XMOS

AN03000: XCORE Input and Output

Publication Date: 2025/3/11
Document Number: XM-015252-AN v1.0.0

IN THIS DOCUMENT

1 Overview

Input and output (I/O) operations are fundamental to embedded systems, enabling them to interact with the physical world. The XMOS XCore processor provides a simple and efficient way to perform I/O operations through the use of a logical abstraction called a **port**.

A port connects a processor to one or more physical pins and as such defines the interface between a processor and its environment. The port logic can drive its pins high or low, or it can sample the value on its pins, optionally waiting for a particular condition.

Ports are not memory mapped; instead they are accessed using dedicated instructions that make it easy to express operations on ports. Fig. 1 illustrates these operations.

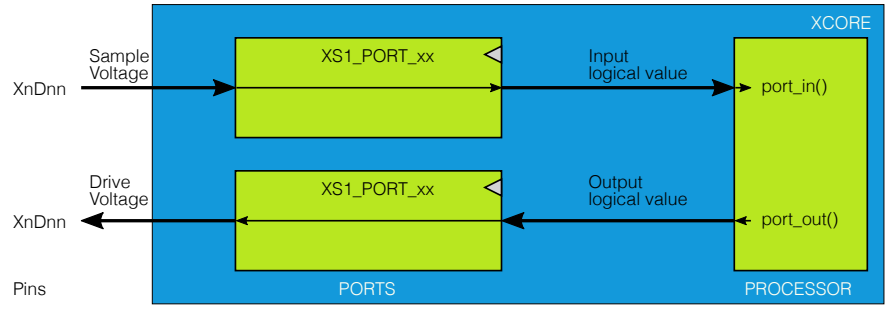

Fig. 1: Input and Output Operations

In the XCORE architecture, ports are typically referred to by a symbolic name and are labelled as `XS1_PORT_xy` where `xy` is port identifier as described in AN03007: XCORE Ports.

Data rates can be controlled using hardware timers that delay the execution of the input and output instructions for a defined period and the processor can also be made to

wait for an input from more than one port, enabling multiple I/O devices to be interfaced concurrently.

The following sections describe how to perform basic I/O operations on ports followed by more advanced operations such as conditional input and controlling I/O rates with timers. A case study is included which demonstrates how to implement a UART function using XCORE ports.

2 Data Output

The most basic form of I/O operation is to output a value to a port. A simple program that toggles a pin high and low is shown below.

```
#include <xs1.h>
#include <xcore/port.h>

port_t p = XS1_PORT_1A;

int main(void) {
    port_enable(p);
    port_out(p, 1);
    port_out(p, 0);
}
```

The declaration

```
port_t p = XS1_PORT_1A;
```

declares an output port named **p**, which refers to the 1-bit port identifier 1A. The value **XS1_PORT_1A** is defined in the header file **<xs1.h>**.

One can give the ports different names by either using a **#define** or by giving them a name in a board-description file, also known as an XN file. Names from the latter are defined in a generated header file **<platform.h>**. This allows you to use more intuitive names for ports such as **PORT_UART_TX** and **PORT_LED_A**. Further details on using XN files can be found in AN02039: Ports, Pins, and the XN file.

The statement

```
port_enable(p);
```

switches the port on. By default ports are dormant and cannot be used until they are enabled.

The statement

```
port_out(p, 1);
```

outputs the value 1 to the port **p**, causing the port to drive its corresponding pin *high*. The port continues to drive its pin high until execution of the next statement

```
port_out(p, 0);
```

which outputs the value 0 to the port, causing the port to drive its pin low. Fig. 2 shows the signals generated by this program.

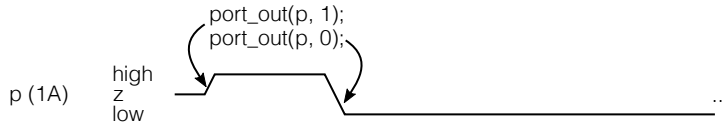

Fig. 2: Output waveform diagram

The pin is initially not driven; after the first output is executed it is driven high; and after the second output is executed it is driven low.

