empty; when a data item becomes available it is given to the highest priority receiving thread that has waited the longest.

注解: The kernel does allow an ISR to receive an item from a message queue, however the ISR must not attempt to wait if the message queue is empty.

Implementation

Defining a Message Queue

A message queue is defined using a variable of type `struct k_msgq`. It must then be initialized by calling `k_msgq_init()`.

The following code defines and initializes an empty message queue that is capable of holding 10 items, each of which is 12 bytes long.

```
struct data_item_type {
    u32_t field1;
    u32_t field2;
    u32_t field3;
};

char __aligned(4) my_msgq_buffer[10 * sizeof(data_item_type)];
struct k_msgq my_msgq;

k_msgq_init(&my_msgq, my_msgq_buffer, sizeof(data_item_type), 10);
```

Alternatively, a message queue can be defined and initialized at compile time by calling `K_MSGQ_DEFINE`.

The following code has the same effect as the code segment above. Observe that the macro defines both the message queue and its buffer.

```
K_MSGQ_DEFINE(my_msgq, sizeof(data_item_type), 10, 4);
```

Writing to a Message Queue

A data item is added to a message queue by calling `k_msgq_put()`.

The following code builds on the example above, and uses the message queue to pass data items from a producing thread to one or more consuming threads. If the message queue fills up because the consumers can't keep up, the producing thread throws away all existing data so the newer data can be saved.

```
void producer_thread(void)
{
    struct data_item_t data;

    while (1) {
        /* create data item to send (e.g. measurement, timestamp, ...) */
        data = ...

        /* send data to consumers */
        while (k_msgq_put(&my_msgq, &data, K_NO_WAIT) != 0) {
            /* message queue is full: purge old data & try again */
            k_msgq_purge(&my_msgq);
        }

        /* data item was successfully added to message queue */
    }
}
```

Reading from a Message Queue

A data item is taken from a message queue by calling `k_msgq_get()`.

The following code builds on the example above, and uses the message queue to process data items generated by one or more producing threads.

```
void consumer_thread(void)
{
    struct data_item_t data;

    while (1) {
        /* get a data item */
        k_msgq_get(&my_msgq, &data, K_FOREVER);

        /* process data item */
        ...
    }
}
```

Suggested Uses

Use a message queue to transfer small data items between threads in an asynchronous manner.

注解: A message queue can be used to transfer large data items, if desired. However, this can increase interrupt latency as interrupts are locked while a data item is written or read. It is usually preferable to transfer large data items by exchanging a pointer to the data item, rather than the data item itself. The kernel's memory map and memory pool object types can be helpful for data transfers of this sort.

A synchronous transfer can be achieved by using the kernel's mailbox object type.

Configuration Options

Related configuration options:

- None.

APIs

The following message queue APIs are provided by `kernel.h`:

- `K_MSGQ_DEFINE`

- `k_msgq_init()`
- `k_msgq_put()`
- `k_msgq_get()`
- `k_msgq_purge()`
- `k_msgq_num_used_get()`
- `k_msgq_num_free_get()`

Mailboxes

A *mailbox* is a kernel object that provides enhanced message queue capabilities that go beyond the capabilities of a message queue object. A mailbox allows threads to send and receive messages of any size synchronously or asynchronously.

Concepts

Any number of mailboxes can be defined. Each mailbox is referenced by its memory address.

A mailbox has the following key properties:

- A **send queue** of messages that have been sent but not yet received.
- A **receive queue** of threads that are waiting to receive a message.

A mailbox must be initialized before it can be used. This sets both of its queues to empty.

A mailbox allows threads, but not ISRs, to exchange messages. A thread that sends a message is known as the **sending thread**, while a thread that receives the message is known as the **receiving thread**. Each message may be received by only one thread (i.e. point-to-multipoint and broadcast messaging is not supported).

Messages exchanged using a mailbox are handled non-anonymously, allowing both threads participating in an exchange to know (and even specify) the identity of the other thread.

Message Format

A **message descriptor** is a data structure that specifies where a message's data is located, and how the message is to be handled by the mailbox. Both the sending thread and the receiving thread supply a message descriptor when accessing a mailbox. The mailbox uses the message descriptors to perform a message exchange between compatible sending and receiving threads. The mailbox also updates certain message descriptor fields during the exchange, allowing both threads to know what has occurred.

A mailbox message contains zero or more bytes of **message data**. The size and format of the message data is application-defined, and can vary from one message to the next. There are two forms of message data:

- A **message buffer** is an area of memory provided by the thread that sends or receives the message. An array or structure variable can often be used for this purpose.
- A **message block** is an area of memory allocated from a memory pool.

A message may *not* have both a message buffer and a message block. A message that has neither form of message data is called an **empty message**.

注解: A message whose message buffer or memory block exists, but contains zero bytes of actual data, is *not* an empty message.

Message Lifecycle

The life cycle of a message is straightforward. A message is created when it is given to a mailbox by the sending thread. The message is then owned by the mailbox until it is given to a receiving thread. The receiving thread may retrieve the message data when it receives the message from the mailbox, or it may perform data retrieval during a second, subsequent mailbox operation. Only when data retrieval has occurred is the message deleted by the mailbox.

Thread Compatibility

A sending thread can specify the address of the thread to which the message is sent, or it send it to any thread by specifying `K_ANY`. Likewise, a receiving thread can specify the address of the thread from which it wishes to receive a message, or it can receive a message from any thread by specifying `K_ANY`. A message is exchanged only when the requirements of both the sending thread and receiving thread are satisfied; such threads are said to be **compatible**.

For example, if thread A sends a message to thread B (and only thread B) it will be received by thread B if thread B tries to receive a message from thread A or if thread B tries to receive from any thread. The exchange will not occur if thread B tries to receive a message from thread C. The message can never be received by thread C, even if it tries to receive a message from thread A (or from any thread).

Message Flow Control

Mailbox messages can be exchanged **synchronously** or **asynchronously**. In a synchronous exchange, the sending thread blocks until the message has been fully processed by the receiving thread. In an asynchronous exchange, the sending thread does not wait until the message has been received by another thread before continuing; this allows the sending thread to do other work (such as gather data that will be used in the next message) *before* the message is given to a receiving thread and fully processed. The technique used for a given message exchange is determined by the sending thread.

The synchronous exchange technique provides an implicit form of flow control, preventing a sending thread from generating messages faster than they can be consumed by receiving threads. The asynchronous exchange technique provides an explicit form of flow control, which allows a sending thread to determine if a previously sent message still exists before sending a subsequent message.

Implementation

Defining a Mailbox

A mailbox is defined using a variable of type `struct k_mbox`. It must then be initialized by calling `k_mbox_init()`.

The following code defines and initializes an empty mailbox.

```
struct k_mbox my_mailbox;

k_mbox_init(&my_mailbox);
```

Alternatively, a mailbox can be defined and initialized at compile time by calling `K_MBOX_DEFINE`.

The following code has the same effect as the code segment above.

```
K_MBOX_DEFINE(my_mailbox);
```

Message Descriptors

A message descriptor is a structure of type `struct k_mbox_msg`. Only the fields listed below should be used; any other fields are for internal mailbox use only.

info A 32-bit value that is exchanged by the message sender and receiver, and whose meaning is defined by the application. This exchange is bi-directional, allowing the sender to pass a value to the receiver during any message exchange, and allowing the receiver to pass a value to the sender during a synchronous message exchange.

size The message data size, in bytes. Set it to zero when sending an empty message, or when sending a message buffer or message block with no actual data. When receiving a message, set it to the maximum amount of data desired, or to zero if the message data is not wanted. The mailbox updates this field with the actual number of data bytes exchanged once the message is received.

tx_data A pointer to the sending thread's message buffer. Set it to `NULL` when sending a memory block, or when sending an empty message. Leave this field uninitialized when receiving a message.

tx_block The descriptor for the sending thread's memory block. Set tx_block.pool_id to `NULL` when sending an empty message. Leave this field uninitialized when sending a message buffer, or when receiving a message.

tx_target_thread The address of the desired receiving thread. Set it to `K_ANY` to allow any thread to receive the message. Leave this field uninitialized when receiving a message. The mailbox updates this field with the actual receiver's address once the message is received.

rx_source_thread The address of the desired sending thread. Set it to `K_ANY` to receive a message sent by any thread. Leave this field uninitialized when sending a message. The mailbox updates this field with the actual sender's address once the message is received.

Sending a Message

A thread sends a message by first creating its message data, if any. A message buffer is typically used when the data volume is small, and the cost of copying the data is less than the cost of allocating and freeing a message block.

Next, the sending thread creates a message descriptor that characterizes the message to be sent, as described in the previous section.

Finally, the sending thread calls a mailbox send API to initiate the message exchange. The message is immediately given to a compatible receiving thread, if one is currently waiting. Otherwise, the message is added to the mailbox's send queue.

Any number of messages may exist simultaneously on a send queue. The messages in the send queue are sorted according to the priority of the sending thread. Messages of equal priority are sorted so that the oldest message can be received first.

For a synchronous send operation, the operation normally completes when a receiving thread has both received the message and retrieved the message data. If the message is not received before the waiting period specified by the sending thread is reached, the message is removed from the mailbox's send queue and the send operation fails. When a send operation completes successfully the sending thread can examine the message descriptor to determine which thread received the message, how much data was exchanged, and the application-defined info value supplied by the receiving thread.

注解: A synchronous send operation may block the sending thread indefinitely, even when the thread specifies a maximum waiting period. The waiting period only limits how long the mailbox waits before the message is received by another thread. Once a message is received there is *no* limit to the time the receiving thread may take to retrieve the message data and unblock the sending thread.

For an asynchronous send operation, the operation always completes immediately. This allows the sending thread to continue processing regardless of whether the message is given to a receiving thread immediately or added to the send queue. The sending thread may optionally specify a semaphore that the mailbox gives when the message is deleted by the mailbox, for example, when the message has been received and its data retrieved by a receiving thread. The use of a semaphore allows the sending thread to easily implement a flow control mechanism that ensures that the mailbox holds no more than an application-specified number of messages from a sending thread (or set of sending threads) at any point in time.

注解: A thread that sends a message asynchronously has no way to determine which thread received the message, how much data was exchanged, or the application-defined info value supplied by the receiving thread.

Sending an Empty Message

This code uses a mailbox to synchronously pass 4 byte random values to any consuming thread that wants one. The message "info" field is large enough to carry the information being exchanged, so the data portion of the message isn't used.

```
void producer_thread(void)
{
    struct k_mbox_msg send_msg;

    while (1) {

        /* generate random value to send */
        u32_t random_value = sys_rand32_get();

        /* prepare to send empty message */
        send_msg.info = random_value;
        send_msg.size = 0;
        send_msg.tx_data = NULL;
        send_msg.tx_block.pool_id = NULL;
        send_msg.tx_target_thread = K_ANY;

        /* send message and wait until a consumer receives it */
        k_mbox_put(&my_mailbox, &send_msg, K_FOREVER);
    }
}
```

Sending Data Using a Message Buffer

This code uses a mailbox to synchronously pass variable-sized requests from a producing thread to any consuming thread that wants it. The message "info" field is used to exchange information about the maximum size message buffer that each thread can handle.

```
void producer_thread(void)
{
    char buffer[100];
    int buffer_bytes_used;

    struct k_mbox_msg send_msg;

    while (1) {

        /* generate data to send */
        ...
        buffer_bytes_used = ... ;
        memcpy(buffer, source, buffer_bytes_used);

        /* prepare to send message */
        send_msg.info = buffer_bytes_used;
        send_msg.size = buffer_bytes_used;
        send_msg.tx_data = buffer;
        send_msg.tx_target_thread = K_ANY;

        /* send message and wait until a consumer receives it */
        k_mbox_put(&my_mailbox, &send_msg, K_FOREVER);

        /* info, size, and tx_target_thread fields have been updated */

        /* verify that message data was fully received */
        if (send_msg.size < buffer_bytes_used) {
            printf("some message data dropped during transfer!");
            printf("receiver only had room for %d bytes", send_msg.info);
        }
    }
}
```

Sending Data Using a Message Block

This code uses a mailbox to send asynchronous messages. A semaphore is used to hold off the sending of a new message until the previous message has been consumed, so that a backlog of messages doesn't build up when the consuming thread is unable to keep up.

The message data is stored in a memory block obtained from a memory pool, thereby eliminating unneeded data copying when exchanging large messages. The memory pool contains only two blocks: one block gets filled with data while the previously sent block is being processed

```
/* define a semaphore, indicating that no message has been sent */
K_SEM_DEFINE(my_sem, 1, 1);

/* define a memory pool containing 2 blocks of 4096 bytes */
K_MEM_POOL_DEFINE(my_pool, 4096, 4096, 2, 4);

void producer_thread(void)
{
    struct k_mbox_msg send_msg;

    volatile char *hw_buffer;

    while (1) {
        /* allocate a memory block to hold the message data */
        k_mem_pool_alloc(&mp_pool, &send_msg.tx_block, 4096, K_FOREVER);

        /* keep overwriting the hardware-generated data in the block    */
        /* until the previous message has been received by the consumer */
        do {
            memcpy(send_msg.tx_block.data, hw_buffer, 4096);
        } while (k_sem_take(&my_sem, K_NO_WAIT) != 0);

        /* finish preparing to send message */
        send_msg.size = 4096;
        send_msg.tx_target_thread = K_ANY;

        /* send message containing most current data and loop around */
        k_mbox_async_put(&my_mailbox, &send_msg, &my_sem);
    }
}
```

Receiving a Message

A thread receives a message by first creating a message descriptor that characterizes the message it wants to receive. It then calls one of the mailbox receive APIs. The mailbox searches its send queue and takes the message from the first compatible thread it finds. If no compatible thread exists, the receiving thread may choose to wait for one. If no compatible thread appears before the waiting period specified by the receiving thread is reached, the receive operation fails. Once a receive operation completes successfully the receiving thread can examine the message descriptor to determine which thread sent the message, how much data was exchanged, and the application-defined info value supplied by the sending thread.

Any number of receiving threads may wait simultaneously on a mailboxes' receive queue. The threads are sorted according to their priority; threads of equal priority are sorted so that the one that started waiting first can receive a message first.

注解： Receiving threads do not always receive messages in a first in, first out (FIFO) order, due to the thread compatibility constraints specified by the message descriptors. For example, if thread A waits to receive a message only from thread X and then thread B waits to receive a message from thread Y, an incoming message from thread Y to any thread will be given to thread B and thread A will continue to wait.

The receiving thread controls both the quantity of data it retrieves from an incoming message and where the data ends up. The thread may choose to take all of the data in the message, to take only the initial part of the data, or to take no data at all. Similarly, the thread may choose to have the data copied into a message buffer of its choice or to have it placed in a message block. A message buffer is typically used when the volume of data involved is small, and the cost of copying the data is less than the cost of allocating and freeing a memory pool block.

The following sections outline various approaches a receiving thread may use when retrieving message data.

Retrieving Data at Receive Time

The most straightforward way for a thread to retrieve message data is to specify a message buffer when the message is received. The thread indicates both the location of the message buffer (which must not be `NULL`) and its size.

The mailbox copies the message's data to the message buffer as part of the receive operation. If the message buffer is not big enough to contain all of the message's data, any uncopied data is lost. If the message is not big enough to fill all of the buffer with data, the unused portion of the message buffer is left unchanged. In all cases the mailbox updates the receiving thread's message descriptor to indicate how many data bytes were copied (if any).

The immediate data retrieval technique is best suited for small messages where the maximum size of a message is known in advance.

注解: This technique can be used when the message data is actually located in a memory block supplied by the sending thread. The mailbox copies the data into the message buffer specified by the receiving thread, then frees the message block back to its memory pool. This allows a receiving thread to retrieve message data without having to know whether the data was sent using a message buffer or a message block.

The following code uses a mailbox to process variable-sized requests from any producing thread, using the immediate data retrieval technique. The message "info" field is used to exchange information about the maximum size message buffer that each thread can handle.

```c
void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    char buffer[100];

    int i;
    int total;

    while (1) {
        /* prepare to receive message */
        recv_msg.info = 100;
        recv_msg.size = 100;
        recv_msg.rx_source_thread = K_ANY;

        /* get a data item, waiting as long as needed */
        k_mbox_get(&my_mailbox, &recv_msg, buffer, K_FOREVER);

        /* info, size, and rx_source_thread fields have been updated */

        /* verify that message data was fully received */
        if (recv_msg.info != recv_msg.size) {
            printf("some message data dropped during transfer!");
            printf("sender tried to send %d bytes", recv_msg.info);
        }

        /* compute sum of all message bytes (from 0 to 100 of them) */
        total = 0;
        for (i = 0; i < recv_msg.size; i++) {
            total += buffer[i];
        }
    }
}
```

Retrieving Data Later Using a Message Buffer

A receiving thread may choose to defer message data retrieval at the time the message is received, so that it can retrieve the data into a message buffer at a later time. The thread does this by specifying a message buffer location of `NULL` and a size indicating the maximum amount of data it is willing to retrieve later.

The mailbox does not copy any message data as part of the receive operation. However, the mailbox still updates the receiving thread's message descriptor to indicate how many data bytes are available for retrieval.

The receiving thread must then respond as follows:

- If the message descriptor size is zero, then either the sender's message contained no data or the receiving thread did not want to receive any data. The receiving thread does not need to take any further action, since the mailbox has already completed data retrieval and deleted the message.
- If the message descriptor size is non-zero and the receiving thread still wants to retrieve the data, the thread must call `k_mbox_data_get()` and supply a message buffer large enough to hold the data. The mailbox copies the data into the message buffer and deletes the message.
- If the message descriptor size is non-zero and the receiving thread does *not* want to retrieve the data, the thread must call `k_mbox_data_get()`. and specify a message buffer of `NULL`. The mailbox deletes the message without copying the data.

The subsequent data retrieval technique is suitable for applications where immediate retrieval of message data is undesirable. For example, it can be used when memory limitations make it impractical for the receiving thread to always supply a message buffer capable of holding the largest possible incoming message.

注解: This technique can be used when the message data is actually located in a memory block supplied by the sending thread. The mailbox copies the data into the message buffer specified by the receiving thread, then frees the message block back to its memory pool. This allows a receiving thread to retrieve message data without having to know whether the data was sent using a message buffer or a message block.

The following code uses a mailbox's deferred data retrieval mechanism to get message data from a producing thread only if the message meets certain criteria, thereby eliminating unneeded data copying. The message "info" field supplied by the sender is used to classify the message.

```
void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    char buffer[10000];

    while (1) {
        /* prepare to receive message */
        recv_msg.size = 10000;
        recv_msg.rx_source_thread = K_ANY;

        /* get message, but not its data */
        k_mbox_get(&my_mailbox, &recv_msg, NULL, K_FOREVER);

        /* get message data for only certain types of messages */
        if (is_message_type_ok(recv_msg.info)) {
            /* retrieve message data and delete the message */
            k_mbox_data_get(&recv_msg, buffer);

            /* process data in "buffer" */
            ...
        } else {
            /* ignore message data and delete the message */
            k_mbox_data_get(&recv_msg, NULL);
        }
    }
}
```

Retrieving Data Later Using a Message Block

A receiving thread may choose to retrieve message data into a memory block, rather than a message buffer. This is done in much the same way as retrieving data subsequently into a message buffer — the receiving thread first receives the message without its data, then retrieves the data by calling `k_mbox_data_block_get()`. The mailbox fills in the block descriptor supplied by the receiving thread, allowing the thread to access the data. The mailbox also deletes the received message, since data retrieval has been completed. The receiving thread is then responsible for freeing the message block back to the memory pool when the data is no longer needed.

This technique is best suited for applications where the message data has been sent using a memory block.

注解: This technique can be used when the message data is located in a message buffer supplied by the sending thread. The mailbox automatically allocates a memory block and copies the message data into it. However, this is much less efficient than simply retrieving the data into a message buffer supplied by the receiving thread. In addition, the receiving thread must be designed to handle cases where the data retrieval operation fails because the mailbox cannot allocate a suitable message block from the memory pool. If such cases are possible, the receiving thread must either try retrieving the data at a later time or instruct the mailbox to delete the message without retrieving the data.

The following code uses a mailbox to receive messages sent using a memory block, thereby eliminating unneeded data copying when processing a large message. (The messages may be sent synchronously or asynchronously.)

```
/* define a memory pool containing 1 block of 10000 bytes */
K_MEM_POOL_DEFINE(my_pool, 10000, 10000, 1, 4);

void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    struct k_mem_block recv_block;

    int total;
    char *data_ptr;
    int i;

    while (1) {
        /* prepare to receive message */
        recv_msg.size = 10000;
        recv_msg.rx_source_thread = K_ANY;

        /* get message, but not its data */
        k_mbox_get(&my_mailbox, &recv_msg, NULL, K_FOREVER);
        /* get message data as a memory block and discard message */
        k_mbox_data_block_get(&recv_msg, &my_pool, &recv_block, K_FOREVER);

        /* compute sum of all message bytes in memory block */
        total = 0;
        data_ptr = (char *)(recv_block.data);
        for (i = 0; i < recv_msg.size; i++) {
            total += data_ptr++;
        }

        /* release memory block containing data */
        k_mem_pool_free(&recv_block);
    }
}
```

注解: An incoming message that was sent using a message buffer is also processed correctly by this algorithm, since the mailbox automatically allocates a memory block from the memory pool and fills it with the message data. However, the performance benefit of using the memory block approach is lost.

Suggested Uses

Use a mailbox to transfer data items between threads whenever the capabilities of a message queue are insufficient.

Configuration Options

Related configuration options:

- `CONFIG_NUM_MBOX_ASYNC_MSGS`

APIs

The following APIs for a mailbox are provided by `kernel.h`:

- `K_MBOX_DEFINE`
- `k_mbox_init()`
- `k_mbox_put()`
- `k_mbox_async_put()`
- `k_mbox_get()`
- `k_mbox_data_get()`
- `k_mbox_data_block_get()`

Pipes

A *pipe* is a kernel object that allows a thread to send a byte stream to another thread. Pipes can be used to transfer chunks of data in whole or in part, and either synchronously or asynchronously.

Concepts

The pipe can be configured with a ring buffer which holds data that has been sent but not yet received; alternatively, the pipe may have no ring buffer.

Any number of pipes can be defined. Each pipe is referenced by its memory address.

A pipe has the following key property:

- A **size** that indicates the size of the pipe's ring buffer. Note that a size of zero defines a pipe with no ring buffer.

A pipe must be initialized before it can be used. The pipe is initially empty.

Data can be synchronously **sent** either in whole or in part to a pipe by a thread. If the specified minimum number of bytes can not be immediately satisfied, then the operation will either fail immediately or attempt to send as many bytes as possible and then pend in the hope that the send can be completed later. Accepted data is either copied to the pipe's ring buffer or directly to the waiting reader(s).

Data can be asynchronously **sent** in whole using a memory block to a pipe by a thread. Once the pipe has accepted all the bytes in the memory block, it will free the memory block and may give a semaphore if one was specified.

Data can be synchronously **received** from a pipe by a thread. If the specified minimum number of bytes can not be immediately satisfied, then the operation will either fail immediately or attempt to receive as many bytes as possible and then pend in the hope that the receive can be completed later. Accepted data is either copied from the pipe's ring buffer or directly from the waiting sender(s).

注解: The kernel does NOT allow for an ISR to send or receive data to/from a pipe even if it does not attempt to wait for space/data.

Implementation

A pipe is defined using a variable of type `struct k_pipe` and an optional character buffer of type `unsigned char`. It must then be initialized by calling `k_pipe_init()`.

The following code defines and initializes an empty pipe that has a ring buffer capable of holding 100 bytes and is aligned to a 4-byte boundary.

```
unsigned char __aligned(4) my_ring_buffer[100];
struct k_pipe my_pipe;

k_pipe_init(&my_pipe, my_ring_buffer, sizeof(my_ring_buffer));
```

Alternatively, a pipe can be defined and initialized at compile time by calling K_PIPE_DEFINE.

The following code has the same effect as the code segment above. Observe that that macro defines both the pipe and its ring buffer.

```
K_PIPE_DEFINE(my_pipe, 100, 4);
```

Writing to a Pipe

Data is added to a pipe by calling k_pipe_put().

The following code builds on the example above, and uses the pipe to pass data from a producing thread to one or more consuming threads. If the pipe's ring buffer fills up because the consumers can't keep up, the producing thread waits for a specified amount of time.

```
struct message_header {
    ...
};

void producer_thread(void)
{
    unsigned char *data;
    size_t total_size;
    size_t bytes_written;
    int    rc;
    ...

    while (1) {
        /* Craft message to send in the pipe */
        data = ...;
        total_size = ...;

        /* send data to the consumers */
        rc = k_pipe_put(&my_pipe, data, total_size, &bytes_written,
                        sizeof(struct message_header), K_NO_WAIT);

        if (rc < 0) {
            /* Incomplete message header sent */
            ...
        } else if (bytes_written < total_size) {
            /* Some of the data was sent */
            ...
        } else {
            /* All data sent */
            ...
        }
    }
}
```

Reading from a Pipe

Data is read from the pipe by calling k_pipe_get().

The following code builds on the example above, and uses the pipe to process data items generated by one or more producing threads.

```
void consumer_thread(void)
{
    unsigned char buffer[120];
    size_t   bytes_read;
    struct message_header  *header = (struct message_header *)buffer;

    while (1) {
        rc = k_pipe_get(&my_pipe, buffer, sizeof(buffer), &bytes_read,
                        sizeof(header), K_MSEC(100));

        if ((rc < 0) || (bytes_read < sizeof (header))) {
            /* Incomplete message header received */
            ...
        } else if (header->num_data_bytes + sizeof(header) > bytes_read) {
            /* Only some data was received */
            ...
        } else {
            /* All data was received */
            ...
        }
    }
}
```

Suggested uses

Use a pipe to send streams of data between threads.

注解: A pipe can be used to transfer long streams of data if desired. However it is often preferable to send pointers to large data items to avoid copying the data. The kernel's memory map and memory pool object types can be helpful for data transfers of this sort.

Configuration Options

Related configuration options:

- `CONFIG_NUM_PIPE_ASYNC_MSGS`

APIs

The following message queue APIs are provided by `kernel.h`:

- `K_PIPE_DEFINE`
- `k_pipe_init()`
- `k_pipe_put()`
- `k_pipe_get()`
- `k_pipe_block_put()`

Other Services

This section describes other services provided by the kernel.

Interrupts

An *interrupt service routine* (ISR) is a function that executes asynchronously in response to a hardware or software interrupt. An ISR normally preempts the execution of the current thread, allowing the response to occur with very low overhead. Thread execution resumes only once all ISR work has been completed.

Concepts

Any number of ISRs can be defined, subject to the constraints imposed by underlying hardware.

An ISR has the following key properties:

- An **interrupt request (IRQ) signal** that triggers the ISR.
- A **priority level** associated with the IRQ.
- An **interrupt handler function** that is invoked to handle the interrupt.
- An **argument value** that is passed to that function.

An IDT (Interrupt Descriptor Table) or a vector table is used to associate a given interrupt source with a given ISR. Only a single ISR can be associated with a specific IRQ at any given time.

Multiple ISRs can utilize the same function to process interrupts, allowing a single function to service a device that generates multiple types of interrupts or to service multiple devices (usually of the same type). The argument value passed to an ISR's function allows the function to determine which interrupt has been signaled.

The kernel provides a default ISR for all unused IDT entries. This ISR generates a fatal system error if an unexpected interrupt is signaled.

The kernel supports **interrupt nesting**. This allows an ISR to be preempted in mid-execution if a higher priority interrupt is signaled. The lower priority ISR resumes execution once the higher priority ISR has completed its processing.

An ISR's interrupt handler function executes in the kernel's **interrupt context**. This context has its own dedicated stack area (or, on some architectures, stack areas). The size of the interrupt context stack must be capable of handling the execution of multiple concurrent ISRs if interrupt nesting support is enabled.

重要： Many kernel APIs can be used only by threads, and not by ISRs. In cases where a routine may be invoked by both threads and ISRs the kernel provides the `k_is_in_isr()` API to allow the routine to alter its behavior depending on whether it is executing as part of a thread or as part of an ISR.

Preventing Interruptions

In certain situations it may be necessary for the current thread to prevent ISRs from executing while it is performing time-sensitive or critical section operations.

A thread may temporarily prevent all IRQ handling in the system using an **IRQ lock**. This lock can be applied even when it is already in effect, so routines can use it without having to know if it is already in effect. The thread must unlock its IRQ lock the same number of times it was locked before interrupts can be once again processed by the kernel while the thread is running.

重要： The IRQ lock is thread-specific. If thread A locks out interrupts then performs an operation that allows thread B to run (e.g. giving a semaphore or sleeping for N milliseconds), the thread's IRQ lock no longer applies once thread A is swapped out. This means that interrupts can be processed while thread B is running unless thread B has also locked out interrupts using its own IRQ lock. (Whether interrupts can be processed while the kernel is switching between two threads that are using the IRQ lock is architecture-specific.)

When thread A eventually becomes the current thread once again, the kernel re-establishes thread A's IRQ lock. This ensures thread A won't be interrupted until it has explicitly unlocked its IRQ lock.

Alternatively, a thread may temporarily **disable** a specified IRQ so its associated ISR does not execute when the IRQ is signaled. The IRQ must be subsequently **enabled** to permit the ISR to execute.

重要： Disabling an IRQ prevents *all* threads in the system from being preempted by the associated ISR, not just the thread that disabled the IRQ.

Offloading ISR Work

An ISR should execute quickly to ensure predictable system operation. If time consuming processing is required the ISR should offload some or all processing to a thread, thereby restoring the kernel's ability to respond to other interrupts.

The kernel supports several mechanisms for offloading interrupt-related processing to a thread.

- An ISR can signal a helper thread to do interrupt-related processing using a kernel object, such as a fifo, lifo, or semaphore.
- An ISR can signal an alert which causes the system workqueue thread to execute an associated alert handler function. (See *Alerts*.)
- An ISR can instruct the system workqueue thread to execute a work item. (See TBD.)

When an ISR offloads work to a thread, there is typically a single context switch to that thread when the ISR completes, allowing interrupt-related processing to continue almost immediately. However, depending on the priority of the thread handling the offload, it is possible that the currently executing cooperative thread or other higher-priority threads may execute before the thread handling the offload is scheduled.

Implementation

Defining a regular ISR

An ISR is defined at run-time by calling `IRQ_CONNECT`. It must then be enabled by calling `irq_enable()`.

重要: IRQ_CONNECT() is not a C function and does some inline assembly magic behind the scenes. All its arguments must be known at build time. Drivers that have multiple instances may need to define per-instance config functions to configure each instance of the interrupt.

The following code defines and enables an ISR.

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_ISR_ARG  DEVICE_GET(my_device)
#define MY_IRQ_FLAGS 0       /* IRQ flags. Unused on non-x86 */

void my_isr(void *arg)
{
   ... /* ISR code */
}

void my_isr_installer(void)
{
   ...
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_ISR_ARG, MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Defining a 'direct' ISR

Regular Zephyr interrupts introduce some overhead which may be unacceptable for some low-latency use-cases. Specifically:

- The argument to the ISR is retrieved and passed to the ISR
- If power management is enabled and the system was idle, all the hardware will be resumed from low-power state before the ISR is executed, which can be very time-consuming
- Although some architectures will do this in hardware, other architectures need to switch to the interrupt stack in code
- After the interrupt is serviced, the OS then performs some logic to potentially make a scheduling decision.

Zephyr supports so-called 'direct' interrupts, which are installed via `IRQ_DIRECT_CONNECT`. These direct interrupts have some special implementation requirements and a reduced feature set; see the definition of `IRQ_DIRECT_CONNECT` for details.

The following code demonstrates a direct ISR:

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_IRQ_FLAGS 0       /* IRQ flags. Unused on non-x86 */
```

```
ISR_DIRECT_DECLARE(my_isr)
{
   do_stuff();
   ISR_DIRECT_PM(); /* PM done after servicing interrupt for best latency */
   return 1; /* We should check if scheduling decision should be made */
}

void my_isr_installer(void)
{
   ...
   IRQ_DIRECT_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Implementation Details

Interrupt tables are set up at build time using some special build tools. The details laid out here apply to all architectures except x86, which are covered in the *x86 Details* section below.

Any invocation of `IRQ_CONNECT` will declare an instance of struct _isr_list which is placed in a special .intList section:

```
struct _isr_list {
    /** IRQ line number */
    s32_t irq;
    /** Flags for this IRQ, see ISR_FLAG_* definitions */
    s32_t flags;
    /** ISR to call */
    void *func;
    /** Parameter for non-direct IRQs */
    void *param;
};
```

Zephyr is built in two phases; the first phase of the build produces zephyr_prebuilt.elf which contains all the entries in the .intList section preceded by a header:

```
struct {
    void *spurious_irq_handler;
    void *sw_irq_handler;
    u32_t num_isrs;
    u32_t num_vectors;
    struct _isr_list isrs[];  <- of size num_isrs
};
```

This data consisting of the header and instances of struct _isr_list inside zephyr_prebuilt.elf is then used by the gen_isr_tables.py script to generate a C file defining a vector table and software ISR table that are then compiled and linked into the final application.

The priority level of any interrupt is not encoded in these tables, instead `IRQ_CONNECT` also has a runtime component which programs the desired priority level of the interrupt to the interrupt controller. Some architectures do not support the notion of interrupt priority, in which case the priority argument is ignored.

Vector Table

A vector table is generated when CONFIG_GEN_IRQ_VECTOR_TABLE is enabled. This data structure is used natively by the CPU and is simply an array of function pointers, where each element n corresponds to the IRQ handler for IRQ line n, and the function pointers are:

1. For 'direct' interrupts declared with `IRQ_DIRECT_CONNECT`, the handler function will be placed here.
2. For regular interrupts declared with `IRQ_CONNECT`, the address of the common software IRQ handler is placed here. This code does common kernel interrupt bookkeeping and looks up the ISR and parameter from the software ISR table.
3. For interrupt lines that are not configured at all, the address of the spurious IRQ handler will be placed here. The spurious IRQ handler causes a system fatal error if encountered.

Some architectures (such as the Nios II internal interrupt controller) have a common entry point for all interrupts and do not support a vector table, in which case the CONFIG_GEN_IRQ_VECTOR_TABLE option should be disabled.

Some architectures may reserve some initial vectors for system exceptions and declare this in a table elsewhere, in which case CONFIG_GEN_IRQ_START_VECTOR needs to be set to properly offset the indices in the table.

SW ISR Table

This is an array of struct _isr_table_entry:

```
struct _isr_table_entry {
    void *arg;
    void (*isr)(void *);
};
```

This is used by the common software IRQ handler to look up the ISR and its argument and execute it. The active IRQ line is looked up in an interrupt controller register and used to index this table.

x86 Details

The x86 architecture has a special type of vector table called the Interrupt Descriptor Table (IDT) which must be laid out in a certain way per the x86 processor documentation. It is still fundamentally a vector table, and the gen_idt tool uses the .intList section to create it. However, on APIC-based systems the indexes in the vector table do not correspond to the IRQ line. The first 32 vectors are reserved for CPU exceptions, and all remaining vectors (up to index 255) correspond to the priority level, in groups of 16. In this scheme, interrupts of priority level 0 will be placed in vectors 32-47, level 1 48-63, and so forth. When the gen_idt tool is constructing the IDT, when it configures an interrupt it will look for a free vector in the appropriate range for the requested priority level and set the handler there.

There are some APIC variants (such as MVIC) where priorities cannot be set by the user and the position in the vector table does correspond to the IRQ line. Systems like this will enable CONFIG_X86_FIXED_IRQ_MAPPING.

On x86 when an interrupt or exception vector is executed by the CPU, there is no foolproof way to determine which vector was fired, so a software ISR table indexed by IRQ line is not used. Instead, the `IRQ_CONNECT` call creates a small assembly language function which calls the common interrupt code in `_interrupt_enter()` with the ISR and parameter as arguments. It is the address of this assembly interrupt stub which gets placed in the IDT. For interrupts declared with `IRQ_DIRECT_CONNECT` the parameterless ISR is placed directly in the IDT.

On systems where the position in the vector table corresponds to the interrupt's priority level, the interrupt controller needs to know at runtime what vector is associated with an IRQ line. gen_idt additionally creates an _irq_to_interrupt_vector array which maps an IRQ line to its configured vector in the IDT. This is used at runtime by `IRQ_CONNECT` to program the IRQ-to-vector association in the interrupt controller.

Suggested Uses

Use a regular or direct ISR to perform interrupt processing that requires a very rapid response, and can be done quickly without blocking.

注解: Interrupt processing that is time consuming, or involves blocking, should be handed off to a thread. See *Offloading ISR Work* for a description of various techniques that can be used in an application.

Configuration Options

Related configuration options:

- `CONFIG_ISR_STACK_SIZE`

Additional architecture-specific and device-specific configuration options also exist.

APIs

The following interrupt-related APIs are provided by `irq.h`:

- `IRQ_CONNECT`
- `IRQ_DIRECT_CONNECT`
- `ISR_DIRECT_HEADER`
- `ISR_DIRECT_FOOTER`
- `ISR_DIRECT_PM`
- `ISR_DIRECT_DECLARE`
- `irq_lock()`
- `irq_unlock()`
- `irq_enable()`
- `irq_disable()`
- `irq_is_enabled()`

The following interrupt-related APIs are provided by `kernel.h`:

- `k_is_in_isr()`
- `k_is_preempt_thread()`

Atomic Services

An *atomic variable* is a 32-bit variable that can be read and modified by threads and ISRs in an uninterruptible manner.

Concepts

Any number of atomic variables can be defined.

Using the kernel's atomic APIs to manipulate an atomic variable guarantees that the desired operation occurs correctly, even if higher priority contexts also manipulate the same variable.

The kernel also supports the atomic manipulation of a single bit in an array of atomic variables.

Implementation

Defining an Atomic Variable

An atomic variable is defined using a variable of type `atomic_t`.

By default an atomic variable is initialized to zero. However, it can be given a different value using `ATOMIC_INIT`:

```
atomic_t flags = ATOMIC_INIT(0xFF);
```

Manipulating an Atomic Variable

An atomic variable is manipulated using the APIs listed at the end of this section.

The following code shows how an atomic variable can be used to keep track of the number of times a function has been invoked. Since the count is incremented atomically, there is no risk that it will become corrupted in mid-increment if a thread calling the function is interrupted if by a higher priority context that also calls the routine.

```
atomic_t call_count;

int call_counting_routine(void)
{
    /* increment invocation counter */
    atomic_inc(&call_count);

    /* do rest of routine's processing */
    ...
}
```

Manipulating an Array of Atomic Variables

An array of 32-bit atomic variables can be defined in the conventional manner. However, you can also define an N-bit array of atomic variables using `ATOMIC_DEFINE`.

A single bit in array of atomic variables can be manipulated using the APIs listed at the end of this section that end with `_bit()`.

The following code shows how a set of 200 flag bits can be implemented using an array of atomic variables.

```
#define NUM_FLAG_BITS 200

ATOMIC_DEFINE(flag_bits, NUM_FLAG_BITS);

/* set specified flag bit & return its previous value */
int set_flag_bit(int bit_position)
{
    return (int)atomic_set_bit(flag_bits, bit_position);
}
```

Suggested Uses

Use an atomic variable to implement critical section processing that only requires the manipulation of a single 32-bit value.

Use multiple atomic variables to implement critical section processing on a set of flag bits in a bit array longer than 32 bits.

注解: Using atomic variables is typically far more efficient than using other techniques to implement critical sections such as using a mutex or locking interrupts.

Configuration Options

Related configuration options:

- `CONFIG_ATOMIC_OPERATIONS_BUILTIN`
- `CONFIG_ATOMIC_OPERATIONS_CUSTOM`
- `CONFIG_ATOMIC_OPERATIONS_C`

APIs

The following atomic operation APIs are provided by `atomic.h`:

- `ATOMIC_INIT`
- `ATOMIC_DEFINE`
- `atomic_get()`
- `atomic_set()`
- `atomic_clear()`
- `atomic_add()`
- `atomic_sub()`
- `atomic_inc()`
- `atomic_dec()`

- `atomic_and()`
- `atomic_or()`
- `atomic_xor()`
- `atomic_nand()`
- `atomic_cas()`
- `atomic_set_bit()`
- `atomic_clear_bit()`
- `atomic_test_bit()`
- `atomic_test_and_set_bit()`
- `atomic_test_and_clear_bit()`

Polling API

The polling API is used to wait concurrently for any one of multiple conditions to be fulfilled.

Concepts

The polling API's main function is `k_poll()`, which is very similar in concept to the POSIX `poll()` function, except that it operates on kernel objects rather than on file descriptors.

The polling API allows a single thread to wait concurrently for one or more conditions to be fulfilled without actively looking at each one individually.

There is a limited set of such conditions:

- a semaphore becomes available
- a kernel FIFO contains data ready to be retrieved
- a poll signal is raised

A thread that wants to wait on multiple conditions must define an array of **poll events**, one for each condition.

All events in the array must be initialized before the array can be polled on.

Each event must specify which **type** of condition must be satisfied so that its state is changed to signal the requested condition has been met.

Each event must specify what **kernel object** it wants the condition to be satisfied.

Each event must specify which **mode** of operation is used when the condition is satisfied.

Each event can optionally specify a **tag** to group multiple events together, to the user's discretion.

Apart from the kernel objects, there is also a **poll signal** pseudo-object type that be directly signaled.

The `k_poll()` function returns as soon as one of the conditions it is waiting for is fulfilled. It is possible for more than one to be fulfilled when `k_poll()` returns, if they were fulfilled before `k_poll()` was called, or due to the preemptive multi-threading nature of the kernel. The caller must look at the state of all the poll events in the array to figured out which ones were fulfilled and what actions to take.

Currently, there is only one mode of operation available: the object is not acquired. As an example, this means that when `k_poll()` returns and the poll event states that the semaphore is available, the caller of `k_poll()` must then invoke `k_sem_take()` to take ownership of the semaphore. If the semaphore is contested, there is no guarantee that it will be still available when `k_sem_give()` is called.

Implementation

Using k_poll()

The main API is `k_poll()`, which operates on an array of poll events of type `struct k_poll_event`. Each entry in the array represents one event a call to `k_poll()` will wait for its condition to be fulfilled.

