XCORE Software Development Kit - Programming Guide

Release: 1.0.0
Publication Date: 2023/05/02

Table of Contents

1 Introduction **1**

1.1 Installation . . . 1

1.1.1 Prerequisites . . . 1

Windows . . . 1

macOS . . . 1

1.1.2 Install Steps . . . 1

Step 1. Cloning the repository . . . 2

Step 2. Install Host Applications . . . 2

Optional Step 3. Install Python and Python Requirements . . . 2

Build & Run Your First Application . . . 3

1.2 Getting Started Tutorials . . . 3

1.3 Advanced Tutorials . . . 3

1.3.1 Platform . . . 3

Architecture & Hardware Guide . . . 3

Programming Guide . . . 3

1.3.2 Bare-Metal . . . 3

Bare-metal Code Examples . . . 3

1.3.3 FreeRTOS . . . 5

FreeRTOS Code Examples . . . 5

FreeRTOS FAQs . . . 24

FreeRTOS Common Issues . . . 25

1.4 Frequently Asked Questions . . . 26

1.4.1 Build Issues . . . 26

Submodule updates . . . 26

fatfs_mkimage: not found . . . 26

xcc2clang.exe: error: no such file or directory . . . 26

1.4.2 FreeRTOS . . . 26

1.5 Copyright & Disclaimer . . . 26

1.6 Licenses . . . 27

1.6.1 XMOS . . . 27

1.6.2 Third-Party . . . 27

1 Introduction

XCORE-IOT is a collection of C/C++ software libraries designed to simplify and accelerate application development on xcore processors. It is composed of the following components:

- Peripheral IO libraries including; UART, I2C, I2S, SPI, QSPI, PDM microphones, and USB. These libraries support bare-metal and RTOS application development.
- Libraries core to DSP applications, including vectorized math. These libraries support bare-metal and RTOS application development.
- Libraries that enable multi-core FreeRTOS development on xcore including a wide array of RTOS drivers and middleware.
- Code Examples - Examples showing a variety of xcore features based on bare-metal and FreeRTOS programming.
- Documentation - Tutorials, references and API guides.

XCORE-IOT is designed to be used in conjunction with the xcore.ai Evaluation Kit (XK-EVK-XU316). Further information about the xcore.ai Evaluation Kit and xcore.ai devices is available to on www.xmos.ai.

1.1 Installation

1.1.1 Prerequisites

XTC Tools 15.2.1 or newer and CMake 3.21 or newer are required for building the example applications. If necessary, download and follow the installation instructions for those components.

Windows

A standard C/C++ compiler is required to build applications for the host PC. Windows users may use Build Tools for Visual Studio command-line interface.

Host build should also work using other Windows GNU development environments like GNU Make, MinGW or Cygwin.

libusb The DFU example requires dfu-util which requires `libusb v1.0`. `libusb` requires the installation of a driver for use on a Windows host. Driver installation should be done using a third-party installation tool like Zadig.

macOS

A standard C/C++ compiler is required to build applications for the host PC. Mac users may use the Xcode command-line tools.

1.1.2 Install Steps

Follow the following steps to install and setup XCORE-IOT:

Step 1. Cloning the repository

Clone the XCORE-IOT repository with the following command:

```
git clone --recurse-submodules git@github.com:xmos/xcore_iot.git
```

Step 2. Install Host Applications

XCORE-IOT includes utilities that run on the PC host. Run the following command to build and install these utilities:

Linux and MacOS

```
cmake -B build_host
cd build_host
sudo make install
```

This command installs the applications at `/opt/xmos/bin/` directory. You may wish to append this directory to your `PATH` variable.

```
export PATH=$PATH:/opt/xmos/bin/
```

Some host applications require that the location of `xscope_endpoint.so` be added to your `LD_LIBRARY_PATH` environment variable. This environment variable will be set if you run the host application in the XTC Tools command-line environment. For more information see Configuring the command-line environment.

Or, you may prefer to set this environment variable manually.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<path-to-XTC-Tools>/lib
```

Windows Windows users must run the x86 native tools command prompt from Visual Studio

```
cmake  -G "NMake Makefiles" -B build_host
cd build_host
nmake install
```

This command installs the applications at `<USERPROFILE>\.xmos\bin\` directory. You may wish to add this directory to your `PATH` variable.

Some host applications require that the location of `xscope_endpoint.dll` be added to your PATH. This environment variable will be set if you run the host application in the XTC Tools command-line environment. For more information see Configuring the command-line environment.

Optional Step 3. Install Python and Python Requirements

XCORE-IOT does not require installing Python, however, several example applications do utilize Python scripts. To run these scripts, Python 3 is needed, we recommend and test with Python 3.8. Install Python and install the dependencies using the following commands:

Note: You can also setup a Python virtual environment using Conda or other virtual environment tool.

Install `pip` if needed:

```
python -m pip install --upgrade pip
```

Then use `pip` to install the required modules.

```
pip install -r tools/install/requirements.txt
```

Build & Run Your First Application

Once your have installed XCORE-IOT, the next step is to *build and run your first xcore application.*

1.2 Getting Started Tutorials

Follow these 3 steps:

1. *Check the system requirements and prerequisites*
2. Install XCORE-IOT
3. Select the Getting Started guide below based on your preferred development path

- Build and run your first FreeRTOS application on xcore

Now you are ready to dive into the advanced tutorials.

1.3 Advanced Tutorials

1.3.1 Platform

Architecture & Hardware Guide

See the Architecture & Hardware Guide in the XTC Tools documentation for an introduction to the xcore platform architecture and hardware.

Programming Guide

See the Programming Guide in the XTC Tools documentation for an introduction to xcore platform programming.

1.3.2 Bare-Metal

Note: Stay tuned for a bare-metal application development Getting Started Guide.

Bare-metal Code Examples

Several example bare-metal applications are included to illustrate the fundamental tool flow and provide a starting point for basic evaluation. The examples do not seek to exhibit the full potential of the platform, and are purposely basic to provide instruction. Select an example below for more information on what the example demonstrates, how to build the example, and how to run it.

Explorer Board This example application demonstrates various capabilities of the Explorer board.

The example consists of pdm_mics to a simple audio processing pipeline which applies a variable gain. Pressing button 0 will increase the gain. Pressing button 1 will decrease the gain. The processed audio is sent to the DAC.

When button 0 is pressed, LED 0 will be lit. When button 1 is pressed, LED 1 will be lit. When the gain adjusted audio passes a frame power threshold, LED 2 will be lit. Lastly, LED 3 will blink periodically.

Additionally, the example demonstrates a simple flash, UART loopback and SPI setup.

Preparing the hardware The UART loopback section of the demo requires that a jumper cable be connected between X1D36 and X1D39. This connects the Tx pin to the Rx pin.

Deploying the firmware with Linux or macOS

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_bare_metal_explorer_board
```