Note: The functionality described above applies equally to ports of any width. In general, when outputting to an *n*-bit port, the least significant *n* bits of the output value are driven on the pins and the rest are ignored.

The timings of the output are not controlled in any way in this example; they happen nanoseconds after the program executes the statement. Output timings can be made precise, either to a wall clock or to an application clock, this is discussed in AN03001: XCORE Clocked Input and Output

It is good practice to not use the same port in two variables, as each port should only be used by one thread. Passing a port to a function is allowed as normal.

3 Data Input

An XCORE port can also sample the values on the device pins, and the width of the port determines the number of pins that are sampled in a single operation, and these samples are converted to a value for further processing.

For example, the program below continuously samples the 4 pins of an input port, driving an output port high whenever the sampled value exceeds 9.

```
#include <xs1.h>
#include <xcore/port.h>

port_t  inP = XS1_PORT_4A;
port_t outP = XS1_PORT_1A;

int main(void) {
    int x;
    port_enable(inP);
    port_enable(outP);
    while (1) {
        x = port_in(inP);
        if (x > 9)
            port_out(outP, 1);
        else
            port_out(outP, 0);
    }
}
```

The declaration

```
port_t inP = XS1_PORT_4A;
```

declares an input port named `inP`, which refers to the 4-bit port identifier 4A.

The statement

```
x = port_int(inP);
```

inputs the value sampled by the port **inP** into the variable **x**. Fig. 3 shows example input stimuli and expected output for this program.

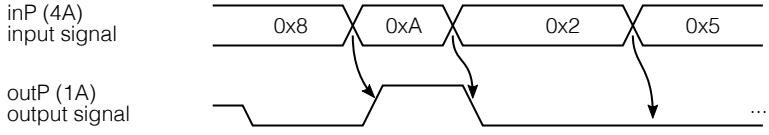

Fig. 3: Input waveform diagram

The program continuously inputs from the port **inP**: when 0x8 is sampled the output is driven low, when 0xA is sampled the output is driven high and when 0x2 is sampled the output is again driven low. Each input value may be sampled many times, depending on the relative speed of the program and the I/O.

4 Waiting for a Condition on an Input Pin

XCORE ports are however much more powerful that simple digital I/O pins.

For example an input operation can be made to wait for one of two conditions on a pin: equal to or not equal to some value. Fig. 4 shows an input signal triggering an action when it changes.

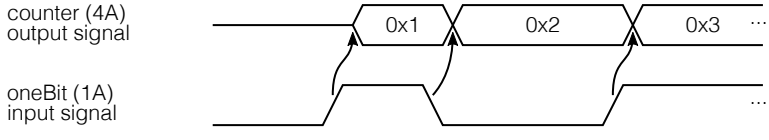

Fig. 4: Conditional input waveform diagram

This is implemented by the program below which uses a *conditional input* to count the number of transitions on its input pin.

Note: The program below has an **#include <xcore/port.h>** statement which is required to use these more advanced input functions.

```
#include <xs1.h>
#include <xcore/port.h>

port_t oneBit  = XS1_PORT_1A;
port_t counter = XS1_PORT_4A;

int main(void) {
    int x;
    int i = 0;
    port_enable(oneBit);
    port_enable(counter);
    x = port_in(oneBit);
    while (1) {
        port_set_trigger_in_not_equal(oneBit, x);
        x = port_in(oneBit);
        port_out(counter, ++i);
    }
}
```

The statement

```
port_set_trigger_in_not_equal(oneBit, x);
```

instructs the port `oneBit` to not allow further inputs until the value on its pins is not equal to `x`. The subsequent `port_in(oneBit)` will then block before sampling and providing it to the processor to store in `x`. In a 1-bit port there is only two values that are each other's complement, but on wider ports (eg, 4-bit) one can trigger on any pattern.

As another example, the sequence required to wait for an Ethernet preamble on a 4-bit port is:

```
port_set_trigger_in_equal(ethData, 0xD);
(void) port_in(ethData);
```

Note: The processor must complete an input operation from the port once a condition is met, even if the input value is not required. This is expressed in C by casting the result of `port_in` to `void`.

