XMOS

AN03009: Placing interfaces on the XCORE

Publication Date: 2025/11/5
Document Number: XM-015415-AN v1.0.0

IN THIS DOCUMENT

1 Introduction

Unlike many microcontrollers, the XCORE architecture does not include a fixed set of built-in interface peripherals. Instead, the architecture provides a set of general-purpose I/O ports that support software modules which implement a wide variety of interface protocols. This approach provides considerable flexibility in selecting and configuring the interface functionality present on a device.

XMOS provides numerous software interface libraries, designed to use the general-purpose I/O ports, that implement common interface protocols including ADAT, Ethernet MAC, I^2C, I^2S, S/PDIF, SPI, UART, and USB. These libraries come as source code, enabling specialisation of their functionality when necessary.

This application note describes a procedure for placing interface functionality on an XCORE device. It presumes that the interface protocols needed by the design have been identified and characterised (e.g., their type, number of instances, specific roles, widths, sample rates, and clock domains). The example application includes an interface peripheral that uses an XMOS-supplied software interface library and two GPIO interfaces that use a custom handler instead of an XMOS-supplied library.

Prior to explaining interface placement, this note presents the necessary background on electrical pins, architectural ports, and packages that bring out ports to pins.

2 Ports

A port connects an XCORE processor through one or more pins to an external device. Ports can be narrow (1-bit ports) or wide (up to 32-bit ports). Each XCORE device has a set 1-, 4-, 8-, 16-, and 32- bit ports available. The number of ports of each width depends on the specific XCORE package.

Each port includes logic to support a variety of interface operations including simple I/O, clocked I/O, buffered I/O, and serialisation. AN03007: XCORE Ports provides an introduction to XCORE ports and pointers to further information.

Ports are part of a tile (e.g., tile 0 or tile 1). The pin number associated with a port will start with X*n*, where *n* is the tile number. For an interface that uses multiple ports, choose ones on the same tile, otherwise the software cannot control them.

3 Interface placement and package selection

3.1 Factors in package selection

Choosing an XCORE package for a particular design requires consideration of multiple factors including:

- The number and type of interfaces required.
- The amount of memory required.
- The amount of compute capability needed to perform the necessary computational functions within the system's time constraints.
- The number of independent threads of processing and the communication channels between them to implement the design in a timely manner.

These factors interact with one another. New information about one factor may have an impact on the others. For example, every interface peripheral added to a design uses, at least partially, a thread of processing and takes up some of the available memory and computational bandwidth. Similarly, adding a purely computational module will use more memory and may need a separate thread of processing or a separate communication channel.

XMOS offers a range of XCORE packages. While each package internally has the same number of ports per tile, the number of external pins varies. Physically smaller packages with fewer pins do not bring out each port to an external pin. Moving from a smaller package to a larger one within the same generation provides more interface capability.

In addition, the XCORE architecture has been designed for XCORE processors to network with one another, forming a cohesive system of computational nodes that function as a single device. No special hardware or software is needed to communicate between multiple XCORE processors even if they reside on different physical devices. To the programmer, the network of processors appears as a single device with more memory, more computational power, and more ports. This feature allows the system architect to easily add more resources of any needed type without incurring additional software development effort.

3.2 Interface placement within the design process

Often the set of interface peripherals needed by a design becomes apparent during development of the system architecture, and it becomes fixed at that point. For this reason, one approach when designing a system to operate on an XCORE places the interface peripherals once they are known. In this way, the system architect can choose an XCORE package to fit the needed set of peripherals. If subsequent development reveals that the system needs more resources, the decisions made during interface placement can often be retained with little or no software modification (e.g., by altering the set of ports or the tile that hosts a peripheral).

4 General considerations

4.1 Constraints on interface placement

The XCORE architecture provides considerable flexibility in placing interface peripherals, however some hardware constraints exist. Certain features available in an XCORE device overlay specific ports. When used, these features make the overlaid ports unavailable for other uses. Examples include connection to:

- Flash memory,
- LPDDR memory,
- MIPI,
- USB, and
- Other XCORE devices via xLinks.