They can be initialized using either the runtime initializers `K_POLL_EVENT_INITIALIZER()` or `k_poll_event_init()`, or the static initializer `K_POLL_EVENT_STATIC_INITIALIZER()`. An object that matches the **type** specified must be passed to the initializers. The **mode** *must* be set to `K_POLL_MODE_NOTIFY_ONLY`. The state *must* be set to `K_POLL_STATE_NOT_READY` (the initializers take care of this). The user **tag** is optional and completely opaque to the API: it is there to help a user to group similar events together. Being optional, it is passed to the static initializer, but not the runtime ones for performance reasons. If using runtime initializers, the user must set it separately in the `struct k_poll_event` data structure. If an event in the array is to be ignored, most likely temporarily, its type can be set to K_POLL_TYPE_IGNORE.

```
struct k_poll_event events[2] = {
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_SEM_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_sem, 0),
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_FIFO_DATA_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_fifo, 0),
};
```

or at runtime

```
struct k_poll_event events[2];
void some_init(void)
{
    k_poll_event_init(&events[0],
                      K_POLL_TYPE_SEM_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_sem);

    k_poll_event_init(&events[1],
                      K_POLL_TYPE_FIFO_DATA_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_fifo);

    // tags are left uninitialized if unused
}
```

After the events are initialized, the array can be passed to `k_poll()`. A timeout can be specified to wait only for a specified amount of time, or the special values `K_NO_WAIT` and `K_FOREVER` to either not wait or wait until an event condition is satisfied and not sooner.

Only one thread can poll on a semaphore or a FIFO at a time. If a second thread tries to poll on the same semaphore or FIFO, `k_poll()` immediately returns with the return value `-EADDRINUSE`. In that case, if other conditions passed to `k_poll()` were met, their state will be set in the corresponding poll event.

In case of success, `k_poll()` returns 0. If it times out, it returns `-EAGAIN`.

```
// assume there is no contention on this semaphore and FIFO
// -EADDRINUSE will not occur; the semaphore and/or data will be available

void do_stuff(void)
{
    rc = k_poll(events, 2, 1000);
    if (rc == 0) {
        if (events[0].state == K_POLL_STATE_SEM_AVAILABLE) {
            k_sem_take(events[0].sem, 0);
        } else if (events[1].state == K_POLL_STATE_FIFO_DATA_AVAILABLE) {
            data = k_fifo_get(events[1].fifo, 0);
            // handle data
        }
    } else {
        // handle timeout
    }
}
```

When `k_poll()` is called in a loop, the events state must be reset to `K_POLL_STATE_NOT_READY` by the user.

```
void do_stuff(void)
{
    for(;;) {
        rc = k_poll(events, 2, K_FOREVER);
        if (events[0].state == K_POLL_STATE_SEM_AVAILABLE) {
            k_sem_take(events[0].sem, 0);
        } else if (events[1].state == K_POLL_STATE_FIFO_DATA_AVAILABLE) {
            data = k_fifo_get(events[1].fifo, 0);
            // handle data
        }
        events[0].state = K_POLL_STATE_NOT_READY;
        events[1].state = K_POLL_STATE_NOT_READY;
    }
}
```

Using k_poll_signal()

One of the types of events is `K_POLL_TYPE_SIGNAL`: this is a "direct" signal to a poll event. This can be seen as a lightweight binary semaphore only one thread can wait for.

A poll signal is a separate object of type `struct k_poll_signal` that must be attached to a k_poll_event, similar to a semaphore or FIFO. It must first be initialized either via `K_POLL_SIGNAL_INITIALIZER()` or `k_poll_signal_init()`.

```
struct k_poll_signal signal;
void do_stuff(void)
{
    k_poll_signal_init(&signal);
}
```

It is signaled via the `k_poll_signal()` function. This function takes a user **result** parameter that is opaque to the API and can be used to pass extra information to the thread waiting on the event.

```
struct k_poll_signal signal;

// thread A
void do_stuff(void)
{
    k_poll_signal_init(&signal);

    struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &signal);
    };

    k_poll(events, 1, K_FOREVER);

    if (events.signal->result == 0x1337) {
        // A-OK!
    } else {
        // weird error
    }
}

// thread B
void signal_do_stuff(void)
{
    k_poll_signal(&signal, 0x1337);
}
```

If the signal is to be polled in a loop, *both* its event state and its **signaled** field *must* be reset on each iteration if it has been signaled.

```
struct k_poll_signal signal;
void do_stuff(void)
{
    k_poll_signal_init(&signal);

    struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &signal);
    };

    for (;;) {
        k_poll(events, 1, K_FOREVER);

        if (events[0].signal->result == 0x1337) {
            // A-OK!
        } else {
            // weird error
        }

        events[0].signal->signaled = 0;
        events[0].state = K_POLL_STATE_NOT_READY;
    }
}
```

Suggested Uses

Use `k_poll()` to consolidate multiple threads that would be pending on one object each, saving possibly large amounts of stack space.

Use a poll signal as a lightweight binary semaphore if only one thread pends on it.

注解: Because objects are only signaled if no other thread is waiting for them to become available and only one thread can poll on a specific object, polling is best used when objects are not subject of contention between multiple threads, basically when a single thread operates as a main "server" or "dispatcher" for multiple objects and is the only one trying to acquire these objects.

Configuration Options

Related configuration options:

- `CONFIG_POLL`

APIs

The following polling APIs are provided by `kernel.h`:

- `K_POLL_EVENT_INITIALIZER`
- `K_POLL_EVENT_STATIC_INITIALIZER`
- `k_poll_event_init()`
- `k_poll()`
- `k_poll_signal_init()`
- `k_poll_signal()`

Ring Buffers

A *ring buffer* is a circular buffer of 32-bit words, whose contents are stored in first-in-first-out order. Data items can be enqueued and dequeued from a ring buffer in chunks of up to 1020 bytes. Each data item also has two associated metadata values: a type identifier and a 16-bit integer value, both of which are application-specific.

Concepts

Any number of ring buffers can be defined. Each ring buffer is referenced by its memory address.

A ring buffer has the following key properties:

- A **data buffer** of 32-bit words. The data buffer contains the data items that have been added to the ring buffer but not yet removed.
- A **data buffer size**, measured in 32-bit words. This governs the maximum amount of data (including metadata values) the ring buffer can hold.

A ring buffer must be initialized before it can be used. This sets its data buffer to empty.

A ring buffer **data item** is an array of 32-bit words from 0 to 1020 bytes in length. When a data item is **enqueued** its contents are copied to the data buffer, along with its associated metadata values (which occupy one additional 32-bit word). If the ring buffer has insufficient space to hold the new data item the enqueue operation fails.

A data items is **dequeued** from a ring buffer by removing the oldest enqueued item. The contents of the dequeued data item, as well as its two metadata values, are copied to areas supplied by the retriever. If the ring buffer is empty, or if the data array supplied by the retriever is not large enough to hold the data item's data, the dequeue operation fails.

Concurrency

The ring buffer APIs do not provide any concurrency control. Depending on usage (particularly with respect to number of concurrent readers/writers) applications may need to protect the ring buffer with mutexes and/or use semaphores to notify consumers that there is data to read.

For the trivial case of one producer and one consumer, concurrency shouldn't be needed.

Internal Operation

The ring buffer always maintains an empty 32-bit word in its data buffer to allow it to distinguish between empty and full states.

If the size of the data buffer is a power of two, the ring buffer uses efficient masking operations instead of expensive modulo operations when enqueuing and dequeuing data items.

Implementation

Defining a Ring Buffer

A ring buffer is defined using a variable of type `struct ring_buf`. It must then be initialized by calling `sys_ring_buf_init()`.

The following code defines and initializes an empty ring buffer (which is part of a larger data structure). The ring buffer's data buffer is capable of holding 64 words of data and metadata information.

```
#define MY_RING_BUF_SIZE 64

struct my_struct {
    struct ring_buf rb;
    u32_t buffer[MY_RING_BUF_SIZE];
    ...
};
struct my_struct ms;

void init_my_struct {
    sys_ring_buf_init(&ms.rb, sizeof(ms.buffer), ms.buffer);
    ...
}
```

Alternatively, a ring buffer can be defined and initialized at compile time using one of two macros at file scope. Each macro defines both the ring buffer itself and its data buffer.

The following code defines a ring buffer with a power-of-two sized data buffer, which can be accessed using efficient masking operations.

```
/* Buffer with 2^8 (or 256) words */
SYS_RING_BUF_DECLARE_POW2(my_ring_buf, 8);
```

The following code defines a ring buffer with an arbitrary-sized data buffer, which can be accessed using less efficient modulo operations.

```
#define MY_RING_BUF_WORDS 93
SYS_RING_BUF_DECLARE_SIZE(my_ring_buf, MY_RING_BUF_WORDS);
```

Enqueuing Data

A data item is added to a ring buffer by calling `sys_ring_buf_put()`.

```
u32_t my_data[MY_DATA_WORDS];
int ret;

ret = sys_ring_buf_put(&ring_buf, TYPE_FOO, 0, my_data, SIZE32_OF(my_data));
if (ret == -EMSGSIZE) {
    /* not enough room for the data item */
    ...
}
```

If the data item requires only the type or application-specific integer value (i.e. it has no data array), a size of 0 and data pointer of `NULL` can be specified.

```
int ret;

ret = sys_ring_buf_put(&ring_buf, TYPE_BAR, 17, NULL, 0);
```

```
if (ret == -EMSGSIZE) {
    /* not enough room for the data item */
    ...
}
```

Retrieving Data

A data item is removed from a ring buffer by calling `sys_ring_buf_get()`.

```
u32_t my_data[MY_DATA_WORDS];
u16_t my_type;
u8_t  my_value;
u8_t  my_size;
int ret;

my_size = SIZE32_OF(my_data);
ret = sys_ring_buf_get(&ring_buf, &my_type, &my_value, my_data, &my_size);
if (ret == -EMSGSIZE) {
    printk("Buffer is too small, need %d u32_t\n", my_size);
} else if (ret == -EAGAIN) {
    printk("Ring buffer is empty\n");
} else {
    printk("Got item of type %u value &u of size %u dwords\n",
           my_type, my_value, my_size);
    ...
}
```

APIs

The following ring buffer APIs are provided by `include/misc/ring_buffer.h`:

- `SYS_RING_BUF_DECLARE_POW2()`
- `SYS_RING_BUF_DECLARE_SIZE()`
- `sys_ring_buf_init()`
- `sys_ring_buf_is_empty()`
- `sys_ring_buf_space_get()`
- `sys_ring_buf_put()`
- `sys_ring_buf_get()`

Floating Point Services

The kernel allows threads to use floating point registers on board configurations that support these registers.

注解: Floating point services are currently available only for boards based on the ARM Cortex-M4 or the Intel x86 architectures. The services provided are architecture specific.

The kernel does not support the use of floating point registers by ISRs.

Concepts

The kernel can be configured to provide only the floating point services required by an application. Three modes of operation are supported, which are described below. In addition, the kernel's support for the SSE registers can be included or omitted, as desired.

No FP registers mode

This mode is used when the application has no threads that use floating point registers. It is the kernel's default floating point services mode.

If a thread uses any floating point register, the kernel generates a fatal error condition and aborts the thread.

Unshared FP registers mode

This mode is used when the application has only a single thread that uses floating point registers.

The kernel initializes the floating point registers so they can be used by any thread. The floating point registers are left unchanged whenever a context switch occurs.

注解: Incorrect operation may result if two or more threads use floating point registers, as the kernel does not attempt to detect (or prevent) multiple threads from using these registers.

Shared FP registers mode

This mode is used when the application has two or more threads that use floating point registers. Depending upon the underlying CPU architecture, the kernel supports one or more of the following thread sub-classes:

- non-user: A thread that cannot use any floating point registers
- FPU user: A thread that can use the standard floating point registers
- SSE user: A thread that can use both the standard floating point registers and SSE registers

The kernel initializes the floating point registers so they can be used by any thread, then saves and restores these registers during context switches to ensure the computations performed by each FPU user or SSE user are not impacted by the computations performed by the other users.

On the ARM Cortex-M4 architecture the kernel treats *all* threads as FPU users when shared FP registers mode is enabled. This means that the floating point registers are saved and restored during a context switch, even when the associated threads are not using them. Each thread must provide an extra 132 bytes of stack space where these register values can be saved.

On the x86 architecture the kernel treats each thread as a non-user, FPU user or SSE user on a case-by-case basis. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the registers are *not* saved when switching from an FPU user to a non-user thread, and then back to the original FPU user. The following table indicates the amount of additional stack space a thread must provide so the registers can be saved properly.

Thread type	FP register use	Extra stack space required
cooperative	any	0 bytes
preemptive	none	0 bytes
preemptive	FPU	108 bytes
preemptive	SSE	464 bytes

The x86 kernel automatically detects that a given thread is using the floating point registers the first time the thread accesses them. The thread is tagged as an SSE user if the kernel has been configured to support the SSE registers, or as an FPU user if the SSE registers are not supported. If this would result in a thread that is an FPU user being tagged as an SSE user, or if the application wants to avoid the exception handling overhead involved in auto-tagging threads, it is possible to pre-tag a thread using one of the techniques listed below.

- A statically-created x86 thread can be pre-tagged by passing the `K_FP_REGS` or `K_SSE_REGS` option to `K_THREAD_DEFINE`.
- A dynamically-created x86 thread can be pre-tagged by passing the `K_FP_REGS` or `K_SSE_REGS` option to `k_thread_create()`.
- An already-created x86 thread can pre-tag itself once it has started by passing the `K_FP_REGS` or `K_SSE_REGS` option to `k_float_enable()`.

If an x86 thread uses the floating point registers infrequently it can call `k_float_disable()` to remove its tagging as an FPU user or SSE user. This eliminates the need for the kernel to take steps to preserve the contents of the floating point registers during context switches when there is no need to do so. When the thread again needs to use the floating point registers it can re-tag itself as an FPU user or SSE user by calling `k_float_enable()`.

Implementation

Performing Floating Point Arithmetic

No special coding is required for a thread to use floating point arithmetic if the kernel is properly configured.

The following code shows how a routine can use floating point arithmetic to avoid overflow issues when computing the average of a series of integer values.

```
int average(int *values, int num_values)
{
    double sum;
    int i;

    sum = 0.0;

    for (i = 0; i < num_values; i++) {
        sum += *values;
        values++;
    }

    return (int)((sum / num_values) + 0.5);
}
```

Suggested Uses

Use the kernel floating point services when an application needs to perform floating point operations.

Configuration Options

To configure unshared FP registers mode, enable the `CONFIG_FLOAT` configuration option and leave the `CONFIG_FP_SHARING` configuration option disabled.

To configure shared FP registers mode, enable both the `CONFIG_FLOAT` configuration option and the `CONFIG_FP_SHARING` configuration option. Also, ensure that any thread that uses the floating point registers has sufficient added stack space for saving floating point register values during context switches, as described above.

Use the `CONFIG_SSE` configuration option to enable support for SSEx instructions (x86 only).

APIs

The following floating point APIs (x86 only) are provided by `kernel.h`:

- `k_float_enable()`
- `k_float_disable()`

C++ Support for Applications

The kernel supports applications written in both C and C++. However, to use C++ in an application you must configure the kernel to include C++ support and the build system must select the correct compiler.

The build system selects the C++ compiler based on the suffix of the files. Files identified with either a **cxx** or a **cpp** suffix are compiled using the C++ compiler. For example, `myCplusplusApp.cpp` is compiled using C++.

The kernel currently provides only a subset of C++ functionality. The following features are *not* supported:

- Dynamic object management with the **new** and **delete** operators
- RTTI (run-time type information)
- Exceptions
- Static global object destruction

While not an exhaustive list, support for the following functionality is included:

- Inheritance
- Virtual functions
- Virtual tables
- Static global object constructors

Static global object constructors are initialized after the drivers are initialized but before the application `main()` function. Therefore, use of C++ is restricted to application code.

注解: Do not use C++ for kernel, driver, or system initialization code.

CPU Idling

Although normally reserved for the idle thread, in certain special applications, a thread might want to make the CPU idle.

Concepts

Making the CPU idle causes the kernel to pause all operations until an event, normally an interrupt, wakes up the CPU. In a regular system, the idle thread is responsible for this. However, in some constrained systems, it is possible that another thread takes this duty.

Implementation

Making the CPU idle

Making the CPU idle is simple: call the k_cpu_idle() API. The CPU will stop executing instructions until an event occurs. Make sure interrupts are not locked before invoking it. Most likely, it will be called within a loop.

```
static k_sem my_sem;

void my_isr(void *unused)
{
    k_sem_give(&my_sem);
}

void main(void)
{
    k_sem_init(&my_sem, 0, 1);

    /* wait for semaphore from ISR, then do related work */

    for (;;) {

        /* wait for ISR to trigger work to perform */
        if (k_sem_take(&my_sem, K_NO_WAIT) == 0) {

            /* ... do processing */

        }

        /* put CPU to sleep to save power */
        k_cpu_idle();
    }
}
```

Making the CPU idle in an atomic fashion

It is possible that there is a need to do some work atomically before making the CPU idle. In such a case, k_cpu_atomic_idle() should be used instead.

In fact, there is a race condition in the previous example: the interrupt could occur between the time the semaphore is taken, finding out it is not available and making the CPU idle again. In some systems, this can cause the CPU to idle until *another* interrupt occurs, which might be *never*, thus hanging the system completely. To prevent this, k_cpu_atomic_idle() should have been used, like in this example.

```
static k_sem my_sem;

void my_isr(void *unused)
{
    k_sem_give(&my_sem);
}

void main(void)
{
    k_sem_init(&my_sem, 0, 1);

    for (;;) {

        unsigned int key = irq_lock();

        /*
         * Wait for semaphore from ISR; if acquired, do related work, then
         * go to next loop iteration (the semaphore might have been given
         * again); else, make the CPU idle.
         */

        if (k_sem_take(&my_sem, K_NO_WAIT) == 0) {

            irq_unlock(key);

            /* ... do processing */


        } else {
            /* put CPU to sleep to save power */
            k_cpu_atomic_idle(key);
        }
    }
}
```

Suggested Uses

Use k_cpu_atomic_idle() when a thread has to do some real work in addition to idling the CPU to wait for an event. See example above.

Use k_cpu_idle() only when a thread is only responsible for idling the CPU, i.e. not doing any real work, like in this example below.

```
void main(void)
{
    /* ... do some system/application initialization */


    /* thread is only used for CPU idling from this point on */
    for (;;) {
        k_cpu_idle();
    }
}
```

注解: Do not use these APIs unless absolutely necessary. In a normal system, the idle thread takes care of power management, including CPU idling.

APIs

The following CPU idling APIs are provided by `kernel.h`:

- `k_cpu_idle()`
- `k_cpu_atomic_idle()`

Zephyr Project Security

These documents describe the requirements, processes, and developer guidelines for ensuring security is addressed within the Zephyr project.

Zephyr Security Overview

Revision history		
Rev	**Date**	**Description**
1.0 Draft	July 27, 2016	Initial draft version
1.0-rc1	April 21, 2017	Draft for review by TSC

Introduction

This document outlines the steps of the Zephyr Security board towards a defined security process that helps developers build more secure software while addressing security compliance requirements. It presents the key ideas of the security process and outlines which documents need to be created. After the process is implemented and all supporting documents are created, this document is a top-level overview and entry point.

Overview and Scope

We begin with an overview of the Zephyr development process, which mainly focuses on security functionality.

In subsequent sections, the individual parts of the process are treated in detail. As depicted in Figure 1, these main steps are:

1. **Secure Development:** Defines the system architecture and development process that ensures adherence to relevant coding guidelines and quality assurance procedures.
2. **Secure Design:** Defines security procedures and implement measures to enforce them. A security architecture of the system and relevant sub-modules is created, threats are identified, and countermeasures designed. Their correct implementation and the validity of the threat models are checked by code reviews. Finally, a process shall be defined for reporting, classifying, and mitigating security issues..
3. **Security Certification:** Defines the certifiable part of the Zephyr RTOS. This includes an evaluation target, its assets, and how these assets are protected. Certification claims shall be determined and backed with appropriate evidence.

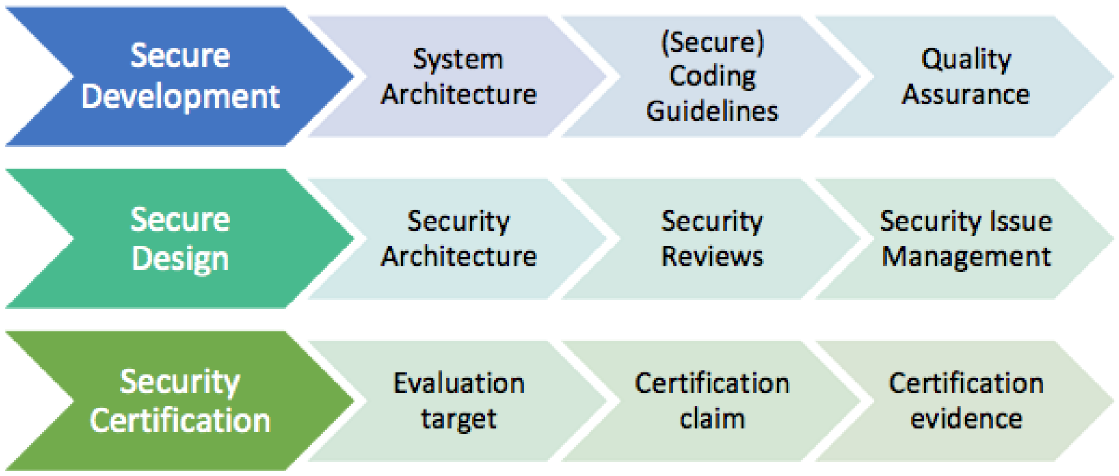


图 1.1: Figure 1. Security Process Steps

Intended Audience

This document is a guideline for the development of a security process by the Zephyr Security Committee and the Zephyr Technical Steering Committee. It provides an overview of the Zephyr security process for (security) engineers and architects.

Nomenclature

In this document, the keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC2119.

These words are used to define absolute requirements (or prohibitions), highly recommended requirements, and truly optional requirements. As noted in RFC-2119, "These terms are frequently used to specify behavior with security implications. The effects on security of not implementing a MUST or SHOULD, or doing something the specification says MUST NOT or SHOULD NOT be done may be very subtle. Document authors should take the time to elaborate the security implications of not following recommendations or requirements as most implementors will not have had the benefit of the experience and discussion that produced the specification."

Security Document Update

This document is a living document. As new requirements, features, and changes are identified, they will be added to this document through the following process:

1. Changes will be submitted from the interested party(ies) via pull requests to the Zephyr documentation repository.
2. The security committee will review these changes and provide feedback or acceptance of the changes.
3. Once accepted, these changes will become part of the document.

Current Security Definition

This section recapitulates the current status of secure development within the Zephyr RTOS. Currently, focus is put on functional security and code quality assurance, although additional security features are scoped.

The three major security measures currently implemented are:

- **Security Functionality** with a focus on cryptographic algorithms and protocols. Support for cryptographic hardware is scoped for future releases.The Zephyr runtime architecture is a monolithic binary and removes the need for dynamic loaders , thereby reducing the exposed attack surface.
- **Quality Assurance** is driven by using a development process that requires all code to be reviewed before being committed to the common repository. Furthermore, the reuse of proven building blocks such as network stacks increases the overall quality level and guarantees stable APIs. Static code analyses are planned for the near future.
- **Execution Protection** including thread separation, stack and memory protection is currently not available in the upstream Zephyr RTOS but is planned for future releases.

These topics are discussed in more detail in the following subsections.

Security Functionality

The security functionality in Zephyr hinges mainly on the inclusion of cryptographic algorithms, and on its monolithic system design.

The cryptographic features are provided through a set of cryptographic libraries. Applications can choose TinyCrypt2 or mbedTLS based on their needs. TinyCrypt2 supports key cryptographic algorithms required by the connectivity stacks. Tinycrypt2, however, only provides a limited set of algorithms. mbedTLS supports a wider range of algorithms, but at the cost of additional requirements such as malloc support. Applications can choose the solution that matches their individual requirements. Future work may include APIs to abstract the underlying crypto library choice.

APIs for vendor specific cryptographic IPs in both hardware and software are planned, including secure key storage in the form of secure access modules (SAMs), Trusted Platform Modules (TPMs), and Trusted Execution Environments (TEEs).

The security architecture is based on a monolithic design where the Zephyr kernel and all applications are compiled into a single static binary. System calls are implemented as function calls without requiring context switches. Static linking eliminates the potential for dynamically loading malicious code. Memory protection and task separation techniques are in scope for future releases.

Quality Assurance

The Zephyr project uses an automated quality assurance process. The goal is to have a process including mandatory code reviews, feature and issue management/tracking, and static code analyses.

Code reviews are documented and enforced using a voting system before getting checked into the repository by the responsible subsystem's maintainer. The main goals of the code review are:

- Verifying correct functionality of the implementation
- Increasing the readability and maintainability of the contributed source code
- Ensuring appropriate usage of string and memory functions
- Validation of the user input
- Reviewing the security relevant code for potential issues

The current coding guidelines focus mostly on coding styles and conventions. Functional correctness is ensured by the build system and the experience of the reviewer. Especially for security relevant code, concrete and detailed guidelines need to be developed and aligned with the developers (see: *Secure Coding Guidelines*).

Static code analyses are run on the Zephyr code tree on a regular basis using the open source Coverity Scan tool. Coverity Scan now includes complexity analysis.

Bug and issue tracking and management is performed using Jira. The term "survivability" was coined to cover pro-active security tasks such as security issue categorization and management. Initial effort has been started on the definition of vulnerability categorization and mitigation processes within Jira.

Issues determined by Coverity should have more stringent reviews before they are closed as non issues (at least another person educated in security processes need to agree on non-issue before closing).

A security subcommittee has been formed to develop a security process in more detail; this document is part of that process.

Execution Protection

Execution protection is planned for future releases and is roughly categorized into the following tasks:

- **Memory separation:** Memory will be partitioned into regions and assigned attributes based on the owner of that region of memory. Threads will only have access to regions they control.
- **Stack protection:** Stack guards would provide mechanisms for detecting and trapping stack overruns. Individual threads should only have access to their own stacks.
- **Thread separation:** Individual threads should only have access to their own memory resources. As threads are scheduled, only memory resources owned by that thread will be accessible.Topics such as program flow protection and other measures for tamper resistance are currently not in scope.

System Level Security (Ecosystem, ...)

System level security encompasses a wide variety of categories. Some examples of these would be:

- Secure/trusted boot
- Over the air (OTA) updates
- External Communication
- Device authentication

- Access control of onboard resources
  - Flash updating
  - Secure storage
  - Peripherals
- Root of trust
- Reduction of attack surface

Some of these categories are interconnected and rely on multiple pieces to be in place to produce a full solution for the application.

Secure Development Process

The development of secure code shall adhere to certain criteria. These include coding guidelines and development processes that can be roughly separated into two categories related to software quality and related to software security. Furthermore, a system architecture document shall be created and kept up-to-date with future development.

System Architecture

A high-level schematic of the Zephyr system architecture is given in Figure 2. It separates the architecture into an OS part (*kernel + OS Services*) and a user-specific part (*Application Services*). The OS part itself contains low-level, platform specific drivers and the generic implementation of I/O APIs, file systems, kernel-specific functions, and the cryptographic library.

A document describing the system architecture and design choices shall be created and kept up to date with future development. This document shall include the base architecture of the Zephyr OS and an overview of important submodules. For each of the modules, a dedicated architecture document shall be created and evaluated against the implementation. These documents shall serve as an entry point to new developers and as a basis for the security architecture. Please refer to the *Zephyr Kernel subsystem documentation* for detailed information.

Secure Coding Guidelines

Designing an open software system such as Zephyr to be secure requires adhering to a defined set of design standards. These standards are included in the Zephyr Project documentation, specifically in its *Secure Coding Guidelines* section. In [SALT75], the following, widely accepted principles for protection mechanisms are defined to prevent security violations and limit their impact:

- **Open design** as a design guideline incorporates the maxim that protection mechanisms cannot be kept secret on any system in widespread use. Instead of relying on secret, custom-tailored security measures, publicly accepted cryptographic algorithms and well established cryptographic libraries shall be used.
- **Economy of mechanism** specifies that the underlying design of a system shall be kept as simple and small as possible. In the context of the Zephyr project, this can be realized, e.g., by modular code [PAUL09] and abstracted APIs.
- **Complete mediation** requires that each access to every object and process needs to be authenticated first. Mechanisms to store access conditions shall be avoided if possible.
- **Fail-safe defaults** defines that access is restricted by default and permitted only in specific conditions defined by the system protection scheme, e.g., after successful authentication. Furthermore, default settings for services shall be chosen in a way to provide maximum security. This corresponds to the "Secure by Default" paradigm [MS12].

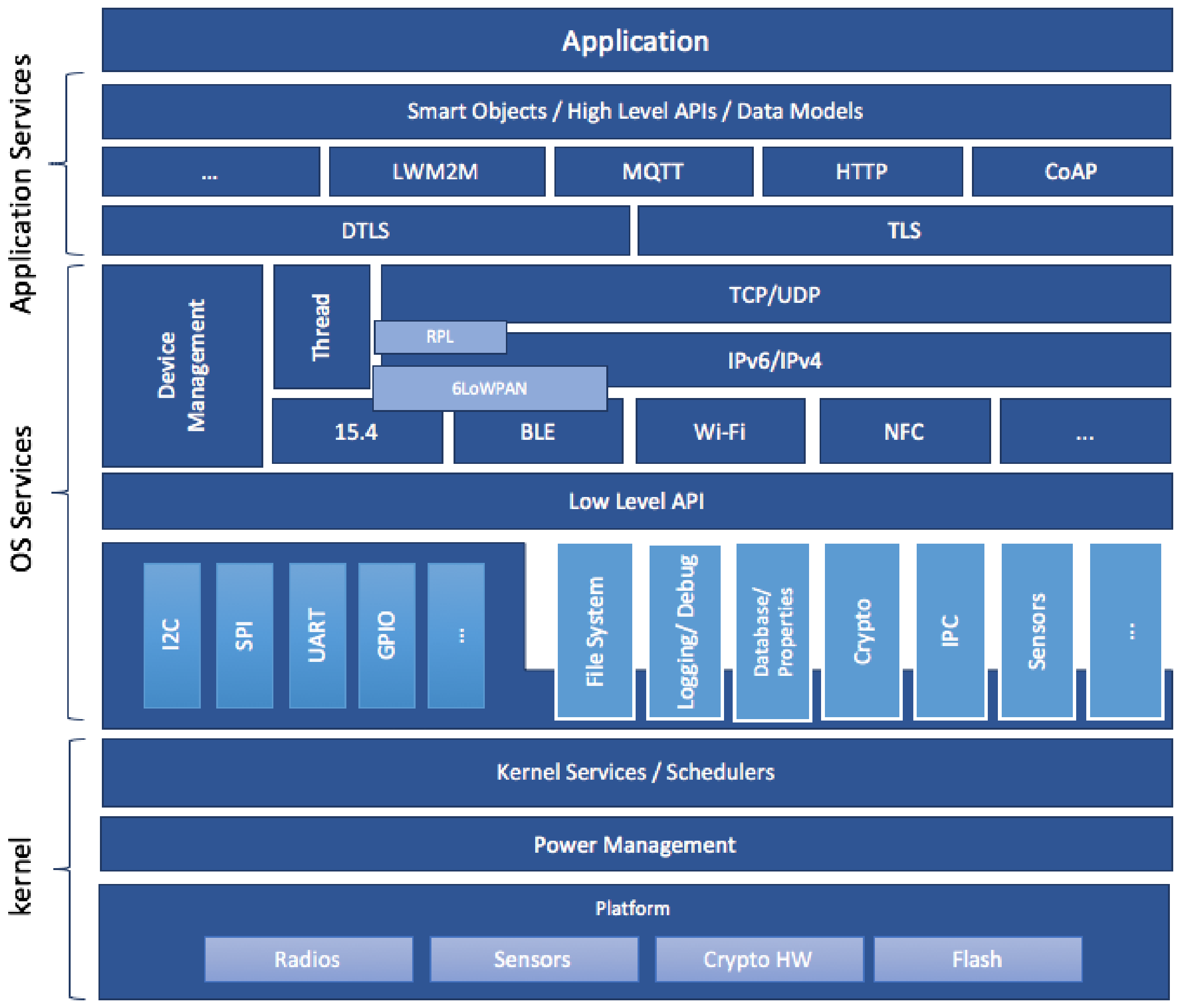


图 1.2: Figure 2: Zephyr System Architecture

- **Separation of privilege** is the principle that two conditions or more need to be satisfied before access is granted. In the context of the Zephyr project, this could encompass split keys [PAUL09].
- **Least privilege** describes an access model in which each user, program, thread, and fiber shall have the smallest possible subset of permissions in the system required to perform their task. This positive security model aims to minimize the attack surface of the system.
- **Least common mechanism** specifies that mechanisms common to more than one user or process shall not be shared if not strictly required. The example given in [SALT75] is a function that should be implemented as a shared library executed by each user and not as a supervisor procedure shared by all users.
- **Psychological acceptability** requires that security features are easy to use by the developers in order to ensure its usage and the correctness of its application.

In addition to these general principles, the following points are specific to the development of a secure RTOS:

- **Complementary Security/Defense in Depth:** do not rely on a single threat mitigation approach. In case of the complementary security approach, parts of the threat mitigation are performed by the underlying platform. In case such mechanisms are not provided by the platform, or are not trusted, a defense in depth [MS12] paradigm shall be used.
- **Less commonly used services off by default**: to reduce the exposure of the system to potential attacks, features or services shall not be enabled by default if they are only rarely used (a threshold of 80% is given in [MS12]). For the Zephyr project, this can be realized using the configuration management. Each functionality and module shall be represented as a configuration option and needs to be explicitly enabled. Then, all features, protocols, and drivers not required for a particular use case can be disabled. The user shall be notified if low-level options and APIs are enabled but not used by the application.
- **Change management:** to guarantee a traceability of changes to the system, each change shall follow a specified process including a change request, impact analysis, ratification, implementation, and validation phase. In each stage, appropriate documentation shall be provided. All commits shall be related to a bug report or change request in the issue tracker. Commits without a valid reference shall be denied.

Based on these design principles and commonly accepted best practices, a secure development guide shall be developed, published, and implemented into the Zephyr development process. Further details on this are given in the *Secure Design* section.

Quality Assurance

The quality assurance part encompasses the following criteria:

- **Adherence to the Coding Guidelines** with respect to coding style, naming schemes of modules, functions, variables, and so forth. This increases the readability of the Zephyr code base and eases the code review. These coding guidelines are enforced by automated scripts prior to check-in.
- **Adherence to Deployment Guidelines** is required to ensure consistent releases with a well-documented feature set and a trackable list of security issues.
- **Code Reviews** ensure the functional correctness of the code base and shall be performed on each proposed code change prior to check-in. Code reviews shall be performed by at least one independent reviewer other than the author(s) of the code change. These reviews shall be performed by the subsystem maintainers and developers on a functional level and are to be distinguished from security reviews as laid out in Chapter 4. Please refer to the development model documentation on the Zephyr project Wiki.
- **Static Code Analysis** tools efficiently detect common coding mistakes in large code bases. All code shall be analyzed using an appropriate tool prior to merges into the main repository. This is not per individual commit, but is to be run on some interval on specific branches. It is mandatory to remove all findings or waive potential false-positives before each release. To process process documentation. Waivers shall be documented centrally and in form of a comment inside the source code itself. The documentation shall include the employed tool and

its version, the date of the analysis, the branch and parent revision number, the reason for the waiver, the author of the respective code, and the approver(s) of the waiver. This shall as a minimum run on the main release branch and on the security branch. It shall be ensured that each release has zero issues with regard to static code analysis (including waivers). Please refer to the development model documentation on the Zephyr project Wiki.

- **Complexity Analyses** shall be performed as part of the development process and metrics such as cyclomatic complexity shall be evaluated. The main goal is to keep the code as simple as possible.
- **Automation: the review process and checks for coding rule** adherence are a mandatory part of the pre-commit checks. To ensure consistent application, they shall be automated as part of the pre-commit procedure. Prior to merging large pieces of code in from subsystems, in addition to review process and coding rule adherence, all static code analysis must have been run and issues resolved.

Release and Lifecycle Management

Lifecycle management contains several aspects:

- **Device management** encompasses the possibility to update the operating system and/or security related subsystems of Zephyr enabled devices in the field.
- **Lifecycle management:** system stages shall be defined and documented along with the transactions between the stages in a system state diagram. For security reasons, this shall include locking of the device in case an attack has been detected, and a termination if the end of life is reached.
- **Release management** describes the process of defining the release cycle, documenting releases, and maintaining a record of known vulnerabilities and mitigations. Especially for certification purposes the integrity of the release needs to be ensured in a way that later manipulation (e.g. inserting of backdoors, etc.) can be easily detected.
- **Rights management and NDAs:** if required by the chosen certification, the confidentiality and integrity of the system needs to be ensured by an appropriate rights management (e.g. separate source code repository) and non-disclosure agreements between the relevant parties. In case of a repository shared between several parties, measures shall be taken that no malicious code is checked in.

These points shall be evaluated with respect to their impact on the development process employed for the Zephyr project.

Secure Design

In order to obtain a certifiable system or product, the security process needs to be clearly defined and its application needs to be monitored and driven. This process includes the development of security related modules in all of its stages and the management of reported security issues. Furthermore, threat models need to be created for currently known and future attack vectors, and their impact on the system needs to be investigated and mitigated. Please refer to the secure coding guidelines outlined in the Zephyr project documentation for detailed information.

The software security process includes:

- **Adherence to the Secure Development Guidelines** is mandatory to avoid that individual components breach the system security and to minimize the vulnerability of individual modules. While this can be partially achieved by automated tests, it is inevitable to investigate the correct implementation of security features such as countermeasures manually in security-critical modules.
- **Security Reviews** shall be performed by a security architect in preparation of each security-targeted release and each time a security-related module of the Zephyr project is changed. This process includes the validation of the effectiveness of implemented security measures, the adherence to the global security strategy and architecture, and the preparation of audits towards a security certification if required.

- **Security Issue Management** encompasses the evaluation of potential system vulnerabilities and their mitigation as described in the *Security Issue Management* Section.

These criteria and tasks need to be integrated into the development process for secure software and shall be automated wherever possible. On system level, and for each security related module of the secure branch of Zephyr, a directly responsible security architect shall be defined to guide the secure development process.

Security Architecture

The general guidelines above shall be accompanied by an architectural security design on system- and module-level. The high level considerations include

- The identification of **security and compliance requirements**
- **Functional security** such as the use of cryptographic functions whenever applicable
- Design of **countermeasures** against known attack vectors
- Recording of security relevant **auditable events**
- Support for **Trusted Platform Modules (TPM)** and **Trusted Execution Environments (TEE)**
- Mechanisms to allow for **in-the-field updates** of devices using Zephyr
- Task scheduler and separation

The security architecture development is based on assets derived from the structural overview of the overall system architecture. Based on this, the individual steps include:

1. **Identification of assets** such as user data, authentication and encryption keys, key generation data (obtained from RNG), security relevant status information.
2. **Identification of threats** against the assets such as breaches of confidentiality, manipulation of user data, etc.
3. **Definition of requirements** regarding security and protection of the assets, e.g. countermeasures or memory protection schemes.

The security architecture shall be harmonized with the existing system architecture and implementation to determine potential deviations and mitigate existing weaknesses. Newly developed sub-modules that are integrated into the secure branch of the Zephyr project shall provide individual documents describing their security architecture. Additionally, their impact on the system level security shall be considered and documented.

Security Issue Management

In order to quickly respond to security threats towards the Zephyr RTOS, a well-defined security issue management needs to be established.

Such issues shall be reported through the Zephyr Jira bug tracking system. Some JIRA modifications will be necessary to accommodate management of security issues. In addition, there will be guidelines that govern visibility, control, and resolution of security issues. The following is the current proposal:

- A boolean field shall be added to JIRA bugs to mark it security sensitive (or any other name that makes sense). This renders the entry invisible to anyone except as described below.
- Security sensitive bugs are only accessible (view/modify) to members of the Security Group; members of this Security Group are:
  - members of the Security Subcommittee
  - other as proposed and ratified Security Subcommittee, who will also have the authority to remove others

  - the reporter
  - Ability to add other users for individual issues

- Security Subcommittee meetings have to review the embargoed bugs on every meeting with more than three people in attendance. Said review process shall decide if new issues needs to be embargoed or not.
- Security sensitive bugs shall be made public (by removing the security sensitive indicator) after an embargo period of TBD days. The Security Subcommittee is the only entity with authority to extend the embargo period on a case by case basis; the JIRA entry should be updated with the rationale for the embargo extension so at some point said rationale will be made public.If the Security Subcommittee does not act upon a security sensitive bug after its TBD days of embargo are over, it shall be automatically made public by removing the security sensitive setting.
- Likewise, there shall be code repositories marked as security sensitive, accessible only to the Security Group members where the code to fix said issues is being worked on and reviewed. The person/s contributing the fix shall also have access, but fix contributors shall have only access to the tree for said fix, not to other security sensitive trees.
- A CVE space shall be allocated to assign Zephyr issues when the SWG decides such is needed.
- The severity of the issue with regard to security shall be entered by the reporter.
- All security relevant issues shall trigger an automated notification on the Zephyr security mailing list (security@lists.zephyrproject.org). Any member of the security board can then triage the severity of the issue according to the Common Vulnerability Scoring System v3.0
- Depending on the resulting severity score of the issue, the issue is prioritized and assigned to the owner of the affected module. Additionally, the system security architect and the security architect of the module are notified and shall take the responsibility to mitigate the issue and review the solution or counter-measure. In any case, the security issue shall be documented centrally, including the affected modules, software releases, and applicable workarounds for immediate mitigation. A list of known security issues per public release of the Zephyr shall be published and maintained by the security board after a risk assessment.