Using a conditional input is more power efficient than polling the port in software, because it allows the processor to idle, consuming less power, while the port remains active monitoring its pins.

5 Controlling I/O Data Rates with Timers

A timer is a special type of port used for measuring and controlling the time between events. A timer has a 32-bit counter that is continually incremented at a rate of 100MHz and whose value can be input at any time. An input on a timer can also be delayed until a time in the future.

Note: The timer functions are enabled with the `#include <xcore/hwtimer.h>` statement in a programme.

The code below uses a timer to control the rate at which a 1-bit port is toggled.

```
#include <xs1.h>
#include <xcore/port.h>
#include <xcore/hwtimer.h>

#define DELAY 50000000

port_t p = XS1_PORT_1A;

int main(void) {
    unsigned state = 1, time;
    port_enable(p);
    hwtimer_t t = hwtimer_alloc();
    time = hwtimer_get_time(t);
    while (1) {
        port_out(p, state);
        time += DELAY;
        hwtimer_set_trigger_time(t, time);
        (void) hwtimer_get_time(t);
        state = !state;
    }
    hwtimer_free(t);
}
```

The declaration

```
hwtimer_t t = hwtimer_alloc();
```

declares a timer named `t`, obtaining a timer resource from the XCORE's pool of available timers.

The statement

```
time = hwtimer_get_time(t);
```

inputs the value of **t**'s counter into the variable **time**. This variable is then incremented by the value **DELAY**, which specifies a number of counter increments. The timer has a period of 10 ns, giving a time in the future of 50,000,000 * 10 ns = 0.5 s.

Similar to ports, we can ask the timer to set a trigger time that stops the timer from providing input values until the trigger time has passed:

```
hwtimer_set_trigger_time(t, time);
(void) hwtimer_get_time(t);
```

waits until this time is reached, completing the input just afterwards. The input is required and is where the program will wait.

Fig. 5 shows the data driven by this program.

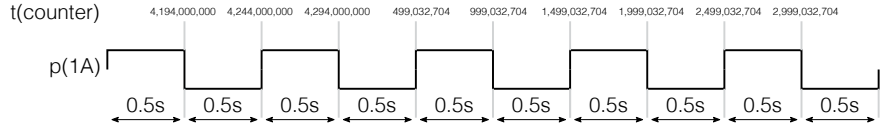

Fig. 5: Timed output waveform diagram

The function **hwtimer_set_trigger_time** treats the timer's counter as having two separate ranges, as illustrated in Fig. 6.

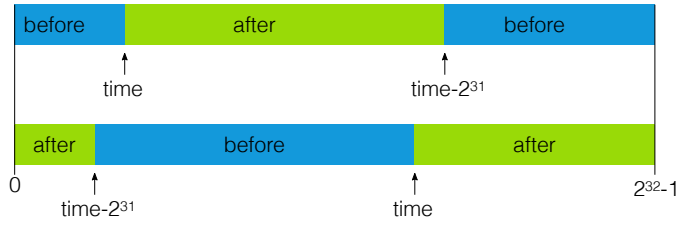

Fig. 6: Range of **hwtimer_set_trigger_time**

All values in the range (**time-** 2^{31} , … , **time-1**) are considered to come before **time**, with values in the range (**time+1** , … , **time+** 2^{32} - 1 , 0 , … , **time-** 2^{31}) considered to come afterwards. If the delay between the two input values fits in 31 bits, **hwtimer_set_trigger_time** is guaranteed to behave correctly, otherwise it may behave incorrectly due to overflow or underflow. This means that a timer can be used to measure up to a total of 2^{31}/100,000,000 = 21 s.

A subtle error may be introduced by inputting the new time instead of ignoring it with a cast to **void**, as in:

```
hwtimer_set_trigger_time(t, time);
time = hwtimer_get_time(t);
```

Even thought the the processor completes when the time is reached, the inputted value may be slightly higher, incrementing the value of **time** by a small additional amount. This means that the timing slowly *skids*.

Note that using a timer is a great way to provide signals that are timed approximately, typically without an application clock. This includes, for example, a UART. For signals

that are accompanied by an application clock, and for signals that require precise timing, one should use **clocked I/O**. Clocked I/O enables ports to input and output a signal at precisely defined times and this functionality is described in AN03001: XCORE Clocked Input and Output.

