

# XC-1 Development Card Tutorial

---

(VERSION 9.7)



2009/08/11

*Authors:*

XMOS LTD.

Copyright © 2009, XMOS Ltd.  
All Rights Reserved

## 1 Introduction

The XC-1 is an Event-Driven Processor development board based on the XMOS XS1-G4. It comprises a single XS1-G4, 16 LEDs, four press-buttons, a speaker, JTAG and serial interfaces, and a through-hole prototyping area for connecting external components.

The XS1-G4 consists of four XCore tiles, each comprising an event-driven multi-threaded processor core with tightly integrated general purpose I/O. Each tile provides up to eight threads, 400 MIPS and 64 KBytes of RAM. The XS1-G4 pins are connected to the components on the board using ports (see Section 9).

The XMOS originated XC language [1] is based upon C, providing additional constructs that simplify control over I/O operations, time and concurrent behaviour. The XMOS design tools support XC and C, allowing complete systems to be built and debugged from within a single development environment.

This tutorial provides an introduction to start developing for Event-Driven Processor devices using features of the XC-1 and the XC language. It assumes that you are familiar with C [2]. In this tutorial you will:

- illuminate the LEDs on the card
- flash an LED on the card
- flash an LED while cycling it around a 12-LED circle on the card
- respond to a button press, outputting an audible tone to the speaker, at the same time as flashing the LEDs in a cycle
- communicate between two threads so that the flashing LED changes colour when a button is pressed
- implement a UART protocol and send the message “Hello World” to your host PC
- implement a multicore program using a loopback connection on the through-hole area on the card

Each section of the tutorial introduces a new feature of XC; the corresponding keywords and operators are included in the section title. The example programs

are intended to illustrate how particular language constructs simplify the implementation of Event-Driven Processor designs.

The examples in this tutorial apply to version 9.7 of the XMOS Design Tools. Information on downloading, installing and using these tools is provided in the Development Tools User Guide [3].

## 2 Illuminate an LED: port, <:

This part of the tutorial shows you how to use XC ports and an output statement to illuminate the LEDs on your XC-1.

The XC-1 has 16 LEDs. Four of these green LEDs are positioned next to the four press-buttons (collectively referred to as button-LEDs) and contain green diodes. Another 12 red/green LEDs are positioned around the XS1-G4 in a circle (collectively referred to as clock-LEDs). A Roman numeral is printed next to each LED.

The following program illuminates a single button-LED on your XC-1:

```
#include <platform.h>

out port bled = PORT_BUTTONLED;

int main() {
    bled <: 0x10;
    while(1)
        ;
    return 0;
}
```

The second line of this program declares a port variable `bled` and initialises it with a generic port identifier `PORT_BUTTONLED`. The `XC-1.xn` file maps the `PORT_BUTTONLED` identifier to the `XS1_PORT_8D` port which originates from the `xs1.h` header file. The `8D` refers to the pin width (8 bits) and name (D). On the XC-1, the top four bits of pins `8D` are connected to the four button-LEDs. The LEDs are active high.

Ports are used to transfer data to and from the pins on the processor, thereby interfacing with external components. Integrated input and output XC statements make it easy to express I/O operations on these pins.


The first statement in `main` is an output statement:


```
bled <: 0x10;
```

The value specified to the right of `<:` (`0b00010000`) is output to the port specified to its left (`bled`). This value sets one of the button-LED pins high, causing the corresponding LED to illuminate. Outputting a different value to `bled` causes a different combination of the button-LEDs to illuminate.

The infinite loop introduced after the output statement prevents the program terminating, ensuring that the LED remains illuminated.