Running the firmware From the build folder run:

```
make run_example_bare_metal_explorer_board
```

Debugging the firmware with xgdb From the build folder run:

```
make debug_example_bare_metal_explorer_board
```

Deploying the firmware with Windows

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_bare_metal_explorer_board
```

Running the firmware From the build folder run:

```
nmake run_example_bare_metal_explorer_board
```

Debugging the firmware with xgdb From the build folder run:

```
nmake debug_example_bare_metal_explorer_board
```

1.3.3 FreeRTOS

FreeRTOS Code Examples

Several FreeRTOS code examples are included to illustrate the fundamental tool flow and provide a starting point for new applications. The examples do not seek to exhibit the full potential of the platform, and are purposely basic to provide instruction. Select an example below for more information on what the example demonstrates, how to build the example, and how to run it.

Audio Mux This example application can be configured for onboard mic, usb audio, or i2s input. Outputs are usb audio and i2s. No DSP is performed on the audio, but the example contains an empty 2 tile pipeline skeleton for a user to populate. In this example all USB audio endpoints are sychronous.

Preparing the host On Linux and macOS the user may need to update their `udev` rules for USB configurations. Add a custom `udev` rule for USB device with VID `0x20B1` and PID `0x0021`.

Deploying the firmware with Linux or macOS

Building the firmware Run the following commands in the repository root folder.

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_audio_mux
```

Running the firmware Run the following commands in the build folder.

```
make run_example_audio_mux
```

Debugging the firmware with xgdb Run the following commands in the build folder.

```
make debug_example_audio_mux
```

Deploying the firmware with Windows

Building the firmware Run the following commands in the repository root folder.

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_audio_mux
```

Running the firmware Run the following commands in the build folder.

```
nmake run_example_audio_mux
```

Debugging the firmware with xgdb Run the following commands in the build folder.

```
nmake debug_example_audio_mux
```

Device Control This example application demonstrates how to use device control over USB and I2C.

Deploying the firmware with Linux or macOS

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_freertos_device_control
```

Running the firmware From the build folder run:

```
make run_example_freertos_device_control
```

Debugging the firmware with xgdb From the build folder run:

```
make debug_example_freertos_device_control
```

Building the host application With the firmware running in its own terminal, in a new window, run the following commands in the xcore_sdk root folder to build the host app:

```
cmake -B build_host
cd build_host
make example_freertos_device_control_host
```

Running the host application From the *xcore_sdk/build_host/examples/freertos/device_control/host* folder run:

```
./example_freertos_device_control_host -g test_cmd
```

Deploying the firmware with Windows

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_freertos_device_control
```

Running the firmware From the build folder run:

```
nmake run_example_freertos_device_control
```

Debugging the firmware with xgdb From the build folder run:

```
nmake debug_example_freertos_device_control
```

Building the host application With the firmware running in its own terminal, in a new window, run the following commands in the xcore_sdk root folder to build the host app:

```
cmake -G "NMake Makefiles" -B build_host
cd build_host
nmake example_freertos_device_control_host
```

Running the host application From the *xcore_sdk/build_host/examples/freertos/device_control/host* folder run:

```
example_freertos_device_control_host.exe -g test_cmd
```

Verifying a successful build After running the host application, you should see the following output in your console:

```
Command test_cmd sent with resid 3
Bytes received are:
50462976
```

DFU This example application demonstrates a method to add DFU to a FreeRTOS application on XCORE.

Preparing the host This application supports any host host application that is capable of USB DFU Class V1.1.

The application was verified using dfu-util.

Installation instructions for respective operating system can be found here

If on Linux the user may need to add the USB device to their udev rules. This example defaults to Vendor ID 0xCAFE with Product ID 0x4000.

If on Windows the user may need to use a tool such as Zadig to install USB drivers.

Deploying the firmware with Linux or macOS

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_freertos_dfu_v1
make example_freertos_dfu_v2
```

Preparing the hardware It is recommended to begin from an erased flash. To erase flash run:

```
make erase_all_example_freertos_dfu_v1
```

This target will use `xflash` to erase the flash of the device specified by the provided target XN file.

After building the firmware and erasing the flash, the factory image must be flashed. From the build folder run:

> make flash_app_example_freertos_dfu_v1

This target will use `xflash` to flash the application as a factory image with a boot partition size specified in `dfu.cmake`.

The board may then be power cycled and will boot up the application.

```
make create_upgrade_img_example_freertos_dfu_v2
```

This target will use `xflash` to create an upgrade image for the specified target.

Running the firmware After flashed, the factory image will run by default. The user may opt to manually run via `xrun` to see debug messages.

```
make run_example_freertos_dfu_v1
```

Debugging the firmware with xgdb From the build folder run:

```
make debug_example_freertos_dfu_v1
```

Deploying the firmware with Windows

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_freertos_dfu_v1
nmake example_freertos_dfu_v2
```

Preparing the hardware It is recommended to begin from an erased flash. To erase flash run:

```
nmake erase_all_example_freertos_dfu_v1
```

This target will use `xflash` to erase the flash of the device specified by the provided target XN file.

After building the firmware and erasing the flash, the factory image must be flashed. From the build folder run:

```
nmake flash_app_example_freertos_dfu_v1
```

This target will use `xflash` to flash the application as a factory image with a boot partition size specified in `dfu.cmake`.

The board may then be power cycled and will boot up the application.