4.2 Interface placement strategy

We recommend the following initial strategy when placing interface peripherals on an XCORE device:

1. Place constrained peripherals.
2. Place peripherals that require wide ports.
3. Place serial peripherals that use narrow ports.
4. Place GPIO peripherals using remaining narrow or wide ports.

The software on a tile can only directly access ports and pins on that tile. Therefore, when placing an interface peripheral, keep associated wires on the same tile. For debugging purposes, consider using one xLink to connect to a host PC via an xTAG. If the design uses multiple XCORE devices or may evolve to do so, consider reserving one or more xLinks per device for inter-device communication.

The strategy outlined here provides a good starting point. Depending on the number and type of interface peripherals needed and on your familiarity with XMOS technology, a different order may prove more effective.

4.2.1 Placing constrained interfaces

When booting an XCORE device, the state of three pins in a specific 4-bit port determines the boot mode. After booting, this port may be used for other purposes, however care must be taken to avoid conflicts with the boot process. Booting an XCORE device from SPI flash memory use a specific set of four 1-bit ports. Booting from QSPI flash memory uses one specific 4-bit port and two specific 1-bit ports. The datasheet for the XCORE device provides details about the ports used during the boot process.

A USB interface overlays all or a part of certain ports on any XCORE device. When using a multi-tile package, the designer has the option to choose the tile hosting the USB interface.

On an XCORE.AI device, MIPI likewise overlays certain ports, but again, the designer can choose the tile hosting it.

Connection to LPDDR memory, which is only available on the largest XCORE.AI package, overlays some ports on both tiles.

The Describe a target platform section of the XTC Tools User Guide provides some additional information about constrained interfaces.

4.2.2 Placing wide-port interfaces

Ports of different widths frequently use the same pins as one another. For instance, two 8-bit ports will connect to the same pins as one 16-bit port. In some cases, two 4-bit ports will connect to the same pins as one 8-bit port. In other cases, four 1-bit ports will connect to some of the same pins as those used by an 8-bit port. Using a narrower port makes the corresponding pins in the wider port(s) unavailable for use.

In general, an XCORE device will have more 4-bit ports than 8-bit ports, more 8-bit ports than 16-bit ports, and so on. Consequently, it makes sense to place interface peripherals that require wider ports before placing those that can use a narrower one. For instance, if a product includes an OLED display with an 8-bit parallel interface, choose an 8-bit port for it. The chosen port cannot be in use by any constraining interface, and it will need to have all eight bits brought out to pins. If the design requires several 4-bit ports but few 1-bit ports, choose an 8-bit port that shares pins with 1-bit ports. Conversely, if the design requires many 1-bit ports but few or no 4-bit ports, choose an 8-bit port that shares pins with 4-bit ports.

After allocating ports at one width, move on to the next narrower port size.

Some interface protocols that use a wide port for data require a narrow port for clock signals, selection or enablement signals, and so forth. For example, placing an Ethernet MII PHY will need two 4-bit ports (for RXD0..3 and TXD0..3), and six 1-bit ports (for RXCLK, TXCLK, RXEN, TXEN, MDC, and MDIO).

4.2.3 Placing serial interfaces

Many 1-bit ports do not share their pin with any wider port, which makes them ideal for use by serial protocols. Each clock signal must use a 1-bit port.

Some serial protocols (e.g., I^2S) allow multiple data wires to use a single clock signal. Aggregating data wires in this way saves on the number of 1-bit ports needed at the expense of using the same clock rate. For example, placing a 4-channel I^2S requires six or seven 1-bit ports: LRCLK, MCLK, BCLK, ADC0, ADC1, DAC0, and DAC1 (the inclusion of MCLK is optional in some cases).

A design that requires multiple sample rates or interface roles, (e.g., Controller or Target) will need separate clock signals.

It may be possible to place a collection of 1-bit signals together on an unused wide port. For example, if the product includes multiple chips of the same type (e.g., CODECs), a separate bit in a wide port can control the RESET line for each one.

