XMOS

# AN03003: XCORE Port Serialisation and Strobing

Publication Date: 2025/3/11
Document Number: XM-015262-AN v1.0.0

IN THIS DOCUMENT


1 Overview . . . 1
2 Serialising Output Data using a Port . . . 1
3 Deserialising Input Data using a Port . . . 2
4 Input Data Accompanied by a Data Valid Signal . . . 3
5 Data Output with a Data Valid Signal . . . 4
6 Case Study: Ethernet MII . . . 6
7 Summary . . . 11


## 1 Overview

The XMOS architecture provides hardware support for operations that frequently arise in communication protocols. For example a port can be configured to perform *serialization*, useful if data must be communicated over ports that are only a few bits wide, and *strobing*, which is useful if data is accompanied by a separate data valid signal.

Offloading these tasks to the ports frees up more processor time for executing computations.

This application note describes how to use the serialization and strobing capabilities of XCORE ports and is intended to be read in conjunction with application notes

- AN03000: XCORE Input and Output,
- AN03001: XCORE Clocked Input and Output, and
- AN03002: XCORE Port Buffering.

In addition, the reader should be familiar with the concept of XCORE ports as described in

- AN03007: XCORE Ports, and
- AN02039: Ports, Pins, and the XN file.

## 2 Serialising Output Data using a Port

A clocked port can serialize data, reducing the number of instructions required to perform an output. This is done using an extension to the buffered port mechanism described in AN03002: XCORE Port Buffering and is enabled by including the `#include <xcore/port.h>` and `#include <xcore/clock.h>` directives in your code.

The example programme below outputs a 32-bit value onto 8 pins, using a clock to determine how long each 8-bit value is driven on the output pins.

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

port_t outP    = XS1_PORT_8A;
port_t inClock = XS1_PORT_1A;
xclock_t clk   = XS1_CLKBLK_1;
```

(continues on next page)

(continued from previous page)

```
int main(void) {
    port_start_buffered(outP, 32);
    port_enable(inClock);
    clock_enable(clk);

    clock_set_source_port(clk, inClock);
    port_out(outP, 0);
    port_set_clock(outP, clk);

    clock_start(clk);

    port_out(outP, 0xAA00FFFF);
    port_out(outP, 0x12345678);
    port_out(outP, 0x55AAA111);
    port_sync(outP);            // Used to wait before stopping the program
}
```

The declaration

```
port_start_buffered(outP, 32);
```

configures the port **outP** to drive 8 pins from a 32-bit *shift register*. The second parameter, 32, specifies the number of bits that are transferred in each output operation (the *transfer width*). Fig. 1 shows the data driven by this program.

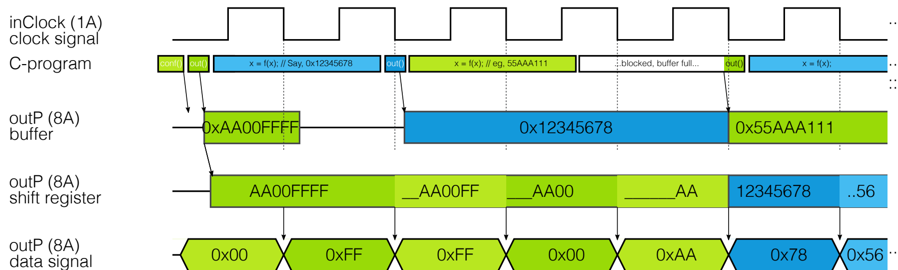


Fig. 1: Serialized output waveform diagram

By offloading the serialization to the port, the processor has only to output once every 4 clock periods. On each falling edge of the clock, the least significant 8 bits of the shift register are driven on the pins; the shift register is then right-shifted by 8 bits.

**Note:** Ports used for serialization must be initialised with the `port_start_buffered`.

## 3 Deserialising Input Data using a Port

As the complement to the functionally described above, an XCORE port can deserialize data, reducing the number of instructions required to input data. The programme below performs a 4-to-8 bit conversion on an input port, controlled by a 25 MHz clock.

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

port_t inP       = XS1_PORT_4C;
port_t outClock = XS1_PORT_1E;
xclock_t clk25   = XS1_CLKBLK_1;

int main(void) {
    port_start_buffered(inP, 8);
    port_enable(outClock);
    clock_enable(clk25);

    clock_set_divide(clk25, 2); // divide by 4, 25 MHz
    port_set_clock(inP, clk25);
```