6 Case Study: UART (Part 1)

A Universal Asynchronous Receiver/Transmitter (UART) component translates data between parallel and serial forms for communication over two 1-bit wires at fixed data rates. Each bit of data is driven for the time defined by the data rate, and the receiver must sample the data during this time.

Fig. 7 shows the transmission of a single byte of data at a rate of 115200 bits/s.

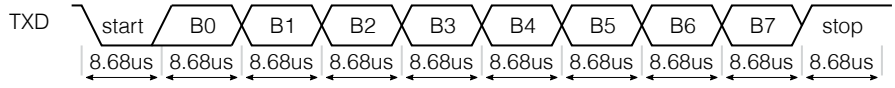

Fig. 7: UART timing diagram

The quiescent state of the wire is high. A byte is sent by first driving a *start bit* (0), followed by the eight data bits and finally a *stop bit* (1). A rate of 115200 bits/s means that each bit is driven for 1/115200 = 8.68 µs.

UARTs are often implemented with microcontrollers by using interrupts to schedule memory-mapped input and output operations. Implementing a UART with an XMOS device is easy due to its dedicated I/O instructions. The program below defines a UART transmitter that outputs data on a 1-bit.

```
void transmitter(port_t TXD) {
    unsigned byte, time;
    port_enable(TXD);
    hwtimer_t t = hwtimer_alloc();

    while (1) {
        // get next byte to transmit
        byte = get_byte();          // defined elsewhere
        time = hwtimer_get_time(t);

        // output start bit
        port_out(TXD, 0);
        time += BIT_TIME;
        hwtimer_set_trigger_time(t, time);
        (void) hwtimer_get_time(t);

        // output data bits
        for (int i=0; i<8; i++) {
            byte = port_out_shift_right(TXD, byte);
            time += BIT_TIME;
            hwtimer_set_trigger_time(t, time);
            (void) hwtimer_get_time(t);
        }

        // output stop bit
        port_out(TXD, 1);
        time += BIT_TIME;
        hwtimer_set_trigger_time(t, time);
        (void) hwtimer_get_time(t);
    }
    hwtimer_free(t);
} // transmitter
```

The transmitter outputs a byte by first outputting a start bit, followed by a conditional input on a timer that waits for the bit time to elapse; the data bits and stop bit are output in the same way.

The output statement in the `for` loop

```
byte = port_out_shift_right(TXD, byte);
```

calls the function `port_out_shift_right`, which right-shifts the value of `byte` by the port width (1 bit) after outputting the least significant port-width bits. This operation is

performed in the same instruction as the output, making it more efficient than performing the shift as a separate operation afterwards.

The function below receives a stream of bytes over a 1-bit wire connected to a port RXD.

```
void receiver(port_t RXD) {
    unsigned byte, time;
    hwtimer_t t = hwtimer_alloc();
    port_enable(RXD);

    while (1) {
        // wait for start bit
        port_set_trigger_in_equal(RXD, 0);
        (void) port_in(RXD);
        time = hwtimer_get_time(t);
        time += BIT_TIME/2;

        byte = 0;
        // input data bits
        for (int i=0; i<8; i++) {
            time += BIT_TIME;
            hwtimer_set_trigger_time(t, time);
            (void) hwtimer_get_time(t);
            byte = port_in_shift_right(RXD, byte);
        }

        // input stop bit
        time += BIT_TIME;
        hwtimer_set_trigger_time(t, time);
        (void) hwtimer_get_time(t);
        (void) port_in(RXD);

        put_byte(byte >> 24); // defined elsewhere
    }
    hwtimer_free(t);
} // receiver
```

The receiver samples the incoming signal, waiting for a start bit. After receiving this bit, it waits for 1.5 times the bit time and then samples the wire at the midpoint of the the first byte transmission, with subsequent bits being sampled at 8.68 µs (BIT_TIME) increments. The input statement in the `for` loop

```
byte = port_in_shift_right(RXD, byte);
```

calls the function `port_in_shift_right`, which first right-shifts the value of byte by the port width (1 bit) and then inputs the next sample into its most significant port-width bits. The expression in the final statement

```
putByte(byte >> 24);
```

right-shifts the bits in the integer byte by 24 bits so that the input value ends up in its least significant bits.