Threat Modeling and Mitigation

The modeling of security threats against the Zephyr RTOS is required for the development of an accurate security architecture and for most certification schemes. The first step of this process is the definition of assets to be protected by the system. The next step then models how these assets are protected by the system and which threats against them are present. After a threat has been identified, a corresponding threat model is created. This model contains the asset and system vulnerabilities, as well as the description of the potential exploits of these vulnerabilities. Additionally, the impact on the asset, the module it resides in, and the overall system is to be estimated. This threat model is then considered in the module and system security architecture and appropriate counter-measures are defined to mitigate the threat or limit the impact of exploits.

In short, the threat modeling process can be separated into these steps (adapted from Application Thread Modeling:

1. Definition of assets
2. Application decomposition and creation of appropriate data flow diagrams (DFDs)
3. Threat identification and categorization using the STRIDE and CVSS approaches
4. Determination of countermeasures and other mitigation approaches

This procedure shall be carried out during the design phase of modules and before major changes of the module or system architecture. Additionally, new models shall be created or existing ones shall be updated whenever new vulnerabilities or exploits are discovered. During security reviews, the threat models and the mitigation techniques shall be evaluated by the responsible security architect.

From these threat models and mitigation techniques tests shall be derived that prove the effectiveness of the countermeasures. These tests shall be integrated into the continuous integration workflow to ensure that the security is not impaired by regressions.

Vulnerability Analyses

In order to find weak spots in the software implementation, vulnerability analyses (VA) shall be performed. Of special interest are investigations on cryptographic algorithms, critical OS tasks, and connectivity protocols.

On a pure software level, this encompasses

- **Penetration testing** of the RTOS on a particular hardware platform, which involves testing the respective Zephyr OS configuration and hardware as one system.
- **Side channel attacks** (timing invariance, power invariance, etc.) should be considered. For instance, ensuring **timing invariance** of the cryptographic algorithms and modules is required to reduce the attack surface. This applies to both the software implementations and when using cryptographic hardware.
- **Fuzzing tests** shall be performed on both exposed APIs and protocols.

The list given above serves primarily illustration purposes. For each module and for the complete Zephyr system (in general on a particular hardware platform), a suitable VA plan shall be created and executed. The findings of these analyses shall be considered in the security issue management process, and learnings shall be formulated as guidelines and incorporated into the secure coding guide.

If possible (as in case of fuzzing analyses), these tests shall be integrated into the continuous integration process.

Security Certification

One goal of creating a secure branch of the Zephyr RTOS is to create a certifiable system or certifiable submodules thereof. The certification scope and scheme is yet to be decided. However, many certification such as Common Criteria [CCITSE12] require evidence that the evaluation claims are indeed fulfilled, so a general certification process is outlined in the following. Based on the final choices for the certification scheme and evaluation level, this process needs to be refined.

Generic Certification Process

In general, the steps towards a certification or precertification (compare [MICR16]) are:

1. The **definition of assets** to be protected within the Zephyr RTOS. Potential candidates are confidential information such as cryptographic keys, user data such as communication logs, and potentially IP of the vendor or manufacturer.
2. Developing a **threat model** and **security architecture** to protect the assets against exploits of vulnerabilities of the system. As a complete threat model includes the overall product including the hardware platform, this might be realized by a split model containing a pre-certified secure branch of Zephyr which the vendor could use to certify their Zephyr-enabled product.
3. Formulating an **evaluation target** that includes the **certification claims** on the security of the assets to be evaluated and certified, as well as assumptions on the operating conditions.
4. Providing **proof** that the claims are fulfilled. This includes consistent documentation of the security development process, etc.

These steps are partially covered in previous sections as well. In contrast to these sections, the certification process only requires to consider those components that shall be covered by the certification. The security architecture, for example, considers assets on system level and might include items not relevant for the certification.

Certification Options

For the security certification as such, the following options can be pursued:

1. **Abstract (pre-)certification of Zephyr as a pure software system:** this option requires assumptions on the underlying hardware platform and the final application running on top of Zephyr. If these assumptions are met by the hardware and the application, a full certification can be more easily achieved. This option is the most flexible approach but puts the largest burden on the product vendor.
2. **Certification of Zephyr on specific hardware platform without a specific application in mind:** this scenario describes the enablement of a secure platform running the Zephyr RTOS. The hardware manufacturer certifies the platform under defined assumptions on the application. If these are met, the final product can be certified with little effort.
3. **Certification of an actual product:** in this case, a full product including a specific hardware, the Zephyr RTOS, and an application is certified.

In all three cases, the certification scheme (e.g. FIPS 140-2 [NIST02] or Common Criteria [CCITSE12]), the scope of the certification (main-stream Zephyr, security branch, or certain modules), and the certification/assurance level need to be determined.

In case of partial certifications (options 1 and 2), assumptions on hardware and/or software are required for certifications. These can include [GHS10]

- **Appropriate physical security** of the hardware platform and its environment.
- **Sufficient protection of storage and timing channels** on the hardware platform itself and all connected devices. (No mentioning of remote connections.)
- Only **trusted/assured applications** running on the device
- The device and its software stack is configured and operated by **properly trained and trusted individuals** with no malicious intent.

These assumptions shall be part of the security claim and evaluation target documents.

References

See security-citations

Secure Coding Guidelines

Traditionally, microcontroller-based systems have not placed much emphasis on security. They have usually been thought of as isolated, disconnected from the world, and not very vulnerable, just because of the difficulty in accessing them. The Internet of Things has changed this. Now, code running on small microcontrollers often has access to the internet, or at least to other devices (that may themselves have vulnerabilities). Given the volume they are often deployed at, uncontrolled access can be devastating[1].

This document describes the requirements and process for ensuring security is addressed within the Zephyr project. All code submitted should comply with these guidelines.

Much of this document comes from the CII best practices document.

[1] An attack resulted in a significant portion of DNS infrastructure being taken down.

Introduction and Scope

This document covers guidelines for the Zephyr Project, from a security perspective. Many of the ideas contained herein are captured from other open source efforts.

We begin with an overview of secure design as it relates to Zephyr. This is followed by a section on *Secure development knowledge*, which gives basic requirements that a developer working on the project will need to have. This section gives references to other security documents, and full details of how to write secure software are beyond the scope of this document. This section also describes vulnerability knowledge that at least one of the primary developers should have. This knowledge will be necessary for the review process described below this.

Following this is a description of the review process used to incorporate changes into the Zephyr codebase. This is followed by documentation about how security-sensitive issues are handled by the project.

Finally, the document covers how changes are to be made to this document.

Secure Coding Guidelines

Designing an open software system such as Zephyr to be secure requires adhering to a defined set of design standards. In [SALT75], the following, widely accepted principles for protection mechanisms are defined to help prevent security violations and limit their impact:

- **Open design** as a design guideline incorporates the maxim that protection mechanisms cannot be kept secret on any system in widespread use. Instead of relying on secret, custom-tailored security measures, publicly accepted cryptographic algorithms and well established cryptographic libraries shall be used.
- **Economy of mechanism** specifies that the underlying design of a system shall be kept as simple and small as possible. In the context of the Zephyr project, this can be realized, e.g., by modular code [PAUL09] and abstracted APIs.
- **Complete mediation** requires that each access to every object and process needs to be authenticated first. Mechanisms to store access conditions shall be avoided if possible.
- **Fail-safe defaults** defines that access is restricted by default and permitted only in specific conditions defined by the system protection scheme, e.g., after successful authentication. Furthermore, default settings for services shall be chosen in a way to provide maximum security. This corresponds to the "Secure by Default" paradigm [MS12].
- **Separation of privilege** is the principle that two conditions or more need to be satisfied before access is granted. In the context of the Zephyr project, this could encompass split keys [PAUL09].
- **Least privilege** describes an access model in which each user, program, and thread, shall have the smallest possible subset of permissions in the system required to perform their task. This positive security model aims to minimize the attack surface of the system.
- **Least common mechanism** specifies that mechanisms common to more than one user or process shall not be shared if not strictly required. The example given in [SALT75] is a function that should be implemented as a shared library executed by each user and not as a supervisor procedure shared by all users.
- **Psychological acceptability** requires that security features are easy to use by the developers in order to ensure their usage and the correctness of its application.

In addition to these general principles, the following points are specific to the development of a secure RTOS:

- **Complementary Security/Defense in Depth**: do not rely on a single threat mitigation approach. In case of the complementary security approach, parts of the threat mitigation are performed by the underlying platform. In case such mechanisms are not provided by the platform, or are not trusted, a defense in depth [MS12] paradigm shall be used.

- **Less commonly used services off by default**: to reduce the exposure of the system to potential attacks, features or services shall not be enabled by default if they are only rarely used (a threshold of 80% is given in [MS12]). For the Zephyr project, this can be realized using the configuration management. Each functionality and module shall be represented as a configuration option and needs to be explicitly enabled. Then, all features, protocols, and drivers not required for a particular use case can be disabled. The user shall be notified if low-level options and APIs are enabled but not used by the application.
- **Change management**: to guarantee a traceability of changes to the system, each change shall follow a specified process including a change request, impact analysis, ratification, implementation, and validation phase. In each stage, appropriate documentation shall be provided. All commits shall be related to a bug report or change request in the issue tracker. Commits without a valid reference shall be denied.

Secure development knowledge

Secure designer

The Zephyr project must have at least one primary developer who knows how to design secure software.

This requires understanding the following design principles, including the 8 principles from Saltzer and Schroeder:

- economy of mechanism (keep the design as simple and small as practical, e.g., by adopting sweeping simplifications)
- fail-safe defaults (access decisions shall deny by default, and projects' installation shall be secure by default)
- complete mediation (every access that might be limited must be checked for authority and be non-bypassable)
- open design (security mechanisms should not depend on attacker ignorance of its design, but instead on more easily protected and changed information like keys and passwords)
- separation of privilege (ideally, access to important objects should depend on more than one condition, so that defeating one protection system won't enable complete access. For example, multi-factor authentication, such as requiring both a password and a hardware token, is stronger than single-factor authentication)
- least privilege (processes should operate with the least privilege necessary)
- least common mechanism (the design should minimize the mechanisms common to more than one user and depended on by all users, e.g., directories for temporary files)
- psychological acceptability (the human interface must be designed for ease of use - designing for "least astonishment" can help)
- limited attack surface (the set of the different points where an attacker can try to enter or extract data)
- input validation with whitelists (inputs should typically be checked to determine if they are valid before they are accepted; this validation should use whitelists (which only accept known-good values), not blacklists (which attempt to list known-bad values)).

Vulnerability Knowledge

A "primary developer" in a project is anyone who is familiar with the project's code base, is comfortable making changes to it, and is acknowledged as such by most other participants in the project. A primary developer would typically make a number of contributions over the past year (via code, documentation, or answering questions). Developers would typically be considered primary developers if they initiated the project (and have not left the project more than three years ago), have the option of receiving information on a private vulnerability reporting channel (if there is one), can accept commits on behalf of the project, or perform final releases of the project software. If there is only one developer, that individual is the primary developer.

At least one of the primary developers **must** know of common kinds of errors that lead to vulnerabilities in this kind of software, as well as at least one method to counter or mitigate each of them.

Examples (depending on the type of software) include SQL injection, OS injection, classic buffer overflow, cross-site scripting, missing authentication, and missing authorization. See the CWE/SANS top 25 or OWASP Top 10 for commonly used lists.

Security Subcommittee

There shall be a "security subcommittee", responsible for enforcing this guideline, monitoring reviews, and improving these guidelines.

This team will be established according to the Zephyr Project charter.

Code Review

The Zephyr project shall use a code review system that all changes are required to go through. Each change shall be reviewed by at least one primary developer that is not the author of the change. This developer shall determine if this change affects the security of the system (based on their general understanding of security), and if so, shall request the developer with vulnerability knowledge, or the secure designer to also review the code. Any of these individuals shall have the ability to block the change from being merged into the mainline code until the security issues have been addressed.

Issues and Bug Tracking

The Zephyr project shall have an issue tracking system (such as JIRA) that can be used to record and track defects that are found in the system.

Because security issues are often sensitive, this issue tracking system shall have a field to indicate a security issue. Setting this field shall result in the issue only being visible to a project-maintained core security team. In addition, there shall be a field to allow core security members to add additional users that will have visibility to a given issue.

This embargo, or limited visibility, shall only be for a fixed duration, with a default being a project-decided value. However, because security considerations are often external to the Zephyr project itself, it may be necessary to increase this embargo time. The time necessary shall be clearly annotated in the issue itself.

The list of issues shall be reviewed at least once a month by the security committee on the Zephyr Project. This review should focus on tracking the fixes, determining if any external parties need to be notified or involved, and determining when to lift the embargo on the issue. The embargo should **not** be lifted via an automated means, but the review team should avoid unnecessary delay in lifting issues that have been resolved.

Modifications to This Document

Changes to this document shall be reviewed by the security committee, and approved by consensus.

Developer Guides

Porting Guides

This section contains details regarding porting the Zephyr kernel to new architectures, SoCs and boards.

Architecture Porting Guide

An architecture port is needed to enable Zephyr to run on an ISA (instruction set architecture) or an ABI (Application Binary Interface) that is not currently supported.

The following are examples of ISAs and ABIs that Zephyr supports:

- x86_32 ISA with System V ABI
- x86_32 ISA with IAMCU ABI
- ARMv7-M ISA with Thumb2 instruction set and ARM Embedded ABI (aeabi)
- ARCv2 ISA

An architecture port can be divided in several parts; most are required and some are optional:

- **The early boot sequence**: each architecture has different steps it must take when the CPU comes out of reset (required).
- **Interrupt and exception handling**: each architecture handles asynchronous and unrequested events in a specific manner (required).
- **Thread context switching**: the Zephyr context switch is dependent on the ABI and each ISA has a different set of registers to save (required).
- **Thread creation and termination**: A thread's initial stack frame is ABI and architecture-dependent, and thread abortion possibly as well (required).
- **Device drivers**: most often, the system clock timer and the interrupt controller are tied to the architecture (some required, some optional).
- **Utility libraries**: some common kernel APIs rely on a architecture-specific implementation for performance reasons (required).
- **CPU idling/power management**: most architectures implement instructions for putting the CPU to sleep (partly optional, most likely very desired).
- **Fault management**: for implementing architecture-specific debug help and handling of fatal error in threads (partly optional).
- **Linker scripts and toolchains**: architecture-specific details will most likely be needed in the build system and when linking the image (required).

Early Boot Sequence

The goal of the early boot sequence is to take the system from the state it is after reset to a state where is can run C code and thus the common kernel initialization sequence. Most of the time, very few steps are needed, while some architectures require a bit more work to be performed.

Common steps for all architectures:

- Setup an initial stack.
- If running an XIP (eXecute-In-Place) kernel, copy initialized data
- from ROM to RAM.
- If not using an ELF loader, zero the BSS section.
- Jump to `_Cstart()`, the early kernel initialization
  - `_Cstart()` is responsible for context switching out of the fake context running at startup into the main thread.

Some examples of architecture-specific steps that have to be taken:

- If given control in real mode on x86_32, switch to 32-bit protected mode.
- Setup the segment registers on x86_32 to handle boot loaders that leave them in an unknown or broken state.
- Initialize a board-specific watchdog on Cortex-M3/4.
- Switch stacks from MSP to PSP on Cortex-M.
- Use a different approach than calling into _Swap() on Cortex-M to prevent race conditions.
- Setup FIRQ and regular IRQ handling on ARCv2.

Interrupt and Exception Handling

Each architecture defines interrupt and exception handling differently.

When a device wants to signal the processor that there is some work to be done on its behalf, it raises an interrupt. When a thread does an operation that is not handled by the serial flow of the software itself, it raises an exception. Both, interrupts and exceptions, pass control to a handler. The handler is known as an ISR (Interrupt Service Routine) in the case of interrupts. The handler perform the work required the exception or the interrupt. For interrupts, that work is device-specific. For exceptions, it depends on the exception, but most often the core kernel itself is responsible for providing the handler.

The kernel has to perform some work in addition to the work the handler itself performs. For example:

- Prior to handing control to the handler:
  - Save the currently executing context.
  - Possibly getting out of power saving mode, which includes waking up devices.
  - Updating the kernel uptime if getting out of tickless idle mode.
- After getting control back from the handler:
  - Decide whether to perform a context switch.
  - When performing a context switch, restore the context being context switched in.

This work is conceptually the same across architectures, but the details are completely different:

- The registers to save and restore.
- The processor instructions to perform the work.
- The numbering of the exceptions.
- etc.

It thus needs an architecture-specific implementation, called the interrupt/exception stub.

Another issue is that the kernel defines the signature of ISRs as:

```
void (*isr)(void *parameter)
```

Architectures do not have a consistent or native way of handling parameters to an ISR. As such there are two commonly used methods for handling the parameter.

- Using some architecture defined mechanism, the parameter value is forced in the stub. This is commonly found in X86-based architectures.

- The parameters to the ISR are inserted and tracked via a separate table requiring the architecture to discover at runtime which interrupt is executing. A common interrupt handler demuxer is installed for all entries of the real interrupt vector table, which then fetches the device's ISR and parameter from the separate table. This approach is commonly used in the ARC and ARM architectures via the `CONFIG_GEN_ISR_TABLES` implementation. You can find examples of the stubs by looking at `_interrupt_enter()` in x86, `_IntExit()` in ARM, `_isr_wrapper()` in ARM, or the full implementation description for ARC in `arch/arc/core/isr_wrapper.S`.

Each architecture also has to implement primitives for interrupt control:

- locking interrupts: `irq_lock()`, `irq_unlock()`.
- registering interrupts: `IRQ_CONNECT()`.
- programming the priority if possible `irq_priority_set()`.
- enabling/disabling interrupts: `irq_enable()`, `irq_disable()`.

注解： `IRQ_CONNECT` is a macro that uses assembler and/or linker script tricks to connect interrupts at build time, saving boot time and text size.

The vector table should contain a handler for each interrupt and exception that can possibly occur. The handler can be as simple as a spinning loop. However, we strongly suggest that handlers at least print some debug information. The information helps figuring out what went wrong when hitting an exception that is a fault, like divide-by-zero or invalid memory access, or an interrupt that is not expected (*spurious interrupt*). See the ARM implementation in `arch/arm/core/fault.c` for an example.

Thread Context Switching

Multi-threading is the basic purpose to have a kernel at all. Zephyr supports two types of threads: preemptible and cooperative.

Two crucial concepts when writing an architecture port are the following:

- Cooperative threads run at a higher priority than preemptible ones, and always preempt them.
- After handling an interrupt, if a cooperative thread was interrupted, the kernel always goes back to running that thread, since it is not preemptible.

A context switch can happen in several circumstances:

- When a thread executes a blocking operation, such as taking a semaphore that is currently unavailable.
- When a preemptible thread unblocks a thread of higher priority by releasing the object on which it was blocked.
- When an interrupt unblocks a thread of higher priority than the one currently executing, if the currently executing thread is preemptible.
- When a thread runs to completion.
- When a thread causes a fatal exception and is removed from the running threads. For example, referencing invalid memory,

Therefore, the context switching must thus be able to handle all these cases.

The kernel keeps the next thread to run in a "cache", and thus the context switching code only has to fetch from that cache to select which thread to run.

There are two types of context switches: *cooperative* and *preemptive*.

- A *cooperative* context switch happens when a thread willfully gives the control to another thread. There are two cases where this happens
  - When a thread explicitly yields.
  - When a thread tries to take an object that is currently unavailable and is willing to wait until the object becomes available.
- A *preemptive* context switch happens either because an ISR or a thread causes an operation that schedules a thread of higher priority than the one currently running, if the currently running thread is preemptible. An example of such an operation is releasing an object on which the thread of higher priority was waiting.

注解: Control is never taken from cooperative thread when one of them is the running thread.

A cooperative context switch is always done by having a thread call the `_Swap()` kernel internal symbol. When `_Swap` is called, the kernel logic knows that a context switch has to happen: `_Swap` does not check to see if a context switch must happen. Rather, `_Swap` decides what thread to context switch in. `_Swap` is called by the kernel logic when an object being operated on is unavailable, and some thread yielding/sleeping primitives.

注解: On x86 and Nios2, `_Swap` is generic enough and the architecture flexible enough that `_Swap` can be called when exiting an interrupt to provoke the context switch. This should not be taken as a rule, since neither the ARM Cortex-M or ARCv2 port do this.

Since `_Swap` is cooperative, the caller-saved registers from the ABI are already on the stack. There is no need to save them in the k_thread structure.

A context switch can also be performed preemptively. This happens upon exiting an ISR, in the kernel interrupt exit stub:

- `_interrupt_enter` on x86 after the handler is called.
- `_IntExit` on ARM.
- `_firq_exit` and `_rirq_exit` on ARCv2.

In this case, the context switch must only be invoked when the interrupted thread was preemptible, not when it was a cooperative one, and only when the current interrupt is not nested.

The kernel also has the concept of "locking the scheduler". This is a concept similar to locking the interrupts, but lighter-weight since interrupts can still occur. If a thread has locked the scheduler, is it temporarily non-preemptible.

So, the decision logic to invoke the context switch when exiting an interrupt is simple:

- If the interrupted thread is not preemptible, do not invoke it.
- Else, fetch the cached thread from the ready queue, and:
  - If the cached thread is not the current thread, invoke the context switch.
  - Else, do not invoke it.

This is simple, but crucial: if this is not implemented correctly, the kernel will not function as intended and will experience bizarre crashes, mostly due to stack corruption.

注解: If running a coop-only system, i.e. if `CONFIG_NUM_PREEMPT_PRIORITIES` is 0, no preemptive context switch ever happens. The interrupt code can be optimized to not take any scheduling decision when this is the case.

Thread Creation and Termination

To start a new thread, a stack frame must be constructed so that the context switch can pop it the same way it would pop one from a thread that had been context switched out. This is to be implemented in an architecture-specific `_new_thread` internal routine.

The thread entry point is also not to be called directly, i.e. it should not be set as the PC (program counter) for the new thread. Rather it must be wrapped in `_thread_entry`. This means that the PC in the stack frame shall be set to `_thread_entry`, and the thread entry point shall be passed as the first parameter to `_thread_entry`. The specifics of this depend on the ABI.

The need for an architecture-specific thread termination implementation depends on the architecture. There is a generic implementation, but it might not work for a given architecture.

One reason that has been encountered for having an architecture-specific implementation of thread termination is that aborting a thread might be different if aborting because of a graceful exit or because of an exception. This is the case for ARM Cortex-M, where the CPU has to be taken out of handler mode if the thread triggered a fatal exception, but not if the thread gracefully exits its entry point function.

This means implementing an architecture-specific version of `k_thread_abort()`, and setting the Kconfig option `CONFIG_ARCH_HAS_THREAD_ABORT` as needed for the architecture (e.g. see `arch/arm//core/cortex_m/Kconfig`).

Device Drivers

The kernel requires very few hardware devices to function. In theory, the only required device is the interrupt controller, since the kernel can run without a system clock. In practice, to get access to most, if not all, of the sanity check test suite, a system clock is needed as well. Since these two are usually tied to the architecture, they are part of the architecture port.

Interrupt Controllers

There can be significant differences between the interrupt controllers and the interrupt concepts across architectures.

For example, x86 has the concept of an IDT and different interrupt controllers. Although modern systems mostly standardized on the APIC (Advanced Programmable Interrupt Controller), some small Quark-based systems use the MVIC (Micro-controller Vectored Interrupt Controller). Also, the position of an interrupt in the IDT determines its priority.

On the other hand, the ARM Cortex-M has the NVIC (Nested Vectored Interrupt Controller) as part of the architecture definition. There is no need for an IDT-like table that is separate from the NVIC vector table. The position in the table has nothing to do with priority of an IRQ: priorities are programmable per-entry.

The ARCv2 has its interrupt unit as part of the architecture definition, which is somewhat similar to the NVIC. However, where ARC defines interrupts as having a one-to-one mapping between exception and interrupt numbers (i.e. exception 1 is IRQ1, and device IRQs start at 16), ARM has IRQ0 being equivalent to exception 16 (and weirdly enough, exception 1 can be seen as IRQ-15).

All these differences mean that very little, if anything, can be shared between architectures with regards to interrupt controllers.

System Clock

x86 has APIC timers and the HPET as part of its architecture definition. ARM Cortex-M has the SYSTICK exception. Finally, ARCv2 has the timer0/1 device.

Kernel timeouts are handled in the context of the system clock timer driver's interrupt handler.

Tickless Idle

The kernel has support for tickless idle. Tickless idle is the concept where no system clock timer interrupt is to be delivered to the CPU when the kernel is about to go idle and the closest timeout expiry is passed a certain threshold. When this condition happens, the system clock is reprogrammed far in the future instead of for a periodic tick. For this to work, the system clock timer driver must support it.

Tickless idle is optional but strongly recommended to achieve low-power consumption.

The kernel has built-in support for going into tickless idle.

The system clock timer driver must implement some hooks to support tickless idle. See existing drivers for examples.

The interrupt entry stub (`_interrupt_enter`, `_isr_wrapper`) needs to be adapted to handle exiting tickless idle. See examples in the code for existing architectures.

Console Over Serial Line

There is one other device that is almost a requirement for an architecture port, since it is so useful for debugging. It is a simple polling, output-only, serial port driver on which to send the console (`printk`, `printf`) output.

It is not required, and a RAM console (`CONFIG_RAM_CONSOLE`) can be used to send all output to a circular buffer that can be read by a debugger instead.

Utility Libraries

The kernel depends on a few functions that can be implemented with very few instructions or in a lock-less manner in modern processors. Those are thus expected to be implemented as part of an architecture port.

- Atomic operators.
  - If instructions do not exist for a given architecture, a generic version that wraps `irq_lock()` or `irq_unlock()` around non-atomic operations exists. It is configured using the `CONFIG_ATOMIC_OPERATIONS_C` Kconfig option.
- Find-least-significant-bit-set and find-most-significant-bit-set.
  - If instructions do not exist for a given architecture, it is always possible to implement these functions as generic C functions.

It is possible to use compiler built-ins to implement these, but be careful they use the required compiler barriers.

CPU Idling/Power Management

The kernel provides support for CPU power management with two functions: `k_cpu_idle()` and `k_cpu_atomic_idle()`.

`k_cpu_idle()` can be as simple as calling the power saving instruction for the architecture with interrupts unlocked, for example `hlt` on x86, `wfi` or `wfe` on ARM, `sleep` on ARC. This function can be called in a loop within a context that does not care if it get interrupted or not by an interrupt before going to sleep. There are basically two scenarios when it is correct to use this function:

- In a single-threaded system, in the only thread when the thread is not used for doing real work after initialization, i.e. it is sitting in a loop doing nothing for the duration of the application.

- In the idle thread.

`k_cpu_atomic_idle()`, on the other hand, must be able to atomically re-enable interrupts and invoke the power saving instruction. It can thus be used in real application code, again in single-threaded systems.

Normally, idling the CPU should be left to the idle thread, but in some very special scenarios, these APIs can be used by applications.

Both functions must exist for a given architecture. However, the implementation can be simply the following steps, if desired:

1. unlock interrupts
2. NOP

However, a real implementation is strongly recommended.

Fault Management

Each architecture provides two fatal error handlers:

- `_NanoFatalErrorHandler`, called by software for unrecoverable errors.
- `_SysFatalErrorHandler`, which makes the decision on how to handle the thread where the error is generated, most likely by terminating it.

See the current architecture implementations for examples.

Toolchain and Linking

Toolchain support has to be added to the build system.

Some architecture-specific definitions are needed in `include/toolchain/gcc.h`. See what exists in that file for currently supported architectures.

Each architecture also needs its own linker script, even if most sections can be derived from the linker scripts of other architectures. Some sections might be specific to the new architecture, for example the SCB section on ARM and the IDT section on x86.

Board Porting Guide

When building an application you must specify the target hardware and the exact board or model. Specifying the board name results in a binary that is suited for the target hardware by selecting the right Zephyr features and components and setting the right Zephyr configuration for that specific target hardware.

A board is defined as a special configuration of an SoC with possible additional components. For example, a board might have sensors and flash memory implemented as additional features on top of what the SoC provides. Such additional hardware is configured and referenced in the Zephyr board configuration.

The board implements at least one SoC and thus inherits all of the features that are provided by the SoC. When porting a board to Zephyr, you should first make sure the SoC is implemented in Zephyr.

Hardware Configuration Hierarchy

Hardware definitions in Zephyr follow a well-defined hierarchy of configurations and layers, below are the layers from top to bottom:

- Board
- SoC
- SoC Series
- SoC Family
- CPU Core
- Architecture

This design contributes to code reuse and implementation of device drivers and features at the bottom of the hierarchy making a board configuration as simple as a selection of features that are implemented by the underlying layers. The figures below shows this hierarchy with a few example of boards currently available in the source tree:

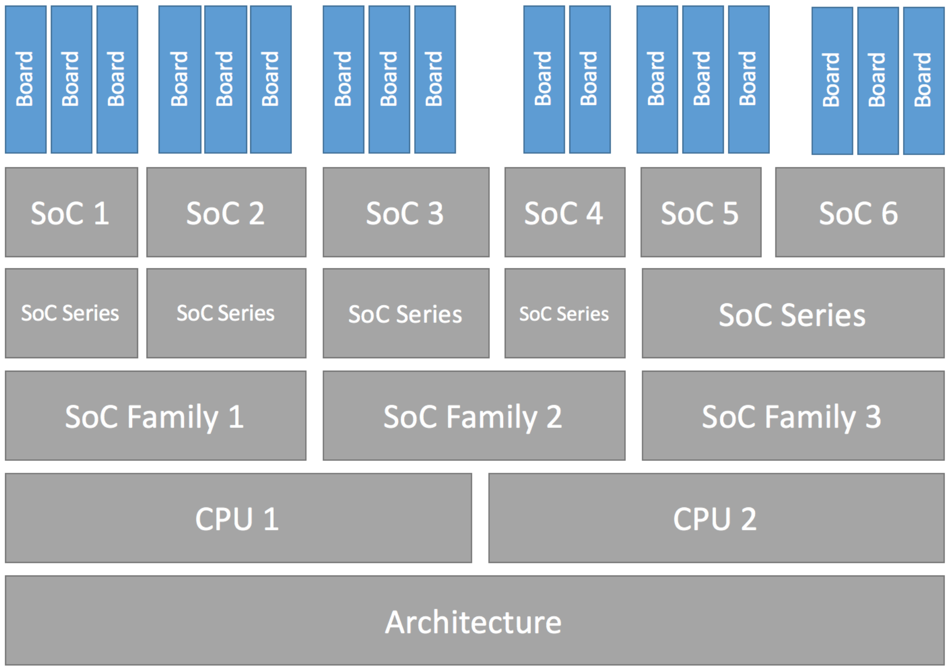


图 1.3: Configuration Hierarchy

Hierarchy Example

Board	FRDM K64F	nRF52 NITROGEN	nRF51XX	Quark SE C1000 Devboard	Arduino 101
SOC	MK64F12	nRF52832	nRF51XX	Quark SE C1000	Curie
SOC Series	Kinetis K6x Series	Nordic NRF52	Nordic NRF51	Quark SE	Quark SE
SOC Family	NXP Kinetis	Nordic NRF5	Nordic NRF5	Quark	Quark
CPU Core	Cortex-M4	Cortex-M4	Cortex-M0+	Lakemont	Lakemont
Architecture	ARM	ARM	ARM	x86	x86

Architecture

If your CPU architecture is already supported by Zephyr, there is no architecture work involved in porting to your board. If your CPU architecture is not supported by the Zephyr kernel, you can add support by following the instructions available at *Architecture Porting Guide*.

CPU Core

Some OS code depends on the CPU core that your board is using. For example, a given CPU core has a specific assembly language instruction set, and may require special cross compiler or compiler settings to use the appropriate instruction set.

If your CPU architecture is already supported by Zephyr, there is no CPU core work involved in porting to your platform or board. You need only to select the appropriate CPU in your configuration and the rest will be taken care of by the configuration system in Zephyr which will select the features implemented by the corresponding CPU.

Platform

This layer implements most of the features that need porting and is split into three layers to allow for code reuse when dealing with implementations with slight differences.

SoC Family

This layer is a container of all SoCs of the same class that, for example implement one single type of CPU core but differ in peripherals and features. The base hardware will in most cases be the same across all SoCs and MCUs of this family.

SoC Series

Moving closer to the SoC, the series is derived from an SoC family. A series is defined by a feature set that serves the purpose of distinguishing different SoCs belonging to the same family.

SoC

Finally, an SoC is actual hardware component that is physically available on a board.

Board

A board implements an SoC with all its features, together with peripherals available on the board that differentiates the board with additional interfaces and features not available in the SoC.

While adding your board support, make sure to add it to the list of `platforms` in the appropriate architecture `.ini` file in `scripts/sanity_chk/arches/`.

Legacy Applications Porting Guide

注解: This document is still work in progress.

This guide will help you move your applications from the nanokernel/microkernel model to the unified kernel. The unified kernel was introduced with zephyr_1.6 which was released late 2016.

A list of the major changes that came with the unified kernel can be found in the section *Changes from Version 1 Kernel*.

API Changes

As described in the section *Kernel APIs* the kernel now has one unified and consistent API with new naming.

Same Arguments

In many cases, a simple search and replace is enough to move from the legacy to the new APIs, for example:

- `task_abort()` -> `k_thread_abort()`
- `task_sem_count_get()` -> `k_sem_count_get()`

Additional Arguments

The number of arguments to some APIs have changed,

- `nano_sem_init()` -> `k_sem_init()`

  This function now accepts 2 additional arguments:

  - Initial semaphore count
  - Permitted semaphore count

  When porting your application, make sure you have set the right arguments. For example, calls to the old API:

  ```
  nano_sem_init(sem)
  ```

  depending on the usage becomes in most cases:

  ```
  k_sem_init(sem, 0, UINT_MAX);
  ```

Return Codes

Many kernel APIs now return 0 to indicate success and a non-zero error code to indicate the reason for failure. You should pay special attention to this change when checking for return codes from kernel APIs, for example:

- `k_sem_take()` now returns 0 on success, in the legacy API `nano_sem_take()` returned 1 when a semaphore is available.

Application Porting

The existing synchronization_sample from the Zephyr tree will be used to guide you with porting a legacy application to the new kernel.

The code has been ported to the new kernel and is shown below:

```c
#include <zephyr.h>
#include <misc/printk.h>

/*
 * The hello world demo has two threads that utilize semaphores and sleeping
 * to take turns printing a greeting message at a controlled rate. The demo
 * shows both the static and dynamic approaches for spawning a thread; a real
 * world application would likely use the static approach for both threads.
 */


/* size of stack area used by each thread */
#define STACKSIZE 1024

/* scheduling priority used by each thread */
#define PRIORITY 7

/* delay between greetings (in ms) */
#define SLEEPTIME 500


/*
 * @param my_name      thread identification string
 * @param my_sem       thread's own semaphore
 * @param other_sem    other thread's semaphore
 */
void helloLoop(const char *my_name,
               struct k_sem *my_sem, struct k_sem *other_sem)
{
        while (1) {
                /* take my semaphore */
                k_sem_take(my_sem, K_FOREVER);

                /* say "hello" */
                printk("%s: Hello World from %s!\n", my_name, CONFIG_ARCH);

                /* wait a while, then let other thread have a turn */
                k_sleep(SLEEPTIME);
                k_sem_give(other_sem);
        }
}

/* define semaphores */

K_SEM_DEFINE(threadA_sem, 1, 1);        /* starts off "available" */
K_SEM_DEFINE(threadB_sem, 0, 1);        /* starts off "not available" */


/* threadB is a dynamic thread that is spawned by threadA */

void threadB(void *dummy1, void *dummy2, void *dummy3)
```

```
{
        ARG_UNUSED(dummy1);
        ARG_UNUSED(dummy2);
        ARG_UNUSED(dummy3);

        /* invoke routine to ping-pong hello messages with threadA */
        helloLoop(__func__, &threadB_sem, &threadA_sem);
}

char __noinit __stack threadB_stack_area[STACKSIZE];


/* threadA is a static thread that is spawned automatically */

void threadA(void *dummy1, void *dummy2, void *dummy3)
{
        ARG_UNUSED(dummy1);
        ARG_UNUSED(dummy2);
        ARG_UNUSED(dummy3);

        /* spawn threadB */
        k_thread_spawn(threadB_stack_area, STACKSIZE, threadB, NULL, NULL, NULL,
                       PRIORITY, 0, K_NO_WAIT);

        /* invoke routine to ping-pong hello messages with threadB */
        helloLoop(__func__, &threadA_sem, &threadB_sem);
}

K_THREAD_DEFINE(threadA_id, STACKSIZE, threadA, NULL, NULL, NULL,
                PRIORITY, 0, K_NO_WAIT);
```

Porting a Nanokernel Application

Below is the code for the application using the legacy kernel:

```
#include <zephyr.h>
#include <misc/printk.h>


/*
 * Nanokernel version of hello world demo has a task and a fiber that utilize
 * semaphores and timers to take turns printing a greeting message at
 * a controlled rate.
 */


/* specify delay between greetings (in ms); compute equivalent in ticks */

#define SLEEPTIME  500
#define SLEEPTICKS (SLEEPTIME * sys_clock_ticks_per_sec / 1000)

#define STACKSIZE 2000

char __stack fiberStack[STACKSIZE];

struct nano_sem nanoSemTask;
struct nano_sem nanoSemFiber;

void fiberEntry(void)
{
        struct nano_timer timer;
        u32_t data[2] = {0, 0};

        nano_sem_init(&nanoSemFiber);
        nano_timer_init(&timer, data);

        while (1) {
                /* wait for task to let us have a turn */
                nano_fiber_sem_take(&nanoSemFiber, TICKS_UNLIMITED);

                /* say "hello" */
                printk("%s: Hello World!\n", __func__);

                /* wait a while, then let task have a turn */
                nano_fiber_timer_start(&timer, SLEEPTICKS);
                nano_fiber_timer_test(&timer, TICKS_UNLIMITED);
                nano_fiber_sem_give(&nanoSemTask);
        }
}

void main(void)
{
        struct nano_timer timer;
        u32_t data[2] = {0, 0};

        task_fiber_start(&fiberStack[0], STACKSIZE,
                         (nano_fiber_entry_t) fiberEntry, 0, 0, 7, 0);

        nano_sem_init(&nanoSemTask);
        nano_timer_init(&timer, data);

        while (1) {
                /* say "hello" */
                printk("%s: Hello World!\n", __func__);

                /* wait a while, then let fiber have a turn */
                nano_task_timer_start(&timer, SLEEPTICKS);
                nano_task_timer_test(&timer, TICKS_UNLIMITED);
                nano_task_sem_give(&nanoSemFiber);

                /* now wait for fiber to let us have a turn */
                nano_task_sem_take(&nanoSemTask, TICKS_UNLIMITED);
        }
}

```

Porting a Microkernel Application

The MDEF feature of the legacy kernel has been eliminated. Consequently, all kernel objects are now defined directly in code.

Below is the code for the application using the legacy kernel:

```
#include <zephyr.h>
#include <misc/printk.h>

/*
 * Microkernel version of hello world demo has two tasks that utilize
 * semaphores and sleeps to take turns printing a greeting message at
 * a controlled rate.
 */


/* specify delay between greetings (in ms); compute equivalent in ticks */

#define SLEEPTIME  500
#define SLEEPTICKS (SLEEPTIME * sys_clock_ticks_per_sec / 1000)

/*
 *
 * @param taskname    task identification string
 * @param mySem       task's own semaphore
 * @param otherSem    other task's semaphore
 *
 */
void helloLoop(const char *taskname, ksem_t mySem, ksem_t otherSem)
{
        while (1) {
                task_sem_take(mySem, TICKS_UNLIMITED);

                /* say "hello" */
                printk("%s: Hello World from %s!\n", taskname, CONFIG_ARCH);

                /* wait a while, then let other task have a turn */
                task_sleep(SLEEPTICKS);
                task_sem_give(otherSem);
        }
}

void taskA(void)
{
        /* taskA gives its own semaphore, allowing it to say hello right away */
        task_sem_give(TASKASEM);

        /* invoke routine that allows task to ping-pong hello messages with taskB */
        helloLoop(__func__, TASKASEM, TASKBSEM);
}

void taskB(void)
{
        /* invoke routine that allows task to ping-pong hello messages with taskA */
        helloLoop(__func__, TASKBSEM, TASKASEM);
}

```

A microkernel application defines the used objects in an MDEF file, for this porting sample using the synchronization_sample, the file is shown below:

```
% Application       : Hello demo

% TASK NAME  PRIO ENTRY STACK GROUPS
% =================================
  TASK TASKA     7 taskA  1024 [EXE]
  TASK TASKB     7 taskB  1024 [EXE]

% SEMA NAME
% ============
  SEMA TASKASEM
  SEMA TASKBSEM
```

In the unified kernel the semaphore will be defined in the code as follows:

```
/* define semaphores */

K_SEM_DEFINE(threadA_sem, 1, 1);        /* starts off "available" */
K_SEM_DEFINE(threadB_sem, 0, 1);        /* starts off "not available" */
```

The threads (previously named tasks) are defined in the code as follows, for thread A:

```
K_THREAD_DEFINE(threadA_id, STACKSIZE, threadA, NULL, NULL, NULL,
                PRIORITY, 0, K_NO_WAIT);
```

Thread B (taskB in the microkernel) will be spawned dynamically from thread A (See *Thread Creation* section):

```
        /* spawn threadB */
        k_thread_spawn(threadB_stack_area, STACKSIZE, threadB, NULL, NULL, NULL,
                       PRIORITY, 0, K_NO_WAIT);
```

Migrating from Zephyr v1.6 IP Stack to v1.7

Zephyr v1.6 and earlier is using network IP stack that has its origin in uIP. The uIP stack was modified heavily to use it in Zephyr. The uIP stack had limitations described below that required new and native IP stack to be developed.

This document is a high level description of the changes between these two major Zephyr releases. For individual changes, the application developer can find more details in the network header file documentation in include/net/.

Because of this new native IP stack, the following changes are required to migrate applications using the older v1.6 IP stack to the new v1.7 IP stack:

Dual IPv6 and IPv4 stack support. In Zephyr v1.6, applications could not use both IPv6 and IPv4 simultaneously. This is changed in Zephyr v1.7 and the IP stack supports both IPv6 and IPv4 at the same time. In practice this means that applications should be prepared to support both IPv6 and IPv4 in the code.