(continues on next page)

(continued from previous page)

```
    port_set_clock(outClock, clk25);
    port_set_out_clock(outClock);
    clock_start(clk25);

    int data[10];
    for(int i = 0; i < 10; i++) {
        data[i] = port_in(inP);
    }
    for(int i = 0; i < 10; i++) {
        printf("<%02x>\n", data[i]);
    }
}
```

The programme configures `inP` to be a 4-bit wide port (`XS1_PORT_4C`) with an 8-bit transfer width (the **8** in `port_start_buffered(inP, 8)`), meaning that two 4-bit values can be sampled by the port before they must be input by the processor.

As with serialized output, the deserializer reduces the number of instructions required to obtain the data. Fig. 2 shows example input stimuli and the period during which the data is available in the port's buffer for input.

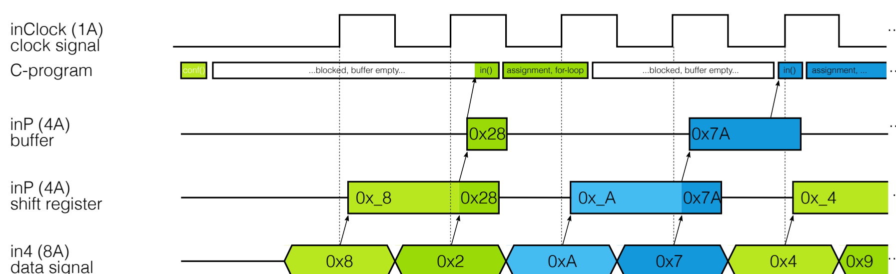


Fig. 2: Deserialized input waveform diagram

Data is sampled on the rising edges of the clock and, when shifting, the least significant 4-bits are read first. The sampled data is available in the port's buffer until the next time that the shift register is full. The first two values input are *0x28* and *0x7A*.

# 4 Input Data Accompanied by a Data Valid Signal

A clocked port can interpret a *ready-in* strobe signal that determines the validity of the accompanying data. This functionally can be enabled by including the `#include <xcore/port_protocol.h>` directive in your code.

The programme below inputs data from a clocked port only when a ready-in signal is high.

```
#include <xcore/port.h>
#include <xcore/port_protocol.h>
#include <xcore/clock.h>
#include <xs1.h>
#include <stdio.h>

port_t inP      = XS1_PORT_4C;
port_t inReady  = XS1_PORT_1B;
port_t inClock  = XS1_PORT_1A;
xclock_t clk    = XS1_CLKBLK_1;

int main(void) {
    port_start_buffered(inP, 8);
    port_enable(inReady);
    port_enable(inClock);
    clock_enable(clk);

    clock_set_source_port(clk, inClock);
    port_protocol_in_strobed_slave(inP, inReady, clk);
    clock_start(clk);

    int data0 = port_in(inP);
    int data1 = port_in(inP);
```

(continues on next page)

(continued from previous page)

```
    printf("%02x %02x\n", data0, data1);
}
```

The statement

```
port_protocol_in_strobed_slave(inP, inReady, clk);
```

configures the input port **inP** to be sampled only when the value sampled on the port **inReady** equals 1. The ready-in port must be 1-bit wide. Fig. 3 shows example input stimuli and the data input by this program.