```
nmake create_upgrade_img_example_freertos_dfu_v2
```

This target will use `xflash` to create an upgrade image for the specified target.

Running the firmware After flashed, the factory image will run by default. The user may opt to manually run via `xrun` to see debug messages.

From the build folder run:

```
nmake run_example_freertos_dfu_v1
```

Debugging the firmware with xgdb From the build folder run:

```
nmake debug_example_freertos_dfu_v1
```

Upgrading the firmware via DFU Once the application is running, a USB DFU v1.1 tool can be used to perform various actions. This example will demonstrate with dfu-util commands.

MacOS users may need to sudo the following commands.

To verify the device is running run:

```
dfu-util -l
```

The output of this command will very based on which image is running. For example_freertos_dfu_v1, the output should contain:

```
Found DFU: [cafe:4000] ver=0100, devnum=53, cfg=1, intf=0, path="3-4.1", alt=2, name=
↪"DFU dev DATAPARTITION v1", serial="123456"
Found DFU: [cafe:4000] ver=0100, devnum=53, cfg=1, intf=0, path="3-4.1", alt=1, name=
↪"DFU dev UPGRADE v1", serial="123456"
Found DFU: [cafe:4000] ver=0100, devnum=53, cfg=1, intf=0, path="3-4.1", alt=0, name=
↪"DFU dev FACTORY v1", serial="123456"
```

For example_freertos_dfu_v2, the output should contain:

```
Found DFU: [cafe:4000] ver=0100, devnum=53, cfg=1, intf=0, path="3-4.1", alt=2, name=
↪"DFU dev DATAPARTITION v2", serial="123456"
Found DFU: [cafe:4000] ver=0100, devnum=53, cfg=1, intf=0, path="3-4.1", alt=1, name=
↪"DFU dev UPGRADE v2", serial="123456"
Found DFU: [cafe:4000] ver=0100, devnum=53, cfg=1, intf=0, path="3-4.1", alt=0, name=
↪"DFU dev FACTORY v2", serial="123456"
```

The factory image can be read back by running:

```
dfu-util -e -d 4000 -a 0 -U readback_factory_img.bin
```

From the build folder, the upgrade image can be written by running:

```
dfu-util -e -d 4000 -a 1 -D example_freertos_dfu_v2_upgrade.bin
```

After updating the upgrade image it may be necessary to unplug the USB device to initiate a host re-enumeration.

The upgrade image can be read back by running:

```
dfu-util -e -d 4000 -a 1 -U readback_upgrade_img.bin
```

The data partition image can be read back by running:

```
dfu-util -e -d 4000 -a 2 -U readback_data_partition_img.bin
```

The data partition image can be written by running:

```
dfu-util -e -d 4000 -a 2 -D readback_data_partition_img.bin
```

If running the application with the run_example_freertos_dfu_v1 target, information is printed to verify behavior.

Initially, the debug prints will contain:

```
DFU Image Info
Factory:
    Addr:0x1C70
    Size:103108
    Version:0
Upgrade:
    Addr:0x1B000
    Size:0
    Version:0
Data Partition
    Addr:0x100000
First word at data partition start is: 0xFFFFFFFF
```

After writing an upgrade image the debug prints will contain:

```
DFU Image Info
Factory:
    Addr:0x1C70
    Size:103108
    Version:0
Upgrade:
    Addr:0x1B000
    Size:103108
    Version:0
Data Partition
    Addr:0x100000
First word at data partition start is: 0xFFFFFFFF
```

The debug prints include the value of the first word at the start of the data partition. Writing a text file containing "XMOS" will result in:

```
DFU Image Info
Factory:
    Addr:0x1C70
    Size:103108
    Version:0
Upgrade:
    Addr:0x1B000
    Size:103108
    Version:0
Data Partition
    Addr:0x100000
First word at data partition start is: 0x534F4D58
```

Explorer Board This example application demonstrates various capabilities of the Explorer board using FreeRTOS. The application uses I2C, I2S, SPI, UART, flash, mic array, and GPIO devices.

The FreeRTOS application creates a single stage audio pipeline which applies a variable gain. The output audio is sent to the DAC and can be listened to via the 3.5mm audio jack. The audio gain can be adjusted via GPIO, where button A is volume up and button B is volume down.

Preparing the hardware The UART loopback section of the demo requires that a jumper cable be connected between X1D36 and X1D39. This connects the Tx pin to the Rx pin.

Deploying the firmware with Linux or macOS

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_freertos_explorer_board
```

Note: The host applications are required to create the filesystem. See the XCORE-IOT installation instructions for more information.

From the build folder, create the filesystem and flash the device with the following command:

```
make flash_app_example_freertos_explorer_board
```

Running the firmware From the build folder run:

```
make run_example_freertos_explorer_board
```

Debugging the firmware with xgdb From the build folder run:

```
make debug_example_freertos_explorer_board
```

Deploying the firmware with Windows

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_freertos_explorer_board
```

Note: The host applications are required to create the filesystem. See the XCORE-IOT installation instructions for more information.

From the build folder, create the filesystem and flash the device with the following command:

```
nmake flash_app_example_freertos_explorer_board
```

Running the firmware From the build folder run:

```
nmake run_example_freertos_explorer_board
```

Debugging the firmware with xgdb From the build folder run:

```
nmake debug_example_freertos_explorer_board
```

IoT This example demonstrates how to control GPIO using MQTT.

Networking configuration In this example, we demonstrate using the Eclipse Mosquitto MQTT broker. Ensure that you have installed Mosquitto by following the instructions here: https://mosquitto.org/download/.

Note: You can modify the example code to connect to a different MQTT broker. When doing so, you will also need to modify the filesystem setup scripts before running them. This is to ensure that the correct client certificate, private key, and CA certificate are flashed. See `filesystem_support/create_fs.sh` and the instructions for setting up the filesystem below.

Next, configure the example software to connect to the proper MQTT broker. If you are running the MQTT broker on your local PC, you will need to know that PC's IP address. This can be determined a number of ways including the `ifconfig` and `ipconfig` commands in Linux/macoS and Windows operating systems, respectively.

Lastly, in `appconf.h`, set `appconfMQTT_HOSTNAME` to your MQTT broker IP address or URL:

```
#define appconfMQTT_HOSTNAME "your endpoint here"
```

Deploying the firmware with Linux or macOS

Building the firmware Run the following commands in the repo's root folder:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_freertos_iot
```

Setting up the hardware

Note: The host applications are required to create the filesystem. See the installation instructions for more information.