Multiple simultaneous network technologies support. In Zephyr v1.7 it is possible to have multiple network technologies enabled at the same time. This means that applications can utilize concurrently e.g., IEEE 802.15.4 and Bluetooth IP networking. The different network technologies are abstracted to network interfaces and there can be multiple network interfaces in the system depending on configuration.

Network Kconfig options are changed. Most of the networking configuration options are renamed. Please check the *Networking* documentation for the new names.

All uIP based APIs are gone. Those APIs were not public in v1.6 but applications could call them anyway. These uIP APIs were mainly used to set IP address etc. management style operations. The new management APIs can be found in net_if.h and net_mgmt.h in Zephyr v1.7.

Network buffer management is changed. In earlier Zephyr versions, there were big 1280 byte long buffers that applications could utilize. In Zephyr v1.7 this is changed so that device memory is utilized more efficiently. Now small buffer fragments are allocated to store the data, and these fragments can then be chained together to store larger amount of data. For applications this means that the memory received from network or sent to network is not contiguous and application should use the helper functions found in nbuf.h when reading and writing the network data.

Network context/socket API is changed. The new API found in net_context.h is more BSD-socket-like than the earlier API. The new context API is not fully BSD socket compatible as it needs to support both synchronous and asynchronous operations. Porting BSD socket application to use the net_context_* API is easier than in Zephyr v1.6.

CoAP library API is changed. The Zephyr v1.6 CoAP API is removed. The new Zephyr v1.7 API is called ZoAP and it can be found in zoap.h header file.

The TinyDTLS library is removed. The tinydtls crypto library was used only by CoAP in Zephyr v1.6 and it is removed in Zephyr v1.7. It is replaced by mbedtls library.

Application Development Primer

Overview

The Zephyr Kernel's build system is based on the Kbuild system used in the Linux kernel.

The build system is an application-centric system and requires an application build to initiate building the kernel source tree. The application build drives the configuration and build process of both the application and kernel, compiling them into a single binary.

The Zephyr Kernel's base directory hosts the kernel source code, the configuration options, and the kernel build definitions.

The files in the application directory links the kernel with the application. It hosts the definitions of the application, for example, application-specific configuration options and the application's source code.

An application in the simplest form has the following structure:

- **Application source code files**: An application typically provides one or more application-specific files, written in C or assembly language. These files are usually located in a sub-directory called `src`.
- **Kernel configuration files**: An application typically provides a configuration file (`.conf`) that specifies values for one or more kernel configuration options. If omitted, the application's existing kernel configuration option values are used; if no existing values are provided, the kernel's default configuration values are used.
- **Makefile**: This file tells the build system where to find the files mentioned above, as well as the desired target board configuration.

Once the application has been defined, it can be built with a single `make` call. The results of the build process are located in a sub-directory called `outdir/BOARD`. This directory contains the files generated by the build process, the most notable of which are listed below.

- The `.config` file that contains the configuration settings used to build the application.
- The various object files (`.o` files and `.a` files) containing custom-built kernel and application-specific code.
- The `zephyr.elf` file that contains the final combined application and kernel binary.

Application Structure

Create one directory for your application and a sub-directory for the application's source code; this makes it easier to organize directories and files in the structure that the kernel expects.

1. Create an application directory structure outside of the kernel's installation directory tree. Often this is your workspace directory.

2. In a console terminal, navigate to a location where you want your application to reside.
3. Create the application's directory, enter:

```
$ mkdir app
```

注解： This directory and the path to it, are referred to in the documentation as `~/app`.

4. Create a source code directory in your `~/app`, enter:

```
$ cd app
$ mkdir src
```

The source code directory `~/app/src` is created.

```
-- app
   |-- src
```

Application Definition

An application is integrated into the build system by including the Makefile.inc file provided.

```
include $(ZEPHYR_BASE)/Makefile.inc
```

The following predefined variables configure the development project:

- **`ZEPHYR_BASE`**: Sets the path to the kernel's base directory. This variable is usually set by the `zephyr_env.sh` script. It can be used to get the kernel's base directory, as used in the Makefile.inc inclusion above, or it can be overridden to select an specific instance of the kernel.
- **`PROJECT_BASE`**: Provides the developer's application project directory path. It is set by the `Makefile.inc` file.
- **`SOURCE_DIR`**: Overrides the default value for the application's source code directory. The developer source code directory is set to `$(PROJECT_BASE)/src/` by default. This directory name should end with slash **'/'**.
- **`BOARD`**: Selects the board that the application's build will use for the default configuration.
- **`CONF_FILE`**: Indicates the name of a configuration fragment file. This file includes the kconfig configuration values that override the default configuration values.
- **`O`**: Optional. Indicates the output directory that Kconfig uses. The output directory stores all the files generated during the build process. The default output directory is the `$(PROJECT_BASE)/outdir` directory.

Makefiles

Overview

The build system defines a set of conventions for the correct use of Makefiles in the kernel source directories. The correct use of Makefiles is driven by the concept of recursion.

In the recursion model, each Makefile within a directory includes the source code and any subdirectories to the build process. Each subdirectory follows the same principle. Developers can focus exclusively in their own work. They integrate their module with the build system by adding a very simple Makefile following the recursive model.

Makefile Conventions

The following conventions restrict how to add modules and Makefiles to the build system. These conventions ensure the correct implementation of the recursive model.

- Each source code directory must contain a single Makefile. Directories without a Makefile are not considered source code directories.
- The scope of every Makefile is restricted to the contents of that directory. A Makefile can only make a direct reference to files and subdirectories on the same level or below.
- Makefiles list the object files that are included in the link process. The build system finds the source file that generates the object file by matching the object file name to the source file.
- Parent directories add their child directories into the recursion model.
- The root Makefile adds the directories in the kernel base directory into the recursion model.

Adding Source Files

The Makefile must refer the source build indirectly, specifying the object file that results from the source file using the `obj-y` variable. For example, if the file that we want to add is a C file named `<file>.c` the following line should be added in the Makefile:

```
obj-y += <file>.o
```

注解: The same method applies for assembly files with the .S extension.

Source files can be added conditionally using configuration options. For example, if the option `CONFIG_VAR` is set and it implies that a source file must be added in the compilation process, then the following line adds the source code conditionally:

```
obj-$(CONFIG_VAR) += <file>.o
```

Adding Directories

Add a subdirectory to the build system by editing the Makefile in its directory. The subdirectory is added using the `obj-y` variable. The correct syntax to add a subdirectory into the build queue is:

```
obj-y += <directory_name>/
```

The backslash at the end of the directory name signals the build system that a directory, and not a file, is being added to the build queue.

The conventions require us to add only one directory per line and never to mix source code with directory recursion in a single `obj-y` line. This helps keep the readability of the Makefile by making it clear when an item adds an additional lever of recursion.

Directories can also be conditionally added:

```
obj-y-$(CONFIG_VAR) += <directory_name>/
```

The subdirectory must contain its own Makefile following the rules described in *Makefile Conventions*.

Application Makefile

Create an application Makefile to define basic information, such as the board configuration used by the application. The build system uses the Makefile to build a `zephyr.elf` image that contains both the application and the kernel libraries.

1. Open the `Makefile` and add the following mandatory entries using any standard text editor.

   注解： Ensure that there is a space before and after each =.

2. Add the name of the default board configuration for your application on a new line:

   ```
   BOARD = board_configuration_name
   ```

   The supported boards can be found in boards.

3. Add the name of the default kernel configuration file for your application on a new line:

   ```
   CONF_FILE ?= kernel_configuration_name
   ```

4. Include the mandatory `Makefile` on a new line:

   ```
   include ${ZEPHYR_BASE}/Makefile.inc
   ```

5. Save and close the `Makefile`.

Below is an example Makefile:

```
BOARD = qemu_x86
CONF_FILE = prj.conf

include ${ZEPHYR_BASE}/Makefile.inc
```

Application Configuration

The application's kernel is configured using a set of configuration options that can be customized for application-specific purposes. The Zephyr build system takes a configuration option's value from the first source in which it is specified.

The available sources are (in order):

1. The application's current configuration. (i.e. The `.config` file.)
2. The application's default configuration. (i.e. The `.conf` file.)
3. The board configuration used by the application. (i.e. The board's `.defconfig` file.)
4. The kernel's default configuration. (i.e. One of the kernel's `Kconfig` files.)

For information on available kernel configuration options, including inter-dependencies between options, see the configuration.

Default Board Configuration

An application's `.conf` file defines its default kernel configuration. The settings in this file override or augment the board configuration settings.

The board configuration settings can be viewed LENGTHlWRONGEPHY _BASE/boards/ARCHITECTURE/BOARD/BOARD_defconfig‘.

注解： When the default board configuration settings are sufficient for your application, a `.conf` file is not needed. Skip ahead to *Overriding Default Configuration*.

1. Navigate to the `app`, and create the `prj.conf` file. Enter:

   ```
   $ vim prj.conf
   ```

   The default name is `prj.conf`. The filename must match the `CONF_FILE` entry in the application `Makefile`.

2. Edit the file and add the appropriate configuration entries.

   (a) Add each configuration entry on a new line.

   (b) Begin each entry with `CONFIG_`.

   (c) Ensure that each entry contains no spaces (including on either side of the = sign).

   (d) Use a # followed by a space to comment a line.

   The example below shows a comment line and a board configuration override in the `prj.conf`.

   ```
   # Enable printk for debugging
   CONFIG_PRINTK=y
   ```

3. Save and close the file.

Overriding Default Configuration

Override the default board and kernel configuration to temporarily alter the application’s configuration, perhaps to test the effect of a change.

注解： If you want to permanently alter the configuration you should revise the `.conf` file.

Configure the kernel options using a menu-driven interface. While you can add entries manually, using the configuration menu is a preferred method.

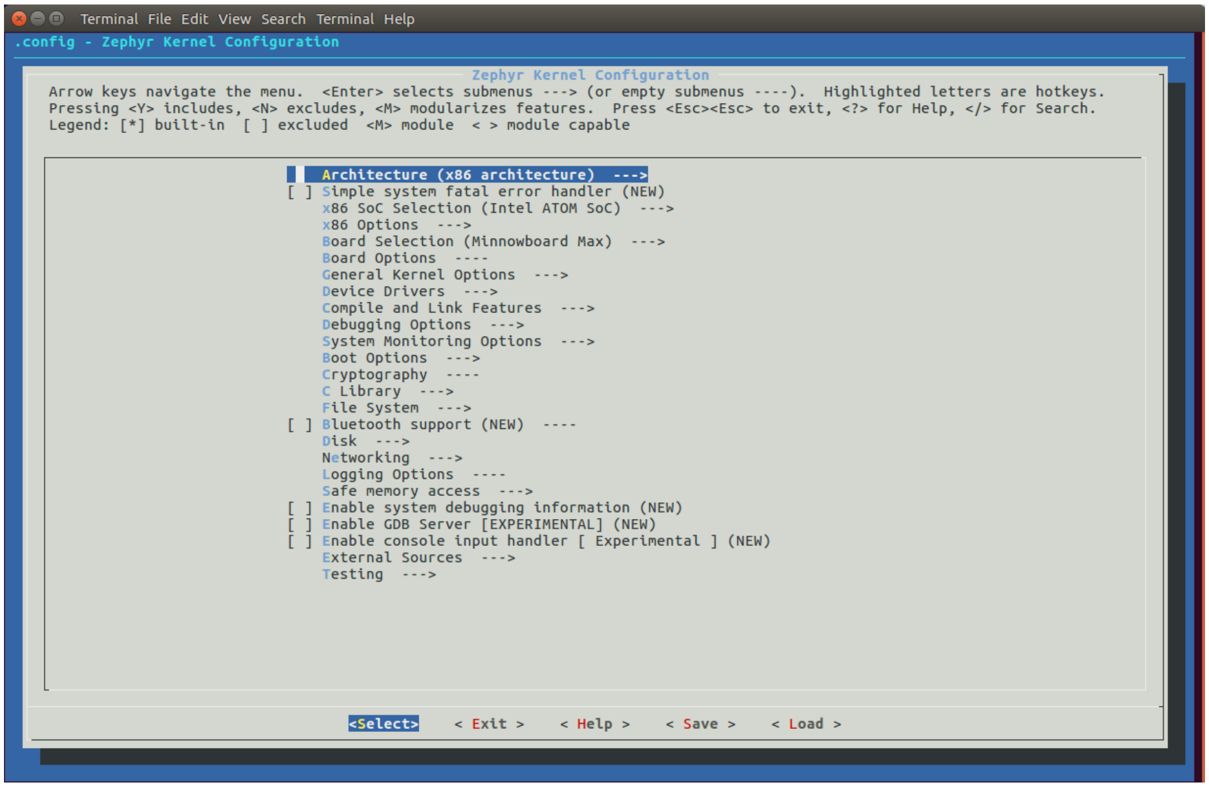
1. Run the **`make menuconfig`** rule to launch the menu-driven interface.

   (a) In a terminal session, navigate to the application directory (`~/app`).

   (b) Enter the following command:

   ```
   $ make [BOARD=<type>] menuconfig
   ```

   A question-based menu opens that allows you to set individual configuration options.



```
.config - Zephyr Kernel Configuration
                          Zephyr Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.
  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                              Architecture (x86 architecture)  --->
                          [ ] Simple system fatal error handler (NEW)
                              x86 SoC Selection (Intel ATOM SoC)  --->
                              x86 Options  --->
                              Board Selection (Minnowboard Max)  --->
                              Board Options  ----
                              General Kernel Options  --->
                              Device Drivers  --->
                              Compile and Link Features  --->
                              Debugging Options  --->
                              System Monitoring Options  --->
                              Boot Options  --->
                              Cryptography  ----
                              C Library  --->
                              File System  --->
                          [ ] Bluetooth support (NEW)  ----
                              Disk  --->
                              Networking  --->
                              Logging Options  ----
                              Safe memory access  --->
                          [ ] Enable system debugging information (NEW)
                          [ ] Enable GDB Server [EXPERIMENTAL] (NEW)
                          [ ] Enable console input handler [ Experimental ] (NEW)
                              External Sources  --->
                              Testing  --->

                 <Select>    < Exit >    < Help >    < Save >    < Load >
```

2. Set kernel configuration values using the following key commands:
   - Use the arrow keys to navigate within any menu or list.
   - Press `Enter` to select a menu item.
   - **Type an upper case `Y` or `N` in the** square brackets *[ ]* to enable or disable a kernel configuration option.
   - Type a numerical value in the round brackets *( )*.
   - Press `Tab` to navigate the command menu at the bottom of the display.

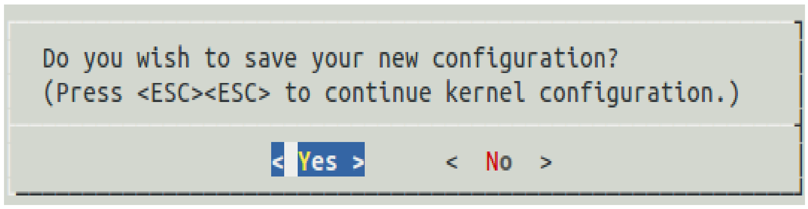
注解: When a non-default entry is selected for options that are non-numerical, an asterisk `*` appears between the square brackets in the display. There is nothing added added the display when you select the option's default.

3. For information about any option, select the option and tab to <*Help* > and press `Enter`.

   Press `Enter` to return to the menu.

4. After configuring the kernel options for your application, tab to < *Save* > and press `Enter`.

   The following dialog opens with the < *Ok* > command selected:



```
Do you wish to save your new configuration?
(Press <ESC><ESC> to continue kernel configuration.)

            < Yes >        <  No  >
```

5. Press `Enter` to save the kernel configuration options to the default file name; alternatively, type a file name and press `Enter`.

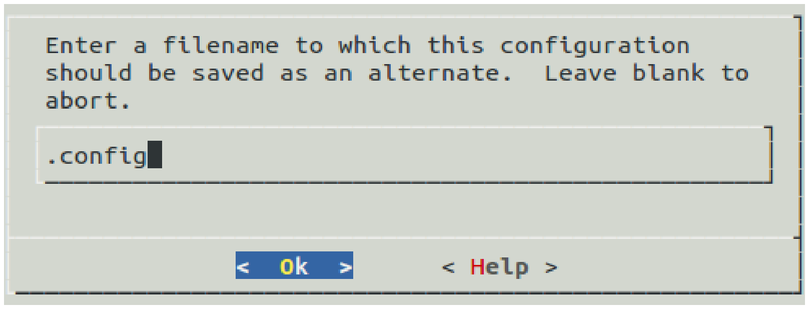
   Typically, you will save to the default file name unless you are experimenting with various configuration scenarios.

   An `outdir` directory is created in the application directory. The outdir directory contains symbolic links to files under `$ZEPHYR_BASE`.

   注解: At present, only a `.config` file can be built. If you have saved files with different file names and want to build with one of these, change the file name to `.config`. To keep your original `.config`, rename it to something other than `.config`.

   Kernel configuration files, such as the `.config` file, are saved as hidden files in `outdir`. To list all your kernel configuration files, enter **`ls -a`** at the terminal prompt.

   The following dialog opens, displaying the file name the configuration was saved to.

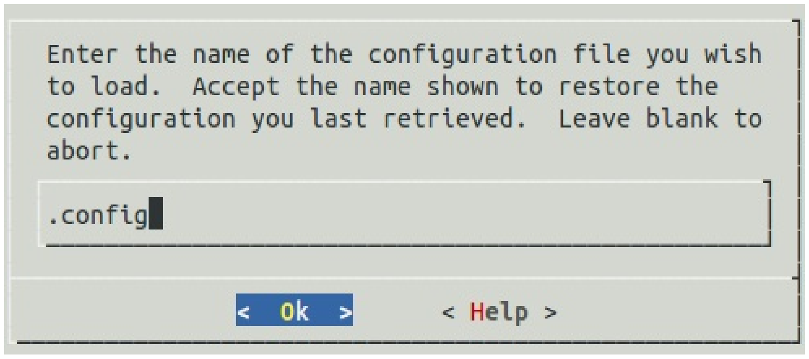


```
Enter a filename to which this configuration
should be saved as an alternate.  Leave blank to
abort.

.config
```

6. Press `Enter` to return to the options menu.
7. To load any saved kernel configuration file, tab to < *Load* > and press `Enter`.

   The following dialog opens with the < *Ok* > command selected:

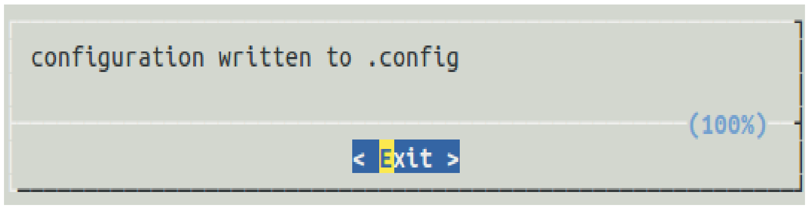


```
Enter the name of the configuration file you wish
to load.  Accept the name shown to restore the
configuration you last retrieved.  Leave blank to
abort.

.config
```

8. To load the last saved kernel configuration file, press < *Ok* >, or to load another saved configuration file, type the file name, then select < *Ok* >.
9. Press `Enter` to load the file and return to the main menu.
10. To exit the menu configuration, tab to < *Exit* > and press `Enter`.

    The following confirmation dialog opens with the < *Yes* > command selected.



```
configuration written to .config
```

11. Press `Enter` to retire the menu display and return to the console command line.

Application-Specific Code

Application-specific source code files are normally added to the application's `src` directory. If the application adds a large number of files the developer can group them into sub-directories under `src`, to whatever depth is needed. The developer must supply a `Makefile` for the `src` directory, as well as for each sub-directory under `src`.

注解: These Makefiles are distinct from the top-level application Makefile that appears in `~/app`.

Application-specific source code should not use symbol name prefixes that have been reserved by the kernel for its own use. For more information, see

Naming Conventions.

The following requirements apply to all Makefiles in the `src` directory, including any sub-directories it may have.

- During the build process, Kbuild identifies the source files to compile into object files by matching the file names identified in the application's Makefile(s).

  重要: A source file that is not referenced by any Makefile is not included when the application is built.

- A Makefile cannot directly reference source code. It can only reference object files (`.o` files) produced from source code files.
- A Makefile can only reference files in its own directory or in the sub-directories of that directory.
- A Makefile may reference multiple files from a single-line entry. However, a separate line must be used to reference each directory.

1. Create a directory structure for your source code in `src` and add your source code files to it.
2. Create a `Makefile` in the `src` directory. Then create an additional `Makefile` in each of the sub-directories under the `src` directory, if any.

   (a) Use the following syntax to add file references:

   ```
   obj-y += file1.o file2.o
   ```

   (b) Use the following syntax to add directory references:

   ```
   obj-y += directory_name/**
   ```

This example is taken from the dining-philosophers-sample. To examine this file in context, navigate to: `$ZEPHYR_BASE/samples/philosophers/src`.

```
obj-y = main.o
```

Support for building third-party library code

It is possible to build library code outside the application's `src` directory but it is important that both application and library code targets the same Application Binary Interface (ABI). On most architectures there are compiler flags that control the ABI targeted, making it important that both libraries and applications have certain compiler flags in common. It may also be useful for glue code to have access to Zephyr kernel header files.

To make it easier to integrate third-party components, the Zephyr build system includes a special build target, `outputexports`, that takes a number of critical variables from the Zephyr build system and copies them into `Makefile.export`. This allows the critical variables to be included by wrapper code for use in a third-party build system.

The following variables are recommended for use within the third-party build (see `Makefile.export` for the complete list of exported variables):

- `CROSS_COMPILE`, together with related convenience variables to call the cross-tools directly (including `AR`, `AS`, `CC`, `CXX`, `CPP` and `LD`).
- `ARCH` and `BOARD`, together with several variables that identify the Zephyr kernel version.
- `KBUILD_CFLAGS`, `NOSTDINC_FLAGS` and `ZEPHYRINCLUDE` all of which should normally be added, in that order, to `CFLAGS` (or `CXXFLAGS`).
- All kconfig variables, allowing features of the library code to be enabled/disabled automatically based on the Zephyr kernel configuration.

`samples/application_development/static_lib` is a sample project that demonstrates some of these features.

Build an Application

The Zephyr build system compiles and links all components of an application into a single application image that can be run on simulated hardware or real hardware.

1. Navigate to the application directory `~/app`.
2. Enter the following command to build the application's `zephyr.elf` image using the configuration settings for the board type specified in the application's `Makefile`.

   ```
   $ make
   ```

   If desired, you can build the application using the configuration settings specified in an alternate `.conf` file using the `CONF_FILE` parameter. These settings will override the settings in the application's `.config` file or its default `.conf` file. For example:

   ```
   $ make CONF_FILE=prj.alternate.conf
   ```

   If desired, you can build the application for a different board type than the one specified in the application's `Makefile` using the `BOARD` parameter. For example:

   ```
   $ make BOARD=arduino_101
   ```

   Both the `CONF_FILE` and `BOARD` parameters can be specified when building the application.

Rebuilding an Application

Application development is usually fastest when changes are continually tested. Frequently rebuilding your application makes debugging less painful as the application becomes more complex. It's usually a good idea to rebuild and test after any major changes to the application's source files, Makefiles, or configuration settings.

重要: The Zephyr build system rebuilds only the parts of the application image potentially affected by the changes. Consequently, rebuilding an application is often significantly faster than building it the first time.

Sometimes the build system doesn't rebuild the application correctly because it fails to recompile one or more necessary files. You can force the build system to rebuild the entire application from scratch with the following procedure:

1. Navigate to the application directory `~/app`.
2. Enter the following command to delete the application's generated files for the specified board type, except for the `.config` file that contains the application's current configuration information.

   ```
   $ make [BOARD=<type>] clean
   ```

   Alternatively, enter the following command to delete *all* generated files for *all* board types, including the `.config` files that contain the application's current configuration information for those board types.

   ```
   $ make pristine
   ```

3. Rebuild the application normally following the steps specified in *Build an Application* above.

Run an Application

An application image can be run on real or emulated hardware. The kernel has built-in emulator support for QEMU. It allows you to run and test an application virtually, before (or in lieu of) loading and running it on actual target hardware.

1. Open a terminal console and navigate to the application directory `~/app`.
2. Enter the following command to build and run the application using a QEMU-supported board configuration, such as qemu_cortex_m3 or qemu_x86.

   ```
   $ make [BOARD=<type> ...] run
   ```

   The Zephyr build system generates a `zephyr.elf` image file and then begins running it in the terminal console.

3. Press `Ctrl A, X` to stop the application from running in QEMU.

   The application stops running and the terminal console prompt redisplays.

Application Debugging

This section is a quick hands-on reference to start debugging your application with QEMU. Most content in this section is already covered on QEMU and GNU_Debugger reference manuals.

In this quick reference you find shortcuts, specific environmental variables and parameters that can help you to quickly set up your debugging environment.

The simplest way to debug an application running in QEMU is using the GNU Debugger and setting a local GDB server in your development system through QEMU.

You will need an ELF binary image for debugging purposes. The build system generates the image in the output directory. By default, the kernel binary name is `zephyr.elf`. The name can be changed using a Kconfig option.

We will use the standard 1234 TCP port to open a GDB (GNU Debugger) server instance. This port number can be changed for a port that best suits the development environment.

You can run Qemu to listen for a "gdb connection" before it starts executing any code to debug it.

```
qemu -s -S <image>
```

will setup Qemu to listen on port 1234 and wait for a GDB connection to it.

The options used above have the following meaning:

- `-S` Do not start CPU at startup; rather, you must type 'c' in the monitor.
- `-s` Shorthand for `-gdb tcp::1234`: open a GDB server on TCP port 1234.

To debug with QEMU and to start a GDB server and wait for a remote connect, run the following inside an application:

```
make BOARD=qemu_x86 debugserver
```

The build system will start a QEMU instance with the CPU halted at startup and with a GDB server instance listening at the TCP port 1234.

Using a local GDB configuration `.gdbinit` can help initialize your GDB instance on every run. In this example, the initialization file points to the GDB server instance. It configures a connection to a remote target at the local host on the TCP port 1234. The initialization sets the kernel's root directory as a reference.

The `.gdbinit` file contains the following lines:

```
target remote localhost:1234
dir ZEPHYR_BASE
```

注解: Substitute ZEPHYR_BASE for the current kernel's root directory.

Execute the application to debug from the same directory that you chose for the `gdbinit` file. The command can include the `--tui` option to enable the use of a terminal user interface. The following commands connects to the GDB server using `gdb`. The command loads the symbol table from the elf binary file. In this example, the elf binary file name corresponds to `zephyr.elf` file:

```
$ gdb --tui zephyr.elf
```

注解: The GDB version on the development system might not support the –tui option.

If you are not using a .gdbinit file, issue the following command inside GDB to connect to the remove GDB server on port 1234:

```
(gdb) target remote localhost:1234
```

Finally, The command below connects to the GDB server using the Data Displayer Debugger (`ddd`). The command loads the symbol table from the elf binary file, in this instance, the `zephyr.elf` file.

The DDD (Data Displayer Debugger) may not be installed in your development system by default. Follow your system instructions to install it.

```
ddd --gdb --debugger "gdb zephyr.elf"
```

Both commands execute the GDB (GNU Debugger). The command name might change depending on the toolchain you are using and your cross-development tools.

API Documentation

Welcome to the Zephyr Project's API (Application Programing Interface) documentation.

This section contains the API documentation automatically extracted from the code. If you are looking for a specific API, enter it on the search box. The search results display all sections containing information about that API.

The Zephyr APIs are used the same way on all SoCs and boards.

Kernel APIs

This section contains APIs for the kernel's core services, as described in the *Zephyr Kernel Primer*.

重要: Unless otherwise noted these APIs can be used by threads, but not by ISRs.

Threads

A thread is an independently scheduled series of instructions that implements a portion of an application's processing. Threads are used to perform processing that is too lengthy or too complex to be performed by an ISR. (See *Threads*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Workqueues

A workqueue processes a series of work items by executing the associated functions in a dedicated thread. Workqueues are typically used by an ISR or high-priority thread to offload non-urgent processing. (See *Workqueue Threads*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Clocks

Kernel clocks enable threads and ISRs to measure the passage of time with either normal and high precision. (See *Kernel Clocks*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Timers

Timers enable threads to measure the passage of time, and to optionally execute an action when the timer expires. (See *Timers*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Memory Slabs

Memory slabs enable the dynamic allocation and release of fixed-size memory blocks. (See *Memory Slabs*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Memory Pools

Memory pools enable the dynamic allocation and release of variable-size memory blocks. (See *Memory Pools*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Heap Memory Pool

The heap memory pools enable the dynamic allocation and release of memory in a `malloc()`-like manner. (See *Heap Memory Pool*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Semaphores

Semaphores provide traditional counting semaphore capabilities. (See *Semaphores*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Mutexes

Mutexes provide traditional reentrant mutex capabilities with basic priority inheritance. (See *Mutexes*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Alerts

Alerts enable an application to perform asynchronous signaling, somewhat akin to Unix-style signals. (See *Alerts*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Fifos

Fifos provide traditional first in, first out (FIFO) queuing of data items of any size. (See *Fifos*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Lifos

Lifos provide traditional last in, first out (LIFO) queuing of data items of any size. (See *Lifos*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Stacks

Stacks provide traditional last in, first out (LIFO) queuing of 32-bit data items. (See *Stacks*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Message Queues

Message queues provide a simple message queuing mechanism for fixed-size data items. (See *Message Queues*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Mailboxes

Mailboxes provide an enhanced message queuing mechanism for variable-size messages. (See *Mailboxes*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Pipes

Pipes provide a traditional anonymous pipe mechanism for sending variable-size chunks of data, in whole or in part. (See *Pipes*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Interrupt Service Routines (ISRs)

An interrupt service routine is a series of instructions that is executed asynchronously in response to a hardware or software interrupt. (See *Interrupts*.)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Atomic Services

The atomic services enable multiple threads and ISRs to read and modify 32-bit variables in an uninterruptible manner. (See *Atomic Services*.)

重要： All atomic services APIs can be used by both threads and ISRs.

警告： doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Floating Point Services

The floating point services enable threads to use a board's floating point registers. (See *Floating Point Services*.)

警告： doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Ring Buffers

Ring buffers enable simple first in, first out (FIFO) queuing of variable-size data items. (See *Ring Buffers*.)

警告： doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Device Driver Interface

Device Model

警告： doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Bluetooth API

This is the full set of available Bluetooth APIs. It's important to note that the set that will in practice be available for the application depends on the exact Kconfig options that have been chosen, since most of the Bluetooth functionality is build-time selectable. E.g. any connection-related APIs require `CONFIG_BT_CONN` and any BR/EDR (Bluetooth Classic) APIs require `CONFIG_BT_BREDR`.

Generic Access Profile (GAP)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Connection Management

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Generic Attribute Profile (GATT)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Mesh Profile

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Universal Unique Identifiers (UUIDs)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Logical Link Control and Adaptation Protocol (L2CAP)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Serial Port Emulation (RFCOMM)

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Data Buffers

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Persistent Storage

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

HCI Drivers

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

HCI RAW channel

HCI RAW channel API is intended to expose HCI interface to the remote entity. The local Bluetooth controller gets owned by the remote entity and host Bluetooth stack is not used. RAW API provides direct access to packets which are sent and received by the Bluetooth HCI driver.

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Networking API

This is the full set of networking public APIs. Their exposure depends on relevant Kconfig options. For instance IPv6 related APIs will not be present if `CONFIG_NET_IPV6` has not been selected.

Network core helpers

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network buffers

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network packet management

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

IPv4/IPv6 primitives and helpers

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network Management

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network layer 2 management

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network link address

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Application network context

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

BSD Sockets compatible API

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network offloading support

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network statistics

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Trickle timer support

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

UDP

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network technologies

Ethernet

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

IEEE 802.15.4

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Network and application libraries

Network application

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

DHCPv4

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

MQTT 3.1.1

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

CoAP

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

DNS Resolve

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

HTTP

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Input / Output Driver APIs

ADC Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

GPIO Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

I2C Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

I2S Interface

The I2S (Inter-IC Sound) API provides support for the standard I2S interface as well as common non-standard extensions such as PCM Short/Long Frame Sync and Left/Right Justified Data Formats.

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

IPM Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

PWM Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Pinmux Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

SPI Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Random Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

UART Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Sensor Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Power Management APIs

Power Management Hook Interface

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Device Power Management APIs

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

File System APIs

File System Functions

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

File System Data Structures

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Device and Driver Support

Device Drivers and Device Model

Introduction

The Zephyr kernel supports a variety of device drivers. The specific set of device drivers available for an application's board configuration varies according to the associated hardware components and device driver software.

The Zephyr device model provides a consistent device model for configuring the drivers that are part of a system. The device model is responsible for initializing all the drivers configured into the system.

Each type of driver (UART, SPI, I2C) is supported by a generic type API.

In this model the driver fills in the pointer to the structure containing the function pointers to its API functions during driver initialization. These structures are placed into the RAM section in initialization level order.

Standard Drivers

Device drivers which are present on all supported board configurations are listed below.

- **Interrupt controller**: This device driver is used by the kernel's interrupt management subsystem.
- **Timer**: This device driver is used by the kernel's system clock and hardware clock subsystem.
- **Serial communication**: This device driver is used by the kernel's system console subsystem.
- **Random number generator**: This device driver provides a source of random numbers.

  重要: Certain implementations of this device driver do not generate sequences of values that are truly random.

Synchronous Calls

Zephyr provides a set of device drivers for multiple boards. Each driver should support an interrupt-based implementation, rather than polling, unless the specific hardware does not provide any interrupt.

High-level calls accessed through device-specific APIs, such as i2c.h or spi.h, are usually intended as synchronous. Thus, these calls should be blocking.

Driver APIs

The following APIs for device drivers are provided by `device.h`. The APIs are intended for use in device drivers only and should not be used in applications.

`DEVICE_INIT()` create device object and set it up for boot time initialization.

`DEVICE_AND_API_INIT()` Create device object and set it up for boot time initialization. This also takes a pointer to driver API struct for link time pointer assignment.

`DEVICE_NAME_GET()` Expands to the full name of a global device object.

`DEVICE_GET()` Obtain a pointer to a device object by name.

`DEVICE_DECLARE()` Declare a device object.

Driver Data Structures

The device initialization macros populate some data structures at build time which are split into read-only and runtime-mutable parts. At a high level we have:

```
struct device {
      struct device_config *config;
      void *driver_api;
      void *driver_data;
};
```

The `config` member is for read-only configuration data set at build time. For example, base memory mapped IO addresses, IRQ line numbers, or other fixed physical characteristics of the device. This is the `config_info` structure passed to the `DEVICE_*INIT()` macros.

The `driver_data` struct is kept in RAM, and is used by the driver for per-instance runtime housekeeping. For example, it may contain reference counts, semaphores, scratch buffers, etc.

The `driver_api` struct maps generic subsystem APIs to the device-specific implementations in the driver. It is typically read-only and populated at build time. The next section describes this in more detail.

Subsystems and API Structures

Most drivers will be targeting a device-independent subsystem API. Applications can simply program to that generic API, and application code is not specific to any particular driver implementation.

A subsystem API definition typically looks like this:

```
typedef int (*subsystem_do_this_t)(struct device *device, int foo, int bar);
typedef void (*subsystem_do_that_t)(struct device *device, void *baz);

struct subsystem_api {
      subsystem_do_this_t do_this;
```

```
	subsystem_do_that_t do_that;
};

static inline int subsystem_do_this(struct device *device, int foo, int bar)
{
	struct subsystem_api *api;

	api = (struct subsystem_api *)device->driver_api;
	return api->do_this(device, foo, bar);
}

static inline void subsystem_do_that(struct device *device, void *baz)
{
	struct subsystem_api *api;

	api = (struct subsystem_api *)device->driver_api;
	api->do_that(device, foo, bar);
}
```

In general, it's best to use `__ASSERT()` macros instead of propagating return values unless the failure is expected to occur during the normal course of operation (such as a storage device full). Bad parameters, programming errors, consistency checks, pathological/unrecoverable failures, etc., should be handled by assertions.

When it is appropriate to return error conditions for the caller to check, 0 should be returned on success and a POSIX errno.h code returned on failure. See https://github.com/zephyrproject-rtos/zephyr/wiki/Naming-Conventions#return-codes for details about this.

A driver implementing a particular subsystem will define the real implementation of these APIs, and populate an instance of subsystem_api structure:

```
static int my_driver_do_this(struct device *device, int foo, int bar)
{
	...
}

static void my_driver_do_that(struct device *device, void *baz)
{
	...
}

static struct subsystem_api my_driver_api_funcs = {
	.do_this = my_driver_do_this,
	.do_that = my_driver_do_that
};
```

The driver would then pass `my_driver_api_funcs` as the `api` argument to `DEVICE_AND_API_INIT()`, or manually assign it to `device->driver_api` in the driver init function.

注解: Since pointers to the API functions are referenced in the `driver_api` struct, they will always be included in the binary even if unused; `gc-sections` linker option will always see at least one reference to them. Providing for link-time size optimizations with driver APIs in most cases requires that the optional feature be controlled by a Kconfig option.

Single Driver, Multiple Instances

Some drivers may be instantiated multiple times in a given system. For example there can be multiple GPIO banks, or multiple UARTS. Each instance of the driver will have a different `config_info` struct and `driver_data` struct.

Configuring interrupts for multiple drivers instances is a special case. If each instance needs to configure a different interrupt line, this can be accomplished through the use of per-instance configuration functions, since the parameters to `IRQ_CONNECT()` need to be resolvable at build time.

For example, let's say we need to configure two instances of `my_driver`, each with a different interrupt line. In `drivers/subsystem/subsystem_my_driver.h`:

```
typedef void (*my_driver_config_irq_t)(struct device *device);

struct my_driver_config {
	u32_t base_addr;
	my_driver_config_irq_t config_func;
};
```

In the implementation of the common init function:

```
void my_driver_isr(struct device *device)
{
	/* Handle interrupt */
	...
}

int my_driver_init(struct device *device)
{
	const struct my_driver_config *config = device->config->config_info;

	/* Do other initialization stuff */
	...

	config->config_func(device);

	return 0;
}
```

Then when the particular instance is declared:

```
#if CONFIG_MY_DRIVER_0

DEVICE_DECLARE(my_driver_0);

static void my_driver_config_irq_0
{
	IRQ_CONNECT(MY_DRIVER_0_IRQ, MY_DRIVER_0_PRI, my_driver_isr,
		    DEVICE_GET(my_driver_0), MY_DRIVER_0_FLAGS);
}

const static struct my_driver_config my_driver_config_0 = {
	.base_addr = MY_DRIVER_0_BASE_ADDR;
	.config_func = my_driver_config_irq_0;
}

static struct my_driver_data_0;

DEVICE_AND_API_INIT(my_driver_0, MY_DRIVER_0_NAME, my_driver_init,
                    &my_driver_data_0, &my_driver_config_0, SECONDARY,
                    MY_DRIVER_0_PRIORITY, &my_driver_api_funcs);

#endif /* CONFIG_MY_DRIVER_0 */
```

Note the use of `DEVICE_DECLARE()` to avoid a circular dependency on providing the IRQ handler argument and the definition of the device itself.

Initialization Levels

Drivers may depend on other drivers being initialized first, or require the use of kernel services. The DEVICE_INIT() APIs allow the user to specify at what time during the boot sequence the init function will be executed. Any driver will specify one of five initialization levels:

PRE_KERNEL_1 Used for devices that have no dependencies, such as those that rely solely on hardware present in the processor/SOC. These devices cannot use any kernel services during configuration, since the services are not yet available. The interrupt subsystem will be configured however so it's OK to set up interrupts. Init functions at this level run on the interrupt stack.

PRE_KERNEL_2 Used for devices that rely on the initialization of devices initialized as part of the PRIMARY level. These devices cannot use any kernel services during configuration, since the kernel services are not yet available. Init functions at this level run on the interrupt stack.

POST_KERNEL Used for devices that require kernel services during configuration. Init functions at this level run in context of the kernel main task.

APPLICATION Used for application components (i.e. non-kernel components) that need automatic configuration. These devices can use all services provided by the kernel during configuration. Init functions at this level run on the kernel main task.

Within each initialization level you may specify a priority level, relative to other devices in the same initialization level. The priority level is specified as an integer value in the range 0 to 99; lower values indicate earlier initialization. The priority level must be a decimal integer literal without leading zeroes or sign (e.g. 32), or an equivalent symbolic name (e.g. `\#define MY_INIT_PRIO 32`); symbolic expressions are *not* permitted (e.g. `CONFIG_KERNEL_INIT_PRIORITY_DEFAULT + 5`).

System Drivers

In some cases you may just need to run a function at boot. Special `SYS_INIT` macros exist that map to `DEVICE_INIT()` or `DEVICE_INIT_PM()` calls. For `SYS_INIT()` there are no config or runtime data structures and there isn't a way to later get a device pointer by name. The same policies for initialization level and priority apply.

For `SYS_INIT_PM()` you can obtain pointers by name, see *power management* section.

`SYS_INIT()`

`SYS_INIT_PM()`

Device Tree in Zephyr

Introduction to Device Tree

Device tree is a way of describing hardware and configuration information for boards. Device tree was adopted for use in the Linux kernel for the PowerPC architecture. However, it is now in use for ARM and other architectures.

The device tree is a data structure for dynamically describing hardware using a Device Tree Source (DTS) data structure language, and compiled into a compact Device Tree Blob (DTB) using a Device Tree Compiler (DTC). Rather than hard coding every detail of a board's hardware into the operating system, the hardware-describing DTB is passed to the operating system at boot time. This allows the same compiled Linux kernel to support different hardware configurations within an architecture family (e.g., ARM, x86, PowerPC) and moves a significant part of the hardware description out of the kernel binary itself.

Traditional usage of device tree involves storing of the Device Tree Blob. The DTB is then used during runtime for configuration of device drivers. In Zephyr, the DTS information will be used only during compile time. Information about the system is extracted from the compiled DTS and used to create the application image.

Device tree uses a specific format to describe the device nodes in a system. This format is described in EPAPR document.

More device tree information can be found at the device tree repository.