In some cases (e.g., I^2C), XMOS supplies an interface library that can use a wide port for a serial protocol that normally uses several 1-bit ports.

4.2.4 Placing GPIO interfaces

GPIO can use either wide or narrow ports. Judicious use of bit masks and bit shifting allows multiple GPIO connections to share a single wide port. One constraint is that all signals sharing a port must use the same direction (input or output).

5 The portmap

The portmap table lists the pins and functions of an XCORE device. It acts as a useful design aid and can be extended to document the interface choices made. You can download a portmap for XCORE.AI as a spreadsheet from the XMOS website. Each row lists a pin (typically named XnDmm, where *n* is the tile number and *mm* is the pin number on that tile), which port(s) this pin belongs to, and the features that can overlay it.

As an example, the table below shows a portion of the portmap for tile[0] of the XU316-1024-FB265 and XU316-1024-TQ128 devices. An empty cell in a **Package Pin** column (the final two columns) indicates that the corresponding port has not been brought out to a pin for that specific package. In the example below, bit 0 of four-bit port 4A (pin X0D02) is brought out on the FB265 package but not on the TQ128 package.

Port				Alt Func			Options				Pkg	
1b	4B	8B	16B	32B	xLINKS	QSPI Mstr	SPI Mstr	SPI Slave	MIPI	USB	Pin Name	FB 265	TQ 128
P1A0					xlink4_rx3		MISO	SS			X0D00	D1	6
P1B0						SS	SS				X0D01	C1	2
	P4A0	P8A0	P16A0	P32A20	xlink7_rx0				RXD0	TXD0	X0D02	T8	
	P4A1	P8A1	P16A1	P32A21	xlink7_tx0				RXD1	TXD1	X0D03	U8	
	P4B0	P8A2	P16A2	P32A22		IO0			RXD2	TXD2	X0D04	A1	127
	P4B1	P8A3	P16A3	P32A23		IO1			RXD3	TXD3	X0D05	C2	128
	P4B2	P8A4	P16A4	P32A24		IO2			RXD4	TXD4	X0D06	B2	1
	P4B3	P8A5	P16A5	P32A25		IO3			RXD5	TXD5	X0D07	B1	3
	P4A2	P8A6	P16A6	P32A26	xlink7_tx1				RXD6	TXD6	X0D08	T9	
	P4A3	P8A7	P16A7	P32A27	xlink7_tx1				RXD7	TXD7	X0D09	U9	
P1C0					xlink4_rx4	SCKL	SCLK	SCLK			X0D10	D2	4
P1D0					xlink4_rx2		MOSI	MOSI			X0D11	E2	7
P1E0					xlink7_rx4				RXA	FLAG0	X0D12	P7	46
P1F0					xlink7_rx3					FLAG1	X0D13	R7	47
	P4C0	P8B0	P16A8	P32A28	xlink4_rx1					RXD0	X0D14	D4	8
	P4C1	P8B1	P16A9	P32A29	xlink4_rx0					RXD1	X0D15	D3	
	P4D0	P8B2	P16A10		xlink4_tx0					RXD2	X0D16	E4	9
	P4D1	P8B3	P16A11		xlink4_tx1					RXD3	X0D17	E3	
	P4D2	P8B4	P16A12		xlink4_tx2					RXD4	X0D18	E1	
	P4D3	P8B5	P16A13		xlink4_tx3					RXD5	X0D19	F2	
	P4C2	P8B6	P16A14	P32A30	xlink4_tx4					RXD6	X0D20	F1	
	P4C3	P8B7	P16A15	P32A31	xlink5_rx4					RXD7	X0D21	G2	

By adding columns to the portmap, the designer can document the particular interface type and signal assigned to each port. In particular, developers may find it helpful to give each port in use a meaningful name, e.g., `PORT_I2S_BCLK`, `PORT_I2S_LRCLK`, `PORT_I2S_DIN`, etc.

After finishing the first version of the portmap, making a PCB-floorplan with the selected package(s) will reveal potential routing issues such as cross-overs, long traces, etc. At this stage, the system architect has flexibility to move signals around. In addition to best practice for GPIO signals, any xLinks should be routed as documented in the datasheet.