Before the demo can be run, the filesystem must be configured and flashed.

```
make flash_app_example_freertos_iot
```

The script will create TLS credentials and prompt you for WiFi credentials:

```
Enter the WiFi network SSID:
Enter the WiFi network password:
Enter the security (0=open, 1=WEP, 2=WPA):
Add another WiFi network? y/[n]:
Enter the MQTT server's IP/hostname:
```

Note: The MQTT server's IP/hostname is what is entered into the "Common Name" (CN) of the certification generation process. If a hostname was specified for the MQTT server, a DNS server must be available that is configured to resolve that name.

Note: Once these files have been created they will be automatically be used. If the WiFi profile or MQTT certificates/keys need to be changed, delete the corresponding components (i.e. `networks.dat` or files contained in `mqtt_broker_certs`).

Running the firmware Run the following commands in the repo's build folder:

```
make run_example_freertos_iot
```

Deploying the firmware with Windows In order to generate the certificates/keys, `OpenSSL` must be installed. There are various options for obtaining a Windows version of `OpenSSL` that include `MinGW` and `Git` installations as well as standalone installations.

Prior to running the commands below, ensure the host system has been setup to permit PowerShell execution. By default, Windows systems are set to the `Restricted` execution policy, for more information see about_Execution_Policies. Setting the policy to `RemotedSigned` should be sufficient for proper execution; this can be set from an Administrator PowerShell instance via the command:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

Note: These scripts are not digitally signed, so depending on how they were acquired/downloaded, the policy set above may still prevent execution. These files may be unblocked via PowerShell using the cmdlet `Unblock-File`.

Building the firmware Run the following commands in the repo's root folder:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_freertos_iot
```

Setting up the hardware

Note: The host applications are required to create the filesystem. See the installation instructions for more information.

Before the demo can be run, the filesystem must be configured and flashed.

```
nmake flash_app_example_freertos_iot
```

The script will create TLS credentials and prompt you for WiFi credentials:

```
Enter the WiFi network SSID:
Enter the WiFi network password:
Enter the security (0=open, 1=WEP, 2=WPA):
Add another WiFi network? y/[n]:
Enter the MQTT server's IP/hostname:
```

Note: The MQTT server's IP/hostname is what is entered into the "Common Name" (CN) of the certification generation process. If a hostname was specified for the MQTT server, a DNS server must be available that is configured to resolve that name.

Note: Once these files have been created they will be automatically be used. If the WiFi profile or MQTT certificates/keys need to be changed, delete the corresponding components (i.e. `networks.dat` or files contained in `mqtt_broker_certs`).

Running the firmware Run the following commands in the repo's build folder:

```
nmake run_example_freertos_iot
```

Testing MQTT Messages

Running the broker From the root folder of the iot example run:

```
cd mosquitto
mosquitto -v -c mosquitto.conf
```

Note: You may need to modify permissions of the cryptocredentials for mosquitto to use them.

Sending messages To turn LED 0 on, from the IoT example's *filesystem_support* subdirectory, run the following command (replacing <*MQTT_SERVER*> with the value used during certificate generation):

```
mosquitto_pub -h <MQTT_SERVER> --cafile mqtt_broker_certs/ca.crt --cert mqtt_broker_
↪certs/client.crt --key mqtt_broker_certs/client.key -d -t "explorer/ledctrl" -m '{"LED
↪": "0", "status": "on"}'
```

Supported values for "LED" are ["0", "1", "2", "3"], supported values for "status" are ["on", "off"].

L2 Cache Example The L2 cache example demonstrates how to use the software defined L2 cache.

Deploying the firmware with Linux or macOS

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_freertos_l2_cache
```

Setting up the hardware Before running the firmware, the swmem must be flashed.

```
make flash_example_freertos_l2_cache_swmem
```

Running the firmware Running with hardware.

```
make run_example_freertos_l2_cache
```

Deploying the firmware with Windows

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_freertos_l2_cache
```

Setting up the hardware Before running the firmware, the swmem must be flashed.

```
nmake flash_example_freertos_l2_cache_swmem
```

Running the firmware Running with hardware.

```
nmake run_example_freertos_l2_cache
```

Tracealyzer Example This is a simple multi-tile FreeRTOS example application illustrating how to use FreeRTOS' trace functionality with Percepio's Tracealyzer. The application illustrates a timeout issue in an example state machine which can be visualized/diagnosed with Tracealyzer. In the absence of Tracealyzer, it is possible to define another trace implementation, see FreeRTOS Trace Macros documentation for more details. For such instances, an ASCII trace implementation is available as a good starting point. This can be enabled by changing the trace mode define in the cmake file to: `USE_TRACE_MODE=TRACE_MODE_XSCOPE_ASCII`.

The application starts the FreeRTOS scheduler running on both `tile[0]` and `tile[1]`. `tile[0]` has 11 tasks, whereas `tile[1]` has only 1 task runing. Both `tile[0]` and `tile[1]` share the same logic for a "hello" task which prints a message every second. The other 10 `tile[0]` tasks serve to demonstrate an issue that can be introduced on command by the user by interacting with the buttons on the xCORE.AI Explorer board. Pressing button 1 will increase a counter up to a maximum value of 8 (while button 0 decreases this counter down to a minimum value of 0). This value affects how many `subprocess` tasks sequentially interrupt the main `process` task. The main `process` task monitors timing while in its `RUN` state. If it detects an interruption greater than or equal to a configured threshold, the `process` will momentarily transition to a `timeout` state. Pressing Button 1 four or more consecutive times should result in this timeout event. Using tools such as Tracealyzer reduces the effort involved in diagnosing multi-core/task applications.

Limitations and Known Issues The following are the currently known issues/limitations for this example:

- Tracing is performed on a single tile at a time. In this example, Tracealyzer is setup on `tile[0]`.
- Tracealyzer's snapshot mode is not supported.
- It may be necessary to disable certain trace events (see `trcConfig.h`), limit user events (i.e. via xTracePrint), or disable additional xSCOPE probes to reduce the bandwidth requirements over xSCOPE. In some cases the application may exit prematurely or drop trace data when there are exceptionally high number of trace events being recorded. This behavior may be attributed to the host PC's USB controller or general performance factors regarding the offloading of trace data from the XTAG. In such cases, xscope2psf will log a "missing events" warning.

Deploying the firmware with Linux or macOS

Building the host application Run the following commands in the root folder to build the host application using your native x86 Toolchain:

Note: Permissions may be required to install the host applications.