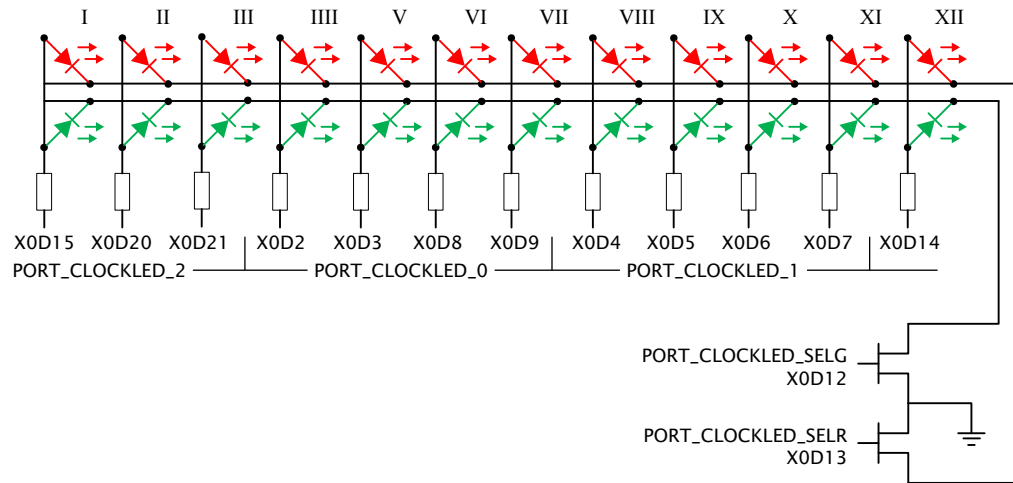
**Note:** Ports must be declared as global variables. The optional `out` qualifier allows the compiler to check for correct usage, thereby helping to reduce programming errors.

 Compile and run this program on your XC-1. (See the Development Tools User Guide for information on using the Development Tools [3].) A single LED should illuminate.

 Modify the value output to the button-LED port (`0xf0`) to illuminate all four button-LEDs.

To reduce the number of pins required for the 12 clock-LEDs, the LED anodes are connected to three 4-bit ports (4A, 4B and 4C, which are referenced using the `PORT_CLOCKLED_0/1/2` identifiers) and the cathodes are connected to two 1-bit ports (1E for green and 1F for red, which are referenced using the `PORT_CLOCKLED_SELG` and `PORT_CLOCKLED_SELR` identifiers) that are active high. The schematic for the clock-LEDs is shown in Figure 1.

The following program illuminates the red diode of an LED connected to `PORT_CLOCKLED_0`:




**Figure 1** Schematic for the 12 bi-colour clock-LEDs on the XC-1


```
#include <platform.h>

out port cled0 = PORT_CLOCKLED_0;
out port cledG = PORT_CLOCKLED_SELG;
out port cledR = PORT_CLOCKLED_SELR;

int main(void) {
    cledG <: 0;    // disable GREEN line
    cledR <: 1;    // enable RED line
    cled0 <: 0x1; // LED pattern
    while (1);
    return 0;
}
```

PORT\_CLOCKLED\_SELG is driven low, PORT\_CLOCKLED\_SELR is driven high and the value 0x1 is output to PORT\_CLOCKLED\_0 (IIII, V, VI, VII).

 Compile and run this program on your XC-1. The clock-LED numbered VII should illuminate red.

 Modify this program to illuminate the green diodes of all 12 LEDs (output

the pattern `0xf` to `PORT_CLOCKLED_0`, `PORT_CLOCKLED_1` and `PORT_CLOCKLED_2`).

### 3 Flash an LED: `timer, :>`

This part of the tutorial shows you how to use an XC timer with an input statement to flash an LED green-red.

Timers are a special type of port that, when input from, return the current time. Timers provide a view onto a 100 MHz reference clock, and can be used to determine when an event happens or to delay execution until a particular time.

The following code declares a timer named `tmr` and then inputs the time into the variable `t`:

```
timer tmr;  
tmr :> t;
```

Having recorded the current time, you can increment the time and then delay a following input until after this time is reached:

```
t += FLASH_PERIOD;  
tmr when timerafter(t) :> void;
```


The processor must complete an input operation once a condition is met, even if the input value is not required. This is expressed in XC as an input to `void`.