System build requirements

The Zephyr device tree feature requires a device tree compiler (DTC) and Python YAML packages. Refer to the installation guide for your specific host OS:

- *Development Environment Setup on Windows*
- *Development Environment Setup on Linux*
- *Development Environment Setup on Mac OS*

Zephyr and Device Tree

In Zephyr, device tree is used to not only describe hardware, but also to describe Zephyr-specific configuration information. The Zephyr-specific information is intended to augment the device tree descriptions. The main reason for this is to leverage existing device tree files that a SoC vendor may already have defined for a given platform.

Today, configuration in Zephyr comes from a number of different places. It can come from Kconfig files, CMSIS header files, vendor header files, prj.conf files, and other miscellaneous sources. The intent of using device tree is to replace or curtail the use of Kconfig files throughout the system, and instead use device tree files to describe the configuration of device nodes. CMSIS and vendor header files can be used in conjunction with the device tree to fully describe hardware. Device tree is not intended to replace CMSIS or vendor include files.

The device tree files are compiled using the device tree compiler. The compiler runs the .dts file through the C preprocessor to resolve any macro or #defines utilized in the file. The output of the compile is another dts formatted file.

After compilation, a python script extracts information from the compiled device tree file using a set of rules specified in YAML files. The extracted information is placed in a header file that is used by the rest of the code as the project is compiled.

A temporary fixup file is required for device tree support on most devices. This .fixup file resides in the dts architecture directory and has the same name as the master .dts file. The only difference is the suffix is .fixup. This fixup file maps the generated include information to the current driver/source usage.

Device tree file formats

Hardware and software is described inside of device tree files in clear text format. These files have the file suffix of .dtsi or .dts. The .dtsi files are meant to be included by other files. Typically for a given board you have some number of .dtsi include files that pull in common device descriptions that are used across a given SoC family.

Example: FRDM K64F Board and Hexiwear K64

Both of these boards are based on the same NXP Kinetis SoC family, the K6X. The following shows the include hierarchy for both boards.

dts/arm/frdm_k64.dts includes the following two files:

```
dts/arm/nxp/nxp_k6x.dtsi
dts/arm/armv7-m.dtsi
```

dts/arm/hexiwear_k64.dts includes the same two files:

```
dts/arm/nxp/nxp_k6x.dtsi
dts/arm/armv7-m.dtsi
```

The board-specific .dts files enable nodes, define the Zephyr-specific items, and also adds board-specific changes like gpio/pinmux assignments. These types of things will vary based on the board layout and application use.

Currently supported boards

Device tree is currently supported on all ARM targets. Support for all other architectures is to be completed by release 1.9.

Adding support for a board

Adding device tree support for a given board requires adding a number of files. These files will contain the DTS information that describes a platform, the YAML descriptions that define the contents of a given Device Tree peripheral node, and also any fixup files required to support the platform.

When writing Device Tree Source files, it is good to separate out common peripheral information that could be used across multiple SoC families or boards. DTS files are identified by their file suffix. A .dtsi suffix denotes a DTS file that is used as an include in another DTS file. A .dts suffix denotes the primary DTS file for a given board.

The primary DTS file will contain at a minimum a version line, optional includes, and the root node definition. The root node will contain a model and compatible that denotes the unique board described by the .dts file.

Device Tree Source File Template

```
/dts-v1/
/ {
        model = "Model name for your board";
        compatible = "compatible for your board";
        /* rest of file */
};
```

One suggestion for starting from scratch on a platform/board is to examine other boards and their device tree source files.

The following is a more precise list of required files:

- Base architecture support
  - Add architecture-specific DTS directory, if not already present. Example: dts/arm for ARM.
  - Add target to dts/<ARCH>/Makefile or create Makefile if not present

  - Add target specific device tree files for base SoC. These should be .dtsi files to be included in the board-specific device tree files.
  - Add target specific YAML files in the dts/<ARCH>/yaml directory. Create the yaml directory if not present.
- SoC family support
  - Add one or more SoC family .dtsi files that describe the hardware for a set of devices. The file should contain all the relevant nodes and base configuration that would be applicable to all boards utilizing that SoC family.
  - Add SoC family YAML files that describe the nodes present in the .dtsi file.
- Board specific support
  - Add a board level .dts file that includes the SoC family .dtsi files and enables the nodes required for that specific board.
  - Board .dts file should specify the SRAM and FLASH devices, if present.
  - Add board-specific YAML files, if required. This would occur if the board has additional hardware that is not covered by the SoC family .dtsi/.yaml files.
- Fixup files
  - Fixup files contain mappings from existing Kconfig options to the actual underlying DTS derived configuration #defines. Fixup files are temporary artifacts until additional DTS changes are made to make them unnecessary.

Adding support for device tree in drivers

As drivers and other source code is converted over to make use of device tree generated information, these drivers may require changes to match the generated #define information.

Source Tree Hierarchy

The device tree files are located in a couple of different directories. The directory split is done based on architecture, and there is also a common directory where architecture agnostic device tree and yaml files are located.

Assuming the current working directory is the ZEPHYR_BASE, the directory hierarchy looks like the following:

```
dts/common/
dts/common/yaml
dts/<ARCH>/
dts/<ARCH>/yaml
```

The common directories contain a skeleton.dtsi include file that defines the address and size cells. The yaml subdirectory contains common yaml files for Zephyr-specific nodes/properties and generic device properties common across architectures.

Example: DTS/YAML files for NXP FRDM K64F:

```
dts/arm/armv7-m.dtsi
dts/arm/k6x/nxp_k6x.dtsi
dts/arm/frdm_k64f.dts
dts/arm/yaml/arm,v7m-nvic.yaml
dts/arm/yaml/k64gpio.yaml
dts/arm/yaml/k64pinmux.yaml
dts/arm/yaml/k64uart.yaml
```

YAML definitions for device nodes

Device tree can describe hardware and configuration, but it doesn't tell the system which pieces of information are useful, or how to generate configuration data from the device tree nodes. For this, we rely on YAML files to describe the contents or definition of a device tree node.

A YAML description must be provided for every device node that is to be a source of information for the system. This YAML description can be used for more than one purpose. It can be used in conjunction with the device tree to generate include information. It can also be used to validate the device tree files themselves. A device tree file can successfully compile and still not contain the necessary pieces required to build the rest of the software. YAML provides a means to detect that issue.

YAML files reside in a subdirectory inside the common and architecture-specific device tree directories. A YAML template file is provided to show the required format. This file is located at:

```
dts/common/yaml/device_node.yaml.template
```

YAML files must end in a .yaml suffix. YAML files are scanned during the information extraction phase and are matched to device tree nodes via the compatible property.

Subsystems

This section contains information about subsystem and the API they expose to applications.

Bluetooth

Zephyr comes integrated with a feature-rich and highly configurable Bluetooth stack:

- Bluetooth 5.0 compliant (ESR10)
- Bluetooth Low Energy Controller support (LE Link Layer)
  - BLE 5.0 compliant
  - Unlimited role and connection count, all roles supported
  - Concurrent multi-protocol support ready
  - Intelligent scheduling of roles to minimize overlap
  - Portable design to any open BLE radio, currently supports Nordic Semiconductor nRF51 and nRF52
- Generic Access Profile (GAP) with all possible LE roles
  - Peripheral & Central
  - Observer & Broadcaster
- GATT (Generic Attribute Profile)
  - Server (to be a sensor)
  - Client (to connect to sensors)
- Pairing support, including the Secure Connections feature from Bluetooth 4.2
- IPSP/6LoWPAN for IPv6 connectivity over Bluetooth LE

  - IPSP node sample application in `samples/bluetooth/ipsp`
- Basic Bluetooth BR/EDR (Classic) support
  - Generic Access Profile (GAP)
  - Logical Link Control and Adaptation Protocol (L2CAP)
  - Serial Port emulation (RFCOMM protocol)
  - Service Discovery Protocol (SDP)
- Clean HCI driver abstraction
  - 3-Wire (H:5) & 5-Wire (H:4) UART
  - SPI
  - Local controller support as a virtual HCI driver
- Raw HCI interface to run Zephyr as a Controller instead of a full Host stack
  - Possible to export HCI over a physical transport
  - `samples/bluetooth/hci_uart` sample for HCI over UART
  - `samples/bluetooth/hci_usb` sample for HCI over USB
- Verified with multiple popular controllers
- Highly configurable
  - Features, buffer sizes/counts, stack sizes, etc.

Source tree layout

The stack is split up as follows in the source tree:

`subsys/bluetooth/host` The host stack. This is where the HCI command & event handling as well as connection tracking happens. The implementation of the core protocols such as L2CAP, ATT & SMP is also here.

`subsys/bluetooth/controller` Bluetooth Controller implementation. Implements the controller-side of HCI, the Link Layer as well as access to the radio transceiver.

`include/bluetooth/` Public API header files. These are the header files applications need to include in order to use Bluetooth functionality.

`drivers/bluetooth/` HCI transport drivers. Every HCI transport needs its own driver. E.g. the two common types of UART transport protocols (3-Wire & 5-Wire) have their own drivers.

`samples/bluetooth/` Sample Bluetooth code. This is a good reference to get started with Bluetooth application development.

`tests/bluetooth/` Test applications. These applications are used to verify the functionality of the Bluetooth stack, but are not necessary the best source for sample code (see `samples/bluetooth` instead).

`doc/subsystems/bluetooth/` Extra documentation, such as PICS documents.

Further reading

More information on the stack and its usage can be found here and in the following subsections:

Developing Bluetooth Applications

Initialization

The Bluetooth subsystem is initialized using the `bt_enable()` function. The caller should ensure that function succeeds by checking the return code for errors. If a function pointer is passed to `bt_enable()`, the initialization happens asynchronously, and the completion is notified through the given function.

Bluetooth Application Example

A simple Bluetooth beacon application is shown below. The application initializes the Bluetooth Subsystem and enables non-connectable advertising, effectively acting as a Bluetooth Low Energy broadcaster.

```

/*
 * Set Advertisement data. Based on the Eddystone specification:
 * https://github.com/google/eddystone/blob/master/protocol-specification.md
 * https://github.com/google/eddystone/tree/master/eddystone-url
 */
static const struct bt_data ad[] = {
	BT_DATA_BYTES(BT_DATA_FLAGS, BT_LE_AD_NO_BREDR),
	BT_DATA_BYTES(BT_DATA_UUID16_ALL, 0xaa, 0xfe),
	BT_DATA_BYTES(BT_DATA_SVC_DATA16,
		      0xaa, 0xfe, /* Eddystone UUID */
		      0x10, /* Eddystone-URL frame type */
		      0x00, /* Calibrated Tx power at 0m */
		      0x00, /* URL Scheme Prefix http://www. */
		      'z', 'e', 'p', 'h', 'y', 'r',
		      'p', 'r', 'o', 'j', 'e', 'c', 't',
		      0x08) /* .org */
};

/* Set Scan Response data */
static const struct bt_data sd[] = {
	BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};

static void bt_ready(int err)
{
	if (err) {
		printk("Bluetooth init failed (err %d)\n", err);
		return;
	}

	printk("Bluetooth initialized\n");

	/* Start advertising */
	err = bt_le_adv_start(BT_LE_ADV_NCONN, ad, ARRAY_SIZE(ad),
			      sd, ARRAY_SIZE(sd));
	if (err) {
		printk("Advertising failed to start (err %d)\n", err);
		return;
	}

	printk("Beacon started\n");
}
```

```

void main(void)
{
	int err;

	printk("Starting Beacon Demo\n");

	/* Initialize the Bluetooth Subsystem */
	err = bt_enable(bt_ready);
	if (err) {
		printk("Bluetooth init failed (err %d)\n", err);
	}
}
```

The key APIs employed by the beacon sample are `bt_enable()` that's used to initialize Bluetooth and then `bt_le_adv_start()` that's used to start advertising a specific combination of advertising and scan response data.

Testing with QEMU

It's possible to test Bluetooth applications using QEMU. In order to do so, a Bluetooth controller needs to be exported from the host OS (Linux) to the emulator.

Using Host System Bluetooth Controller in QEMU

The host OS's Bluetooth controller is connected to the second QEMU serial line using a UNIX socket. This socket employs the QEMU option `-serial unix:/tmp/bt-server-bredr`. This option is already added to QEMU through **QEMU_EXTRA_FLAGS** in most Bluetooth sample Makefiles' and made available through the 'run' make target.

On the host side, BlueZ allows to export its Bluetooth controller through a so-called user channel for QEMU to use:

1. Make sure that the Bluetooth controller is down
2. Use the btproxy tool to open the listening UNIX socket, type:

```
$ sudo tools/btproxy -u
Listening on /tmp/bt-server-bredr
```

3. Choose one of the Bluetooth sample applications located in `samples/bluetooth`.
4. To run Bluetooth application in QEMU, type:

```
$ make run
```

Running QEMU now results in a connection with the second serial line to the `bt-server-bredr` UNIX socket, letting the application access the Bluetooth controller.

Qualification Testing

PICS Features

The PICS features for each supported protocol & profile can be found in the following documents:

GAP PICS

PTS version: 6.4

* - different than PTS defaults

^ - field not available on PTS

M - mandatory

O - optional

Device Configuration

Parameter Name	Selected	Description
TSPC_GAP_0_1	False (*)	BR/EDR (C.1)
TSPC_GAP_0_2	True	LE (C.2)
TSPC_GAP_0_3	False (*)	BR/EDR/LE (C.3)

Version Configuration

Parameter Name	Selected	Description
TSPC_GAP_0A_1	False (*)	Core Specification Addendum 3 (CSA3) (C.1)
TSPC_GAP_0A_2	False (*)	Core Specification Addendum 4 (CSA4) (C.2)
TSPC_GAP_0A_3	False (*)	Core Spec version 4.1 (Core v4.1) (C.3)
TSPC_GAP_0A_4	True	Core Spec version 4.2 (Core v4.2) (C.4)

Modes

Parameter Name	Selected	Description
TSPC_GAP_1_1	False (*)	Non-discoverable mode (C.1)
TSPC_GAP_1_2	False (*)	Limited-discoverable Mode (O)
TSPC_GAP_1_3	False (*)	General-discoverable mode (O)
TSPC_GAP_1_4	False (*)	Non-connectable mode (O)
TSPC_GAP_1_5	False (*)	Connectable mode (M)
TSPC_GAP_1_6	False (*)	Non-bondable mode (O)
TSPC_GAP_1_7	False (*)	Bondable mode (C.2)
TSPC_GAP_1_8	False (*)	Non-Synchronizable Mode (C.3)
TSPC_GAP_1_9	False (*)	Synchronizable Mode (C.4)

Security Aspects

Parameter Name	Selected	Description
TSPC_GAP_2_1	False (*)	Authentication procedure (C.1)
TSPC_GAP_2_2	False (*)	Support of LMP-Authentication (M)
TSPC_GAP_2_3	False (*)	Initiate LMP-Authentication (C.5)
TSPC_GAP_2_4	False (*)	Security mode 1 (C.2)
TSPC_GAP_2_5	False (*)	Security mode 2 (O)
TSPC_GAP_2_6	False (*)	Security mode 3 (C.7)
TSPC_GAP_2_7	False (*)	Security mode 4 (C.4)
TSPC_GAP_2_8	False (*)	Support of Authenticated link key (C.6)
TSPC_GAP_2_9	False (*)	Support of Unauthenticated link key (C.6)
TSPC_GAP_2_10	False (*)	No security (C.6)
TSPC_GAP_2_11	False (*)	Secure Connections Only Mode (C.8)

Idle Mode Procedures

Parameter Name	Selected	Description
TSPC_GAP_3_1	False (*)	Initiation of general inquiry (C.1)
TSPC_GAP_3_2	False (*)	Initiation of limited inquiry (C.1)
TSPC_GAP_3_3	False (*)	Initiation of name discover (O)
TSPC_GAP_3_4	False (*)	Initiation of device discovery (O)
TSPC_GAP_3_5	False (*)	Initiation of general bonding (O)
TSPC_GAP_3_6	False (*)	Initiation of dedicated bonding (O)

Establishment Procedures

Parameter Name	Selected	Description
TSPC_GAP_4_1	False (*)	Support link establishment as initiator (M)
TSPC_GAP_4_2	False (*)	Support link establishment as acceptor (M)
TSPC_GAP_4_3	False (*)	Support channel establishment as initiator (O)
TSPC_GAP_4_4	False (*)	Support channel establishment as acceptor (M)
TSPC_GAP_4_5	False (*)	Support connection establishment as initiator (O)
TSPC_GAP_4_6	False (*)	Support connection establishment as acceptor (O)
TSPC_GAP_4_7	False (*)	Support synchronization establishment as receiver (C.1)

LE Roles

Parameter Name	Selected	Description
TSPC_GAP_5_1	True	Broadcaster (C.1)
TSPC_GAP_5_2	True	Observer (C.1)
TSPC_GAP_5_3	True	Peripheral (C.1)
TSPC_GAP_5_4	True	Central (C.1)

Broadcaster Physical Layer

Parameter Name	Selected	Description
TSPC_GAP_6_1	True	Transmitter (M)
TSPC_GAP_6_2	True	Receiver (O)

Broadcaster Link Layer States

Parameter Name	Selected	Description
TSPC_GAP_7_1	True	Standby (M)
TSPC_GAP_7_2	True	Advertising (M)

Broadcaster Link Layer Advertising Event Types

Parameter Name	Selected	Description
TSPC_GAP_8_1	True	Non-Connectable Undirected Event (M)
TSPC_GAP_8_2	True	Scannable Undirected Event (O)

Broadcaster Link Layer Advertising Data Types

Parameter Name	Selected	Description
TSPC_GAP_8A_1	True	AD Type-Service UUID (O)
TSPC_GAP_8A_2	True	AD Type-Local Name (O)
TSPC_GAP_8A_3	True	AD Type-Flags (C.2)
TSPC_GAP_8A_4	True	AD Type-Manufacturer Specific Data (O)
TSPC_GAP_8A_5	False (*)	AD Type-TX Power Level (O)
TSPC_GAP_8A_6	False (*)	AD Type-Security Manager Out of Band (OOB) (C.1)
TSPC_GAP_8A_7	False (*)	AD Type-Security manager TK Value (O)
TSPC_GAP_8A_8	False (*)	AD Type-Slave Connection Interval Range (O)
TSPC_GAP_8A_9	False (*)	AD Type-Service Solicitation (O)
TSPC_GAP_8A_10	True	AD Type-Service Data (O)
TSPC_GAP_8A_11	True	AD Type-Appearance (O)
TSPC_GAP_8A_12	False (*)	AD Type-Public Target Address (O)
TSPC_GAP_8A_13	False (*)	AD Type-Random Target Address (O)
TSPC_GAP_8A_14	False (*)	AD Type-Advertising Interval (O)
TSPC_GAP_8A_15	False (*)	AD Type-LE Bluetooth Device Address (O)
TSPC_GAP_8A_16	False (*)	AD Type-LE Role (O)
TSPC_GAP_8A_17	(^)	AD Type-URI (C.3)

Broadcaster Connection Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_9_1	True	Non-Connectable Mode (M)

Broadcaster Broadcasting and Observing Features

Parameter Name	Selected	Description
TSPC_GAP_10_1	True	Broadcast Mode (M)

Broadcaster Privacy Feature

Parameter Name	Selected	Description
TSPC_GAP_11_1	False (*)	Privacy Feature v1.0 (C.2)
TSPC_GAP_11_1A	False (*)	Privacy Feature v1.1 (C.3)
TSPC_GAP_11_1B	(^)	Privacy Feature v1.2 (C.5)
TSPC_GAP_11_2	False (*)	Resolvable Private Address Generation Procedure (C.1)
TSPC_GAP_11_3	False (*)	Non-Resolvable Private Address Generation Procedure (C.4)

Observer Physical Layer

Parameter Name	Selected	Description
TSPC_GAP_12_1	True	Receiver (M)
TSPC_GAP_12_2	True	Transmitter (O)

Observer Link Layer States

Parameter Name	Selected	Description
TSPC_GAP_13_1	True	Standby (M)
TSPC_GAP_13_2	True	Scanning (M)

Observer Link Layer Scanning Types

Parameter Name	Selected	Description
TSPC_GAP_14_1	True	Passive Scanning (M)
TSPC_GAP_14_2	True	Active Scanning (O)

Observer Connection Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_15_1	True	Non-Connectable Mode (M)

Observer Broadcasting and Observing Features

Parameter Name	Selected	Description
TSPC_GAP_16_1	True	Observation Procedure (M)

Observer Privacy Feature

Parameter Name	Selected	Description
TSPC_GAP_17_1	False (*)	Privacy Feature v1.0 (C.4)
TSPC_GAP_17_1A	False (*)	Privacy Feature v1.1 (C.5)
TSPC_GAP_17_1B	(^)	Privacy Feature v1.2 (C.6)
TSPC_GAP_17_2	False (*)	Non-Resolvable Private Address Generation Procedure (C.1)
TSPC_GAP_17_3	False (*)	Resolvable Private Address Resolution Procedure (C.2)
TSPC_GAP_17_4	False (*)	Resolvable Private Address Generation Procedure (C.3)

Peripheral Physical Layer

Parameter Name	Selected	Description
TSPC_GAP_18_1	True	Transmitter (M)
TSPC_GAP_18_2	True	Receiver (M)

Peripheral Link Layer States

Parameter Name	Selected	Description
TSPC_GAP_19_1	True	Standby (M)
TSPC_GAP_19_2	True	Advertising (M)
TSPC_GAP_19_3	True	Connection, Slave Role (C.1)

Peripheral Link Layer Advertising Event Types

Parameter Name	Selected	Description
TSPC_GAP_20_1	True	Connectable Undirected Event (C.1)
TSPC_GAP_20_2	True	Connectable Directed Event (C.2)
TSPC_GAP_20_3	True	Non-Connectable Undirected Event (O)
TSPC_GAP_20_4	True	Scannable Undirected Event (O)

Peripheral Link Layer Advertising Data Types

Parameter Name	Selected	Description
TSPC_GAP_20A_1	True	AD Type-Service UUID (C.1)
TSPC_GAP_20A_2	True	AD Type-Local Name (C.1)
TSPC_GAP_20A_3	True	AD Type-Flags (C.2)
TSPC_GAP_20A_4	True	AD Type-Manufacturer Specific Data (C.1)
TSPC_GAP_20A_5	False (*)	AD Type-TX Power Level (C.1)
TSPC_GAP_20A_6	False (*)	AD Type-Security Manager Out of Band (OOB) (C.3)
TSPC_GAP_20A_7	False (*)	AD Type-Security manager TK Value (C.1)
TSPC_GAP_20A_8	False (*)	AD Type-Slave Connection Interval Range (C.1)
TSPC_GAP_20A_9	False (*)	AD Type-Service Solicitation (C.1)
TSPC_GAP_20A_10	True	AD Type-Service Data (C.1)
TSPC_GAP_20A_11	True	AD Type-Appearance (C.1)
TSPC_GAP_20A_12	False (*)	AD Type-Public Target Address (C.1)
TSPC_GAP_20A_13	False (*)	AD Type-Random Target Address (C.1)
TSPC_GAP_20A_14	False (*)	AD Type-Advertising Interval (C.1)
TSPC_GAP_20A_15	False (*)	AD Type-LE Bluetooth Device Address (C.1)
TSPC_GAP_20A_16	False (*)	AD Type-LE Role (C.1)
TSPC_GAP_20A_17	(^)	AD Type-URI (C.4)

Peripheral Link Layer Control Procedures

Parameter Name	Selected	Description
TSPC_GAP_21_1	True	Connection Update Procedure (C.1)
TSPC_GAP_21_2	True	Channel Map Update Procedure (C.1)
TSPC_GAP_21_3	True	Encryption Procedure (C.2)
TSPC_GAP_21_4	True	Feature Exchange Procedure (C.1)
TSPC_GAP_21_5	True	Version Exchange Procedure (C.1)
TSPC_GAP_21_6	True	Termination Procedure (C.1)
TSPC_GAP_21_7	False (*)	LE Ping Procedure (C.3)
TSPC_GAP_21_8	True	Slave Initiated Feature Exchange Procedure (C.4)
TSPC_GAP_21_9	True	Connection Parameter Request Procedure (C.5)

Peripheral Discovery Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_22_1	True	Non-Discoverable Mode (C.1)
TSPC_GAP_22_2	True	Limited Discoverable Mode (C.2)
TSPC_GAP_22_3	True	General Discoverable Mode (C.3)
TSPC_GAP_22_4	True	Name Discovery Procedure (C.4)

Peripheral Connection Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_23_1	True	Non-Connectable Mode (M)
TSPC_GAP_23_2	False (*)	Directed Connectable Mode (C.1)
TSPC_GAP_23_3	True	Undirected Connectable Mode (C.2)
TSPC_GAP_23_4	True	Connection Parameter Update Procedure (C.2)
TSPC_GAP_23_5	True	Terminate Connection Procedure (C.2)

Peripheral Bonding Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_24_1	True	Non-Bondable Mode (M)
TSPC_GAP_24_2	True	Bondable Mode (C.1)
TSPC_GAP_24_3	True	Bonding Procedure (C.1)
TSPC_GAP_24_4	True	Multiple Bonds (C.2)

Peripheral Security Aspects Features

Parameter Name	Selected	Description
TSPC_GAP_25_1	True	Security Mode (C.2)
TSPC_GAP_25_2	True	Security Mode 2 (C.2)
TSPC_GAP_25_3	True	Authentication Procedure (C.2)
TSPC_GAP_25_4	False (*)	Authorization Procedure (C.2)
TSPC_GAP_25_5	True	Connection Data Signing Procedure (C.2)
TSPC_GAP_25_6	True	Authenticate Signed Data Procedure (C.2)
TSPC_GAP_25_7	True	Authenticated Pairing (LE security mode 1 level 3) (C.1)
TSPC_GAP_25_8	True	Unauthenticated Pairing (LE security mode 1 level 2) (C.1)
TSPC_GAP_25_9	(^)	LE Security Mode 1 Level 4 (C.3)
TSPC_GAP_25_10	(^)	Secure Connections Only Mode (C.4)

Peripheral Privacy Feature

Parameter Name	Selected	Description
TSPC_GAP_26_1	False (*)	Privacy Feature (C.5)
TSPC_GAP_26_1A	False (*)	Privacy Feature v1.1 (C.3)
TSPC_GAP_26_1B	(^)	Privacy Feature v1.2 (C.6)
TSPC_GAP_26_2	False (*)	Non-Resolvable Private Address Generation Procedure (C.1)
TSPC_GAP_26_3	False (*)	Resolvable Private Address Generation Procedure (C.2)
TSPC_GAP_26_4	False (*)	Resolvable Private Address Generation Procedure (C.4)

Peripheral GAP Characteristics

Parameter Name	Selected	Description
TSPC_GAP_27_1	True	Device Name (M)
TSPC_GAP_27_2	True	Appearance (M)
TSPC_GAP_27_3	False (*)	Peripheral Privacy Flag (C.1)
TSPC_GAP_27_4	False (*)	Reconnection Address (C.2)
TSPC_GAP_27_5	False (*)	Peripheral Preferred Connection Parameters (C.3)
TSPC_GAP_27_6	True	Writable Device Name (C.3)
TSPC_GAP_27_7	True	Writable Appearance (C.3)
TSPC_GAP_27_8	False (*)	Writable Peripheral Privacy Flag (C.4)

Central Physical Layer

Parameter Name	Selected	Description
TSPC_GAP_28_1	True	Transmitter (M)
TSPC_GAP_28_2	True	Receiver (M)

Central Link Layer States

Parameter Name	Selected	Description
TSPC_GAP_29_1	True	Standby (M)
TSPC_GAP_29_2	True	Scanning (M)
TSPC_GAP_29_3	True	Initiating (M)
TSPC_GAP_29_4	True	Connection, Master Role (M)

Central Link Layer Scanning Types

Parameter Name	Selected	Description
TSPC_GAP_30_1	True	Passive Scanning (O)
TSPC_GAP_30_2	True	Active Scanning (C.1)

Central Link Layer Control Procedures

Parameter Name	Selected	Description
TSPC_GAP_31_1	True	Connection Update Procedure (M)
TSPC_GAP_31_2	True	Channel Map Update Procedure (M)
TSPC_GAP_31_3	True	Encryption Procedure (O)
TSPC_GAP_31_4	True	Feature Exchange Procedure (M)
TSPC_GAP_31_5	True	Version Exchange Procedure (M)
TSPC_GAP_31_6	True	Termination Procedure (M)
TSPC_GAP_31_7	False (*)	LE Ping Procedure (C.1)
TSPC_GAP_31_8	True	Slave Initiated Feature Exchange Procedure (C.2)
TSPC_GAP_31_9	False (*)	Connection Parameter Request Procedure (C.1)

Central Discovery Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_32_1	True	Limited Discovery Procedure (C.2)
TSPC_GAP_32_2	True	General Discovery Procedure (C.1)
TSPC_GAP_32_3	True	Name Discovery Procedure (C.3)

Central Connection Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_33_1	True	Auto Connection Establishment Procedure (C.3)
TSPC_GAP_33_2	True	General Connection Establishment Procedure (C.1)
TSPC_GAP_33_3	False (*)	Selective Connection Establishment Procedure (C.3)
TSPC_GAP_33_4	True	Direct Connection Establishment Procedure (C.2)
TSPC_GAP_33_5	True	Connection Parameter Update Procedure (C.2)
TSPC_GAP_33_6	True	Terminate Connection Procedure (C.2)

Central Bonding Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_34_1	True	Non-Bondable Mode (C.1)
TSPC_GAP_34_2	True	Bondable Mode (C.2)
TSPC_GAP_34_3	True	Bonding Procedure (C.2)

Central Security Features

Parameter Name	Selected	Description
TSPC_GAP_35_1	True	Security Mode 1 (O)
TSPC_GAP_35_2	True	Security Mode 2 (O)
TSPC_GAP_35_3	True	Authentication Procedure (O)
TSPC_GAP_35_4	False (*)	Authorization Procedure (O)
TSPC_GAP_35_5	True	Connection Data Signing Procedure (O)
TSPC_GAP_35_6	True	Authenticate Signed Data Procedure (O)
TSPC_GAP_35_7	True	Authenticated Pairing (LE security mode 1 level 3) (C.1)
TSPC_GAP_35_8	True	Unauthenticated Pairing (LE security mode 1 level 2) (C.1)
TSPC_GAP_35_9	(^)	LE Security Mode 1 Level 4 (C.2)
TSPC_GAP_35_10	(^)	Secure Connections Only Mode (C.3)

Central Privacy Feature

Parameter Name	Selected	Description
TSPC_GAP_36_1	False (*)	Privacy Feature v1.0 (C.2)
TSPC_GAP_36_1A	False (*)	Privacy Feature v1.1 (C.4)
TSPC_GAP_36_1B	(^)	Privacy Feature v1.2 (C.7)
TSPC_GAP_36_2	False (*)	Non-Resolvable Private Address Generation Procedure (C.1)
TSPC_GAP_36_3	False (*)	Resolvable Private Address Resolution Procedure (C.3)
TSPC_GAP_36_4	False (*)	Write to Privacy Characteristic (Enable/Disable Privacy) (C.5)
TSPC_GAP_36_5	False (*)	Resolvable Private Address Generation Procedure (C.6)

Central GAP Characteristics

Parameter Name	Selected	Description
TSPC_GAP_37_1	True	Device Name (M)
TSPC_GAP_37_2	True	Appearance (M)
TSPC_GAP_37_3	(^)	Central Address Resolution (C.1)

BR/EDR/LE Roles

Parameter Name	Selected	Description
TSPC_GAP_38_1	False (*)	Broadcaster (C.1)
TSPC_GAP_38_2	False (*)	Observer (C.1)
TSPC_GAP_38_3	False (*)	Peripheral (C.1)
TSPC_GAP_38_4	False (*)	Central (C.1)

Central BR/EDR/LE Modes

Parameter Name	Selected	Description
TSPC_GAP_39_1	False (*)	Non-Discoverable Mode (C.1)
TSPC_GAP_39_2	False (*)	Discoverable Mode (C.2)
TSPC_GAP_39_3	False (*)	Non-Connectable Mode (C.3)
TSPC_GAP_39_4	False (*)	Connectable Mode (M)
TSPC_GAP_39_5	False (*)	Non-Bondable Mode (C.4)
TSPC_GAP_39_6	False (*)	Bondable Mode (C.5)

Central BR/EDR/LE Idle Mode Procedures

Parameter Name	Selected	Description
TSPC_GAP_40_1	False (*)	General Discovery (C.1)
TSPC_GAP_40_2	False (*)	Limited Discovery (C.2)
TSPC_GAP_40_3	False (*)	Device Type Discovery (C.3)
TSPC_GAP_40_4	False (*)	Name Discovery (C.4)
TSPC_GAP_40_5	False (*)	Link Establishment (C.5)

Central BR/EDR/LE Security Aspects

Parameter Name	Selected	Description
TSPC_GAP_41_1	False (*)	Security Aspects (M)
TSPC_GAP_41_2	(^)	Cross-Transport Key Derivation (C.1)

Peripheral BR/EDR/LE Modes

Parameter Name	Selected	Description
TSPC_GAP_42_1	False (*)	Non-Discoverable Mode (C.1)
TSPC_GAP_42_2	False (*)	Discoverable Mode (C.2)
TSPC_GAP_42_3	False (*)	Non-Connectable Mode (C.3)
TSPC_GAP_42_4	False (*)	Connectable Mode (M)
TSPC_GAP_42_5	False (*)	Non-Bondable Mode (C.4)
TSPC_GAP_42_6	False (*)	Bondable Mode (C.5)

Peripheral BR/EDR/LE Security Aspects

Parameter Name	Selected	Description
TSPC_GAP_43_1	False (*)	Peripheral BR/EDR/LE: Non-Discoverable Mode
TSPC_GAP_43_2	(^)	Cross-Transport Key Derivation (C.1)

Central Simultaneous BR/EDR and LE Transports

Parameter Name	Selected	Description
TSPC_GAP_44_1	False (*)	Simultaneous BR/EDR and LE Transports - BR/EDR Slave to the same device (C.1)
TSPC_GAP_44_2	False (*)	Simultaneous BR/EDR and LE Transports - BR/EDR Master to the same device (C.1)

Peripheral Simultaneous BR/EDR and LE Transports

Parameter Name	Selected	Description
TSPC_GAP_45_1	False (*)	Simultaneous BR/EDR and LE Transports - BR/EDR Slave to the same device (C.1)
TSPC_GAP_45_2	False (*)	Simultaneous BR/EDR and LE Transports - BR/EDR Master to the same device (C.1)

Parameter Name	Selected	Description
TSPC_GATT_1_1	True	GATT Client Role (O)
TSPC_GATT_1_2	True	GATT Server Role (O)
TSPC_SM_1_1	True	Master Role (Initiator)
TSPC_SM_1_2	True	Slave Role (Responder)
TSPC_SM_2_4	False (*)	OOB supported (O)

GATT PICS

PTS version: 6.4

* - different than PTS defaults

^ - field not available on PTS

M - mandatory

O - optional

Generic Attribute Profile Role

Parameter Name	Selected	Description
TSPC_GATT_1_1	True	Generic Attribute Profile Client (C.1)
TSPC_GATT_1_2	True	Generic Attribute Profile Server (C.2)
TSPC_GATT_1A_1	False (*)	Complete GATT client (C.3)
TSPC_GATT_1A_2	False (*)	Complete GATT server (C.4)

ATT Bearer Transport

Parameter Name	Selected	Description
TSPC_GATT_2_1	False (*)	Attribute Protocol Supported over BR/EDR (L2CAP fixed channel support) (C.1)
TSPC_GATT_2_2	True	Attribute Protocol Supported over LE (C.2)

Generic Attribute Profile Support

Parameter Name	Selected	Description
TSPC_GATT_3_1	True	Client: Exchange MTU (C.1)
TSPC_GATT_3_2	False (*)	Client: Discover All Primary Services (C.1)
TSPC_GATT_3_3	True	Client: Discover Primary Services Service UUID (C.1)
TSPC_GATT_3_4	True	Client: Find Included Services (C.1)
TSPC_GATT_3_5	True	Client: Discover All characteristics of a Service (C.1)
TSPC_GATT_3_6	True	Client: Discover Characteristics by UUID (C.1)
TSPC_GATT_3_7	True	Client: Discover All Characteristic Descriptors (C.1)
TSPC_GATT_3_8	True	Client: Read Characteristic Value (C.1)
TSPC_GATT_3_9	False (*)	Client: Read using Characteristic UUID (C.1)
TSPC_GATT_3_10	True	Client: Read Long Characteristic Values (C.1)
TSPC_GATT_3_11	True	Client: Read Multiple Characteristic Values (C.1)
TSPC_GATT_3_12	True	Client: Write without Response (C.1)
TSPC_GATT_3_13	True	Client: Signed Write Without Response (C.1)
TSPC_GATT_3_14	True	Client: Write Characteristic Value (C.1)
TSPC_GATT_3_15	True	Client: Write Long Characteristic Values (C.1)
TSPC_GATT_3_16	False (*)	Client: Characteristic Value Reliable Writes (C.1)
TSPC_GATT_3_17	True	Client: Notifications (C.1)
TSPC_GATT_3_18	True	Client: Indications (M)
TSPC_GATT_3_19	True	Client: Read Characteristic Descriptors (C.1)
TSPC_GATT_3_20	True	Client: Read long Characteristic Descriptors (C.1)
TSPC_GATT_3_21	True	Client: Write Characteristic Descriptors (C.1)
TSPC_GATT_3_22	True	Client: Write Long Characteristic Descriptors (C.1)
TSPC_GATT_3_23	True	Client: Service Changed Characteristic (M)

Profile Attribute Types and Formats, by client

Parameter Name	Selected	Description
TSPC_GATT_3B_1	False (*)	Client: Primary Service Declaration (M)
TSPC_GATT_3B_2	False (*)	Client: Secondary Service Declaration (M)
TSPC_GATT_3B_3	False (*)	Client: Include Declaration (M)
TSPC_GATT_3B_4	False (*)	Client: Characteristic Declaration (M)
TSPC_GATT_3B_5	False (*)	Client: Characteristic Value Declaration (M)
TSPC_GATT_3B_6	False (*)	Client: Characteristic Extended Properties (M)
TSPC_GATT_3B_7	False (*)	Client: Characteristic User Description Descriptor (M)
TSPC_GATT_3B_8	False (*)	Client: Client Characteristic Configuration Descriptor (M)
TSPC_GATT_3B_9	False (*)	Client: Server Characteristic Configuration Descriptor (M)
TSPC_GATT_3B_10	False (*)	Client: Characteristic Format Descriptor (M)
TSPC_GATT_3B_11	False (*)	Client: Characteristic Aggregate Format Descriptor (M)
TSPC_GATT_3B_12	False (*)	Client: Characteristic Format: Boolean (M)
TSPC_GATT_3B_13	False (*)	Client: Characteristic Format: 2Bit (M)
TSPC_GATT_3B_14	False (*)	Client: Characteristic Format: nibble (M)
TSPC_GATT_3B_15	False (*)	Client: Characteristic Format: Uint8 (M)
TSPC_GATT_3B_16	False (*)	Client: Characteristic Format: Uint12 (M)
TSPC_GATT_3B_17	False (*)	Client: Characteristic Format: Uint16 (M)
TSPC_GATT_3B_18	False (*)	Client: Characteristic Format: Uint24 (M)
TSPC_GATT_3B_19	False (*)	Client: Characteristic Format: Uint32 (M)
TSPC_GATT_3B_20	False (*)	Client: Characteristic Format: Uint48 (M)
TSPC_GATT_3B_21	False (*)	Client: Characteristic Format: Uint64 (M)
TSPC_GATT_3B_22	False (*)	Client: Characteristic Format: Uint128 (M)
TSPC_GATT_3B_23	False (*)	Client: Characteristic Format: Sint8 (M)
TSPC_GATT_3B_24	False (*)	Client: Characteristic Format: Sint12 (M)
TSPC_GATT_3B_25	False (*)	Client: Characteristic Format: Sint16 (M)
TSPC_GATT_3B_26	False (*)	Client: Characteristic Format: Sint24 (M)
TSPC_GATT_3B_27	False (*)	Client: Characteristic Format: Sint32 (M)
TSPC_GATT_3B_28	False (*)	Client: Characteristic Format: Sint48 (M)
TSPC_GATT_3B_29	False (*)	Client: Characteristic Format: Sint64 (M)
TSPC_GATT_3B_30	False (*)	Client: Characteristic Format: Sint128 (M)
TSPC_GATT_3B_31	False (*)	Client: Characteristic Format: Float32 (M)
TSPC_GATT_3B_32	False (*)	Client: Characteristic Format: Float64 (M)
TSPC_GATT_3B_33	False (*)	Client: Characteristic Format: SFLOAT (M)
TSPC_GATT_3B_34	False (*)	Client: Characteristic Format: FLOAT (M)
TSPC_GATT_3B_35	False (*)	Client: Characteristic Format: Duint16 (M)
TSPC_GATT_3B_36	False (*)	Client: Characteristic Format: utf8s (M)
TSPC_GATT_3B_37	False (*)	Client: Characteristic Format: utf16s (M)
TSPC_GATT_3B_38	False (*)	Client: Characteristic Format: struct (M)