```
cmake -B build_host
cd build_host
make xscope2psf
make install
```

The host application, `xscope2psf`, will be installed at `/opt/xmos/bin/`, and may be moved if desired.

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_freertos_tracealyzer
```

Running the firmware From the build folder run:

```
make run_xscope_to_file_example_freertos_tracealyzer
```

Deploying the firmware with Windows

Building the host application Run the following commands in the root folder to build the host application using your native x86 Toolchain:

Note: Permissions may be required to install the host applications.

```
cmake -G "NMake Makefiles" -B build_host
cd build_host
nmake xscope2psf
nmake install
```

The host application, `xscope2psf.exe`, will be install at `%USERPROFILE%\.xmos\bin\\`, and may be moved if desired.

The instructions that follow will assume that the path of this binary has been added to your `PATH` variable or the binary has been copied to the current directory.

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_freertos_tracealyzer
```

Running the firmware

```
nmake run_xscope_to_file_example_freertos_tracealyzer
```

Verifying a successful build If the run command is successful, the console should have printed a subset of messages similar to the following:

```
Hello task running from tile 1 on core 4
Entered subprocess task (7) on core 3
Entered subprocess task (6) on core 4
Entered subprocess task (5) on core 5
Entered subprocess task (4) on core 0
Entered subprocess task (3) on core 2
Entered subprocess task (2) on core 3
Entered subprocess task (1) on core 4
Entered subprocess task (0) on core 5
Entered main process on core 0
Hello task running from tile 0 on core 2
Entered gpio task on core 1
Hello from tile 0
Hello from tile 1
Hello from tile 0
Hello from tile 1
```

The LED behavior should be as follows:

- LED 0 should turn on while Button 0 is pressed.
- LED 1 should turn on while Button 0 is pressed.
- LED 2 should toggle when the main process enters the timeout state.
- LED 3 should toggle every 500ms.

There should also be two new files generated:

- `freertos_trace.vcd`
- `freertos_trace.gtkw`

Generating a Tracealyzer PSF File With the previously generated `freertos_trace.vcd` file, from the build directory run:

```
xscope2psf -v -i freertos_trace.vcd -o freertos_trace.psf
```

The output from this command should look similar to what is shown below:

```
Opening input file ...
Opening output file ...
Processing file (Probe: 0) ...
[PSF Header]
- Format Version: 0x000A
- Options: 0x00000000
- Number of Cores: 6
- Platform: FreeRTOS
- Platform ID: 0x1AA1
- Platform Config: 1.0 Patch 0
- ISR Tail-Chaining Threshold: 0
[PSF Timestamp]
- Type: 1
- Frequency: 100000000
- Period: 100000
- Wraparounds: 0
- OS Tick Hz: 1000
- Latest Timestamp: 0
- OS Tick Count: 0
End of file reached.
```

(continues on next page)

(continued from previous page)

```
Read 282879 lines.
Processed 70714 events.
Closing files ...
Done.
```

Successful execution of this command will produce the Percepio Streaming Format (PSF) file that can be opened in Tracealyzer for inspection.

Live Trace Visualization (streaming) The previous steps illustrated a way to save a VCD trace to disk and post process it. Alternatively, this workflow can be changed to visualize the trace live. Two methods are currently available for this which will be discussed in this section.

Before continuing, Tracealyzer must be configured to use the 'File System` as the PSF streaming option. This can be configured via the following steps:

1. From the menubar in Tracealyzer, click `File` -> `Settings`
2. In the `Settings` window's left-hand menu tree, click `Project Settings` -> `PSF Streaming Settings`.
3. Under `Target Connection` select `File System`.
4. This setting will provide an option to specify a PSF file. Specify the `freertos_trace.psf` file that was previously generated.
5. Click `OK`.
6. From the menubar, click `Trace` -> `Open Live Stream Tool`.
7. This will open a new `Live Stream` window, in this window click `Connect`.

With the xrun/xgdb `example_freertos_tracealyzer.xe` and `xscope2psf` applications still running, it should now be possible to click `Start Session` and see the trace data live. Alternatively, the `Start` and `Stop` recording button in the main window's left hand menu bar may be utilized for control.

Note: The `Live Stream` window's reported `Event Rate` and `Data Rate` is useful when optimizing xscope bandwidth utilization and to determine if it is necessary to limit the frequency or types of events being recorded. A `Data Rate` versus time graph can be shown in this window via the menubar's `View` -> `Data Rate` option.

Using –xscope-file From the build folder run:

1. Start the application:

```
xrun --xscope-file freertos_trace example_freertos_tracealyzer.xe
```

2. Start the PSF file generation process:

```
xscope2psf -v -s -i freertos_trace.vcd -o freertos_trace.psf
```

As the VCD file is being written to (via `xscope`), `xscope2psf` will produce status updates on the number of lines processed and how many events have been written to the PSF file. The console output will look similar to the following:

```
Opening input file ...
Opening output file ...
Processing file (Probe: 0) ...
[PSF Header]
- Format Version: 0x000A
```

(continues on next page)

(continued from previous page)