6 Interface placement

Placing an interface peripheral on an XCORE device involves several steps:

1. Associate each interface signal with a port of appropriate width,
2. Define and initialize at file-scope a variable for each port used by the interface peripheral, and
3. Call the entry point function of the interface handler or XMOS-supplied interface library to start a separate thread of processing.

For XMOS supplied libraries you must also:

4. Pass each port variable and possibly an interface definition to the entry point function of the associated interface library,
5. Include the API header file for the library in the application's source code files as necessary, and
6. Add the name of the library to the `APP_DEPENDENT_MODULES` variable in the application's `CMakeLists.txt` file.

7 Example application

7.1 Introduction to the example

This example application demonstrates how to:

- Place a common serial communication interface peripheral on an XCORE device, and
- Place GPIO peripherals for a button and an LED.

The example has been kept intentionally simple to focus attention on the interface placement process. The application includes an I^2S interface peripheral that receives and transmits audio data. A button controls muting and unmuting of the transmitted audio data. An LED indicates the mute status.

The example uses an I^2C interface to configure an external audio codec on the XK-EVK-XU316 board. A XMOS-provided board support package performs the necessary I^2C interface placement and hardware configuration for this board. Consequently, this application note does not discuss I^2C interface placement in any depth.

Unlike many XMOS example applications, this one requires modification to allow it to build successfully.

The modifications involve:

- Defining and initialising a port variable for the button input in the `button_handler.c` source file,
- Defining and initialising a port variable for the LED output in the `led_handler.c` source file,
- Defining and initialising port variables for the I^2S interface in the `main.xc` source file,
- Calling the entry point functions for the custom button and LED handlers,
- Defining an I^2S interface for use by the application in the `main.xc` source file, and
- Calling the entry point function for the I^2S interface peripheral.

Except for the modifications listed above, the provided source files contain all of the logic needed to implement the described functionality.

The following sections illustrate the procedure described in the *Interface placement* section above.

7.2 Step 1: Associating an interface signal with a port

The XCORE architecture assumes that each XCORE tile resides within a network of XCORE tiles. Most XCORE packages include two tiles although some packages include only one tile and others include more than two. The XN file describes the network of tiles. For each tile in the network, the XN file lists the ports available on it.

To associate an interface signal to a port, the designer edits the XN file to give the port a meaningful name. Application note AN02039: Ports, Pins, and the XN file provides more information about the XN file and how to give a name to a port.

Because the example application runs on the XK-EVK-XU316 board, the XN file has already been correctly configured. For other designs, the system architect may have to modify the XN file as the first step of the interface placement procedure.

The snippets below show the relevant lines from the XK-EVK-XU316 XN file for the ports used in the example application.

Listing 1: Button and LED ports

```
<Tile Number="0" Reference="tile[0]">
  .
  .
  .
  <Port Location="XS1_PORT_4C" Name="PORT_LEDS"/>
  <Port Location="XS1_PORT_4D" Name="PORT_BUTTONS"/>
  .
  .
  .
</Tile>
```

These lines specify that the button input uses a 4-bit port 4D on tile 0 and the LED output uses a 4-bit port 4C. They also assign a name to each of these ports.

Listing 2: I^{2}S ports

```
<Tile Number="1" Reference="tile[1]">
  .
  .
  .
  <!-- Audio ports -->
  <Port Location="XS1_PORT_1D" Name="PORT_MCLK_IN"/>
  <Port Location="XS1_PORT_1C" Name="PORT_I2S_BCLK"/>
  <Port Location="XS1_PORT_1B" Name="PORT_I2S_LRCLK"/>
  <Port Location="XS1_PORT_1A" Name="PORT_I2S_DAC_DATA"/>
  <Port Location="XS1_PORT_1N" Name="PORT_I2S_ADC_DATA"/>
  .
  .
  .
</Tile>
```

These lines specify that the I^{2}S interface uses several 1-bit ports on tile 1, one for each signal wire. They also assign a name to each of these ports.