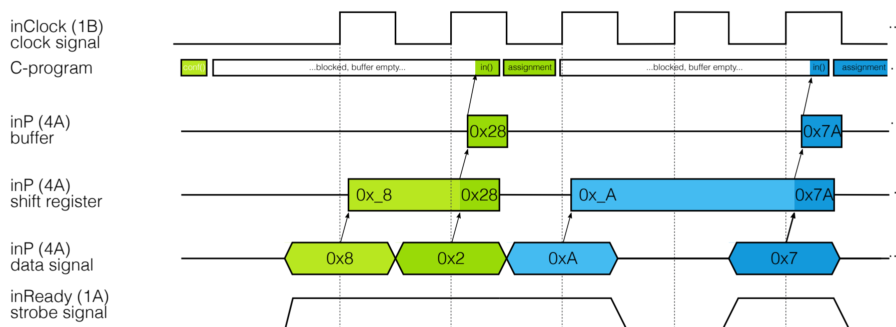


Fig. 3: Input data with data valid signal

Data is sampled on the rising edge of the clock whenever the ready-in signal is high. The port samples two 4-bit values and combines them to produce a single 8-bit value for input by the processor; the data input is *0x28*. XCORE devices have a single-entry buffer, which means that data is available for input until the ready-in signal is high for the next two rising edges of the clock.

# 5 Data Output with a Data Valid Signal

A clocked port can generate a *ready-out* strobe signal whenever data is output. The programme below causes an output port to drive a data valid signal whenever data is driven on a 4-bit port.

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

port_t outP     = XS1_PORT_4D;
port_t outR     = XS1_PORT_1C;
port_t inClock  = XS1_PORT_1A;
xclock_t clk    = XS1_CLKBLK_1;

int main(void) {
    port_start_buffered(outP, 8);
    port_enable(outR);
    port_enable(inClock);
    clock_enable(clk);

    clock_set_source_port(clk, inClock);
    port_protocol_out_strobed_master(outP, outR, clk, 0);
    clock_start(clk);

    port_out(outP, 0x58);
    f();
    port_out(outP, 0x12);
    port_out(outP, 0x34);
}
```

The statement