```
- Options: 0x00000000
- Number of Cores: 6
- Platform: FreeRTOS
- Platform ID: 0x1AA1
- Platform Config: 1.0 Patch 0
- ISR Tail-Chaining Threshold: 0
[PSF Timestamp]
- Type: 1
- Frequency: 100000000
- Period: 100000
- Wraparounds: 0
- OS Tick Hz: 1000
- Latest Timestamp: 0
- OS Tick Count: 0
[STREAM STATUS]
- Read 33027 lines
- Processed 8251 events
[STREAM STATUS]
- Read 41359 lines
- Processed 10334 events
[STREAM STATUS]
- Read 47431 lines
- Processed 11852 events
[STREAM STATUS]
- Read 56771 lines
- Processed 14187 events
```

Using –xscope-port

1. Start the application:

```
xrun --xscope-port localhost:10234 example_freertos_tracealyzer.xe
```

2. Start the PSF file generation process:

```
xscope2psf -v -I localhost:10234 -o freertos_trace.psf
```

As record data is sent to `xscope2psf` it will produce status updates on the number of events written to the PSF file. The console output will look similar to the following:

```
Configuring xscope callbacks ...
Opening output file ...
Connecting to xscope (Probe: 0, Host: localhost, Port: 10234) ...
[REGISTERED] Probe ID: 0, Name: 'freertos_trace'
[PSF Header]
- Format Version: 0x000A
- Options: 0x00000000
- Number of Cores: 6
- Platform: FreeRTOS
- Platform ID: 0x1AA1
- Platform Config: 1.0 Patch 0
- ISR Tail-Chaining Threshold: 0
[PSF Timestamp]
- Type: 1
- Frequency: 100000000
- Period: 100000
```

(continues on next page)

(continued from previous page)

```
- Wraparounds: 0
- OS Tick Hz: 1000
- Latest Timestamp: 0
- OS Tick Count: 0
[STREAM STATUS]
- Processed 162 events
[STREAM STATUS]
- Processed 1585 events
[STREAM STATUS]
- Processed 3902 events
[STREAM STATUS]
- Processed 5288 events
```

In this case the target application's `printf` output will not be present in either xrun/xgdb or `xscope2psf` (while `xscope2psf` is connected). This output can be emitted on xscope2psf by providing the `--print-endpoint` option. It is recommended to use the `-p` and `-v` options separately as the current implementation of this utility does not provide any measures to ensure the target's printf log entries are not interrupted by the regular stream status reporting.

XLINK This example application demonstrates the AN01024 application note in FreeRTOS on xcore.ai.

Note: This example application required XTC Tools version 15.2.1 or newer.

Preparing the hardware This example requires 2 XCORE-AI-EXPLORER boards, and a user provided device to act as an I2C slave.

To setup the board for testing, the following connections must be made:

Table 1.1: XCORE-AI-EXPLORER to XCORE-AI-EXPLORER Connections 2 Wire

BOARD 0	BOARD 1
GND	GND
X1D65	X1D66
X1D66	X1D65
X1D64	X1D67
X1D67	X1D64
X1D63	X1D68
X1D68	X1D63
X1D62	X1D69
X1D69	X1D62
X1D61	X1D70
X1D70	X1D61

Table 1.2: XCORE-AI-EXPLORER to XCORE-AI-EXPLORER Connections 5 Wire Additions

BOARD 0	BOARD 1
X1D63	X1D68
X1D68	X1D63
X1D62	X1D69
X1D69	X1D62
X1D61	X1D70
X1D70	X1D61

Table 1.3: XCORE-AI-EXPLORER Board 0 to Host Connections

BOARD 0	Host
GND	Host GND
SCL	Host SCL
SDA	Host SDA

Building the firmware Run the following commands in the xcore_sdk root folder to build the firmware:

On Linux and Mac:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_freertos_xlink_both
```

On Windows:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_freertos_xlink_both
```

Running the firmware This application requires `example_freertos_xlink_0.xe` to be run on BOARD 0, IE, the board with a host I2C connection.

Use the following command to determine available device:

```
xrun --list-devices
```

From the build folder run:

```
xrun --id 0 example_freertos_xlink_0.xe
```

In another console, from the build folder run:

```
xrun --id 1 example_freertos_xlink_1.xe
```

BOARD 0 will send out status messages and communication details to slave address `0xC`.

The data will contain an ID, followed by a 4 byte payload. The payload is an `int32`, sent least significant byte first.

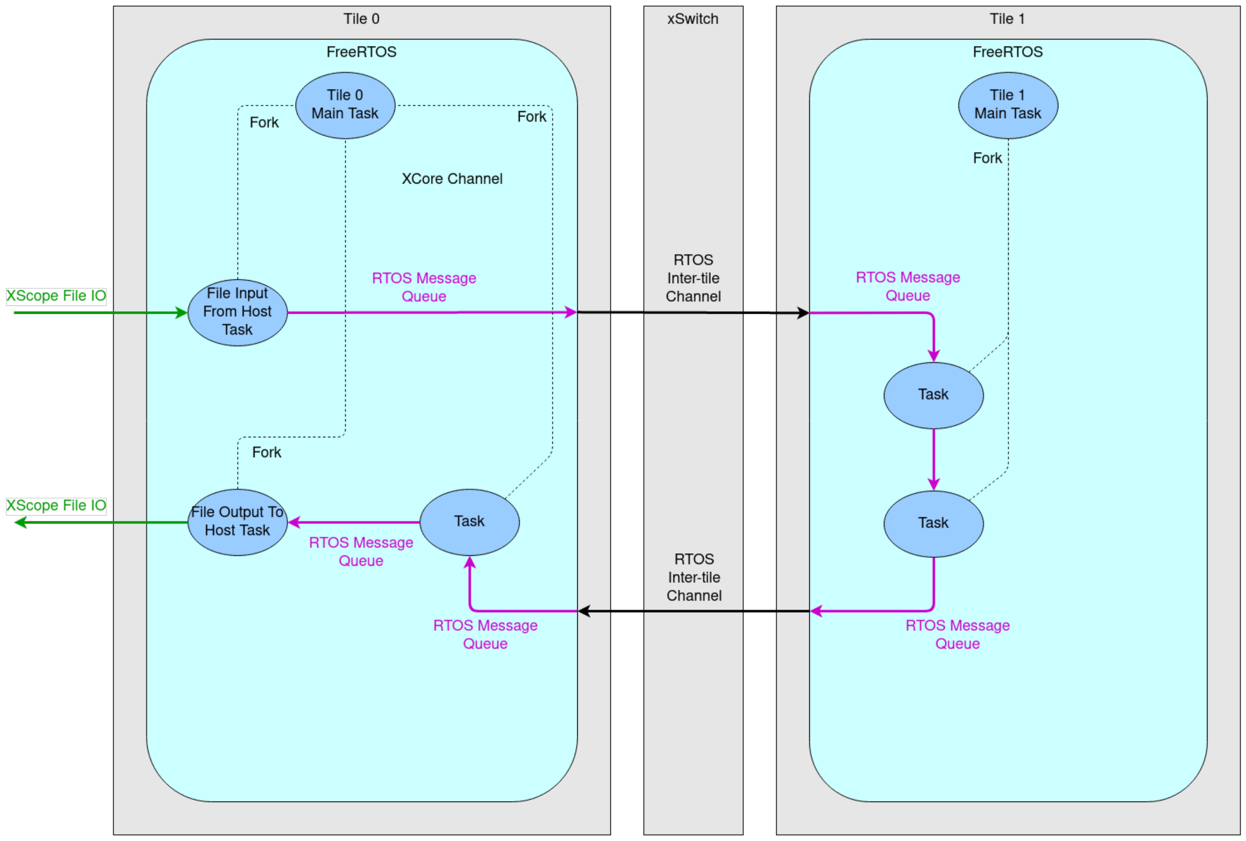
Payloads match to ID per the table below:

Table 1.4: XCORE-AI-EXPLORER to XCORE-AI-EXPLORER Connections 2 Wire

ID	Payload
0x01	RX state
0x82	received data bytes in the last second
0x83	received control tokens in the last second
0x84	timeouts in the last second

Note: Data rates are highly dependant on the electrical characteristics of the physical connection. Refer to xCONNECT Architecture for more information.

XSCOPE File I/O This FreeRTOS example application reads a WAV file from the host over an XSCOPE server, propagates the data through multiple threads across both tiles, and then writes the output to a WAV file on the host PC, also over an XSCOPE server.



The 3-stage pipeline in the example covers both XCORE tiles. Stage #1 and Stage #2 run on tile[1], while Stage #3 runs on tile[0].

Stages #1 and #2 are implemented in the functions `stage_1` and `stage_2` which can be found in the file `src\data_pipeline\src\data_pipeline_tile1.c`. In this example, both stages apply a fixed gain to the PCM audio samples. In `stage_1`, preemption is disabled with the `rtos_interrupt_mask_all()` function to insure the FreeRTOS kernel does not interrupt the task and perform a context switch during a performance critical code section. `stage_2` is a typical FreeRTOS task which can be preempted. However, this example is rather simple so, instead of leaving a context switch up to chance, the `stage_2` function periodically yields to the FreeRTOS kernel - emulating a context switch.

Both the `stage_1` and `stage_2` functions have been instrumented with a stopwatch-like timer to measure the time spent applying the fixed gain.

Stage #3 is implemented in the function `stage_3` which can be found in the file `src\data_pipeline\src\data_pipeline_tile0.c`. In this example, Stage 3 does nothing. It is provided to demonstrate a multi-tile pipeline.

The example application input file name is hard-coded to `in.wav` and the output file file name is hard-coded to `out.wav`. Running the application can be wrapped in a simple script if alternative file names are desired. Simply copy your file to `in.wav`, run the applications, then copy `out.wav` to you preferred output file name.

The example input file provided is 16 KHz, however, 48 KHz will also work. The input file sample rate must be 32 bits per sample.

This example is already configured to link with the XMOS vectorized math library. Users wishing to take advantage of the vector processing unit (VPU) on the XMOS XS3 architecture can use this example application as a starting point.

Deploying the firmware with Linux or macOS

Building the host application Run the following commands in the root folder to build the host application using your native x86 Toolchain:

Note: Permissions may be required to install the host applications.

```
cmake -B build_host
cd build_host
make xscope_host_endpoint
make install
```

The host application, `xscope_host_endpoint`, will be installed at `/opt/xmos/bin/`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

Before running the host application, you may need to add the location of `xscope_endpoint.so` to your `LD_LIBRARY_PATH` environment variable. This environment variable will be set if you run the host application in the XTC Tools command-line environment. For more information see Configuring the command-line environment.

Building the firmware Run the following commands in the root folder to build the embedded application using the XTC Toolchain:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_freertos_xscope_fileio
```

Running the firmware From the build folder run:

```
make run_example_freertos_xscope_fileio
```

In a second console, run the host xscope server:

```
./xscope_host_endpoint 12345
```

Deploying the firmware with Windows

Building the host application Run the following commands in the root folder to build the host application using your native x86 Toolchain:

Note: Permissions may be required to install the host applications.

Before building the host application, you will need to add the path to the XTC Tools to your environment.

```
set "XMOS_TOOL_PATH=<path-to-xtc-tools>"
```

Then build the host application:

```
cmake -G "NMake Makefiles" -B build_host
cd build_host
nmake xscope_host_endpoint
nmake install
```

The host application, `xscope_host_endpoint.exe`, will be install at `<USERPROFILE>\.xmos\bin`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

Before running the host application, you may need to add the location of `xscope_endpoint.dll` to your `PATH`. This environment variable will be set if you run the host application in the XTC Tools command-line environment. For more information see Configuring the command-line environment.