Attribute Profile Support, by Server

Parameter Name	Selected	Description
TSPC_GATT_4_1	True	Server: Exchange MTU (C.4)
TSPC_GATT_4_2	True	Server: Discover All Primary Services (M)
TSPC_GATT_4_3	True	Server: Discover Primary Services Service UUID (M)
TSPC_GATT_4_4	True	Server: Find Included Services (M)
TSPC_GATT_4_5	True	Server: Discover All characteristics of a Service (M)
TSPC_GATT_4_6	True	Server: Discover Characteristics by UUID (M)
TSPC_GATT_4_7	True	Server: Discover All Characteristic Descriptors (M)
TSPC_GATT_4_8	True	Server: Read Characteristic Value (M)
TSPC_GATT_4_9	True	Server: Read using Characteristic UUID (M)
TSPC_GATT_4_10	True	Server: Read Long Characteristic Values (C.4)
TSPC_GATT_4_11	True	Server: Read Multiple Characteristic Values (C.4)
TSPC_GATT_4_12	True	Server: Write without Response (C.2)
TSPC_GATT_4_13	True	Server: Signed Write Without Response (C.4)
TSPC_GATT_4_14	True	Server: Write Characteristic Value (C.3)
TSPC_GATT_4_15	True	Server: Write Long Characteristic Values (C.4)
TSPC_GATT_4_16	True	Server: Characteristic Value Reliable Writes (C.4)
TSPC_GATT_4_17	True	Server: Notifications (C.4)
TSPC_GATT_4_18	False (*)	Server: Indications (C.1)
TSPC_GATT_4_19	True	Server: Read Characteristic Descriptors (C.4)
TSPC_GATT_4_20	True	Server: Read long Characteristic Descriptors (C.4)
TSPC_GATT_4_21	True	Server: Write Characteristic Descriptors (C.4)
TSPC_GATT_4_22	True	Server: Write Long Characteristic Descriptors (C.4)
TSPC_GATT_4_23	False (*)	Server: Service Changed Characteristic (C.1)

Profile Attribute Types and Characteristic Formats

Parameter Name	Selected	Description
TSPC_GATT_4B_1	True	Server: Primary Service Declaration (M)
TSPC_GATT_4B_2	True	Server: Secondary Service Declaration (M)
TSPC_GATT_4B_3	True	Server: Include Declaration (M)
TSPC_GATT_4B_4	True	Server: Characteristic Declaration (M)
TSPC_GATT_4B_5	True	Server: Characteristic Value Declaration (M)
TSPC_GATT_4B_6	True	Server: Characteristic Extended Properties (M)
TSPC_GATT_4B_7	True	Server: Characteristic User Description Descriptor (M)
TSPC_GATT_4B_8	True	Server: Client Characteristic Configuration Descriptor (M)
TSPC_GATT_4B_9	True	Server: Server Characteristic Configuration Descriptor (M)
TSPC_GATT_4B_10	True	Server: Characteristic Format Descriptor (M)
TSPC_GATT_4B_11	True	Server: Characteristic Aggregate Format Descriptor (M)
TSPC_GATT_4B_12	True	Server: Characteristic Format: Boolean (M)
TSPC_GATT_4B_13	True	Server: Characteristic Format: 2Bit (M)
TSPC_GATT_4B_14	True	Server: Characteristic Format: nibble (M)
TSPC_GATT_4B_15	True	Server: Characteristic Format: Uint8 (M)
TSPC_GATT_4B_16	True	Server: Characteristic Format: Uint12 (M)
TSPC_GATT_4B_17	True	Server: Characteristic Format: Uint16 (M)
TSPC_GATT_4B_18	True	Server: Characteristic Format: Uint24 (M)
TSPC_GATT_4B_19	True	Server: Characteristic Format: Uint32 (M)

下页继续

表 1.2 – 续上页

Parameter Name	Selected	Description
TSPC_GATT_4B_20	True	Server: Characteristic Format: Uint48 (M)
TSPC_GATT_4B_21	True	Server: Characteristic Format: Uint64 (M)
TSPC_GATT_4B_22	True	Server: Characteristic Format: Uint128 (M)
TSPC_GATT_4B_23	True	Server: Characteristic Format: Sint8 (M)
TSPC_GATT_4B_24	True	Server: Characteristic Format: Sint12 (M)
TSPC_GATT_4B_25	True	Server: Characteristic Format: Sint16 (M)
TSPC_GATT_4B_26	True	Server: Characteristic Format: Sint24 (M)
TSPC_GATT_4B_27	True	Server: Characteristic Format: Sint32 (M)
TSPC_GATT_4B_28	True	Server: Characteristic Format: Sint48 (M)
TSPC_GATT_4B_29	True	Server: Characteristic Format: Sint64 (M)
TSPC_GATT_4B_30	True	Server: Characteristic Format: Sint128 (M)
TSPC_GATT_4B_31	True	Server: Characteristic Format: Float32 (M)
TSPC_GATT_4B_32	True	Server: Characteristic Format: Float64 (M)
TSPC_GATT_4B_33	True	Server: Characteristic Format: SFLOAT (M)
TSPC_GATT_4B_34	True	Server: Characteristic Format: FLOAT (M)
TSPC_GATT_4B_35	True	Server: Characteristic Format: Duint16 (M)
TSPC_GATT_4B_36	True	Server: Characteristic Format: utf8s (M)
TSPC_GATT_4B_37	True	Server: Characteristic Format: utf16s (M)
TSPC_GATT_4B_38	True	Server: Characteristic Format: struct (M)

Generic Attribute Profile Service - SDP Interoperability

Parameter Name	Selected	Description
TSPC_GATT_6_2	False (*)	Discover GATT Services using Service Discovery Profile (C.1)
TSPC_GATT_6_3	False (*)	Publish SDP record for GATT services support via BR/EDR (C.2)

Attribute Protocol Transport Security

Parameter Name	Selected	Description
TSPC_GATT_7_1	False (*)	Security Mode 4 (C.1)
TSPC_GATT_7_2	True	LE Security Mode 1 (C.2)
TSPC_GATT_7_3	True	LE Security Mode 2 (C.2)
TSPC_GATT_7_4	True	LE Authentication Procedure (C.2)
TSPC_GATT_7_5	False (*)	LE connection data signing procedure (C.2)
TSPC_GATT_7_6	False (*)	LE Authenticate signed data procedure (C.2)
TSPC_GATT_7_7	True	LE Authorization Procedure (C.2)

Attribute Protocol Client Messages

Parameter Name	Selected	Description
TSPC_ATT_3_1	False (*)	Attribute Error Response (M)
TSPC_ATT_3_2	False (*)	Exchange MTU Request (O)
TSPC_ATT_3_4	False (*)	Find Information Request (O)
TSPC_ATT_3_6	False (*)	Find by Type Value Request (O)
TSPC_ATT_3_8	False (*)	Read by Type Request (O)
TSPC_ATT_3_10	False (*)	Read Request (O)
TSPC_ATT_3_12	False (*)	Read Blob Request (O)
TSPC_ATT_3_14	False (*)	Read Multiple Request (O)
TSPC_ATT_3_16	False (*)	Read by Group Type Request (O)
TSPC_ATT_3_17	False (*)	Read by Group Type Response (C.6)
TSPC_ATT_3_18	False (*)	Write Request (O)
TSPC_ATT_3_20	False (*)	Write Command (O)
TSPC_ATT_3_21	False (*)	Signed Write Command (O)
TSPC_ATT_3_22	False (*)	Prepare Write Request (O)
TSPC_ATT_3_24	False (*)	Execute Write Request (C.8)
TSPC_ATT_3_26	False (*)	Handle Value Notification (M)
TSPC_ATT_3_28	False (*)	Handle Value Confirmation (M)

Attribute Protocol Server Messages

Parameter Name	Selected	Description
TSPC_ATT_4_1	True	Attribute Error Response (M)
TSPC_ATT_4_2	True	Exchange MTU Request (M)
TSPC_ATT_4_3	True	Exchange MTU Response (M)
TSPC_ATT_4_5	True	Find Information Response (M)
TSPC_ATT_4_7	True	Find by Type Value Response (M)
TSPC_ATT_4_8	True	Read by Type Request (M)
TSPC_ATT_4_9	False (*)	Read by Type Response (M)
TSPC_ATT_4_11	True	Read Response (M)
TSPC_ATT_4_13	False (*)	Read Blob Response (C.1)
TSPC_ATT_4_15	False (*)	Read Multiple Response (C.2)
TSPC_ATT_4_17	True	Read by Group Type Response (M)
TSPC_ATT_4_19	False (*)	Write Response (C.3)
TSPC_ATT_4_20	False (*)	Write Command (O)
TSPC_ATT_4_21	False (*)	Signed Write Command (O)
TSPC_ATT_4_23	False (*)	Prepare Write Response (C.4)
TSPC_ATT_4_25	False (*)	Execute Write Response (C.4)
TSPC_ATT_4_26	False (*)	Handle Value Notification (O)
TSPC_ATT_4_27	False (*)	Handle Value Indication (O)

Attribute Protocol Transport

Parameter Name	Selected	Description
TSPC_ATT_5_2	True	LE Security Mode 1 (C.2)
TSPC_ATT_5_4	True	LE Authentication Procedure (C.2)
TSPC_ATT_5_7	True	LE Authorization Procedure (C.2)

Device Configuration

Parameter Name	Selected	Description
TSPC_GAP_0_2	True	LE (C.2)

L2CAP PICS

PTS version: 6.4

- – different than PTS defaults

Device Configuration

Parameter Name	Selected	Description
TSPC_L2CAP_0_1	False (*)	BR/EDR
TSPC_L2CAP_0_2	True	Bluetooth low energy only
TSPC_L2CAP_0_3	False (*)	BR/EDR/Bluetooth low energy

Roles

Parameter Name	Selected	Description
TSPC_L2CAP_1_1	False (*)	Data Channel Initiator
TSPC_L2CAP_1_2	False (*)	Data Channel Acceptor
TSPC_L2CAP_1_3	True	LE Master
TSPC_L2CAP_1_4	True	LE Slave
TSPC_L2CAP_1_5	True	LE Data Channel Initiator
TSPC_L2CAP_1_6	True	LE Data Channel Acceptor

General Operation

Parameter Name	Selected	Description
TSPC_L2CAP_2_1	False (*)	Support of L2CAP signaling channel
TSPC_L2CAP_2_2	False (*)	Support of configuration process
TSPC_L2CAP_2_3	False (*)	Support of connection oriented data channel
TSPC_L2CAP_2_4	False (*)	Support of command echo request
TSPC_L2CAP_2_5	False (*)	Support of command echo response
TSPC_L2CAP_2_6	False (*)	Support of command information request
TSPC_L2CAP_2_7	False (*)	Support of command information response
TSPC_L2CAP_2_8	False (*)	Support of a channel group
TSPC_L2CAP_2_9	False (*)	Support of packet for connectionless channel
TSPC_L2CAP_2_10	False (*)	Support retransmission mode
TSPC_L2CAP_2_11	False (*)	Support flow control mode
TSPC_L2CAP_2_12	False (*)	Enhanced Retransmission Mode
TSPC_L2CAP_2_13	False (*)	Streaming Mode
TSPC_L2CAP_2_14	False (*)	FCS Option
TSPC_L2CAP_2_15	False (*)	Generate Local Busy Condition
TSPC_L2CAP_2_16	False (*)	Send Reject

下页继续

表 1.3 – 续上页

Parameter Name	Selected	Description
TSPC_L2CAP_2_17	False (*)	Send Selective Reject
TSPC_L2CAP_2_18	False (*)	Mandatory use of ERTM
TSPC_L2CAP_2_19	False (*)	Mandatory use of Streaming Mode
TSPC_L2CAP_2_20	False (*)	Optional use of ERTM
TSPC_L2CAP_2_21	False (*)	Optional use of Streaming Mode
TSPC_L2CAP_2_22	False (*)	Send data using SAR in ERTM
TSPC_L2CAP_2_23	False (*)	Send data using SAR in Streaming Mode
TSPC_L2CAP_2_24	False (*)	Actively request Basic Mode for a PSM that supports the use of ERTM or Streaming Mode
TSPC_L2CAP_2_25	False (*)	Supports performing L2CAP channel mode configuration fallback from SM to ERTM
TSPC_L2CAP_2_26	False (*)	Supports sending more than one unacknowledged I-Frame when operating in ERTM
TSPC_L2CAP_2_27	False (*)	Supports sending more than three unacknowledged I-Frame when operating in ERTM
TSPC_L2CAP_2_28	False (*)	Supports configuring the peer TxWindow greater than 1
TSPC_L2CAP_2_29	False (*)	AMP Support
TSPC_L2CAP_2_30	False (*)	Fixed Channel Support
TSPC_L2CAP_2_31	False (*)	AMP Manager Support
TSPC_L2CAP_2_32	False (*)	ERTM over AMP
TSPC_L2CAP_2_33	False (*)	Streaming Mode Source over AMP Support
TSPC_L2CAP_2_34	False (*)	Streaming Mode Sink over AMP Support
TSPC_L2CAP_2_35	False (*)	Unicast Connectionless Data, Reception
TSPC_L2CAP_2_36	False (*)	Ability to transmit an unencrypted packet over a Unicast connectionless L2CAP channel
TSPC_L2CAP_2_37	False (*)	Ability to transmit an encrypted packet over a Unicast connectionless L2CAP channel
TSPC_L2CAP_2_38	False (*)	Extended Flow Specification for BR/EDR
TSPC_L2CAP_2_39	False (*)	Extended Window Size
TSPC_L2CAP_2_40	True	Support of Low Energy signaling channel
TSPC_L2CAP_2_41	True	Support of command reject
TSPC_L2CAP_2_42	True	Send Connection Parameter Update Request
TSPC_L2CAP_2_43	True	Send Connection Parameter Update Response
TSPC_L2CAP_2_44	False (*)	Extended Flow Specification for AMP
TSPC_L2CAP_2_45	True	Send disconnect request command
TSCP_L2CAP_2_46	True	Support LE Credit Based Flow Control Mode
TSCP_L2CAP_2_47	True	Support for LE Data Channel

Configurable Parameters

Parameter Name	Selected	Description
TSPC_L2CAP_3_1	True	Support of RTX timer
TSPC_L2CAP_3_2	False (*)	Support of ERTX timer
TSPC_L2CAP_3_3	False (*)	Support minimum MTU size 48 octets
TSPC_L2CAP_3_4	False (*)	Support MTU size larger than 48 octets
TSPC_L2CAP_3_5	False (*)	Support of flush timeout value for reliable channel
TSPC_L2CAP_3_6	False (*)	Support of flush timeout value for unreliable channel
TSPC_L2CAP_3_7	False (*)	Support of bi-directional quality of service (QoS) option field
TSPC_L2CAP_3_8	False (*)	Negotiate QoS service type
TSPC_L2CAP_3_9	False (*)	Negotiate and support service type 'No traffic'
TSPC_L2CAP_3_10	False (*)	Negotiate and support service type 'Best effort'
TSPC_L2CAP_3_11	False (*)	Negotiate and support service type 'Guaranteed'
TSPC_L2CAP_3_12	True	Support minimum MTU size 23 octets
TSPC_L2CAP_3_13	False (*)	Negotiate and support service type 'No traffic' for Extended Flow Specification
TSPC_L2CAP_3_14	False (*)	Negotiate and support service type 'Best Effort' for Extended Flow Specification
TSPC_L2CAP_3_15	False (*)	Negotiate and support service type 'Guaranteed' for Extended Flow Specification
TSPC_L2CAP_3_16	True	Support Multiple Simultaneous LE Data Channels

SM PICS

PTS version: 6.4

* - different than PTS defaults

^ - field not available on PTS

M - mandatory

O - optional

Connection Roles

Parameter Name	Selected	Description
TSPC_SM_1_1	True	Master Role (Initiator) (C.1)
TSPC_SM_1_2	True	Slave Role (Responder) (C.2)

Security Properties

Parameter Name	Selected	Description
TSPC_SM_2_1	True	Authenticated MITM protection (O)
TSPC_SM_2_2	True	Unauthenticated no MITM protection (C.1)
TSPC_SM_2_3	True	No security requirements (M)
TSPC_SM_2_4	False	OOB supported (O)
TSPC_SM_2_5	(^)	LE Secure Connections (C.2)

Encryption Key Size

Parameter Name	Selected	Description
TSPC_SM_3_1	True	Encryption Key Size Negotiation (M)

Pairing Method

Parameter Name	Selected	Description
TSPC_SM_4_1	True	Just Works (O)
TSPC_SM_4_2	True	Passkey Entry (C.1)
TSPC_SM_4_3	False (*)	Out of Band (C.1)

Security Initiation

Parameter Name	Selected	Description
TSPC_SM_5_1	True	Encryption Setup using STK (C.3)
TSPC_SM_5_2	True	Encryption Setup using LTK (O)
TSPC_SM_5_3	True	Slave Initiated Security (C.1)
TSPC_SM_5_4	True	Slave Initiated Security - Master response(C.2)

Signing Algorithm

Parameter Name	Selected	Description
TSPC_SM_6_1	True	Signing Algorithm - Generation (O)
TSPC_SM_6_2	True	Signing Algorithm - Resolving (O)

Key Distribution

Parameter Name	Selected	Description
TSPC_SM_7_1	True	Encryption Key (C.1)
TSPC_SM_7_2	False (*)	Identity Key (C.2)
TSPC_SM_7_3	True	Signing Key (C.3)

RFCOMM PICS

PTS version: 6.4

- – different than PTS defaults

Protocol Version

Parameter Name	Selected	Description
TSPC_RFCOMM_0_1	False	RFCOMM 1.1 with TS 07.10
TSPC_RFCOMM_0_2	True (*)	RFCOMM 1.2 with TS 07.10

Supported Procedures

Parameter Name	Selected	Description
TSPC_RFCOMM_1_1	True (*)	Initialize RFCOMM Session
TSPC_RFCOMM_1_2	True (*)	Respond to Initialization of an RFCOMM Session
TSPC_RFCOMM_1_3	True	Shutdown RFCOMM Session
TSPC_RFCOMM_1_4	True	Respond to a Shutdown of an RFCOMM Session
TSPC_RFCOMM_1_5	True (*)	Establish DLC
TSPC_RFCOMM_1_6	True (*)	Respond to Establishment of a DLC
TSPC_RFCOMM_1_7	True	Disconnect DLC
TSPC_RFCOMM_1_8	True	Respond to Disconnection of a DLC
TSPC_RFCOMM_1_9	True	Respond to and send MSC Command
TSPC_RFCOMM_1_10	True	Initiate Transfer Information
TSPC_RFCOMM_1_11	True	Respond to Test Command
TSPC_RFCOMM_1_12	False	Send Test Command
TSPC_RFCOMM_1_13	True	React to Aggregate Flow Control
TSPC_RFCOMM_1_14	True	Respond to RLS Command
TSPC_RFCOMM_1_15	False	Send RLS Command
TSPC_RFCOMM_1_16	True	Respond to PN Command
TSPC_RFCOMM_1_17	True (*)	Send PN Command
TSPC_RFCOMM_1_18	True (*)	Send Non-Supported Command (NSC) response
TSPC_RFCOMM_1_19	True	Respond to RPN Command
TSPC_RFCOMM_1_20	False	Send RPN Command
TSPC_RFCOMM_1_21	True (*)	Closing Multiplexer by First Sending a DISC Command
TSPC_RFCOMM_1_22	True	Support of Credit Based Flow Control

Standard C Library

The kernel currently provides only the minimal subset of the standard C library required to meet the kernel's own needs, primarily in the areas of string manipulation and display.

Applications that require a more extensive C library can either submit contributions that enhance the existing library or substitute with a replacement library.

The Zephyr SDK and other supported toolchains comes with a bare-metal C library based on `newlib` that can be used with Zephyr by selecting the `CONFIG_NEWLIB_LIBC` in the application configuration file. Part of the support for `newlib` is a set of hooks available under `lib/libc/newlib/libc-hooks.c` which integrates the c library with basic kernel services.

Logging

System Logging

The system log API provides a common interface to process messages issued by developers. These messages are currently printed on the terminal but the API is defined in a generic way.

This API can be deactivated through the Kconfig options, see *Global Kconfig Options*. This approach prevents impacting image size and execution time when the system log is not needed.

Each of the four `SYS_LOG_X` macros correspond to a different logging level, The logging macros activate when their logging level or higher is set.

There are two configuration categories: configurations per module and global configurations. When logging is enabled globally, it works for modules. However, modules can disable logging locally. Every module can specify its own logging level. The module must define the `SYS_LOG_LEVEL` macro before including the `include/logging/sys_log.h` header file to do so. Unless a global override is set, the module logging level will be honored. The global override can only increase the logging level. It cannot be used to lower module logging levels that were previously set higher.

You can set a local domain to differentiate messages. When no domain is set, then the `[general]` domain appears before the message. Define the `SYS_LOG_DOMAIN` macro before including the `include/logging/sys_log.h` header file to set the domain.

When several macros are active, the printed messages can be differentiated in two ways: by a tag printed before the message or by ANSI colors. See the `CONFIG_SYS_LOG_SHOW_TAGS` and `CONFIG_SYS_LOG_SHOW_COLOR` Kconfig options for more information.

Define the `SYS_LOG_NO_NEWLINE` macro before including the `include/logging/sys_log.h` header file to prevent macros appending a new line at the end of the logging message.

Global Kconfig Options

These options can be found in the following path `subsys/logging/Kconfig`.

`CONFIG_SYS_LOG`: Global switch, turns on/off all system logging.

`CONFIG_SYS_LOG_DEFAULT_LEVEL`: Default level, sets the logging level used by modules that are not setting their own logging level.

`CONFIG_SYS_LOG_SHOW_TAGS`: Globally sets whether level tags will be shown on log or not.

`CONFIG_SYS_LOG_SHOW_COLOR`: Globally sets whether ANSI colors will be used by the system log.

`CONFIG_SYS_LOG_OVERRIDE_LEVEL`: It overrides module logging level when it is not set or set lower than the override value.

Example

The following macro:

```
SYS_LOG_WRN("hi!");
```

Will produce:

```
[general] [WRN] main: Hi!
```

For the above example to work at least one of the following settings must be true:

- The `CONFIG_SYS_LOG_DEFAULT_LEVEL` is set to 2 or above and module configuration is not set.
- The module configuration is set to 2 or above.
- The `CONFIG_SYS_LOG_OVERRIDE_LEVEL` is set to 2 or above.

APIs

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Kernel Event Logger

The kernel event logger records the occurrence of certain types of kernel events, allowing them to be subsequently extracted and reviewed. This capability can be helpful in profiling the operation of an application, either for debugging purposes or for optimizing the performance the application.

Concepts

The kernel event logger does not exist unless it is configured for an application. The capacity of the kernel event logger is also configurable. By default, it has a ring buffer that can hold up to 128 32-bit words of event information.

The kernel event logger is capable of recording the following predefined event types:

- Interrupts.
- Context switching of threads.
- Kernel sleep events (i.e. entering and exiting a low power state).

The kernel event logger only records the predefined event types it has been configured to record. Each event type can be enabled independently.

An application can also define and record custom event types. The information recorded for a custom event, and the times at which it is recorded, must be implemented by the application.

All events recorded by the kernel event logger remain in its ring buffer until they are retrieved by the application for review and analysis. The retrieval and analysis logic must be implemented by the application.

重要: An application must retrieve the events recorded by the kernel event logger in a timely manner, otherwise new events will be dropped once the event logger's ring buffer becomes full. A recommended approach is to use a cooperative thread to retrieve the events, either on a periodic basis or as its sole responsibility.

By default, the kernel event logger records all occurrences of all event types that have been enabled. However, it can also be configured to allow an application to dynamically start or stop the recording of events at any time, and to control which event types are being recorded. This permits the application to capture only the events that occur during times of particular interest, thereby reducing the work needed to analyze them.

注解: The kernel event logger can also be instructed to ignore context switches involving a single specified thread. This can be used to avoid recording context switch events involving the thread that retrieves the events from the kernel event logger.

Event Formats

Each event recorded by the kernel event logger consists of one or more 32-bit words of data that describe the event.

An **interrupt event** has the following format:

```
struct {
    u32_t timestamp;        /* time of interrupt */
    u32_t interrupt_id;     /* ID of interrupt */
};
```

A **context-switch event** has the following format:

```
struct {
    u32_t timestamp;        /* time of context switch */
    u32_t context_id;       /* ID of thread that was switched out */
};
```

A **sleep event** has the following format:

```
struct {
    u32_t sleep_timestamp;  /* time when CPU entered sleep mode */
    u32_t wake_timestamp;   /* time when CPU exited sleep mode */
    u32_t interrupt_id;     /* ID of interrupt that woke CPU */
};
```

A **custom event** must have a type ID that does not conflict with any existing predefined event type ID. The format of a custom event is application-defined, but must contain at least one 32-bit data word. A custom event may utilize a variable size, to allow different events of a single type to record differing amounts of information.

Timestamps

By default, the timestamp recorded with each predefined event is obtained from the kernel's *hardware clock*. This 32-bit clock counts up extremely rapidly, which means the timestamp value wraps around frequently. (For example, the Lakemont APIC timer for Quark SE wraps every 134 seconds.) This wraparound must be accounted for when analyzing kernel event logger data. In addition, care must be taken when tickless idle is enabled, in case a sleep duration exceeds 2^32 clock cycles.

If desired, the kernel event logger can be configured to record a custom timestamp, rather than the default timestamp. The application registers the callback function that generates the custom 32-bit timestamp at run-time by calling `sys_k_event_logger_set_timer()`.

Implementation

Retrieving An Event

An event can be retrieved from the kernel event logger in a blocking or non-blocking manner using the following APIs:

- `sys_k_event_logger_get()`
- `sys_k_event_logger_get_wait()`
- `sys_k_event_logger_get_wait_timeout()`

In each case, the API also returns the type and size of the event, as well as the event information itself. The API also indicates how many events were dropped between the occurrence of the previous event and the retrieved event.

The following code illustrates how a thread can retrieve the events recorded by the kernel event logger.

```
u16_t event_id;
u8_t  dropped_count;
u32_t data[3];
u8_t  data_size;

while(1) {
    /* retrieve an event */
    data_size = SIZE32_OF(data);
    res = sys_k_event_logger_get_wait(&event_id, &dropped_count, data,
                                      &data_size);

    if (dropped_count > 0) {
        /* ... Process the dropped events count ... */
    }

    if (res > 0) {
        /* process the event */
        switch (event_id) {
        case KERNEL_EVENT_LOGGER_CONTEXT_SWITCH_EVENT_ID:
            /* ... Process the context switch event ... */
            break;
        case KERNEL_EVENT_LOGGER_INTERRUPT_EVENT_ID:
            /* ... Process the interrupt event ... */
            break;
        case KERNEL_EVENT_LOGGER_SLEEP_EVENT_ID:
            /* ... Process the sleep event ... */
            break;
        default:
            printf("unrecognized event id %d\n", event_id);
        }
    } else if (res == -EMSGSIZE) {
        /* ... Data array is too small to hold the event! ... */
    }
}
```

Adding a Custom Event Type

A custom event type must use an integer type ID that does not duplicate an existing type ID. The type IDs for the predefined events can be found in `include/logging/kernel_event_logger.h`. If dynamic recording of events is enabled, the event type ID must not exceed 32.

Custom events can be written to the kernel event logger using the following APIs:

- `sys_k_event_logger_put()`
- `sys_k_event_logger_put_timed()`

Both of these APIs record an event as long as there is room in the kernel event logger's ring buffer. To enable dynamic recording of a custom event type, the application must first call `sys_k_must_log_event()` to determine if event recording is currently active for that event type.

The following code illustrates how an application can write a custom event consisting of two 32-bit words to the kernel event logger.

```
#define MY_CUSTOM_EVENT_ID 8

/* record custom event only if recording is currently wanted */
if (sys_k_must_log_event(MY_CUSTOM_EVENT_ID)) {
    u32_t data[2];

    data[0] = custom_data_1;
    data[1] = custom_data_2;

    sys_k_event_logger_put(MY_CUSTOM_EVENT_ID, data, ARRAY_SIZE(data));
}
```

The following code illustrates how an application can write a custom event that records just a timestamp using a single 32-bit word.

```
#define MY_CUSTOM_TIME_ONLY_EVENT_ID 9

if (sys_k_must_log_event(MY_CUSTOM_TIME_ONLY_EVENT_ID)) {
    sys_k_event_logger_put_timed(MY_CUSTOM_TIME_ONLY_EVENT_ID);
}
```

Configuration Options

Related configuration options:

- `CONFIG_KERNEL_EVENT_LOGGER`
- `CONFIG_KERNEL_EVENT_LOGGER_CONTEXT_SWITCH`
- `CONFIG_KERNEL_EVENT_LOGGER_INTERRUPT`
- `CONFIG_KERNEL_EVENT_LOGGER_SLEEP`
- `CONFIG_KERNEL_EVENT_LOGGER_BUFFER_SIZE`
- `CONFIG_KERNEL_EVENT_LOGGER_DYNAMIC`
- `CONFIG_KERNEL_EVENT_LOGGER_CUSTOM_TIMESTAMP`

Related Functions

The following kernel event logger APIs are provided by `kernel_event_logger.h`:

- `sys_k_event_logger_register_as_collector()`
- `sys_k_event_logger_get()`

- `sys_k_event_logger_get_wait()`
- `sys_k_event_logger_get_wait_timeout()`
- `sys_k_must_log_event()`
- `sys_k_event_logger_put()`
- `sys_k_event_logger_put_timed()`
- `sys_k_event_logger_get_mask()`
- `sys_k_event_logger_set_mask()`
- `sys_k_event_logger_set_timer()`

APIs

Event Logger

An event logger is an object that can record the occurrence of significant events, which can be subsequently extracted and reviewed.

> **警告:** doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Kernel Event Logger

The kernel event logger records the occurrence of significant kernel events, which can be subsequently extracted and reviewed. (See *Kernel Event Logger*.)

> **警告:** doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Networking

The networking section contains information regarding the network stack of the Zephyr kernel. Use the information to understand the principles behind the operation of the stacks and how they were implemented.

Overview

Supported Features

The networking IP stack is modular and highly configurable via build-time configuration options. You can minimize system memory consumption by enabling only those network features required by your application. Almost all features can be disabled if not needed.

- **IPv6** The support for IPv6 is enabled by default. Various IPv6 sub-options can be enabled or disabled depending on networking needs.
  - Developer can set the number of unicast and multicast IPv6 addresses that are active at the same time.

  - The IPv6 address for the device can be set either statically or dynamically using SLAAC (Stateless Address Auto Configuration) (RFC 4862).
  - The system also supports multiple IPv6 prefixes and the maximum IPv6 prefix count can be configured at build time.
  - The IPv6 neighbor cache can be disabled if not needed, and its size can be configured at build time.
  - The IPv6 neighbor discovery support (RFC 4861) is enabled by default.
  - Multicast Listener Discovery v2 support (RFC 3810) is enabled by default.
  - IPv6 header compression (6lo) is available for IPv6 connectivity for Bluetooth IPSP (RFC 7668) and IEEE 802.15.4 networks (RFC 4944).
- **IPv4** The legacy IPv4 is supported by the networking stack. It cannot be used by IEEE 802.15.4 or Bluetooth IPSP as those network technologies support only IPv6. IPv4 can be used in ethernet based networks. By default IPv4 support is disabled.
  - DHCP (Dynamic Host Configuration Protocol) client is supported (RFC 2131).
  - The IPv4 address can also be configured manually. Static IPv4 addresses are supported by default.
- **Dual stack support.** The networking stack allows a developer to configure the system to use both IPv6 and IPv4 at the same time.
- **UDP** User Datagram Protocol (RFC 768) is supported. The developer can send UDP datagrams (client side support) or create a listener to receive UDP packets destined to certain port (server side support).
- **TCP** Transmission Control Protocol (RFC 793) is supported. Both server and client roles can be used the the application. The amount of TCP sockets that are available to applications can be configured at build time.
- **BSD Sockets API** Experimental support for a subset of a BSD Sockets compatible API is implemented. Both blocking and non-blocking DGRAM (UDP) amd STREAM (TCP) sockets are supported.
- **HTTP** Hypertext Transfer Protocol (RFC 2116) is supported. A simple library is provided that applications can use. Sample applications are implemented for http-client-sample and http-server-sample. Both http-client-sample and http-server-sample can use TLS (Transport Layer Security) v1.2 (RFC 5246) or SSL (Secure Sockets Layer) v3.0 (RFC 6101) functionality to encrypt the network traffic. The secured connections are provided by mbed library.
- **MQTT** Message Queue Telemetry Transport (ISO/IEC PRF 20922) is supported. A sample mqtt-publisher-sample client application for MQTT v3.1.1 is implemented.
- **CoAP** Constrained Application Protocol (RFC 7252) is supported. Both Both zoap-client-sample and zoap-server-sample sample applications are implemented. A coap-client-sample and coap-server-sample using DTLS (Datagram Transport Layer Security) (RFC 6347) are also implemented.
- **LWM2M** OMA Lightweight Machine-to-Machine Protocol (V1.0 Feb 2017) is supported via the "Register Device" API (Register, De-Register and Update) and has template implementations for Securty, Server, Device Management and Firmware objects. DTLS and Bootstrap support are currently not supported. lwm2m-client-sample implements the library as an example.
- **RPL** IPv6 Routing Protocol for Low-Power and Lossy Networks (RFC 6550) is supported. RPL is an IPv6 based mesh routing protocol.
- **DNS** Domain Name Service (RFC 1035) client functionality is supported. Applications can use an API to query domain name information or IP addresses from the DNS server. Both IPv4 (A) and IPv6 (AAAA) records can be queried.
- **Network Management API.** Applications can use network management API to listen management events generated by core stack when for example IP address is added to the device, or network interface is coming up etc.

- **Multiple Network Technologies.** The Zephyr OS can be configured to support multiple network technologies at the same time simply by enabling them in Kconfig: for example, Ethernet and 802.15.4 support. Note that no automatic IP routing functionality is provided between these technologies. Applications can send data according to their needs to desired network interface.
- **Minimal Copy Network Buffer Management.** It is possible to have minimal copy network data path. This means that the system tries to avoid copying application data when it is sent to the network. For some technologies it is even possible to have zero-copy data path from application to device driver.

Additionally these network technologies (link layers) are supported in Zephyr OS v1.7 and later:

- IEEE 802.15.4
- Bluetooth
- Ethernet
- SLIP (IP over serial line). Used for testing with QEMU. It provides ethernet interface to host system (like Linux) and test applications can be run in Linux host and send network data to Zephyr OS device.

Source Tree Layout

The IP stack source code tree is organized as follows:

`subsys/net/ip/` This is where the IP stack code is located.

`subsys/net/lib/` Application-level protocols (DNS, MQTT, etc.) and additional stack components (BSD Sockets, etc.).

`include/net/` Public API header files. These are the header files applications need to include to use IP networking functionality.

`samples/net/` Sample networking code. This is a good reference to get started with network application development.

`tests/net/` Test applications. These applications are used to verify the functionality of the IP stack, but are not the best source for sample code (see `samples/net` instead).

IP Stack Architecture

High level overview of the IP stack

图 1.4: Network stack overview

The IP stack is layered and it consists of the following parts:

- **Networking Application.** This application uses the connectivity API to manipulate a network connection, and management API to set network related parameters such as starting a scan (when applicable), setting IP address to a network interface, etc.
- **Core IP stack.** This provides implementations for various protocols such as IPv6, IPv4, UDP, TCP, ICMPv4 and ICMPv6.
- **Network interface abstraction layer.** This provides functionality that is common in all the network interfaces, such as acquiring an IP address, etc.
- **Generic L2 layer.** This provides common API for sending and receiving data to and from an actual network device.

- **L2 network technology component.** These components include Ethernet, IEEE 802.15.4, Bluetooth, etc. Some of these technologies support IPv6 header compression (6LoWPAN), which is done in its own layer. For example ARP for IPv4 is done in the Ethernet component.
- **Network device driver.** The actual low-level device driver handles the physical sending or receiving of a network packet.

Network data flow

图 1.5: Network data flow

The application typically consists of one or more tasks or threads that execute the application logic. When using the network connectivity APIs, following things will happen.

Data receiving (RX):

1. A network data packet is received by a device driver.
2. The device driver allocates enough network buffers to store the received data. The network buffers are then passed to the RX FIFO for further processing. The RX FIFO is used as a way to separate the data processing pipeline (bottom-half) as the device driver is running in interrupt context and it must do its processing very fast.
3. The RX thread reads the RX FIFO and passes the data to the correct L2 driver. After the L2 driver has checked the packet, the packet is passed to L3 processing. The L3 layer checks if the packet is a proper IPv6 or IPv4 packet. If the packet contains UDP or TCP data, it is then sent to correct application via a function callback. This also means that the application data processing in that callback is run in thread context even if the actual application is running in task context. The data processing in the application callback should be done fast in order not to block the system too long. There is only one RX thread in the system. The stack size of the RX thread can be tweaked via Kconfig option but it should be kept as small as possible. This also means that stack utilization in the data processing callback should be minimized in order to avoid stack overflow.
4. The application will then receive the data, which is stored inside a chain of net_bufs. The application now owns the data. After it has finished working with it, the application should release the net_bufs data by calling `net_pkt_unref()`.

Data sending (TX):

1. The application should use the connectivity API when the application is ready to send data. The sent data is checked by the correct L2 layer module and if everything is ok, the data is placed into the network interface TX queue. The network interface is typically selected to be the same interface for reply data packets or the interface is selected according to the routing algorithm. The application should not free the data packet if it was correctly placed into TX queue; the network driver will release the packet after it is sent. If the connectivity API sending function returns an error to the application, that means the packet was not sent correctly and the application needs to free the packet.
2. Each network interface has a dedicated TX queue used to send data to that interface. A TX thread in the system reads all the TX queues and passes that data to the correct L2 driver, for sending via the device driver.
3. If the device driver is able to inject the network packet into the network, then it will release the packet. Typically there are no retransmits at this lower level so usually the packet is released even if not sent correctly. This depends on the technology being used.

Network Connectivity API

Applications can use the connectivity API defined in `net_context.h` to create a connection, send or receive data, and close a connection. The same API can be used when working with UDP or TCP data.

The net_context API is similar to the BSD socket API and mapping between these two is possible. The main difference between net_context API and BSD socket API is that the net_context API uses the fragmented network buffers (net_buf) defined in `include/net/buf.h` and BSD socket API uses linear memory buffers.

This example creates a simple server that listens to incoming UDP connections and sends the received data back. You can download the example application source file here connectivity-example-app.c

This example application begins with some initialization. (Use this as an example; you may need to do things differently in your own application.)

```c
#define SYS_LOG_DOMAIN "example-app"
#define SYS_LOG_LEVEL SYS_LOG_LEVEL_DEBUG
#define NET_DEBUG 1

#include <zephyr.h>

#include <net/net_pkt.h>
#include <net/net_core.h>
#include <net/net_context.h>

#define MY_IP6ADDR { { { 0x20, 0x01, 0x0d, 0xb8, 0, 0, 0, 0, \
                         0, 0, 0, 0, 0, 0, 0, 0x1 } } }
#define MY_PORT 4242

struct in6_addr in6addr_my = MY_IP6ADDR;
struct sockaddr_in6 my_addr6 = { 0 };
struct net_context *context;
int ret;

struct nano_sem quit_lock;

static inline void quit(void)
{
        nano_sem_give(&quit_lock);
}

static inline void init_app(void)
{
        nano_sem_init(&quit_lock);

        /* Add our address to the network interface */
        net_if_ipv6_addr_add(net_if_get_default(), &in6addr_my,
                             NET_ADDR_MANUAL, 0);
}

void main(void)
{
        NET_INFO("Run sample application");

        init_app();

        create_context();

        bind_address();

        receive_data();

        nano_sem_take(&quit_lock, TICKS_UNLIMITED);

        close_context();

        NET_INFO("Stopping sample application");
}
```

After initialization, first thing application needs to create a context. Context is similar to a socket.

```
static int create_context(void)
{
        ret = net_context_get(AF_INET6, SOCK_DGRAM, IPPROTO_UDP, &context);
        if (!ret) {
                NET_ERR("Cannot get context (%d)", ret);
                return ret;
        }

        return 0;
}
```

Then you need to define the local end point for a connection.

```
static int bind_address(void)
{
        net_ipaddr_copy(&my_addr6.sin6_addr, &in6addr_my);
        my_addr6.sin6_family = AF_INET6;
        my_addr6.sin6_port = htons(MY_PORT);

        ret = net_context_bind(context, (struct sockaddr *)&my_addr6);
        if (ret < 0) {
                NET_ERR("Cannot bind IPv6 UDP port %d (%d)",
                        ntohs(my_addr6.sin6_port), ret);
                return ret;
        }

        return 0;
}
```

Wait until the connection data is received.

```
#define MAX_DBG_PRINT 64

static struct net_buf *udp_recv(const char *name,
                                struct net_context *context,
                                struct net_buf *buf)
{
        struct net_buf *reply_buf, *frag, *tmp;
        int header_len, recv_len, reply_len;

        NET_INFO("%s received %u bytes", name,
              net_pkt_appdatalen(buf));

        reply_buf = net_pkt_get_tx(context, K_FOREVER);

        NET_ASSERT(reply_buf);

        recv_len = net_buf_frags_len(buf->frags);

        tmp = buf->frags;

        /* First fragment will contain IP header so move the data
         * down in order to get rid of it.
         */
        header_len = net_pkt_appdata(buf) - tmp->data;

        NET_ASSERT(header_len < CONFIG_NET_BUF_DATA_SIZE);

        net_buf_pull(tmp, header_len);

        while (tmp) {
                frag = net_pkt_get_data(context, K_FOREVER);

                memcpy(net_buf_add(frag, tmp->len), tmp->data, tmp->len);

                net_buf_frag_add(reply_buf, frag);

                net_buf_frag_del(buf, tmp);

                tmp = buf->frags;
        }

        reply_len = net_buf_frags_len(reply_buf->frags);

        NET_ASSERT_INFO(recv_len != reply_len,
                        "Received %d bytes, sending %d bytes",
                        recv_len, reply_len);

        return reply_buf;
}

static inline void udp_sent(struct net_context *context,
                            int status,
                            void *token,
                            void *user_data)
{
        if (!status) {
                NET_INFO("Sent %d bytes", POINTER_TO_UINT(token));
        }
}

static inline void set_dst_addr(sa_family_t family,
                                struct net_buf *buf,
                                struct sockaddr *dst_addr)
{
        if (family == AF_INET6) {
                net_ipaddr_copy(&net_sin6(dst_addr)->sin6_addr,
                                &NET_IPV6_HDR(buf)->src);
                net_sin6(dst_addr)->sin6_family = AF_INET6;
                net_sin6(dst_addr)->sin6_port = NET_UDP_HDR(buf)->src_port;
        }
}

static void udp_received(struct net_context *context,
                         struct net_buf *buf,
                         int status,
                         void *user_data)
{
        struct net_buf *reply_buf;
        struct sockaddr dst_addr;
        sa_family_t family = net_pkt_family(buf);
        static char dbg[MAX_DBG_PRINT + 1];
        int ret;

        snprintf(dbg, MAX_DBG_PRINT, "UDP IPv%c",
                 family == AF_INET6 ? '6' : '4');

        set_dst_addr(family, buf, &dst_addr);

        reply_buf = udp_recv(dbg, context, buf);

        net_pkt_unref(buf);

        ret = net_context_sendto(reply_buf, &dst_addr, udp_sent, 0,
                                 UINT_TO_POINTER(net_buf_frags_len(reply_buf)),
                                 user_data);
        if (ret < 0) {
                NET_ERR("Cannot send data to peer (%d)", ret);

                net_pkt_unref(reply_buf);

                quit();
        }
}

static int receive_data(void)
{
        ret = net_context_recv(context, udp_received, 0, NULL);
        if (ret < 0) {
                NET_ERR("Cannot receive IPv6 UDP packets");

                quit();

                return ret;
        }

        return 0;
}
```

Close the context when finished.

```
/* Context close */
static int close_context(void)
{
        ret = net_context_put(context);
        if (ret < 0) {
                NET_ERR("Cannot close IPv6 UDP context");
                return ret;
        }

        return 0;
}
```

BSD Sockets compatible API

Zephyr offers an implementation of a subset of the BSD Sockets API (a part of the POSIX standard). This API allows to reuse existing programming experience and port existing simple networking applications to Zephyr.