7 Responding to Multiple Inputs

The examples above implicitly require a separate execution thread for each input port. However, the XCORE architecture allows a single thread to be used to detect events on multiple ports, using the SELECT_RES macro which is defined in is defined in the `<xcore/select.h>` header file.

A SELECT_RES construct allows the processor to wait for events on an arbitrary group of resources, which can include both ports and timers. The code generated waits for an event on any of the resources, and then executes the code associated with that event, which is defined by a CASE_ macro. Full details on SELECT_RES can be found in the lib_xcore documentation.

The program below illustrates the use of this construct. It processes inputs from two streams of data from two separate ports using only a single thread. The availability of data on one of these ports is signalled by the toggling of a pin, with data on another other port being received at a fixed rate.

```
#include <xs1.h>
#include <print.h>
#include <xcore/select.h>
#include <xcore/hwtimer.h>
#include <xcore/port.h>

#define DELAY_Q 2000

port_t toggleP = XS1_PORT_1A;
port_t dataP   = XS1_PORT_4A;
port_t dataQ   = XS1_PORT_4B;

int main(void) {
    hwtimer_t t = hwtimer_alloc();
    unsigned time, x = 0;
    int data;
    port_enable(toggleP);
    port_enable(dataP);
    port_enable(dataQ);

    time = hwtimer_get_time(t);
    time += DELAY_Q;
    port_set_trigger_in_not_equal(toggleP, x);
    hwtimer_set_trigger_time(t, time);
    SELECT_RES(CASE_THEN(toggleP, p_toggled),
               CASE_THEN(t, timer_expired)) {
    p_toggled:
        x = port_in(toggleP);
        port_set_trigger_in_not_equal(toggleP, x);
        data = port_in(dataP);   // Input data from port P
        printchar(data + '0');   // Do something with it
        continue;

    timer_expired:
        time = hwtimer_get_time(t);
        data = port_in(dataQ);   // Input data form port Q
        printchar(data + 'a');   // Do something with it
        time += DELAY_Q;
        hwtimer_set_trigger_time(t, time);
        continue;
    }
    hwtimer_free(t);
}
```

The SELECT_RES construct continually performs an input on either the port **toggleP** or the timer **t**, depending on which of these resources becomes ready to input first. Before the SELECT_RES both are set with a trigger so that both are blocked from completing.

In the SELECT_RES two CASE_THEN are defined, one for each resource. Inside the code for the SELECT_RES, the two cases are spelled out. Each case start with an input, and each case ends by setting the trigger up appropriately.

If both inputs become ready at the same time, only one is selected, the other remaining ready on the next iteration of the loop. After performing an input, the body of code below it is executed. Each body is in this case terminated by a **continue** to ensure it continues looping for more events.

Note: Case statements can only work on input operations, not output operations as the XCORE architecture requires an output operation to complete but allows an input operation to wait until it sees a matching output before committing to its completion.

Each port and timer may appear in only one of the **case** statements. This is because the XCORE architecture restricts each port and timer resource to waiting for just one condition at a time.

In this example, the processor effectively multi-tasks the running of two independent tasks, and it must be fast enough to process both streams of data in real-time. If this is not possible, two separate threads may be used to process the data instead