7.3 Step 2: Define a variable for each port used by the interface peripheral

Initialisation of a port variable can use the underlying name given in the XN file. Assigning a meaningful name in the XN file documents the intention and minimises changes in the source code if later development results in moving the interface to a different port or set of ports.

As shown previously, the XN file already includes meaningful names for the ports used by the example. The table below describes each port's function.

Interface	Function	Port Name	Description
I^{2}S	Bit Clock	PORT_I2S_BCLK	1-bit port for bit clock signal
	Left-Right Clock	PORT_I2S_LRCLK	1-bit port for Left-Right clock signal
	Master Clock	PORT_MCLK_IN	1-bit port for master clock signal
	Data In	PORT_ADC_DATA	1-bit port for audio data from codec
	Data Out	PORT_DAC_DATA	1-bit port for audio data to codec
GPIO	Button Input	PORT_BUTTONS	4-bit port for button input
	LED Output	PORT_LEDS	4-bit port for LED output

Each definition of a port variable must appear at file-scope. Initialisation of a port variable with a port name allows the variable to act as a reference to the corresponding port defined in the XN file.

For interface peripherals that use an XMOS-supplied interface library, the definition and initialisation of the associated port(s) typically occurs in the file containing the applica-

tion's `main()` function. Designs that include a custom protocol handler may define and initialise the associated port variables within that handler or may define and initialise them elsewhere and pass them as an argument to the handler's entry point function.

> **Note**
>
> An XCORE application built with XTC Tools version 15 requires you to place the `main()` function in a source file written in the XC language. Functionality in the tool chain uses information in this file to place threads of processing on specific tiles of the XCORE network. The tool chain also uses information in the XN file to create a `platform.h` file that includes a defined symbol for each port name.

To complete the process of placing the button interface peripheral, edit the `button_handler.c` source file to define and initialise the `p_buttons` port variable at file-scope as shown below. The lib_xcore library distributed with the XTC Tools defines the `port_t` type for defining a port in C.

Listing 3: Definition and initialisation of the button port variable

```
// Define the port for the buttons.
// Initialising p_buttons with PORT_BUTTONS connects it through the XK-EVK-XU316
//   XN file with PORT_4D on tile[0].
// The datasheet for the XU316-1024-FB265 shows PORT_4D of tile[0] connected
//   through signals X0D16, X0D17, X0D18, and X0D19 to balls E4, E3, E1, and F2.
// The schematic for the XK-EVK-XU316 board shows that two of these signals,
//   X0D16 and X0D17, connect directly to Button 0 and Button 1 respectively.
static port_t p_buttons = PORT_BUTTONS;
```

> **Tip**
>
> A comment in the source file indicates where to place the definition.

To complete the process of placing the LED interface peripheral, edit the `led_handler.c` source file to define and initialise the `p_leds` port variable at file-scope as shown below.

Listing 4: Definition and initialisation of the LED port variable

```
// Define the port for the LEDs.
// Initialising p_leds with PORT_LEDS connects it through the XK-EVK-XU316 XN
//   file with PORT_4C on tile[0].
// The datasheet for the XU316-1024-FB265 shows PORT_4C of tile[0] connected
//   through signals X0D14, X0D15, X0D20, and X0D21 to balls D4, D3, F1, and G2.
// The schematic for the XK-EVK-XU316 board shows that these signals connect
//   through a 74AVC4TD245 voltage level shifter to LEDs 0..3.
static port_t p_leds = PORT_LEDS;
```

Complete the process of placing the I^2S interface peripheral by editing the `main.xc` source file to define and initialise the port variables at file-scope as shown below.