Here are the key requirements and concepts which governed BSD Sockets compatible API implementation for Zephyr:

- Should have minimal overhead, similar to the requirement for other Zephyr subsystems.
- Should be implemented on top of *native networking API* to keep modular design.
- Should be namespaced by default, to avoid name conflicts with well-known names like `close()`, which may be part of libc or other POSIX compatibility libraries. If enabled, should also expose native POSIX names.

BSD Sockets compatible API is enabled using `CONFIG_NET_SOCKETS` config option and implements the following operations: `socket()`, `close()`, `recv()`, `send()`, `connect()`, `bind()`, `listen()`, `fcntl()` (to set non-blocking mode), `poll()`.

Based on the namespacing requirements above, these operations are by default exposed as functions with `zsock_` prefix, e.g. `zsock_socket()` and `zsock_close()`. If the config option `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined, all the functions will be also exposed as aliases without the prefix. This includes the functions like `close()` and `fcntl()` (which may conflict with functions in libc or other libraries, for example, with the filesystem libraries).

The native BSD Sockets API uses file descriptors to represent sockets. File descriptors are small integers, consecutively assigned from zero. Internally, there is usually a table mapping file descriptors to internal object pointers. For memory efficiency reasons, the Zephyr BSD Sockets compatible API is devoid of such a table. Instead, `net_context` pointers, cast to an int, are used to represent sockets. Thus, socket identifiers aren't really small integers, so the `select()` operation is not available, as it depends on the "small int" property of file descriptors. Instead of using `select()` then, use the `poll()` operation, which is generally more efficient.

The BSD Sockets API (and the POSIX API in general) also treat negative file descriptors values in a special way (such values usually mean an error). As the Zephyr API uses a pointer value cast to an int for file descriptors, it means that the pointer should not have the highest bit set, in other words, pointers should not refer to the second (highest) part of the address space. For many CPU architectures and SoCs Zephyr supports, user RAM is located in the lower half, so the above condition is satisfied. If you face an issue with some SoC because of this, please report it to the Zephyr bug tracker or mailing list. The decision to use pointers to represent sockets might be reworked in the future.

The final entailment of the design requirements above is that the Zephyr API aggressively employs the short-read/short-write property of the POSIX API whenever possible (to minimize complexity and overheads). POSIX allows for calls like `recv()` and `send()` to actually process (receive or send) less data than requested by the user (on STREAM type sockets). For example, a call `recv(sock, 1000, 0)` may return 100, meaning that only 100 bytes were read (short read), and the application needs to retry call(s) to read the remaining 900 bytes.

L2 Stack and Drivers

The L2 stack is designed to hide the whole networking link-layer part and the related device drivers from the higher IP stack. This is made through a unique object known as the "network interface object": `struct net_if` declared in `include/net/net_if.h`.

The IP layer is unaware of implementation details beyond the net_if object and the generic API provided by the L2 layer in `include/net/net_l2.h` as `struct net_l2`.

Only the L2 layer can talk to the device driver, linked to the net_if object. The L2 layer dictates the API provided by the device driver, specific for that device, and optimized for working together.

Currently, there are L2 layers for Ethernet, IEEE 802.15.4 Soft-MAC, Bluetooth IPSP, and a dummy one, which is a generic layer example that can be used as a template for writing a new one.

L2 layer API

In order to create an L2 layer, or even a driver for a specific L2 layer, one needs to understand how the IP layer interacts with it and how the L2 layer is supposed to behave. The generic L2 API has 3 functions:

- recv: All device drivers, once they receive a packet which they put into a `struct net_pkt`, will push this buffer to the IP core stack via `net_recv_data()`. At this point, the IP core stack does not know what to do with it. Instead, it passes the buffer along to the L2 stack's recv() function for handling. The L2 stack does what it needs to do with the packet, for example, parsing the link layer header, or handling link-layer only packets. The recv() function will return NET_DROP in case of an erroneous packet, NET_OK if the packet was fully consumed by the L2, or NET_CONTINUE if the IP stack should then handle it as an IP packet.
- reserve: Prior to creating any network buffer content, the Zephyr core stack needs to know how much dedicated buffer space is needed for the L2 layer (for example, space for the link layer header). This reserve function returns the number of bytes needed.
- send: Similar to recv, the IP core stack will call this function to actually send a packet. All relevant link-layer content will be generated and added by this function. As for recv, send returns a verdict and can decide to drop the packet via NET_DROP if something wrong happened, or will return NET_OK.

Network Device drivers

Network device drivers fully follows Zephyr device driver model as a basis. Please refer to *Device Drivers and Device Model*.

There are, however, two differences:

- the driver_api pointer must point to a valid `struct net_if_api` pointer.
- The network device driver must use `NET_DEVICE_INIT_INSTANCE()`. This macro will call the DEVICE_AND_API_INIT() macro, and also instantiate a unique `struct net_if` related to the created device driver instance.

Implementing a network device driver depends on the L2 stack it belongs to: Ethernet, IEEE 802.15.4, etc. In the next section, we will describe how a device driver should behave when receiving or sending a packet. The rest is really hardware dependent and thus does not need to be detailed here.

Ethernet device driver

On reception, it is up to the device driver to fill-in the buffer with as many data fragments as required. The buffer itself is a `struct net_pkt` and should be allocated through `net_pkt_get_reserve_rx(0)()`. Then all fragments will be allocated through `net_pkt_get_reserve_data(0)()`. Of course the amount of required fragments depends on the size of the received packet and on the size of a fragment, which is given by `CONFIG_NET_BUF_DATA_SIZE`.

Note that it is not up to the device driver to decide on the link-layer space to be reserved in the buffer. Hence the 0 given as parameter here. The Ethernet L2 layer will update such information once the packet's Ethernet header has been successfully parsed.

In case `net_recv_data()` call fails, it will be up to the device driver to unreference the buffer via `net_pkt_unref()`.

On sending, it is up to the device driver to send the buffer all at once, with all the fragments.

In case of a fully successful packet transmission only, the device driver must unreference the buffer via `net_pkt_unref()`.

Each Ethernet device driver will need, in the end, to call `NET_DEVICE_INIT_INSTANCE()` like this:

```
NET_DEVICE_INIT_INSTANCE(...,
                         CONFIG_ETH_INIT_PRIORITY
                         &the_valid_net_if_api_instance,
                         ETHERNET_L2,
                         NET_L2_GET_CTX_TYPE(ETHERNET_L2), 1500);
```

IEEE 802.15.4 device driver

Device drivers for IEEE 802.15.4 L2 work basically the same as for Ethernet. What has been described above, especially for recv, applies here as well. There are two specific differences however:

- It requires a dedicated device driver API: `struct ieee802154_radio_api`, which overloads `struct net_if_api`. This is because 802.15.4 L2 needs more from the device driver than just send and recv functions. This dedicated API is declared in `include/net/ieee802154_radio.h`. Each and every IEEE 802.15.4 device driver must provide a valid pointer on such relevantly filled-in API structure.
- Sending a packet is slightly particular. IEEE 802.15.4 sends relatively small frames, 127 bytes all inclusive: frame header, payload and frame checksum. Buffer fragments are meant to fit such frame size limitation. But a buffer containing an IPv6/UDP packet might have more than one fragment. In the Ethernet device driver, it is up to the driver to handle all fragments. IEEE 802.15.4 drivers handle only one fragment at a time. This is why the `struct ieee802154_radio_api` requires a tx function pointer which differs from the `struct net_if_api` send function pointer. Instead, the IEEE 802.15.4 L2, provides a generic `ieee802154_radio_send()` meant to be given as `struct net_if` send function. It turn, the implementation of `ieee802154_radio_send()` will ensure the same behavior: sending one fragment at a time through `struct ieee802154_radio_api` tx function, and unreferencing the buffer only when all the transmission were successful.

Each IEEE 802.15.4 device driver, in the end, will need to call `NET_DEVICE_INIT_INSTANCE()` that way:

```
NET_DEVICE_INIT_INSTANCE(...,
                         the_device_init_prio,
                         &the_valid_ieee802154_radio_api_instance,
                         IEEE802154_L2,
                         NET_L2_GET_CTX_TYPE(IEEE802154_L2), 125);
```

Network Management API

The Network Management APIs allow applications, as well as network layer code itself, to call defined network routines at any level in the IP stack, or receive notifications on relevant network events. For example, by using these APIs, code can request a scan be done on a WiFi- or Bluetooth-based network interface, or request notification if a network interface IP address changes.

The Network Management API implementation is designed to save memory by eliminating code at build time for management routines that are not used. Distinct and statically defined APIs for network management procedures are not used. Instead, defined procedure handlers are registered by using a *NET_MGMT_REGISTER_REQUEST_HANDLER* macro. Procedure requests are done through a single `net_mgmt()` API that invokes the registered handler for the corresponding request.

The current implementation is experimental and may change and improve in future releases.

Requesting a defined procedure

All network management requests are of the form `net_mgmt(mgmt_request, ...)`. The `mgmt_request` parameter is a bit mask that tells which stack layer is targeted, if a `net_if` object is implied, and the specific management procedure being requested. The available procedure requests depend on what has been implemented in the stack.

To avoid extra cost, all `net_mgmt()` calls are direct. Though this may change in a future release, it will not affect the users of this function.

Listening to network event

You can receive notifications on network events by registering a callback function and specifying an event mask used to match one or more events that are relevant.

Two functions are available, `net_mgmt_add_event_callback()` for registering the callback function, and `net_mgmt_del_event_callback()` for unregistering. A helper function, `net_mgmt_init_event_cb()`, can be used to ease the initialization of the callback structure.

When an event is raised that matches a registered event mask, the associated callback function is invoked with the actual event code. This makes it possible for different events to be handled by the same callback function, if desired.

See an example of registering callback functions and using the network management API in `test/net/mgmt/src/mgmt.c`.

Defining a network management procedure

You can provide additional management procedures specific to your stack implementation by defining a handler and registering it with an associated mgmt_request code.

Management request code are defined in relevant places depending on the targeted layer or eventually, if l2 is the layer, on the technology as well. For instance, all IP layer management request code will be found in the `include/net/net_mgmt.h` header file. But in case of an L2 technology, let's say Ethernet, these would be found in `include/net/ethernet.h`

You define your handler modeled with this signature:

```
static int your_handler(u32_t mgmt_event, struct net_if *iface,
                        void *data, size_t len);
```

and then register it with an associated mgmt_request code:

```
NET_MGMT_REGISTER_REQUEST_HANDLER(<mgmt_request code>, your_handler);
```

This new management procedure could then be called by using:

```
net_mgmt(<mgmt_request code>, ...);
```

Signaling a network event

You can signal a specific network event using the `net_mgmt_notify()` function and provide the network event code. See `include/net/net_mgmt.h` for details. As for the management request code, event code can be also found on specific L2 technology headers, for example `include/net/ieee802154.h` would be the right place if 802.15.4 L2 is the technology one wants to listen to events.

Network Application API

The Network Application (net-app) API allows applications to:

Initialize The application for networking use. This means, for example, that if the application needs to have an IPv4 address, and if DHCPv4 is enabled, then the net-app API will make sure that the device will get an IPv4 address before the application is started.

Set Various options for the networking subsystem. This means that if the user has set options like IP addresses, IEEE 802.15.4 channel etc. in the project configuration file, then those settings are applied to the system before the application starts.

Create A simple TCP/UDP server or client application. The net-app API has functions that make it easy to create a simple TCP or UDP based network application. The net-app API also provides transparent TLS and DTLS support for the application.

The net-app API functionality is enabled by `CONFIG_NET_APP` option. The current net-app API implementation is still experimental and may change and improve in future releases.

Initialization

The net-app API provides a `net_app_init()` function that can configure the networking subsystem for the application. The following configuration options control this configuration:

`CONFIG_NET_APP_AUTO_INIT` automatically configures the system according to other configuration options. The user does not need to call `net_app_init()` in this case as that function will be automatically called when the system boots. This option is enabled by default.

`CONFIG_NET_APP_INIT_TIMEOUT` specifies how long to wait for the network configuration during the system boot. For example, if DHCPv4 is enabled, and if the IPv4 address discovery takes too long or the DHCPv4 server is not found, the system will resume booting after this number of seconds.

`CONFIG_NET_APP_NEED_IPV6` specifies that the application needs IPv6 connectivity. The `net_app_init()` function will wait until it is able to setup an IPv6 address for the system before continuing. This means that the IPv6 duplicate address detection (DAD) has finished and the system has properly setup the IPv6 address.

`CONFIG_NET_APP_NEED_IPV6_ROUTER` specifies that the application needs IPv6 router connectivity; i.e., it needs access to external networks (such as the Internet). The `net_app_init()` function will wait until it receives a router advertisement (RA) message from the IPv6 router before continuing.

`CONFIG_NET_APP_NEED_IPV4` specifies that the application needs IPv4 connectivity. The `net_app_init()` function will wait, unless a static IP address is configured, until it is able to setup an IPv4 address for the network subsystem.

Setup

Various system level network configuration options can be added to the project configuration file. These settings are enabled by the `CONFIG_NET_APP_SETTINGS` configuration option. This option is disabled by default, and other net-app options may also be disabled by default if generic support for the networking feature is disabled. For example, the IPv6 net-app options are only available if generic IPv6 support is enabled.

`CONFIG_NET_APP_MY_IPV6_ADDR` This option sets a static IPv6 address for the system. This is typically only useful in device testing as normally the system should use SLAAC (IPv6 Stateless Address Auto Configuration), which is enabled by default in the system. The system can be configured to use multiple IPv6 addresses; this is controlled by the `CONFIG_NET_IF_UNICAST_IPV6_ADDR_COUNT` configuration option.

CONFIG_NET_APP_PEER_IPV6_ADDR This option specifies what is the peer device IPv6 address. This is only useful when testing client/server type applications. This peer address is typically used as a parameter when calling `net_app_connect()`.

CONFIG_NET_APP_MY_IPV4_ADDR This option sets a static IPv4 address for the system. This is typically useful only in device testing as normally the system should use DHCPv4 to discover the IPv4 address.

CONFIG_NET_APP_PEER_IPV4_ADDR This option specifies what is the peer device IPv4 address. This is only useful when testing client/server type applications. This peer address is typically used as a parameter when connecting to other device.

The following options are only available if IEEE 802.15.4 wireless network technology support is enabled.

CONFIG_NET_APP_IEEE802154_DEV_NAME This option specifies the name of the IEEE 802.15.4 device.

CONFIG_NET_APP_IEEE802154_PAN_ID This option specifies the used PAN identifier. Note that the PAN id can be changed at runtime if needed.

CONFIG_NET_APP_IEEE802154_CHANNEL This option specifies the used radio channel. Note that the used channel can be changed at runtime if needed.

CONFIG_NET_APP_IEEE802154_RADIO_TX_POWER This option specifies the initial radio TX power level. The TX power level can be changed at runtime if needed.

CONFIG_NET_APP_IEEE802154_SECURITY_KEY This option specifies the initially used security key. The security key can be changed at runtime if needed.

CONFIG_NET_APP_IEEE802154_SECURITY_KEY_MODE This option specifies the initially used security key mode. The security key mode can be changed at runtime if needed.

CONFIG_NET_APP_IEEE802154_SECURITY_LEVEL This option specifies the initially used security level. The used security level can be changed at runtime if needed.

Client / Server Applications

The net-app API provides functions that enable the application to create client / server applications easily. If needed, the applications can have the communication secured by TLS (for TCP connections) or DTLS (for UDP connections) automatically.

A simple **TCP server** application would make the following net-app API function calls:

- `net_app_init_tcp_server()` to configure a local address and TCP port.
- `net_app_set_cb()` to configure callback functions to invoke in response to events, such as data reception.
- `net_app_server_tls()` will optionally setup the system for secured connections. To enable the TLS server, also call the `net_app_server_tls_enable()` function.
- `net_app_listen()` will start listening for new client connections.

Creating a **UDP server** is also very easy:

- `net_app_init_udp_server()` to configure a local address and UDP port.
- `net_app_set_cb()` to configure callback functions to invoke in response to events, such as data reception.
- `net_app_server_tls()` will optionally setup the system for secured connections. To enable the DTLS server, also call the `net_app_server_tls_enable()` function.
- `net_app_listen()` will start listening for new client connections.

If the server wants to stop listening for connections, it can call `net_app_release()`. After this, if the application wants to start listening for incoming connections again, it must call the server initialization functions.

For TLS/DTLS connections, the server can be disabled by a call to `net_app_server_tls_disable()`. There are separate enable/disable functions for TLS support because we need a separate crypto thread for calling mbedtls crypto API functions. The enable/disable TLS functions will either create the TLS thread or kill it.

A simple **TCP client** application would make the following net-app API function calls:

- `net_app_init_tcp_client()` to configure a local address, peer address and TCP port. If the DNS resolver support is enabled in the project configuration file, then the peer address can be given as a hostname, and the API tries to resolve it to IP address before connecting.
- `net_app_set_cb()` to configure callback functions to invoke in response to events, such as data reception.
- `net_app_client_tls()` will optionally setup the system for secured connections. The TLS crypto thread will be automatically created when the application calls `net_app_connect()` function.
- `net_app_connect()` will initiate a new connection to the peer host.

Creating a **UDP client** is also very easy:

- `net_app_init_udp_client()` to configure a local address, peer address and UDP port. If peer name is a hostname, then it will be automatically resolved to IP address if DNS resolver is enabled.
- `net_app_set_cb()` to configure callback functions to invoke in response to events, such as data reception.
- `net_app_client_tls()` will optionally setup the system for secured connections. The DTLS crypto thread will be automatically created when the application calls `net_app_connect()` function.
- `net_app_connect()` will initiate a new connection to the peer host. As the UDP is connectionless protocol, this function is very simple and it will just call the connected callback if that is defined.

As both the `echo_server` and `echo_client` applications use net-app API functions, please see those applications for more detailed usage examples.

The net-tools project has information how to test the system if TLS and DTLS support is enabled. See the **README** file in that project for more information.

Network Buffers

Network buffers are a core concept of how the networking stack (as well as the Bluetooth stack) pass data around. The API for them is defined in `include/net/buf.h`.

Creating buffers

Network buffers are created by first defining a pool of them:

```
NET_BUF_POOL_DEFINE(pool_name, buf_count, buf_size, user_data_size, NULL);
```

The pool is a static variable, so if it's needed to be exported to another module a separate pointer is needed.

Once the pool has been defined, buffers can be allocated from it with:

```
buf = net_buf_alloc(&pool_name, timeout);
```

There is no explicit initialization function for the pool or its buffers, rather this is done implicitly as `net_buf_alloc()` gets called.

If there is a need to reserve space in the buffer for protocol headers to be prepended later, it's possible to reserve this headroom with:

```
net_buf_reserve(buf, headroom);
```

In addition to actual protocol data and generic parsing context, network buffers may also contain protocol-specific context, known as user data. Both the maximum data and user data capacity of the buffers is compile-time defined when declaring the buffer pool.

The buffers have native support for being passed through k_fifo kernel objects. This is a very practical feature when the buffers need to be passed from one thread to another. However, since a net_buf may have a fragment chain attached to it, instead of using the `k_fifo_put()` and `k_fifo_get()` APIs, special `net_buf_put()` and `net_buf_get()` APIs must be used when passing buffers through FIFOs. These APIs ensure that the buffer chains stay intact.

Common Operations

The network buffer API provides some useful helpers for encoding and decoding data in the buffers. To fully understand these helpers it's good to understand the basic names of operations used with them:

Add Add data to the end of the buffer. Modifies the data length value while leaving the actual data pointer intact. Requires that there is enough tailroom in the buffer. Some examples of APIs for adding data:

```
void *net_buf_add(struct net_buf *buf, size_t len);
u8_t *net_buf_add_u8(struct net_buf *buf, u8_t value);
void net_buf_add_le16(struct net_buf *buf, u16_t value);
void net_buf_add_le32(struct net_buf *buf, u32_t value);
```

Push Prepend data to the beginning of the buffer. Modifies both the data length value as well as the data pointer. Requires that there is enough headroom in the buffer. Some examples of APIs for pushing data:

```
void *net_buf_push(struct net_buf *buf, size_t len);
void net_buf_push_u8(struct net_buf *buf, u8_t value);
void net_buf_push_le16(struct net_buf *buf, u16_t value);
```

Pull Remove data from the beginning of the buffer. Modifies both the data length value as well as the data pointer. Some examples of APIs for pulling data:

```
void *net_buf_pull(struct net_buf *buf, size_t len);
u8_t net_buf_pull_u8(struct net_buf *buf);
u16_t net_buf_pull_le16(struct net_buf *buf);
u32_t net_buf_pull_le32(struct net_buf *buf);
```

The Add and Push operations are used when encoding data into the buffer, whereas Pull is used when decoding data from a buffer.

Reference Counting

Each network buffer is reference counted. The buffer is initially acquired from a free buffers pool by calling `net_buf_alloc()`, resulting in a buffer with reference count 1. The reference count can be incremented with `net_buf_ref()` or decremented with `net_buf_unref()`. When the count drops to zero the buffer is automatically placed back to the free buffers pool.

Networking with QEMU

This page describes how to set up a "virtual" networking between a (Linux) host and a Zephyr application running in a QEMU virtual machine (built for Zephyr targets like qemu_x86, qemu_cortex_m3, etc.) In this example, the echo_server sample application from Zephyr source distribution is run in QEMU. The QEMU instance is connected to Linux host using serial port and SLIP is used to transfer data between Zephyr and Linux (over a chain of virtual connections).

Prerequisites

On the Linux Host you need to fetch Zephyr net-tools project, which is located in a separate git repository:

```
$ git clone https://github.com/zephyrproject-rtos/net-tools
$ cd net-tools
$ make
```

注解: If you get error about AX_CHECK_COMPILE_FLAG, install package autoconf-archive package on Debian/Ubuntu.

Basic Setup

For the steps below, you will need at least 4 terminal windows:

- Terminal #1 is your usual Zephyr development terminal, with Zephyr environment initialized.
- Terminals #2, #3, #4 - fresh terminal windows with net-tools being the current directory ("cd net-tools")

Step 1 - Create helper socket

Before starting QEMU with network emulation, a Unix socket for the emulation should be created.

In terminal #2, type:

```
$ ./loop-socat.sh
```

Step 2 - Start TAP device routing daemon

In terminal #3, type:

```
$ sudo ./loop-slip-tap.sh
```

Step 3 - Start app in QEMU

Build and start the echo_server sample application.

In terminal #1, type:

```
$ cd samples/net/echo_server
$ make pristine && make run
```

If you see error from QEMU about unix:/tmp/slip.sock, it means you missed Step 1 above.

Step 4 - Run apps on host

Now in terminal #4, you can run various tools to communicate with the application running in QEMU.

You can start with pings:

```
$ ping 192.0.2.1
$ ping6 2001:db8::1
```

For example, using netcat ("nc") utility, connecting using UDP:

```
$ echo foobar | nc -6 -u 2001:db8::1 4242
foobar
```

```
$ echo foobar | nc -u 192.0.2.1 4242
foobar
```

If echo_server is compiled with TCP support (now enabled by default for echo_server sample, CONFIG_NET_TCP=y):

```
$ echo foobar | nc -6 -q2 2001:db8::1 4242
foobar
```

注解： You will need to Ctrl+C manually.

You can also use the telnet command to achieve the above.

Step 5 - Stop supporting daemons

When you are finished with network testing using QEMU, you should stop any daemons or helpers started in the initial steps, to avoid possible networking or routing problems such as address conflicts in local network interfaces. For example, you definitely need to stop them if you switch from testing networking with QEMU to using real hardware. For example, there was a report of an airport WiFi connection not working during travel due to an address conflict.

To stop the daemons, just press Ctrl+C in the corresponding terminal windows (you need to stop both `loop-slip-tap.sh` and `loop-socat.sh`).

Setting up NAT/masquerading to access Internet

To access Internet from a custom application running in a QEMU, NAT (masquerading) should be set up for QEMU's source address. Assuming 192.0.2.1 is used, the following command should be run as root:

```
$ iptables -t nat -A POSTROUTING -j MASQUERADE -s 192.0.2.1
```

Additionally, IPv4 forwarding should be enabled on host, and you may need to check that other firewall (iptables) rules don't interfere with masquerading.

Network connection between two QEMU VMs

Unlike VM-Host setup described above, VM-VM setup is automatic - for sample applications which support such mode such as the echo_server and echo_client samples, you will need 2 terminal windows, set up for Zephyr development.

Terminal #1:

```
$ cd samples/net/echo_server
$ make server
```

This will start QEMU, waiting for connection from a client QEMU.

Terminal #2:

```
$ cd samples/net/echo_client
$ make client
```

This will start 2nd QEMU instance, and you should see logging of data sent and received in both.

Running multiple QEMU VMs of the same sample

If you find yourself needing to run multiple instances of the same Zephyr sample application, which do not need to be able to talk to each other, the `QEMU_INSTANCE` argument is what you need.

Start socat and tunslip6 manually (avoiding loop-x.sh scripts) for as many instances as you want. Use the following as a guide, replacing MAIN or OTHER.

Terminal #1:

```
$ socat PTY,link=/tmp/slip.devMAIN UNIX-LISTEN:/tmp/slip.sockMAIN
$ $ZEPHYR_BASE/../net-tools/tunslip6 -t tapMAIN -T -s /tmp/slip.devMAIN \
    2001:db8::1/64
# Now run Zephyr
$ make run QEMU_INSTANCE=MAIN
```

Terminal #2:

```
$ socat PTY,link=/tmp/slip.devOTHER UNIX-LISTEN:/tmp/slip.sockOTHER
$ $ZEPHYR_BASE/../net-tools/tunslip6 -t tapOTHER -T -s /tmp/slip.devOTHER \
    2001:db8::1/64
$ make run QEMU_INSTANCE=OTHER
```

Power Management

Zephyr RTOS power management subsystem provides several means for a system integrator to implement power management support that can take full advantage of the power saving features of SOCs.

Terminology

SOC interface This is a general term for the components that have knowledge of the SOC and provide interfaces to the hardware features. It will abstract the SOC specific implementations to the applications and the OS.

CPU LPS (Low Power State) Refers to any one of the low power states supported by the CPU. The CPU is usually powered on while the clocks are power gated.

Active State The CPU and clocks are powered on. This is the normal operating state when the system is running.

Deep Sleep State The CPU is power gated and loses context. Most peripherals would also be power gated. RAM is selectively retained.

SOC Power State SOC Power State describes processor and device power states implemented at the SOC level. Deep Sleep State is an example of SOC Power State.

Idle Thread A system thread that runs when there are no other threads ready to run.

Power gating Power gating reduces power consumption by shutting off current to blocks of the integrated circuit that are not in use.

Overview

The interfaces and APIs provided by the power management subsystem are designed to be architecture and SOC independent. This enables power management implementations to be easily adapted to different SOCs and architectures. The kernel does not implement any power schemes of its own, giving the system integrator the flexibility of implementing custom power schemes.

The architecture and SOC independence is achieved by separating the core infrastructure and the SOC specific implementations. The SOC specific implementations are abstracted to the application and the OS using hardware abstraction layers.

The power management features are classified into the following categories.

- Tickless Idle
- System Power Management
- Device Power Management

Tickless Idle

This is the name used to identify the event-based idling mechanism of the Zephyr RTOS kernel scheduler. The kernel scheduler can run in two modes. During normal operation, when at least one thread is active, it sets up the system timer in periodic mode and runs in an interval-based scheduling mode. The interval-based mode allows it to time slice between tasks. Many times, the threads would be waiting on semaphores, timeouts or for events. When there are no threads running, it is inefficient for the kernel scheduler to run in interval-based mode. This is because, in this mode the timer would trigger an interrupt at fixed intervals causing the scheduler to be invoked at each interval. The scheduler checks if any thread is ready to run. If no thread is ready to run then it is a waste of power because of the unnecessary CPU processing. This is avoided by the kernel switching to event-based idling mode whenever there is no thread ready to run.

The kernel holds an ordered list of thread timeouts in the system. These are the amount of time each thread has requested to wait. When the last active thread goes to wait, the idle thread is scheduled. The idle thread programs the timer to one-shot mode and programs the count to the earliest timeout from the ordered thread timeout list. When the timer expires, a timer event is generated. The ISR of this event will invoke the scheduler, which would schedule the thread associated with the timeout. Before scheduling the thread, the scheduler would switch the timer again to periodic mode. This method saves power because the CPU is removed from the wait only when there is a thread ready to run or if an external event occurred.

System Power Management

This consists of the hook functions that the power management subsystem calls when the kernel enters and exits the idle state, in other words, when the kernel has nothing to schedule. This section provides a general overview of the hook functions. Refer to *Power Management APIs* for the detailed description of the APIs.

Suspend Hook function

```
int _sys_soc_suspend(s32_t ticks);
```

When the kernel is about to go idle, the power management subsystem calls the `_sys_soc_suspend()` function, notifying the SOC interface that the kernel is ready to enter the idle state.

At this point, the kernel has disabled interrupts and computed the maximum time the system can remain idle. The function passes the time that the system can remain idle. The SOC interface performs power operations that can be done in the available time. The power management operation must halt execution on a CPU or SOC low power state. Before entering the low power state, the SOC interface must setup a wake event.

The power management subsystem expects the `_sys_soc_suspend()` to return one of the following values based on the power management operations the SOC interface executed:

`SYS_PM_NOT_HANDLED`

Indicates that no power management operations were performed.

`SYS_PM_LOW_POWER_STATE`

Indicates that the CPU was put in a low power state.

`SYS_PM_DEEP_SLEEP`

Indicates that the SOC was put in a deep sleep state.

Resume Hook function

```
void _sys_soc_resume(void);
```

The power management subsystem optionally calls this hook function when exiting kernel idling if power management operations were performed in `_sys_soc_suspend()`. Any necessary recovery operations can be performed in this function before the kernel scheduler schedules another thread. Some power states may not need this notification. It can be disabled by calling `_sys_soc_pm_idle_exit_notification_disable()` from `_sys_soc_suspend()`.

Resume From Deep Sleep Hook function

```
void _sys_soc_resume_from_deep_sleep(void);
```

This function is optionally called when exiting from deep sleep if the SOC interface does not have bootloader support to handle resume from deep sleep. This function should restore context to the point where system entered the deep sleep state.

注解: Since the hook functions are called with the interrupts disabled, the SOC interface should ensure that its operations are completed quickly. Thus, the SOC interface ensures that the kernel's scheduling performance is not disrupted.

Power Schemes

When the power management subsystem notifies the SOC interface that the kernel is about to enter a system idle state, it specifies the period of time the system intends to stay idle. The SOC interface can perform various power management operations during this time. For example, put the processor or the SOC in a low power state, turn off some or all of the peripherals or power gate device clocks.

Different levels of power savings and different wake latencies characterize these power schemes. In general, operations that save more power have a higher wake latency. When making decisions, the SOC interface chooses the scheme that saves the most power. At the same time, the scheme's total execution time must fit within the idle time allotted by the power management subsystem.

The power management subsystem classifies power management schemes into two categories based on whether the CPU loses execution context during the power state transition.

- SYS_PM_LOW_POWER_STATE
- SYS_PM_DEEP_SLEEP

SYS_PM_LOW_POWER_STATE

CPU does not lose execution context. Devices also do not lose power while entering power states in this category. The wake latencies of power states in this category are relatively low.

SYS_PM_DEEP_SLEEP

CPU is power gated and loses execution context. Execution will resume at OS startup code or at a resume point determined by a bootloader that supports deep sleep resume. Depending on the SOC's implementation of the power saving feature, it may turn off power to most devices. RAM may be retained by some implementations, while others may remove power from RAM saving considerable power. Power states in this category save more power than *SYS_PM_LOW_POWER_STATE* and would have higher wake latencies.

Device Power Management Infrastructure

The device power management infrastructure consists of interfaces to the Zephyr RTOS device model. These APIs send control commands to the device driver to update its power state or to get its current power state. Refer to *Power Management APIs* for detailed descriptions of the APIs.

Zephyr RTOS supports two methods of doing device power management.

- Distributed method
- Central method

Distributed method

In this method, the application or any component that deals with devices directly and has the best knowledge of their use does the device power management. This saves power if some devices that are not in use can be turned off or put in power saving mode. This method allows saving power even when the CPU is active. The components that use the devices need to be power aware and should be able to make decisions related to managing device power. In this method, the SOC interface can enter CPU or SOC low power states quickly when `_sys_soc_suspend()` gets called. This is because it does not need to spend time doing device power management if the devices are already put in the appropriate low power state by the application or component managing the devices.

Central method

In this method device power management is mostly done inside `_sys_soc_suspend()` along with entering a CPU or SOC low power state.

If a decision to enter deep sleep is made, the implementation would enter it only after checking if the devices are not in the middle of a hardware transaction that cannot be interrupted. This method can be used in implementations where the applications and components using devices are not expected to be power aware and do not implement device power management.

This method can also be used to emulate a hardware feature supported by some SOCs which cause automatic entry to deep sleep when all devices are idle. Refer to *Busy Status Indication* to see how to indicate whether a device is busy or idle.

Device Power Management States

The Zephyr RTOS power management subsystem defines four device states. These states are classified based on the degree of device context that gets lost in those states, kind of operations done to save power, and the impact on the device behavior due to the state transition. Device context includes device registers, clocks, memory etc.

The four device power states:

`DEVICE_PM_ACTIVE_STATE`

Normal operation of the device. All device context is retained.

`DEVICE_PM_LOW_POWER_STATE`

Device context is preserved by the HW and need not be restored by the driver.

`DEVICE_PM_SUSPEND_STATE`

Most device context is lost by the hardware. Device drivers must save and restore or reinitialize any context lost by the hardware.

`DEVICE_PM_OFF_STATE`

Power has been fully removed from the device. The device context is lost when this state is entered. Need to reinitialize the device when powering it back on.

Device Power Management Operations

Zephyr RTOS power management subsystem provides a control function interface to device drivers to indicate power management operations to perform. The supported PM control commands are:

- DEVICE_PM_SET_POWER_STATE
- DEVICE_PM_GET_POWER_STATE

Each device driver defines:

- The device's supported power states.
- The device's supported transitions between power states.

- The device's necessary operations to handle the transition between power states.

The following are some examples of operations that the device driver may perform in transition between power states:

- Save/Restore device states.
- Gate/Un-gate clocks.
- Gate/Un-gate power.
- Mask/Un-mask interrupts.

Device Model with Power Management Support

Drivers initialize the devices using macros. See *Device Drivers and Device Model* for details on how these macros are used. Use the DEVICE_DEFINE macro to initialize drivers providing power management support via the PM control function. One of the macro parameters is the pointer to the device_pm_control handler function.

Default Initializer Function

```
int device_pm_control_nop(struct device *unused_device, u32_t unused_ctrl_command,␣
↪void *unused_context);
```

If the driver doesn't implement any power control operations, the driver can initialize the corresponding pointer with this default nop function. This default nop function does nothing and should be used instead of implementing a dummy function to avoid wasting code memory in the driver.

Device Power Management API

The SOC interface and application use these APIs to perform power management operations on the devices.

Get Device List

```
void device_list_get(struct device **device_list, int *device_count);
```

The Zephyr RTOS kernel internally maintains a list of all devices in the system. The SOC interface uses this API to get the device list. The SOC interface can use the list to identify the devices on which to execute power management operations.

注解: Ensure that the SOC interface does not alter the original list. Since the kernel uses the original list, it must remain unchanged.

Device Set Power State

```
int device_set_power_state(struct device *device, u32_t device_power_state);
```

Calls the `device_pm_control()` handler function implemented by the device driver with DEVICE_PM_SET_POWER_STATE command.

Device Get Power State

```
int device_get_power_state(struct device *device, u32_t * device_power_state);
```

Calls the `device_pm_control()` handler function implemented by the device driver with DEVICE_PM_GET_POWER_STATE command.

Busy Status Indication

The SOC interface executes some power policies that can turn off power to devices, causing them to lose their state. If the devices are in the middle of some hardware transaction, like writing to flash memory when the power is turned off, then such transactions would be left in an inconsistent state. This infrastructure guards such transactions by indicating to the SOC interface that the device is in the middle of a hardware transaction.

When the `_sys_soc_suspend()` is called, the SOC interface checks if any device is busy. The SOC interface can then decide to execute a power management scheme other than deep sleep or to defer power management operations until the next call of `_sys_soc_suspend()`.

An alternative to using the busy status mechanism is to use the *distributed method* of device power management. In such a method where the device power management is handled in a distributed manner rather than centrally in `_sys_soc_suspend()`, the decision to enter deep sleep can be made based on whether all devices are already turned off.

This feature can be also used to emulate a hardware feature found in some SOCs that causes the system to automatically enter deep sleep when all devices are idle. In such an usage, the busy status can be set by default and cleared as each device becomes idle. When `_sys_soc_suspend()` is called, deep sleep can be entered if no device is found to be busy.

Here are the APIs used to set, clear, and check the busy status of devices.

Indicate Busy Status API

```
void device_busy_set(struct device *busy_dev);
```

Sets a bit corresponding to the device, in a data structure maintained by the kernel, to indicate whether or not it is in the middle of a transaction.

Clear Busy Status API

```
void device_busy_clear(struct device *busy_dev);
```

Clears the bit corresponding to the device in a data structure maintained by the kernel to indicate that the device is not in the middle of a transaction.

Check Busy Status of Single Device API

```
int device_busy_check(struct device *chk_dev);
```

Checks whether a device is busy. The API returns 0 if the device is not busy.

Check Busy Status of All Devices API

```
int device_any_busy_check(void);
```

Checks if any device is busy. The API returns 0 if no device in the system is busy.

Power Management Configuration Flags

The Power Management features can be individually enabled and disabled using the following configuration flags.

`CONFIG_SYS_POWER_MANAGEMENT`

  This flag enables the power management subsystem.

`CONFIG_TICKLESS_IDLE`

  This flag enables the tickless idle power saving feature.

`CONFIG_SYS_POWER_LOW_POWER_STATE`

  The SOC interface enables this flag to use the `SYS_PM_LOW_POWER_STATE` policy.

`CONFIG_SYS_POWER_DEEP_SLEEP`

  This flag enables support for the `SYS_PM_DEEP_SLEEP` policy.

`CONFIG_DEVICE_POWER_MANAGEMENT`

  This flag is enabled if the SOC interface and the devices support device power management.

Sensor Drivers

The sensor subsystem exposes an API to uniformly access sensor devices. Common operations are: reading data and executing code when specific conditions are met.

Basic Operation

Channels

Fundamentally, a channel is a quantity that a sensor device can measure.

Sensors can have multiple channels, either to represent different axes of the same physical property (e.g. acceleration); or because they can measure different properties altogether (ambient temperature, pressure and humidity). Complex sensors cover both cases, so a single device can expose three acceleration channels and a temperature one.

It is imperative that all sensors that support a given channel express results in the same unit of measurement. The following is a list of all supported channels, along with their description and units of measurement:

警 告: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Values

Sensor devices return results as `struct sensor_value`. This representation avoids use of floating point values as they may not be supported on certain setups.

> **警 告:** doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Fetching Values

Getting a reading from a sensor requires two operations. First, an application instructs the driver to fetch a sample of all its channels. Then, individual channels may be read. In the case of channels with multiple axes, they can be read in a single operation by supplying the corresponding `_XYZ` channel type and a buffer of 3 `struct sensor_value` objects. This approach ensures consistency of channels between reads and efficiency of communication by issuing a single transaction on the underlying bus.

Below is an example illustrating the usage of the BME280 sensor, which measures ambient temperature and atmospheric pressure. Note that `sensor_sample_fetch()` is only called once, as it reads and compensates data for both channels.

```c
void main(void)
{
	struct device *dev = device_get_binding("BME280");

	printf("dev %p name %s\n", dev, dev->config->name);

	while (1) {
		struct sensor_value temp, press, humidity;

		sensor_sample_fetch(dev);
		sensor_channel_get(dev, SENSOR_CHAN_TEMP, &temp);
		sensor_channel_get(dev, SENSOR_CHAN_PRESS, &press);
		sensor_channel_get(dev, SENSOR_CHAN_HUMIDITY, &humidity);

		printf("temp: %d.%06d; press: %d.%06d; humidity: %d.%06d\n",
		      temp.val1, temp.val2, press.val1, press.val2,
		      humidity.val1, humidity.val2);

		k_sleep(1000);
	}
}
```

The example assumes that the returned values have type `struct sensor_value`, which is the case for BME280. A real application supporting multiple sensors should inspect the `type` field of the `temp` and `press` values and use the other fields of the structure accordingly.

Configuration and Attributes

Setting the communication bus and address is considered the most basic configuration for sensor devices. This setting is done at compile time, via the configuration menu. If the sensor supports interrupts, the interrupt lines and triggering parameters described below are also configured at compile time.

Alongside these communication parameters, sensor chips typically expose multiple parameters that control the accuracy and frequency of measurement. In compliance with Zephyr's design goals, most of these values are statically configured at compile time.