8 Case Study: UART (Part 2)

Using the SELECT_RES macro, the UART transmitter described in the Case study above can be optimized to implement both the transmit and receive sides of a UART in a single thread as shown below.

```
#include <xcore/select.h>
#include <xcore/hwtimer.h>
#include <xcore/port.h>

extern void putByte(int b);
extern int  getByte();
extern int  hasData();

void UART(port_t RX, int rxPeriod, port_t TX, int txPeriod) {
    int txByte, rxByte;
    int txI, rxI;
    int rxTime, txTime;
    int isTX = 0;
    int isRX = 0;
    hwtimer_t tmrTX = hwtimer_alloc();
    hwtimer_t tmrRX = hwtimer_alloc();
    port_enable(RX);
    port_enable(TX);
    port_out(TX, 1);

    while (1) {
        if (!isTX && hasData()) {
            isTX = 1;
            txI = 0;
            txByte = getByte();
            port_out(TX, 0);            // transmit start bit
            txTime = hwtimer_get_time(tmrTX) + txPeriod;
        }
        if (isRX) {
            hwtimer_set_trigger_time(tmrRX, rxTime);
            port_clear_trigger_in(RX);
        } else {
            port_set_trigger_in_equal(RX, 0);
        }
        if (isTX)
            hwtimer_set_trigger_time(tmrTX, txTime);

        SELECT_RES(CASE_GUARD_THEN(RX,   !isRX, start_receiver),
                   CASE_GUARD_THEN(tmrRX, isRX, receive_bit),
                   CASE_GUARD_THEN(tmrTX, isTX, transmit_bit)) {
        start_receiver:
            (void) port_in(RX);
            rxTime = hwtimer_get_time(tmrRX) + rxPeriod * 3 / 2;
            isRX = 1;
            rxI = 0;
            rxByte = 0;
            break;

        receive_bit:
            (void) hwtimer_get_time(tmrRX);
            if (rxI < 8) {
                rxByte = port_in_shift_right(RX, rxByte);
                rxI++;
                rxTime += rxPeriod;
            } else { // receive stop bit
                (void) port_in(RX);    // Can be deleted
                putByte(rxByte >> 24);
                isRX = 0;
            }
            break;

        transmit_bit:
            (void) hwtimer_get_time(tmrTX);
            if (txI < 8)
                txByte = port_out_shift_right(TX, txByte);
            else if (txI == 8)
                port_out(TX, 1);      // send stop bit
            else
                isTX = 0;
            txI++;
            txTime += txPeriod;
            break;
        }
    }
    hwtimer_free(tmrTX);
    hwtimer_free(tmrRX);
}
```

The variables `isTX`, `txI`, `isRX` and `rxI` determine which parts of the UART are active and how many bits of data have been transmitted and received.

The `while` loop first checks whether the transmitter is inactive with data available to transmit, in which case it outputs a start bit and sets the timeout for outputting the first data bit.

Before the `SELECT_RES` statement we set up the potential conditions under which we want to continue. The receiver may continue if either a start-bit is received (and we aren't receiving already), or if it is time to sample the next data bit (and we are receiving):

```
if (isRX)
  hwtimer_set_trigger_time(tmrRX, rxTime);
else
  port_set_trigger_in_equal(RX, 0);
```

The transmitter may continue if we are transmitting and it is time to transmit the next bit:

```
if (isTX)
  hwtimer_set_trigger_time(tmrTX, txTime);
```

In the `SELECT_RES` set-up we define the three cases, each with a guard:

```
CASE_GUARD_THEN(start_receiver, !isRX, RX)
CASE_GUARD_THEN(receive_bit,    isRX,  tmrRX)
CASE_GUARD_THEN(transmit_bit,   isTX,  tmrTX)
```

The first case is the case where we start the receiver; this case shall only be executed when we are not already receiving and if the RX port is ready (based on its trigger). The second case is for when we are receiving and the receiver-timer indicates that the next bit is ready. The final case is the case for which we are transmitting and the transmit-timer indicates that the next bit is ready.

Inside the `SELECT_RES` statement we find the three sections of code.

- The body of the first case picks up the start-bit, and then calculates when the first data bit should be sampled.
- The body of the second case inputs the next bit of data and, once all bits are input, it stores the data and sets `isRX` back to zero. It also adjusts the time for the next bit
- The body of the third case outputs the next bit of data and, once all bits are output, it sets `isTX` to zero. It also adjusts the time for the next bit.

9 Further information

This document is one of a group of application notes that describes XCORE ports.

The other documents in this group are:

- AN03007: XCORE Ports
- AN03002: XCORE Port Buffering
- AN03003: XCORE Serialization and Strobing
- AN02039: Ports, Pins, and the XN file

For detailed information on the programming XCORE ports, the lib_xcore documentation is recommend as a reference.