```
port_protocol_out_strobed_master(outP, outR, clk, 0);
```

configures the output port **outP** to drive the port **outR** high whenever data is output. The ready-out port must be 1-bit wide. Fig. 4 shows the data and strobe signals driven by this program.

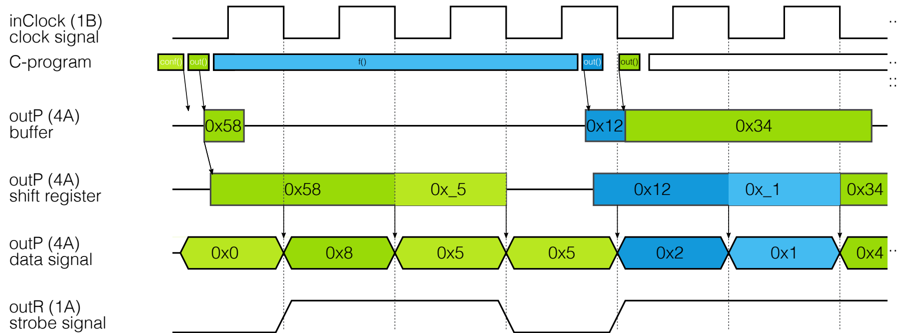


Fig. 4: Output data with data valid signal

The port outputs the first value as two 4-bit values over two clock periods, raising the ready-out signal during this time. In this example, the processing function `f()` has taken so long that the next valid data was a clock late, so the strobe is low for a clock period, then high again to drive subsequent values out.

It is also possible to implement control flow algorithms that output data using a ready-in strobe signal and that input data using a ready-out strobe signal; when both signals are configured, the port implements a symmetric strobe protocol that uses a clock to handshake the communication of the data.

**Note:** On XCORE devices, ports that output on a strobe must be started using `port_start_buffered()`.

# 6 Case Study: Ethernet MII

A single thread on an XCORE device can be used to implement a full duplex 100 Mbps *Ethernet Media Independent Interface (MII) protocol*. This protocol implements the data transfer signals between the link layer and physical device (PHY). The signals are shown in Fig. 5.

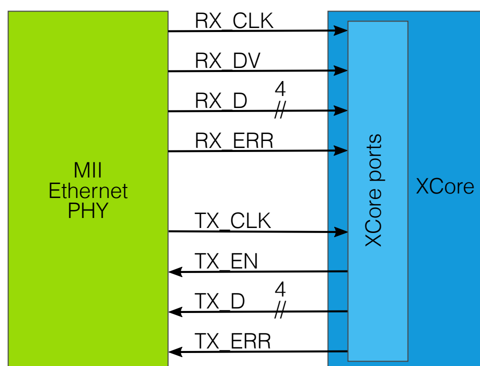


Fig. 5: MII signal diagram

## 6.1 MII Transmit

Fig. 6 shows the transmission of a single frame of data to the PHY. The error signal *TX_ERR* is rarely used and omitted for simplicity.

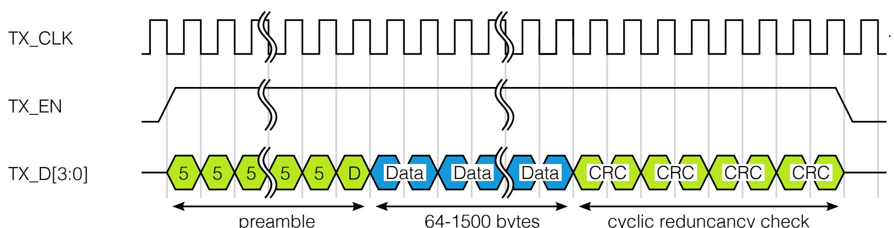


Fig. 6: MII transmit waveform diagram

The signals are as follows:

- *TX_CLK* is a free running 25MHz clock generated by the PHY.
- *TX_EN* is a data valid signal driven high by the transmitter during frame transmission.
- *TX_D* carries a nibble of data per clock period from the transmitter to the PHY. The transmitter starts by sending a preamble of nibbles of value 0x5, followed by two nibbles of values 0x5 and 0xD. The data, which must be in the range of 64 to 1500 bytes, is then transmitted, least significant bit first, followed by four bytes containing a CRC.

Fig. 7 illustrates the port configuration required to serialize the output data and produce a data valid signal.

The port `TX_D` performs a 32-to-4 bit serialization of data onto its pins. It is synchronised to the 1-bit port `TXCLK` and uses the 1-bit port `TX_EN` as a ready-out strobe signal that is driven high whenever data is driven. In this configuration, the processor has only to

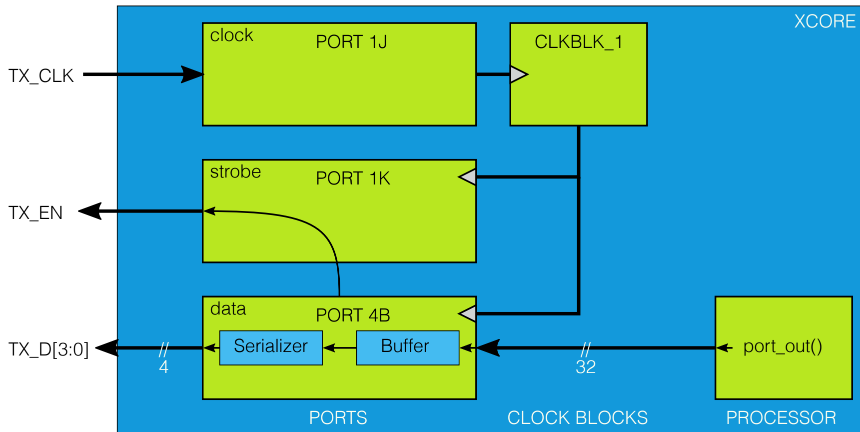


Fig. 7: MII transmit port configuration

output data once every eight clock periods and does not need to explicitly output the data valid signal. The programme below defines and configures the ports in this way.

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

port_t TX_D   = XS1_PORT_4B;
port_t TX_EN  = XS1_PORT_1K;
port_t TX_CLK = XS1_PORT_1J;
static xclock_t clk  = XS1_CLKBLK_1;

void miiConfigTransmit() {
    port_start_buffered(TX_D, 32);
    port_enable(TX_CLK);
    port_enable(TX_EN);
    clock_enable(clk);

    clock_set_source_port(clk, TX_CLK);
    port_protocol_out_strobed_master(TX_D, TX_EN, clk, 0);
    clock_start(clk);
}
```

The function below outputs a frame of data to the MII ports. For simplicity, the error signals and CRC are ignored. We assume that the frame length is stored in the first four bytes.