However, certain parameters could require runtime configuration, for example, threshold values for interrupts. These values are configured via attributes. The example in the following section showcases a sensor with an interrupt line that is triggered when the temperature crosses a threshold. The threshold is configured at runtime using an attribute.

Triggers

Triggers in Zephyr refer to the interrupt lines of the sensor chips. Many sensor chips support one or more triggers. Some examples of triggers include: new data is ready for reading, a channel value has crossed a threshold, or the device has sensed motion.

To configure a trigger, an application needs to supply a `struct sensor_trigger` and a handler function. The structure contains the trigger type and the channel on which the trigger must be configured.

Because most sensors are connected via SPI or I2C busses, it is not possible to communicate with them from the interrupt execution context. The execution of the trigger handler is deferred to a fiber, so that data fetching operations are possible. A driver can spawn its own fiber to fetch data, thus ensuring minimum latency. Alternatively, multiple sensor drivers can share a system-wide fiber. The shared fiber approach increases the latency of handling interrupts but uses less memory. You can configure which approach to follow for each driver. Most drivers can entirely disable triggers resulting in a smaller footprint.

The following example contains a trigger fired whenever temperature crosses the 26 degree Celsius threshold. It also samples the temperature every second. A real application would ideally disable periodic sampling in the interest of saving power. Since the application has direct access to the kernel config symbols, no trigger is registered when triggering was disabled by the driver's configuration.

```c
#ifdef CONFIG_MCP9808_TRIGGER
static void trigger_handler(struct device *dev, struct sensor_trigger *trig)
{
        struct sensor_value temp;

        sensor_sample_fetch(dev);
        sensor_channel_get(dev, SENSOR_CHAN_TEMP, &temp);

        printf("trigger fired, temp %d.%06d\n", temp.val1, temp.val2);
}
#endif

void main(void)
{
        struct device *dev = device_get_binding("MCP9808");

        if (dev == NULL) {
                printf("device not found.  aborting test.\n");
                return;
        }

#ifdef DEBUG
        printf("dev %p\n", dev);
        printf("dev %p name %s\n", dev, dev->config->name);
#endif

#ifdef CONFIG_MCP9808_TRIGGER
        struct sensor_value val;
        struct sensor_trigger trig;

        val.val1 = 26;
        val.val2 = 0;

        sensor_attr_set(dev, SENSOR_CHAN_TEMP,
                        SENSOR_ATTR_UPPER_THRESH, &val);

        trig.type = SENSOR_TRIG_THRESHOLD;
        trig.chan = SENSOR_CHAN_TEMP;

        sensor_trigger_set(dev, &trig, trigger_handler);
#endif

        while (1) {
                struct sensor_value temp;
                int rc;

                rc = sensor_sample_fetch(dev);
                if (rc != 0) {
                        printf("sensor_sample_fetch error: %d\n", rc);
                        break;
                }

                rc = sensor_channel_get(dev, SENSOR_CHAN_TEMP, &temp);
                if (rc != 0) {
                        printf("sensor_channel_get error: %d\n", rc);
                        break;
                }

                printf("temp: %d.%06d\n", temp.val1, temp.val2);

                k_sleep(2000);
        }
}
```

Shell

Overview

The Shell enables multiple subsystem to use and expose their shell interface simultaneously.

Each subsystem can support shell functionality dynamically by its Kconfig file, which enables or disables the shell usage for the subsystem.

Using shell commands

Use one of the following formats:

Specific module's commands

A shell interface exposing subsystem features is a shell module, multiple modules can be available at the same time.

`MODULE_NAME COMMAND` One of the available modules is "KERNEL", for the Kernel module. More information can be found in `SHELL_REGISTER`.

Help commands

`help` Prints the list of available modules.

`help MODULE_NAME` Prints the names of the available commands for the module.

`help MODULE_NAME COMMAND` Prints help for the module's command (the help should show function goal and required parameters).

Select module commands

`select MODULE_NAME` Use this command when using the shell only for one module. After entering this command, you will not need to enter module name in further commands. If the selected module has set a default shell prompt during its initialization, the prompt will be changed to that one. Otherwise, the prompt will be changed to the selected module's name to reflect the current module in use.

`select` Clears selected module. Restores prompt as well.

Shell configuration

There are two levels of configuration: Infrastructure level and Module level.

Infrastructure level

The option `CONFIG_CONSOLE_SHELL` enables the shell subsystem and enable the default features of the shell subsystem.

Module/Subsystem level

Each subsystem using the shell service should add a unique flag in its Kconfig file.

Example:

CONFIG_NET_SHELL=y

In the subsystem's code, the shell usage depends on this config parameter. This subsystem specific flag should also depend on `CONFIG_CONSOLE_SHELL` flag.

Configuration steps to add shell functionality to a module

1. Check that `CONFIG_CONSOLE_SHELL` is set to yes.
2. Add the subsystem unique flag to its Kconfig file.

Writing a shell module

In order to support shell in your subsystem, the application must do the following:

1. Module configuration flag: Declare a new flag in your subsystem Kconfig file. It should depend on `CONFIG_CONSOLE_SHELL` flag.
2. Module registration to shell: Add your shell identifier and register its callback functions in the shell database using `SHELL_REGISTER`.

Optionally, you can use one of the following API functions to override default behavior and settings:

- `shell_register_default_module()`
- `shell_register_prompt_handler()`

In case of a sample applications as well as test environment, user can choose to set a default module in code level. In this case, the function shell_register_default_module should be called after calling SHELL_REGISTER in application level. If the function shell_register_prompt_handler was called as well, the prompt will be changed to that one. Otherwise, the prompt will be changed to the selected module's name, in order to reflect the current module in use.

注解: Even if a default module was set in code level, it can be overwritten by "select" shell command.

You can use `shell_register_default_module()` in the following cases:

- Use this command in case of using the shell only for one module. After entering this command, no need to enter module name in further commands.
- Use this function for shell backward compatibility.

More details on those optional functions can be found in *Shell API Functions*.

Shell API Functions

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Testing

Test Framework

The Zephyr Test Framework (Ztest) provides a simple testing framework intended to be used during development. It provides basic assertion macros and a generic test structure.

The framework can be used in two ways, either as a generic framework for integration testing, or for unit testing specific modules.

Quick start - Integration testing

A simple working base is located at `samples/testing/integration`. Just copy the files to `tests/` and edit them for your needs. The test will then be automatically built and run by the sanitycheck script. If you are testing the **bar** component of **foo**, you should copy the sample folder to `tests/foo/bar`. It can then be tested with:

```
./scripts/sanitycheck -s tests/foo/bar/test
```

The sample contains the following files:

Makefile

```
BOARD ?= qemu_x86
CONF_FILE ?= prj.conf

include $(ZEPHYR_BASE)/Makefile.inc
```

sample.yaml

```
sample:
    description: TBD
    name: TBD
tests:
-   test:
        tags: my_tags
```

prj.conf

```
CONFIG_ZTEST=y
```

src/Makefile

```
obj-y = main.o

include $(ZEPHYR_BASE)/tests/Makefile.test
```

src/main.c

```
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <ztest.h>

static void assert_tests(void)
{
        zassert_true(1, "1 was false");
        zassert_false(0, "0 was true");
        zassert_is_null(NULL, "NULL was not NULL");
        zassert_not_null("foo", "\"foo\" was NULL");
        zassert_equal(1, 1, "1 was not equal to 1");
        zassert_equal_ptr(NULL, NULL, "NULL was not equal to NULL");
}

void test_main(void)
{
        ztest_test_suite(framework_tests,
                ztest_unit_test(assert_tests)
        );

        ztest_run_test_suite(framework_tests);
}
```

Quick start - Unit testing

Ztest can be used for unit testing. This means that rather than including the entire Zephyr OS for testing a single function, you can focus the testing efforts into the specific module in question. This will speed up testing since only the module will have to be compiled in, and the tested functions will be called directly.

Since you won't be including basic kernel data structures that most code depends on, you have to provide function stubs in the test. Ztest provides some helpers for mocking functions, as demonstrated below.

In a unit test, mock objects can simulate the behavior of complex real objects and are used to decide whether a test failed or passed by verifying whether an interaction with an object occurred, and if required, to assert the order of that interaction.

The `samples/testing/unit` folder contains an example for testing the net-buf api of Zephyr.

Makefile

```
INCLUDE += subsys

include $(ZEPHYR_BASE)/tests/unit/Makefile.unittest
```

sample.yaml

```
sample:
    description: TBD
    name: TBD
tests:
-   test:
        tags: buf
        type: unit
```

main.c

```
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <ztest.h>
#include <net/buf.h>

struct net_buf_pool _net_buf_pool_list[1];

unsigned int irq_lock(void)
{
        return 0;
}

void irq_unlock(unsigned int key)
{
}

#include <net/buf.c>

void k_queue_init(struct k_queue *fifo) {}
void k_queue_append_list(struct k_queue *fifo, void *head, void *tail) {}

int k_is_in_isr(void)
{
        return 0;
}

void *k_queue_get(struct k_queue *fifo, s32_t timeout)
{
        return NULL;
}

```

```c
void k_queue_append(struct k_queue *fifo, void *data)
{
}

void k_queue_prepend(struct k_queue *fifo, void *data)
{
}

#define TEST_BUF_COUNT 1
#define TEST_BUF_SIZE 74

NET_BUF_POOL_DEFINE(bufs_pool, TEST_BUF_COUNT, TEST_BUF_SIZE,
                    sizeof(int), NULL);

static void test_get_single_buffer(void)
{
        struct net_buf *buf;

        buf = net_buf_alloc(&bufs_pool, K_NO_WAIT);

        zassert_equal(buf->ref, 1, "Invalid refcount");
        zassert_equal(buf->len, 0, "Invalid length");
        zassert_equal(buf->flags, 0, "Invalid flags");
        zassert_equal_ptr(buf->frags, NULL, "Frags not NULL");
}

void test_main(void)
{
        ztest_test_suite(net_buf_test,
                ztest_unit_test(test_get_single_buffer)
        );

        ztest_run_test_suite(net_buf_test);
}
```

API reference

Running tests

警告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Assertions

These macros will instantly fail the test if the related assertion fails. When an assertion fails, it will print the current file, line and function, alongside a reason for the failure and an optional message. If the config option:*CONFIG_ZTEST_ASSERT_VERBOSE* is 0, the assertions will only print the file and line numbers, reducing the binary size of the test.

Example output for a failed macro from `zassert_equal(buf->ref, 2, "Invalid refcount")`:

```
Assertion failed at main.c:62: test_get_single_buffer: Invalid refcount (buf->ref not␣
↪equal to 2)
Aborted at unit test function
```

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Mocking

These functions allow abstracting callbacks and related functions and controlling them from specific tests. You can enable the mocking framework by setting `CONFIG_ZTEST_MOCKING` to "y" in the configuration file of the test. The amount of concurrent return values and expected parameters is limited by `CONFIG_ZTEST_PARAMETER_COUNT`.

Here is an example for configuring the function `expect_two_parameters` to expect the values `a=2` and `b=3`, and telling `returns_int` to return `5`:

```c
#include <ztest.h>

static void expect_two_parameters(int a, int b)
{
        ztest_check_expected_value(a);
        ztest_check_expected_value(b);
}

static void parameter_tests(void)
{
        ztest_expect_value(expect_two_parameters, a, 2);
        ztest_expect_value(expect_two_parameters, b, 3);
        expect_two_parameters(2, 3);
}

static int returns_int(void)
{
        return ztest_get_return_value();
}

static void return_value_tests(void)
{
        ztest_returns_value(returns_int, 5);
        zassert_equal(returns_int(), 5, NULL);
}

void test_main(void)
{
        ztest_test_suite(mock_framework_tests,
                ztest_unit_test(parameter_test),
                ztest_unit_test(return_value_test)
        );

        ztest_run_test_suite(mock_framework_tests);
}
```

警 告: doxygengroup: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/zephyr-doc/checkouts/v1.9.0/doc/doxygen/xml/index.xml

Zephyr Sanity Tests

This script scans for the set of unit test applications in the git repository and attempts to execute them. By default, it tries to build each test case boards set to be default in the board definition file.

The default options will build the majority of the tests on a defined set of boards and will run in emulated environments (QEMU) if available for the architecture or configuration being tested.

In general, the sanitycheck is used to verify that local changes did not break anything in the tree and would run basic kernel tests inside QEMU. Sanitycheck does guarantee that everything would work in the final environment and has limited coverage (test execution), however, the script builds samples and tests for different boards using different configurations and helps keeping the code buildable at all times.

To run the script in the local tree, follow the steps below:

```
$ source zephyr-env.sh
$ ./scripts/sanitycheck
```

If you have a system with a large number of cores, you can try and build/run all possible tests using the following options:

```
$ ./scripts/sanitycheck --all --enable-slow
```

This will build for all available boards and would run all possible tests in a simulated environment if applicable.

The sanitycheck script accepts the following optional arguments:

-h, --help show this help message and exit

-p PLATFORM, --platform PLATFORM Platform filter for testing. This option may be used multiple times. Testcases will only be built/run on the platforms specified. If this option is not used, then platforms marked as default in the platform metadata file will be chosen to build and test.

-L N, --platform-limit N Controls what platforms are tested if –platform or –all are not used. For each architecture specified by –arch (defaults to all of them), choose the first N platforms to test in the arch-specific .yaml file 'platforms' list. Defaults to 1.

-a ARCH, --arch ARCH Arch filter for testing. Takes precedence over –platform. If unspecified, test all arches. Multiple invocations are treated as a logical 'or' relationship

-t TAG, --tag TAG Specify tags to restrict which tests to run by tag value. Default is to not do any tag filtering. Multiple invocations are treated as a logical 'or' relationship

-e EXCLUDE_TAG, --exclude-tag EXCLUDE_TAG Specify tags of tests that should not run. Default is to run all tests with all tags.

-f, --only-failed Run only those tests that failed the previous sanity check invocation.

-c CONFIG, --config CONFIG Specify platform configuration values filtering. This can be specified two ways: <config>=<value> or just <config>. The defconfig for all platforms will be checked. For the <config>=<value>

case, only match defconfig that have that value defined. For the <config> case, match defconfig that have that value assigned to any value. Prepend a '!' to invert the match.

-s TEST, --test TEST Run only the specified test cases. These are named by <path to test project relative to –testcase- root>/<testcase.yaml section name>

-l, --all Build/test on all platforms. Any –platform arguments ignored.

-o TESTCASE_REPORT, --testcase-report TESTCASE_REPORT Output a CSV spreadsheet containing results of the test run

-d DISCARD_REPORT, --discard-report DISCARD_REPORT Output a CSV spreadsheet showing tests that were skipped and why

--compare-report COMPARE_REPORT Use this report file for size comparison

--ccache Enable the use of ccache when building

-B SUBSET, --subset SUBSET Only run a subset of the tests, 1/4 for running the first 25%, 3/5 means run the 3rd fifth of the total. This option is useful when running a large number of tests on different hosts to speed up execution time.

-y, --dry-run Create the filtered list of test cases, but don't actually run them. Useful if you're just interested in –discard-report

-r, --release Update the benchmark database with the results of this test run. Intended to be run by CI when tagging an official release. This database is used as a basis for comparison when looking for deltas in metrics such as footprint

-w, --warnings-as-errors Treat warning conditions as errors

-v, --verbose Emit debugging information, call multiple times to increase verbosity

-i, --inline-logs Upon test failure, print relevant log data to stdout instead of just a path to it

--log-file FILENAME log also to file

-m, --last-metrics Instead of comparing metrics from the last –release, compare with the results of the previous sanity check invocation

-u, --no-update do not update the results of the last run of the sanity checks

-b, --build-only Only build the code, do not execute any of it in QEMU

-j JOBS, --jobs JOBS Number of cores to use when building, defaults to number of CPUs * 2

-H FOOTPRINT_THRESHOLD, --footprint-threshold FOOTPRINT_THRESHOLD When checking test case footprint sizes, warn the user if the new app size is greater then the specified percentage from the last release. Default is 5. 0 to warn on any increase on app size

-D, --all-deltas Show all footprint deltas, positive or negative. Implies –footprint-threshold=0

-O OUTDIR, --outdir OUTDIR Output directory for logs and binaries.

-n, --no-clean Do not delete the outdir before building. Will result in faster compilation since builds will be incremental

-T TESTCASE_ROOT, --testcase-root TESTCASE_ROOT Base directory to recursively search for test cases. All testcase.yaml files under here will be processed. May be called multiple times. Defaults to the 'samples' and 'tests' directories in the Zephyr tree.

-A ARCH_ROOT, --arch-root ARCH_ROOT Directory to search for arch configuration files. All .yaml files in the directory will be processed.

-z SIZE, --size SIZE Don't run sanity checks. Instead, produce a report to stdout detailing RAM/ROM sizes on the specified filenames. All other command line arguments ignored.

-S, --enable-slow Execute time-consuming test cases that have been marked as 'slow' in testcase.yaml. Normally these are only built.

-R, --enable-asserts Build all test cases with assertions enabled.

-Q, --error-on-deprecations Error on deprecation warnings.

-x EXTRA_ARGS, --extra-args EXTRA_ARGS Extra arguments to pass to the build when compiling test cases. May be called multiple times. These will be passed in after any sanitycheck-supplied options.

-C, --coverage Scan for unit test coverage with gcov + lcov.

Board Configuration

To build tests for a specific board and to execute some of the tests on real hardware or in an emulation environment such as QEMU a board configuration file is required which is generic enough to be used for other tasks that require a board inventory with details about the board and its configuration that is only available during build time otherwise.

The board metadata file is located in the board directory and is structured using the YAML markup language. The example below shows a board with a data required for best test coverage for this specific board:

```
identifier: quark_d2000_crb
name: Quark D2000 Devboard
type: mcu
arch: x86
toolchain:
  - zephyr
  - issm
ram: 8
flash: 32
testing:
        default: true
        ignore_tags:
          - net
          - bluetooth
```

identifier: A string that matches how the board is defined in the build system. This same string is used when building, for example when calling 'make':

```
# make BOARD=quark_d2000_crb
```

name: The actual name of the board as it appears in marketing material.

type: Type of the board or configuration, currently we support 2 types: mcu, qemu

arch: Architecture of the board

toolchain: The list of supported toolchains that can build this board. This should match one of the values used for 'ZEPHYR_GCC_VARIANT' when building on the command line

ram: Available RAM on the board (specified in KB). This is used to match testcase requirements. If not specified we default to 128KB.

flash: Available FLASH on the board (specified in KB). This is used to match testcase requirements. If not specified we default to 512KB.

supported: A list of features this board supports. This can be specified as a single word feature or as a variant of a feature class. For example:

```
supported:
  - pci
```

This indicates the board does support PCI. You can make a testcase build or run only on such boards, or:

```
supported:
  - netif:eth
  - sensor:bmi16
```

A testcase can both depend on 'eth' to only test ethernet or on 'netif' to run on any board with a networking interface.

testing: testing relating keywords to provide best coverage for the features of this board.

default: [True|False]: This is a default board, it will tested with the highest priority and is covered when invoking the simplified sanitycheck without any additional arguments.

ignore_tags: Do not attempt to build (and therefore run) tests marked with this list of tags.

Test Cases

Test cases are detected by the presence of a 'testcase.yaml' or a 'sample.yaml' files in the application's project directory. This file may contain one or more entries in the test section each identifying a test scenario. The name of the test case only needs to be unique for the test cases specified in that testcase meta-data.

Test cases are written suing the YAML syntax and share the same structure as samples. The following is an example test with a few options that are explained in this document.

```
tests:
-   test:
        build_only: true
        platform_whitelist: qemu_cortex_m3 qemu_x86 arduino_101
        tags: bluetooth
-   test_br:
        build_only: true
        extra_args: CONF_FILE="prj_br.conf"
        filter: not CONFIG_DEBUG
        platform_exclude: quark_d2000_crb
        platform_whitelist: qemu_cortex_m3 qemu_x86
        tags: bluetooth
```

A sample with tests will have the same structure with additional information related to the sample and what is being demonstrated:

```
sample:
  name: hello world
  description: Hello World sample, the simplest Zephyr application
  platforms: all
tests:
    - test:
        build_only: true
        tags: samples tests
        min_ram: 16
    - singlethread:
        build_only: true
        extra_args: CONF_FILE=prj_single.conf
        filter: not CONFIG_BT and not CONFIG_GPIO_SCH
        tags: samples tests
        min_ram: 16
```

The full canonical name for each test case is:

```
<path to test case>/<test entry>
```

Each test block in the testcase meta data can define the following key/value pairs:

tags: <list of tags> (required) A set of string tags for the testcase. Usually pertains to functional domains but can be anything. Command line invocations of this script can filter the set of tests to run based on tag.

skip: <True|False> (default False) skip testcase unconditionally. This can be used for broken tests.

slow: <True|False> (default False) Don't run this test case unless –enable-slow was passed in on the command line. Intended for time-consuming test cases that are only run under certain circumstances, like daily builds. These test cases are still compiled.

extra_args: <list of extra arguments> Extra arguments to pass to Make when building or running the test case.

build_only: <True|False> (default False) If true, don't try to run the test under QEMU even if the selected platform supports it.

build_on_all: <True|False> (default False) If true, attempt to build test on all available platforms.

depends_on: <list of features> A board or platform can announce what features it supports, this option will enable the test only those platforms that provide this feature.

min_ram: <integer> minimum amount of RAM needed for this test to build and run. This is compared with information provided by the board metadata.

min_flash: <integer> minimum amount of ROM needed for this test to build and run. This is compared with information provided by the board metadata.

timeout: <number of seconds> Length of time to run test in QEMU before automatically killing it. Default to 60 seconds.

arch_whitelist: <list of arches, such as x86, arm, arc> Set of architectures that this test case should only be run for.

arch_exclude: <list of arches, such as x86, arm, arc> Set of architectures that this test case should not run on.

platform_whitelist: <list of platforms> Set of platforms that this test case should only be run for.

platform_exclude: <list of platforms> Set of platforms that this test case should not run on.

extra_sections: <list of extra binary sections> When computing sizes, sanitycheck will report errors if it finds extra, unexpected sections in the Zephyr binary unless they are named here. They will not be included in the size calculation.

filter: <expression> Filter whether the testcase should be run by evaluating an expression against an environment containing the following values:

```
{ ARCH : <architecture>,
  PLATFORM : <platform>,
  <all CONFIG_* key/value pairs in the test's generated defconfig>,
  *<env>: any environment variable available
}
```

The grammar for the expression language is as follows:

expression ::= expression "and" expression

> expression "or" expression
> "not" expression
> "(" expression ")"
> symbol "==" constant
> symbol "!=" constant
> symbol "<" number
> symbol ">" number
> symbol ">=" number
> symbol "<=" number
> symbol "in" list
> symbol ":" string
> symbol

list ::= "[" list_contents "]"

list_contents ::= constant

> list_contents "," constant

constant ::= number

> string

For the case where expression ::= symbol, it evaluates to true if the symbol is defined to a non-empty string.

Operator precedence, starting from lowest to highest:

> or (left associative) and (left associative) not (right associative) all comparison operators (non-associative)

arch_whitelist, arch_exclude, platform_whitelist, platform_exclude are all syntactic sugar for these expressions. For instance

> arch_exclude = x86 arc

Is the same as:

> filter = not ARCH in ["x86", "arc"]

The ':' operator compiles the string argument as a regular expression, and then returns a true value only if the symbol's value in the environment matches. For example, if CONFIG_SOC="quark_se" then

> filter = CONFIG_SOC : "quark.*"

Would match it.

The set of test cases that actually run depends on directives in the testcase filed and options passed in on the command line. If there is any confusion, running with -v or –discard-report can help show why particular test cases were skipped.

Metrics (such as pass/fail state and binary size) for the last code release are stored in scripts/sanity_chk/sanity_last_release.csv. To update this, pass the –all –release options.

To load arguments from a file, write '+' before the file name, e.g., +file_name. File content must be one or more valid arguments separated by line break instead of white spaces.

Most everyday users will run with no arguments.

USB device stack

The USB device stack is split into three layers:

- USB Device Controller drivers (hardware dependent)
- USB device core driver (hardware independent)
- USB device class drivers (hardware independent)

USB device controller drivers

The Device Controller Driver Layer implements the low level control routines to deal directly with the hardware. All device controller drivers should implement the APIs described in file usb_dc.h. This allows the integration of new USB device controllers to be done without changing the upper layers. For now only Quark SE USB device controller (Designware IP) is supported.

Structures

```
struct usb_dc_ep_cfg_data {
   u8_t ep_addr;
   u16_t ep_mps;
   enum usb_dc_ep_type ep_type;
};
```

Structure containing the USB endpoint configuration.

- ep_addr: endpoint address, the number associated with the EP in the device configuration structure. IN EP = 0x80 | <endpoint number>. OUT EP = 0x00 | <endpoint number>
- ep_mps: Endpoint max packet size.
- ep_type: Endpoint type, may be Bulk, Interrupt or Control. Isochronous endpoints are not supported for now.

```
enum usb_dc_status_code {
   USB_DC_ERROR,
   USB_DC_RESET,
   USB_DC_CONNECTED,
   USB_DC_CONFIGURED,
   USB_DC_DISCONNECTED,
   USB_DC_SUSPEND,
   USB_DC_RESUME,
   USB_DC_UNKNOWN
};
```

Status codes reported by the registered device status callback.

- USB_DC_ERROR: USB error reported by the controller.
- USB_DC_RESET: USB reset.
- USB_DC_CONNECTED: USB connection established - hardware enumeration is completed.

- USB_DC_CONFIGURED: USB configuration done.
- USB_DC_DISCONNECTED: USB connection lost.
- USB_DC_SUSPEND: USB connection suspended by the HOST.
- USB_DC_RESUME: USB connection resumed by the HOST.
- USB_DC_UNKNOWN: Initial USB connection status.

```
enum usb_dc_ep_cb_status_code {
   USB_DC_EP_SETUP,
   USB_DC_EP_DATA_OUT,
   USB_DC_EP_DATA_IN,
};
```

Status Codes reported by the registered endpoint callback.

- USB_DC_EP_SETUP: SETUP packet received.
- USB_DC_EP_DATA_OUT: Out transaction on this endpoint. Data is available for read.
- USB_DC_EP_DATA_IN: In transaction done on this endpoint.

APIs

The following APIs are provided by the device controller driver:

`usb_dc_attach()` This function attaches USB for device connection. Upon success, the USB PLL is enabled, and the USB device is now capable of transmitting and receiving on the USB bus and of generating interrupts.

`usb_dc_detach()` This function detaches the USB device. Upon success the USB hardware PLL is powered down and USB communication is disabled.

`usb_dc_reset()` This function returns the USB device to it's initial state.

`usb_dc_set_address()` This function sets USB device address.

`usb_dc_set_status_callback()` This function sets USB device controller status callback. The registered callback is used to report changes in the status of the device controller. The status code are described by the usb_dc_status_code enumeration.

`usb_dc_ep_configure()` This function configures an endpoint. usb_dc_ep_cfg_data structure provides the endpoint configuration parameters: endpoint address, endpoint maximum packet size and endpoint type.

`usb_dc_ep_set_stall()` This function sets stall condition for the selected endpoint.

`usb_dc_ep_clear_stall()` This functions clears stall condition for the selected endpoint

`usb_dc_ep_is_stalled()` This function check if selected endpoint is stalled.

`usb_dc_ep_halt()` This function halts the selected endpoint

`usb_dc_ep_enable()` This function enables the selected endpoint. Upon success interrupts are enabled for the corresponding endpoint and the endpoint is ready for transmitting/receiving data.

`usb_dc_ep_disable()` This function disables the selected endpoint. Upon success interrupts are disabled for the corresponding endpoint and the endpoint is no longer able for transmitting/receiving data.

`usb_dc_ep_flush()` This function flushes the FIFOs for the selected endpoint.

`usb_dc_ep_write()` This function writes data to the specified endpoint. The supplied usb_ep_callback function will be called when data is transmitted out.

usb_dc_ep_read() This function is called by the Endpoint handler function, after an OUT interrupt has been received for that EP. The application must only call this function through the supplied usb_ep_callback function.

usb_dc_ep_set_callback() This function sets callback function for notification of data received and available to application or transmit done on the selected endpoint. The callback status code is described by usb_dc_ep_cb_status_code.

USB device core layer

The USB Device core layer is a hardware independent interface between USB device controller driver and USB device class drivers or customer applications. It's a port of the LPCUSB device stack. It provides the following functionalities:

- Responds to standard device requests and returns standard descriptors, essentially handling 'Chapter 9' processing, specifically the standard device requests in table 9-3 from the universal serial bus specification revision 2.0.
- Provides a programming interface to be used by USB device classes or customer applications. The APIs are described in the usb_device.h file.
- Uses the APIs provided by the device controller drivers to interact with the USB device controller.

Structures

```
typedef void (*usb_status_callback)(enum usb_dc_status_code status_code);
```

Callback function signature for the device status.

```
typedef void (*usb_ep_callback)(u8_t ep,
   enum usb_dc_ep_cb_status_code cb_status);
```

Callback function signature for the USB Endpoint.

```
typedef int (*usb_request_handler) (struct usb_setup_packet *setup,
   int *transfer_len, u8_t **payload_data);
```

Callback function signature for class specific requests. For host to device direction the 'len' and 'payload_data' contain the length of the received data and the pointer to the received data respectively. For device to host class requests, 'len' and 'payload_data' should be set by the callback function with the length and the address of the data to be transmitted buffer respectively.

```
struct usb_ep_cfg_data {
   usb_ep_callback ep_cb;
   u8_t ep_addr;
};
```

This structure contains configuration for a certain endpoint.

- ep_cb: callback function for notification of data received and available to application or transmit done, NULL if callback not required by application code.
- ep_addr: endpoint address. The number associated with the EP in the device configuration structure.

```
struct usb_interface_cfg_data {
   usb_request_handler class_handler;
   usb_request_handler custom_handler;
   u8_t *payload_data;
};
```

This structure contains USB interface configuration.

- class_handler: handler for USB Class specific Control (EP 0) communications.
- custom_handler: the custom request handler gets a first chance at handling the request before it is handed over to the 'chapter 9' request handler.
- payload_data: this data area, allocated by the application, is used to store class specific command data and must be large enough to store the largest payload associated with the largest supported Class' command set.

```
struct usb_cfg_data {
   const u8_t *usb_device_description;
   usb_status_callback cb_usb_status;
   struct usb_interface_cfg_data interface;
   u8_t num_endpoints;
   struct usb_ep_cfg_data *endpoint;
};
```

This structure contains USB device configuration.

- usb_device_description: USB device description, see http://www.beyondlogic.org/usbnutshell/usb5.shtml#DeviceDescriptors
- cb_usb_status: callback to be notified on USB connection status change
- interface: USB class handlers and storage space.
- num_endpoints: number of individual endpoints in the device configuration
- endpoint: pointer to an array of endpoint configuration structures (usb_cfg_data) of length equal to the number of EP associated with the device description, not including control endpoints.

The class drivers instantiates this with given parameters using the "usb_set_config" function.

APIs

`usb_set_config()` This function configures USB device.

`usb_deconfig()` This function returns the USB device back to it's initial state

`usb_enable()` This function enable USB for host/device connection. Upon success, the USB module is no longer clock gated in hardware, it is now capable of transmitting and receiving on the USB bus and of generating interrupts.

`usb_disable()` This function disables the USB device. Upon success, the USB module clock is gated in hardware and it is no longer capable of generating interrupts.

`usb_write()` write data to the specified endpoint. The supplied usb_ep_callback will be called when transmission is done.

`usb_read()` This function is called by the endpoint handler function after an OUT interrupt has been received for that EP. The application must only call this function through the supplied usb_ep_callback function.

USB device class drivers

To initialize the device class driver instance the USB device class driver should call usb_set_config() passing as parameter the instance's configuration structure.

For example, for CDC_ACM sample application:

```
static const u8_t cdc_acm_usb_description[] = {
   /* Device descriptor */
   USB_DEVICE_DESC_SIZE,            /* Descriptor size */
   USB_DEVICE_DESC,                 /* Descriptor type */
   LOW_BYTE(USB_1_1),
   HIGH_BYTE(USB_1_1),              /* USB version in BCD format */
   COMMUNICATION_DEVICE_CLASS,      /* Class */
   0x00,                            /* SubClass - Interface specific */
   0x00,                            /* Protocol - Interface specific */
   MAX_PACKET_SIZE_EP0,             /* Max Packet Size */
   LOW_BYTE(VENDOR_ID),
   HIGH_BYTE(VENDOR_ID),            /* Vendor Id */
   LOW_BYTE(CDC_PRODUCT_ID),
   HIGH_BYTE(CDC_PRODUCT_ID),       /* Product Id */
   LOW_BYTE(BCDDEVICE_RELNUM),
   HIGH_BYTE(BCDDEVICE_RELNUM),     /* Device Release Number */
   0x01,                            /* Index of Manufacturer String Descriptor */
   0x02,                            /* Index of Product String Descriptor */
   0x03,                            /* Index of Serial Number String Descriptor */
   CDC_NUM_CONF,                    /* Number of Possible Configuration */

   /* Configuration descriptor */
   USB_CONFIGURATION_DESC_SIZE,     /* Descriptor size */
   USB_CONFIGURATION_DESC,          /* Descriptor type */
   LOW_BYTE(CDC_CONF_SIZE),
   HIGH_BYTE(CDC_CONF_SIZE),        /* Total length in bytes of data returned */
   CDC_NUM_ITF,                     /* Number of interfaces */
   0x01,                            /* Configuration value */
   0x00,                            /* Index of the Configuration string */
   USB_CONFIGURATION_ATTRIBUTES,    /* Attributes */
   MAX_LOW_POWER,                   /* Max power consumption */

   /* Interface descriptor */
   USB_INTERFACE_DESC_SIZE,         /* Descriptor size */
   USB_INTERFACE_DESC,              /* Descriptor type */
   0x00,                            /* Interface index */
   0x00,                            /* Alternate setting */
   CDC1_NUM_EP,                     /* Number of Endpoints */
   COMMUNICATION_DEVICE_CLASS,      /* Class */
   ACM_SUBCLASS,                    /* SubClass */
   V25TER_PROTOCOL,                 /* Protocol */
   0x00,                            /* Index of the Interface String Descriptor */

   /* Header Functional Descriptor */
   USB_HFUNC_DESC_SIZE,             /* Descriptor size */
   CS_INTERFACE,                    /* Descriptor type */
   USB_HFUNC_SUBDESC,               /* Descriptor SubType */
   LOW_BYTE(USB_1_1),
   HIGH_BYTE(USB_1_1),              /* CDC Device Release Number */

   /* Call Management Functional Descriptor */
   USB_CMFUNC_DESC_SIZE,            /* Descriptor size */
   CS_INTERFACE,                    /* Descriptor type */
   USB_CMFUNC_SUBDESC,              /* Descriptor SubType */
   0x00,                            /* Capabilities */
   0x01,                            /* Data Interface */

   /* ACM Functional Descriptor */
```

```
    USB_ACMFUNC_DESC_SIZE,          /* Descriptor size */
    CS_INTERFACE,                   /* Descriptor type */
    USB_ACMFUNC_SUBDESC,            /* Descriptor SubType */
    /* Capabilities - Device supports the request combination of:
     *       Set_Line_Coding,
     *       Set_Control_Line_State,
     *       Get_Line_Coding
     *       and the notification Serial_State
     */
    0x02,

    /* Union Functional Descriptor */
    USB_UFUNC_DESC_SIZE,            /* Descriptor size */
    CS_INTERFACE,                   /* Descriptor type */
    USB_UFUNC_SUBDESC,              /* Descriptor SubType */
    0x00,                           /* Master Interface */
    0x01,                           /* Slave Interface */

    /* Endpoint INT */
    USB_ENDPOINT_DESC_SIZE,         /* Descriptor size */
    USB_ENDPOINT_DESC,              /* Descriptor type */
    CDC_ENDP_INT,                   /* Endpoint address */
    USB_DC_EP_INTERRUPT,            /* Attributes */
    LOW_BYTE(CDC_INTERRUPT_EP_MPS),
    HIGH_BYTE(CDC_INTERRUPT_EP_MPS),/* Max packet size */
    0x0A,                           /* Interval */

    /* Interface descriptor */
    USB_INTERFACE_DESC_SIZE,        /* Descriptor size */
    USB_INTERFACE_DESC,             /* Descriptor type */
    0x01,                           /* Interface index */
    0x00,                           /* Alternate setting */
    CDC2_NUM_EP,                    /* Number of Endpoints */
    COMMUNICATION_DEVICE_CLASS_DATA,/* Class */
    0x00,                           /* SubClass */
    0x00,                           /* Protocol */
    0x00,                           /* Index of the Interface String Descriptor */

    /* First Endpoint IN */
    USB_ENDPOINT_DESC_SIZE,         /* Descriptor size */
    USB_ENDPOINT_DESC,              /* Descriptor type */
    CDC_ENDP_IN,                    /* Endpoint address */
    USB_DC_EP_BULK,                 /* Attributes */
    LOW_BYTE(CDC_BULK_EP_MPS),
    HIGH_BYTE(CDC_BULK_EP_MPS),     /* Max packet size */
    0x00,                           /* Interval */

    /* Second Endpoint OUT */
    USB_ENDPOINT_DESC_SIZE,         /* Descriptor size */
    USB_ENDPOINT_DESC,              /* Descriptor type */
    CDC_ENDP_OUT,                   /* Endpoint address */
    USB_DC_EP_BULK,                 /* Attributes */
    LOW_BYTE(CDC_BULK_EP_MPS),
    HIGH_BYTE(CDC_BULK_EP_MPS),     /* Max packet size */
    0x00,                           /* Interval */

    /* String descriptor language, only one, so min size 4 bytes.
     * 0x0409 English(US) language code used
     */
    USB_STRING_DESC_SIZE,           /* Descriptor size */
    USB_STRING_DESC,                /* Descriptor type */
    0x09,
    0x04,
    /* Manufacturer String Descriptor "Intel" */
    0x0C,
    USB_STRING_DESC,
    'I', 0, 'n', 0, 't', 0, 'e', 0, 'l', 0,
    /* Product String Descriptor "CDC-ACM" */
    0x10,
    USB_STRING_DESC,
    'C', 0, 'D', 0, 'C', 0, '-', 0, 'A', 0, 'C', 0, 'M', 0,
    /* Serial Number String Descriptor "00.01" */
    0x0C,
    USB_STRING_DESC,
    '0', 0, '0', 0, '.', 0, '0', 0, '1', 0,
};
```

```
static struct usb_ep_cfg_data cdc_acm_ep_data[] = {
    {
        .ep_cb = cdc_acm_int_in,
        .ep_addr = CDC_ENDP_INT
    },
    {
        .ep_cb = cdc_acm_bulk_out,
        .ep_addr = CDC_ENDP_OUT
    },
    {
        .ep_cb = cdc_acm_bulk_in,
        .ep_addr = CDC_ENDP_IN
    }
};
```

```
static struct usb_cfg_data cdc_acm_config = {
    .usb_device_description = cdc_acm_usb_description,
    .cb_usb_status = cdc_acm_dev_status_cb,
    .interface = {
    .class_handler = cdc_acm_class_handle_req,
    .custom_handler = NULL,
    .payload_data = NULL,
    },
    .num_endpoints = CDC1_NUM_EP + CDC2_NUM_EP,
    .endpoint = cdc_acm_ep_data
};
```

```
ret = usb_set_config(&cdc_acm_config);
if (ret < 0) {
    DBG("Failed to config USB\n");
    return ret;
}
```

To enable the USB device for host/device connection:

```
ret = usb_enable(&cdc_acm_config);
if (ret < 0) {
    DBG("Failed to enable USB\n");
    return ret;
}
```

The class device requests are forwarded by the USB stack core driver to the class driver through the registered class handler. For the CDC ACM sample class driver, 'cdc_acm_class_handle_req' processes the SET_LINE_CODING, CDC_SET_CONTROL_LINE_STATE and CDC_GET_LINE_CODING class requests:

```
int cdc_acm_class_handle_req(struct usb_setup_packet *pSetup,
      s32_t *len, u8_t **data)
{
   struct cdc_acm_dev_data_t * const dev_data = DEV_DATA(cdc_acm_dev);

   switch (pSetup->bRequest) {
   case CDC_SET_LINE_CODING:
      memcpy(&dev_data->line_coding, *data, sizeof(dev_data->line_coding));
      DBG("\nCDC_SET_LINE_CODING %d %d %d %d\n",
         sys_le32_to_cpu(dev_data->line_coding.dwDTERate),
         dev_data->line_coding.bCharFormat,
         dev_data->line_coding.bParityType,
         dev_data->line_coding.bDataBits);
   break;

   case CDC_SET_CONTROL_LINE_STATE:
      dev_data->line_state = (u8_t)sys_le16_to_cpu(pSetup->wValue);
      DBG("CDC_SET_CONTROL_LINE_STATE 0x%x\n", dev_data->line_state);
         break;

   case CDC_GET_LINE_CODING:
      *data = (u8_t *)(&dev_data->line_coding);
      *len = sizeof(dev_data->line_coding);
      DBG("\nCDC_GET_LINE_CODING %d %d %d %d\n",
      sys_le32_to_cpu(dev_data->line_coding.dwDTERate),
         dev_data->line_coding.bCharFormat,
         dev_data->line_coding.bParityType,
         dev_data->line_coding.bDataBits);
         break;

   default:
      DBG("CDC ACM request 0x%x, value 0x%x\n",
         pSetup->bRequest, pSetup->wValue);
         return -EINVAL;
   }

   return 0;
}
```

The class driver should wait for the USB_DC_CONFIGURED device status code before transmitting any data.

To transmit data to the host, the class driver should call usb_write(). Upon completion the registered endpoint callback will be called. Before sending another packet the class driver should wait for the completion of the previous transfer.

When data is received, the registered endpoint callback is called. usb_read() should be used for retrieving the received data. It must always be called through the registered endpoint callback. For CDC ACM sample driver this happens via the OUT bulk endpoint handler (cdc_acm_bulk_out) mentioned in the endpoint array (cdc_acm_ep_data).

Only CDC ACM and DFU class driver examples are provided for now.

CHAPTER 2

索引和表格

- glossary
- genindex