Listing 5: Definition and initialisation of the I2S port variables

```
// Define ports used by the I2S frame-based master functionality in lib_i2s.
// The XN file for the XK-EVK-XU316 shows the association between the native
//   port names and these initialisation constants.
// The datasheet for the XU316-1024-FB265 shows how the ports connect through
//   signals to balls on the device.
// The schematic for the XK-EVK-XU316 board shows the connection between the
//   signals and other parts on the board.
on tile[1]: in port p_mclk =                                PORT_MCLK_IN;
on tile[1]: buffered out port:32 p_lrclk =                  PORT_I2S_LRCLK;
on tile[1]: out port p_bclk =                               PORT_I2S_BCLK;
on tile[1]: buffered out port:32 p_dac[NUM_I2S_LINES] =     {PORT_I2S_DAC_DATA};
on tile[1]: buffered in port:32 p_adc[NUM_I2S_LINES] =      {PORT_I2S_ADC_DATA};
```

The XC language defines `port` as a keyword. It requires the directional qualifier when defining a specific port variable, and it allows the `buffered` and width qualifications to further specify the operation of a port. The `on tile[1]:` phrase states the tile upon which the defined resource resides. The use of an array allows the definition of several 1-bit ports named `p_dac` and `p_adc`. The software can address each port in the array using a zero-based index.

7.4 Step 3: Call the entry point function of each interface handler

Having defined each port used by the custom interface handlers, everything is in place to start those handlers. Edit the `main()` function in the `main.xc` source file to call the buitton and LED entry point functions as shown below.

Listing 6: Invocation of the Button and LED custom interface handlers

```
    // Entry point for the custom button handler
    button_handler(c_button);

    // Entry point for the custom LED handler
    led_handler(c_led);
```

The declaration for the button handler appears in `button_handler.h`. The declaration for the LED handler appears in `led_handler.h`. Each declaration includes a channel end (`chanend`) parameter for communication between the handler and the code that uses it.

7.5 Step 4: Pass each port variable to the entry point function of the associated interface

Starting the I^{2}S interface peripheral involves the creation of an interface as well as calling the entry point function. Edit the `main()` function in the `main.xc` source file to define the I^{2}S interface used by the application as shown below.

Listing 7: Definition of the I2S interface

```
    // Define an interface for I2S frame callback functions
    interface i2s_frame_callback_if i_i2s;
```

The definition of `i2s_frame_callback_if` appears in the `i2s.h` file in lib_i2s.

Likewise, edit the `main()` function of the `main.xc` source file to call the I^{2}S Master entry point function as shown below.

Listing 8: Invocation of the I2S interface library

```
    // Entry point function for the I2S master interface.
    // The API appears in lib_i2s
    i2s_frame_master(
        i_i2s,          // XC-style interface
        p_dac,          // DAC data port(s)
```

(continues on next page)

(continued from previous page)

```
                NUM_I2S_LINES,  // Number of DAC I2S lines
                p_adc,          // ADC data port(s)
                NUM_I2S_LINES,  // Number of ADC I2S lines
                DATA_BITS,      // Data bits per sample
                p_bclk,         // Bit clock port
                p_lrclk,        // Left-right clock port
                p_mclk,         // Master clock port
                bclk            // Clock block for generating the Bit clock
            );
```

The declaration of `i2s_frame_master()` function appears in the `i2s.h` file in lib_i2s.

7.6 Step 5: Include the API header file for the library

The API header files are needed to use an XMOS-supplied interface library. These are already included in the example application in the `main.xc` source file as shown below.

Listing 9: Include APIs for XMOS libraries

```
#include <platform.h>
#include <xs1.h>
#include "i2c.h"
#include "i2s.h"
#include "signalling.h"
#include "xk_evk_xu316/board.h"
```

This snippet shows the inclusion of the I^2S interface library API header file `lib_i2s.h` along with other necessary XMOS tools, board support package, and standard library header files.

7.7 Step 6: Including an XMOS-supplied interface library to the build system

The XCommon-CMake build system makes it easy to add an XMOS-supplied library into an application. Every application has its own `CMakeLists.txt` file. Each application's `CMakeLists.txt` file includes the definition of `APP_DEPENDENT_MODULES`. To include an XMOS-supplied interface library in the set of modules built into an application, add the library's name to the `APP_DEPENDENT_MODULES` definition.