Building the firmware Run the following commands in the root folder to build the embedded application using the XTC Toolchain:

```
set PATH=%PATH%;<path-to-nmake>
```

To build the embedded application:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.
↪cmake
cd build
nmake example_freertos_xscope_fileio
```

Running the firmware From the build folder run:

```
nmake run_example_freertos_xscope_fileio
```

In a second console, run the host xscope server:

```
xscope_host_endpoint.exe 12345
```

FreeRTOS FAQs

1. What is the memory overhead of the FreeRTOS kernel?

   The FreeRTOS kernel requires approximately 9kB of RAM.

2. How do I determine the number of words to allocate for use as a task's stack?

   Since tasks run within FreeRTOS, the RTOS stack requirement must be known at compile time. In FreeRTOS applications on most other microcontrollers, the general practice is to create a task with a large amount of stack, use the FreeRTOS stack debug functions to determine the worst case runtime usage of stack, and then adjust the stack memory value accordingly. The problem with this method is that the stack of any given thread varies greatly based on the functions that are called within, and thus a code or compiler optimization change result in the optimal task stack usage to have to be redetermined. This issue results in many FreeRTOS applications being written in such a way that wastes memory, by providing task with way more stack than they should need. Additionally, stack overflow bugs can remain hidden for a long time and even when bugs do manifest, the source can be difficult to pinpoint.

The XTC Tools address this issue by creating a symbol that represents the maximum stack requirement of any function at compile time. By using the *RTOS_THREAD_STACK_SIZE()* macro, for the stack words argument for creating a FreeRTOS task, it is guaranteed that the optimal stack requirement is used, provided that the function does not call function pointers nor can infinitely recurse.

```
xTaskCreate((TaskFunction_t) example_task,
            "example_task",
            RTOS_THREAD_STACK_SIZE(example_task),
            NULL,
            EXAMPLE_TASK_PRIORITY,
            NULL);
```

If function pointers are used within a thread, then the application programmer must annotate the code with the appropriate function pointer group attribute. For recursive functions, the only option is to specify the stack manually. See Appendix A - Guiding Stack Size Calculation in the XTC Tools documentation for more information.

3. Can I use xcore resources like channels, timers and hw_locks?

   You are free to use channels, ports, timers, etc… in your FreeRTOS applications. However, some considerations need to be made. The RTOS kernel knows about RTOS primitives. For example, if RTOS thread A attempts to take a semaphore, the kernel is free to schedule other tasks in thread A's place while thread A is waiting for some other task to give the semaphore. The RTOS kernel does not know anything about xcore resources. For example, if RTOS thread A attempts to *recv* on a channel, the kernel is **not** free to schedule other tasks in its place while thread A is waiting for some other task to send to the other end of the channel. A developer should be aware that blocking calls on xcore resources will block a FreeRTOS thread. This may be OK as long as it is carefully considered in the application design. There are a variety of methods to handle the decoupling of xcore and RTOS resources. These can be best seen in the various RTOS drivers, which wrap the realtime IO hardware imitation layer.

FreeRTOS Common Issues

Task Stack Space One easy to make mistake in FreeRTOS, is not providing enough stack space for a created task. A vast amount of questions exist online around how to select the FreeRTOS stack size, which the most common answer being to create the task with more than enough stack, force the worst case stack condition (not always trivial), and then use the FreeRTOS debug function *uxTaskGetStackHighWaterMark()* to determine how much you can decrease the stack. This method leaves plenty of room for error and must be done during runtime, and therefore on a build by build basis. The static analysis tools provided by The XTC Tools greatly simplify this process since they calculate the exact stack required for a given function call. The macro *RTOS_THREAD_STACK_SIZE* will return the *nstackwords* symbol for a given thread plus the additional space required for the kernel ISRs. Using this macro for every task create will ensure that there is appropriate stack space for each thread, and thus no stack overflow.

```
xTaskCreate((TaskFunction_t) task_foo,
            "foo",
            RTOS_THREAD_STACK_SIZE(task_foo),
            NULL,
            configMAX_PRIORITIES-1,
            NULL);
```

1.4 Frequently Asked Questions

1.4.1 Build Issues

Submodule updates

XCORE-IOT uses submodules. If you have cloned the repository and later perform an update, it will sometimes also be necessary to update the submodules. To update all submodules, run the following command

```
git submodule update --init --recursive
```

fatfs_mkimage: not found

This issue occurs when the `fatfs_mkimage` utility cannot be found. The most common cause for these issues are an incomplete installation of the XCORE-IOT.

Ensure that the host applications setup has been completed. Verify that the `fatfs_mkimage` binary is installed to a location on PATH, or that the default application installation folder is added to PATH. See the sdk-installation guide for more information on installing the host applications.

xcc2clang.exe: error: no such file or directory

Those strange characters at the beginning of the path are known as a byte-order mark (BOM). CMake adds them to the beginning of the response files it generates during the configure step. Why does it add them? Because the MSVC compiler toolchain requires them. However, some compiler toolchains, like `gcc` and `xcc`, do not ignore the BOM. Why did CMake think the compiler toolchain was MSVC and not the XTC toolchain? Because of a bug in which certain versions of CMake and certain versions of Visual Studio do not play nice together. The good news is that this appears to have been addressed in CMake version 3.22.3.

Update to CMake version 3.22.2 or newer.

1.4.2 FreeRTOS

See the *FreeRTOS FAQs* or *FreeRTOS Common Issues*

1.5 Copyright & Disclaimer

1.6 Licenses

1.6.1 XMOS

All original source code is licensed under the XMOS License.

1.6.2 Third-Party

Additional third party code is included under the following copyrights and licenses:

Table 1.5: Third Party Module Copyrights & Licenses

Module	Copyright & License
Argtable3	Copyright (C) 1998-2001,2003-2011,2013 Stewart Heitmann, licensed under LICENSE
FatFS	Copyright (C) 2017 ChaN, licensed under a BSD-style license
FreeRTOS	Copyright (c) 2017 Amazon.com, Inc., licensed under the MIT License
HTTP Parser	Copyright (c) Joyent, Inc. and other Node contributors, licensed under the MIT license
JSMN JSON Parser	Copyright (c) 2010 Serge A. Zaitsev, licensed under the MIT license
KISS FFT	Copyright (c) 2003-2010 Mark Borgerding, licensed under the SPDX-License-Identifier BSD-3-Clause
Mbed TLS library	Copyright (c) 2006-2018 ARM Limited, licensed under the Apache License 2.0
Paho MQTT C/C++ client for Embedded platforms	Copyright (c) 2013 Eclipse Foundation, licensed under the Eclipse Public License and Eclipse Distribution License
TensorFlow	Copyright (c) 2020 The TensorFlow Authors, licensed under the Apache License
TinyUSB	Copyright (c) 2018 hathach (tinyusb.org), licensed under the MIT license