This code sequence can be used to delay an output operation to the LED pins which, when executed in a loop, flashes the LED green-red. The complete program is shown below:

```
#include <platform.h>  
#define FLASH_PERIOD 20000000  
  
out port cled0 = PORT_CLOCKLED_0;  
out port cledG = PORT_CLOCKLED_SELG;
```

```
out port cledR = PORT_CLOCKLED_SEL_R;

int main(void) {
    timer tmr;
    unsigned ledGreen = 1;
    unsigned t;
    tmr :> t;
    while (1) {
        cledG <: ledGreen;
        cledR <: !ledGreen;
        cled0 <: 0x1;
        t += FLASH_PERIOD;
        tmr when timerafter(t) :> void;
        ledGreen = !ledGreen;
    }
    return 0;
}
```

 Compile and run this program on your XC-1. A single clock-LED should flash green-red.

 Modify this program by reducing the timer period to colour-blend the lights and create the appearance of a constant orange illumination. Note that the red LED is approximately five times as bright as the green LED, which means it should be turned on five times less frequently to create the desired effect.

## 4 Flash and cycle LEDs at different rates: `select`

This part of the tutorial shows you how to use the XC `select` statement to make an LED flash while cycling it around the clockface on your XC-1.

A `select` statement waits for one of a set of inputs to become ready, performs the selected input and then executes a corresponding body of code. Each input is preceded by the keyword `case` and its body must be terminated with a `break` or `return` statement.

The `select` statement in the following example waits for one of two timeouts to occur and then responds to this timeout:

```
select {
  case tmrF when timerafter(timeF) :> void :
    /* respond to timeout,
     * switch LED between on and off */
    ...
    break;
  case tmrC when timerafter(timeC) :> void :
    /* respond to timeout,
     * change which LED is flashing */
    ...
    break;
}
```

The following program uses this `select` statement to update the LED flash and cycle states at different rates:

```
#include <platform.h>


#define FLASH_PERIOD 10000000
#define CYCLE_PERIOD 60000000

out port cled0 = PORT_CLOCKLED_0;
out port cled1 = PORT_CLOCKLED_1;
out port cled2 = PORT_CLOCKLED_2;
out port cledG = PORT_CLOCKLED_SELG;
out port cledR = PORT_CLOCKLED_SELR;

int main(void) {
  unsigned ledOn = 1;
  unsigned ledVal = 1;
  timer tmrF, tmrC;
  unsigned timeF, timeC;
```

```
tmrF :> timeF;
tmrC :> timeC;

while (1) {
  select {
    case tmrF when timerafter(timeF) :> void :
      ledOn = !ledOn;
      cledG <: ledOn;
      timeF += FLASH_PERIOD;
      break;
    case tmrC when timerafter(timeC) :> void :
      cled0 <: ledVal;
      cled1 <: (ledVal >> 4);
      cled2 <: (ledVal >> 8);
      ledVal <<= 1;
      if (ledVal == 0x1000)
        ledVal = 1;
      timeC += CYCLE_PERIOD;
      break;
  }
}
return 0;
}
```

 Compile and run this program on your XC-1. A flashing LED should cycle clockwise around the clockface.

## 5 Respond to a button concurrently: `par`

This part of the tutorial shows you how to use the XC `par` statement to process multiple tasks concurrently. In this case, the flashing LED code from the previous example is run in parallel with a thread that sends an audible tone to the speaker on your XC-1 when a button is pressed.

The `par` statement provides a simple way to execute multiple statements as separate threads in parallel. In the following example, two functions are called concurrently:

```
in port buttons = PORT_BUTTON;
out port speaker = PORT_SPEAKER;

int main(void) {
    par {
        cycleLED(cled0, cled1, cled2,
                cledG, cledR,
                FLASH_PERIOD, CYCLE_PERIOD);
        buttonListener(buttons, speaker);
    }
    return 0;
}
```