Appending a version number in parentheses to a library name will cause the XCommon-CMake build system to use that specific released version of the library. For best results, use the most recently released version when starting a new design.

The application's `CMakeLists.txt` file has already included the necessary XMOS-supplied libraries as shown in the snippet below.

Listing 10: `APP_DEPENDENT_MODULES` definition

```
set(APP_DEPENDENT_MODULES
    "lib_board_support(1.3.0)"
    "lib_i2s(6.0.1)"
) # APP_DEPENDENT_MODULES
```

When building, the XCommon-CMake build system will walk through the dependency tree and pick up dependent modules. Hence, the set of libraries brought into a sandbox may be larger than the set defined by the `APP_DEPENDENT_MODULES` symbol in the application's `CMakeLists.txt` file.

7.8 Building the example

This section assumes that the XMOS XTC Tools have been downloaded and installed. The accompanying `README` specifies the required version, and the XMOS web site contains installation instructions.

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the XCommon-CMake build system, which is provided with the XTC tools and is based on CMake.

Download and extract the `an03009` software ZIP package to a chosen working directory.

To configure the build, run the following commands from an XTC command prompt:

```
cd an03009
cd app_an03009
cmake -G "Unix Makefiles" -B build
```

The software package includes all required dependencies. If any dependencies are missing, this step will retrieve them automatically.

Build the application binaries using `xmake`:

```
xmake -j -C build
```

The build process will generate binary artifacts (.xe files) under the appropriate subdirectories of the `app_an03009/bin` directory — one for each supported build configuration.

For subsequent builds, the `cmake` step may be omitted. If `CMakeLists.txt` or other build files are modified, `cmake` will be re-run automatically by `xmake` as needed.

7.9 Modifying the example

Attempting to build the unmodified example will result in several errors:

- The `p_buttons` port variable has not been defined and initialised in `button_handler.c`.
- The `p_leds` port variable has not been defined and initialised in `led_handler.c`.
- None of the port variables for the I^{2}S interface have been defined and initialised in `main.xc`.
- The custom button and LED handlers have not been started in separate threads by calling them from within a `par` statement in `main.xc`.
- An I^{2}S interface, used by the application to interact with the I^{2}S interface library, has not been defined in `main.xc`.
- The I^{2}S Master function itself (i.e., the entry point function to the XMOS-supplied I^{2}S interface library) has not been started in a separate thread by called it from within a `par` statement in `main.xc`.

After making these modifications, the application should build successfully.

7.10 Running the example

From an XTC command prompt, run the following command from the `an03009/app_an03009` directory:

```
xrun ./bin/app_an03009.xe
```

Alternatively, the application can be programmed into flash memory for standalone execution:

```
xflash ./bin/app_an03009.xe
```

7.11 Testing the example

Plug an analogue audio source into the **LINE IN** jack on the XK-EVK-XU316 board. Plug a speaker or set of headphones into the **LINE OUT** jack on the XK-EVK-XU316 board. Play some audio through the XK-EVK-XU316 from any convenient source. Pressing Button 0 will toggle the mute setting for the audio played out from the **LINE OUT** jack. When muted, LED 0 will light up.

This example application uses a primitive technique to mute and unmute audio. It does so to keep the application as simple as possible. Consequently, the act of muting or un-muting introduces some noticable clicks.

8 Further reading

- AN02039: Ports, Pins, and the XN file
  https://www.xmos.com/documentation/XM-015276-AN/html/doc/rst/an02039.html
- AN03007: XCORE Ports
  https://www.xmos.com/documentation/XM-015272-AN/html/doc/rst/an03007.html
- XCORE.AI port map
  https://www.xmos.com/file/xcore_ai-package-port-map/?version=latest
- XMOS application build and dependency management system: *XCommon-CMake*
  https://www.xmos.com/file/xcommon-cmake-documentation/?version=latest
- XMOS XTC Tools Installation Guide
  https://xmos.com/xtc-install-guide
- XMOS XTC Tools User Guide
  https://www.xmos.com/view/Tools-15-Documentation

Copyright © 2025, All Rights Reserved.

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.