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

void miiTransmitFrame(char *cpkt) {
        int numBytes, tailBytes, tailBits, data;
    int *pkt = (int *)cpkt;

    /* Input size of next packet */
    numBytes = *pkt++;
    tailBytes = numBytes % 4;
    tailBits = tailBytes * 8;

    /* Output row of 0x5s followed by 0xD */
    port_out(TX_D, 0xD5555555);

    /* Output 32-bit words for serialization */
    for (int i=0; i<numBytes-tailBytes; i+=4) {
        data = *pkt++;
        port_out(TX_D, data);
    }

    /* Output remaining bits of data for serialization */
    if (tailBits != 0) {
        data = *pkt;
        port_out_part_word(TX_D, data, tailBits);
    }
```

(continues on next page)

(continued from previous page)

```
    printstr("Done\n");
}
```

The programme first gets the size of the frame in bytes. It then outputs a 32-bit preamble to TX_D, which is driven on the pins as nibbles over eight clock periods. On each iteration of the for loop, the next 32 bits of data are then output to TX_D for serialising onto the pins. This gives the processor enough time to get around the loop before the next block of data must be driven.

The final statement

```
port_out_part_word(TX_D, data, tailBits);
```

performs a *partial output* of the remaining bits of data that represent valid frame data.

## 6.2 MII Receive

Fig. 8 shows the reception of a single frame from the PHY. The error signal *RXER* is omitted for simplicity.

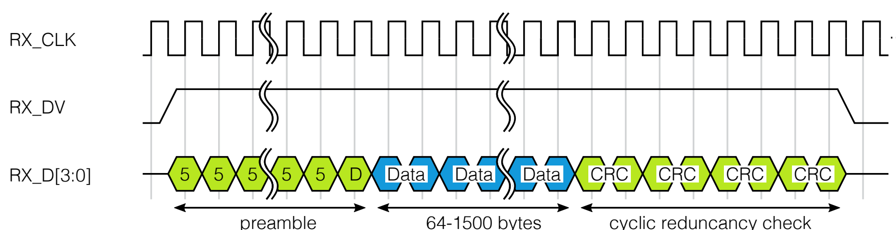


Fig. 8: MII receive waveform diagram

The signals are as follows:

- *RX_CLK* is a free running clock generated by the PHY.
- *RX_DV* is a data valid signal driven high by the PHY during frame transmission.
- *RX_D* carries a nibble of data per clock period from the PHY to the receiver. The receiver waits for a preamble of nibbles of values *0x5*, followed by two nibbles with values *0x5* and *0xD*. The actual data is then received, which is in the range of 64 to 1500 bytes, least significant nibble first, followed by four bytes containing a CRC.

Fig. 9 illustrates the port configuration required to deserialize the input data when a data valid signal is present.

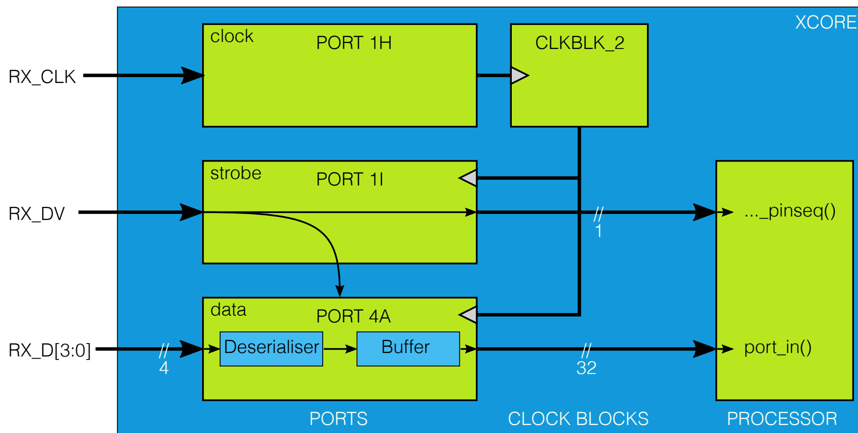


Fig. 9: MII receive port configuration

The port **RX_D** performs a 4-to-32-bit deserialization of data from its pins. It is synchronised to the 1-bit port **RX_CLK** and uses the 1-bit port **RX_DV** as a ready-in strobe signal that causes data to be sampled only when the strobe is high. In this configuration, the port can sample eight values before the data must be input by the processor, and the processor does not need to explicitly wait for the data valid signal. The programme below defines and configures the ports in this way.

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

port_t RX_D   = XS1_PORT_4A;
port_t RX_DV  = XS1_PORT_1I;
port_t RX_CLK = XS1_PORT_1H;
static xclock_t clk  = XS1_CLKBLK_2;

void miiConfigReceive() {
    port_start_buffered(RX_D, 32);
    port_enable(RX_DV);
    port_enable(RX_CLK);
    clock_enable(clk);

    clock_set_source_port(clk, RX_CLK);
    port_protocol_in_strobed_slave(RX_D, RX_DV, clk);
    clock_start(clk);
}
```

The function below receives a single error-free frame and stores it in an array. For simplicity, the error signal and CRC are ignored.

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

void miiReceiveFrame(char *pkt) {
    int data, tail;
    int *o = ((int*)pkt)+1;
    int bytes = 0;

    // Wait for start of frame
    port_set_trigger_in_equal(RX_D, 0xD);
    (void) port_in(RX_D);
    port_set_trigger_in_equal(RX_DV, 0);

    // Receive frame data/crc
    SELECT_RES(
        CASE_THEN(RX_DV, last_data),
        CASE_THEN(RX_D, data_arrived))
    {
    data_arrived:
```

(continues on next page)

(continued from previous page)

```
        // input next 32 bits of data
        data = port_in(RX_D);
        *o++ = data;
        bytes += 4;
        SELECT_CONTINUE_NO_RESET;
    last_data:
        (void) port_in(RX_DV);
        // Input any bits remaining in port
        tail = port_endin(RX_D);
        bytes += tail>>3;
        if (tail > 32) {
            tail -= 32;
            data = port_in(RX_D);
            *o++ = data;
        }
        data = port_in(RX_D);
        *o++ = data >> (32-tail);
        *(int *)pkt = bytes;
        return;
    }
}
```

The processor waits for the last nibble of the preamble (*0xD*) to be sampled by the port `RX_D`:

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

We then set up a trigger for the data valid signal `RX_DV` to go low. We then enter a structure that can select from two choices:

- Data may have arrived on `RX_D`: the data is input, stored, the byte counter is increased and the `SELECT_CONTINUE_NO_RESET` goes around the select structure again (waiting for more data)
- The last data may have arrived (because `RX_DV` triggered), in which case we must first acknowledge that RX_DV was triggered, and then we need to deal with the final nibbles of data.
  An effect of using a port's serialization and strobing capabilities together is that the ready-in signal may go low before a full transfer width's worth of data is received. Hence, the `last_data` label may arrive when there is still data in the port.
  The statement:

  ```
  tail = port_endin(RX_D);
  ```

  causes the port `RX_D` to respond with the remaining number of bits not yet input. It also causes the port to provide this data on the subsequent inputs, even though the data valid signal is low and the shift register is not yet full.
  XCORE devices provide a single-entry buffer up to 32-bits wide and a 32-bit shift register, requiring up to 64 bits of data being input over two input statements once the data valid signal goes low. The last word may not be complete, and has to be manually shifted right.

## 7 Summary

The semantics for I/O on a serialized port are as follows (where *p* refers to the port width and *w* refers to the transfer width of a port):

- An output of a *w*-bit value is driven over *w*/*p* consecutive clock periods, least significant bits first. The ready-out signal is driven high on each of these periods.
- For a timed output, the port waits until its counter equals the specified time before *starting* to serialize the data. The ready-out signal is not driven while waiting to serialize.
- An input of a *w*-bit value is sampled over *w*/*p* clock periods, with earlier bits received ending up in the least significant bits of *w*. (If a ready-in signal is used, the clock periods may not be consecutive.)
- For a timed input, the port provides the *last p* bits of data sampled when its counter equals the specified time.

If a port is configured with a ready-in signal:

- Data is sampled only on rising edges of the port's clock when the ready-in signal is high.

If a port is configured with a ready-out signal:

- The ready-out signal is driven high along with the data and is held for a single period of the clock.

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.