The `cycleLED` function cycles a flashing LED around the clockface (see Section 4). The `buttonListener` function, defined below, waits for input from a button and then outputs a tone to the speaker:


```
#define TDELAY 100000
#define TLENGTH 500

void buttonListener(in port b, out port spkr) {
    timer tmr;
    int t, isOn = 1;
    while (1) {
        b when pinsneq(0xf) :> void;
        tmr :> t;
        for (int i=0; i<TLENGTH; i++) {
            isOn = !isOn;
            t += TDELAY;
            tmr when timerafter(t) :> void;
            spkr <: isOn;
        }
    }
}
```

```
}  
}
```

The `pinsreq` function causes the input to wait until the value sampled on the pins is not equal to the bit pattern `0xf`, which signifies that one or more of the four buttons was pressed. The `for` loop sends a sequence of alternating ones and zeros to the speaker at the frequency specified by the `delay`, for the specified period of time. This causes the speaker to emit an audible tone.

 Modify the example in Section 4 so that the code for cycling a flashing LED around the clockface is implemented as the body of the `cycleLED` function. Compile and run this complete program on your XC-1. A flashing LED should cycle around the clockface; pressing any of the four buttons at any time should result in an audible tone.

 Extend the `buttonListener` function with four additional tone parameters: pressing different buttons should result in different tones being emitted.

## 6 Use a button to change the LED colour: `chan`, `chanend`

This section of the tutorial shows you how to use an XC channel to communicate between threads. In this case, the button listener thread tells the LED thread to change the colour of the flashing LED.

An XC channel provides a synchronous, bi-directional link between two threads. A channel is declared using the `chan` keyword, as in the following example:

```
int main(void) {  
    chan c;  
    par {  
        cycleLED(cled0, cled1, cled2,  
                cledG, cledR,  
                FLASH_PERIOD, CYCLE_PERIOD,  
                c);  
        buttonListener(buttons, speaker, c);  
    }  
}
```

```
    return 0;
}
```


A channel consists of two channel ends, the locations of which are implicitly defined by the use of the channel in two statements of a `par`. A channel end may be explicitly referred to as a function parameter using the `chanend` keyword, for example:


```
void buttonListener(in port b, out port spkr,
                   chanend c);
```

The XC input and output operators can be used for channels, as in the following example:


```
c <: 0;
```

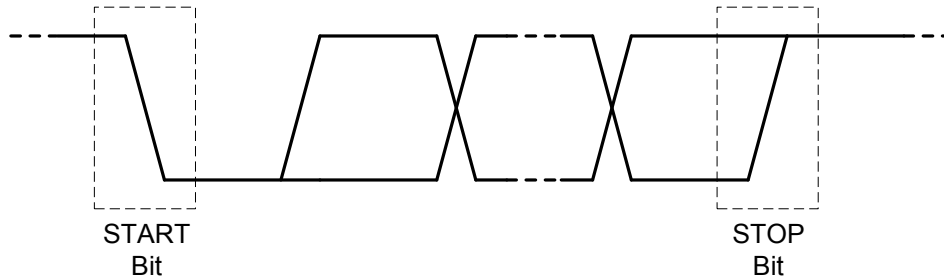
This statement outputs the value 0 to the channel `c`. As the channel is synchronous, the output operation waits until a matching input operation is ready before continuing. Similarly, an input operation waits for a matching output operation before continuing.

 Modify the `buttonListener` function so that immediately after inputting from the button, the value 0 is output to `c`.

 Modify the `cycleLED` function so that it takes an additional channel end argument and declares a local variable `isGreen`. Modify the `select` statement to accept an input from this channel; when selected the colour is changed as follows:

```
case c :> int :
    isGreen = !isGreen;
    break;
```

 Finally, modify the flash timeout body so that it illuminates either the green or red LED, depending on the value of `isGreen`. Compile and run this program



**Figure 2** *UART Transmit Procedure*

on your XC-1. Pressing a button should change the colour of the flashing LED as well as producing an audible tone.

**Note:** No two threads may attempt to write to the same variable in parallel; all communications require the use of channels. This restriction prevents common programming errors such as race conditions, and ensures that the two threads can be run on any two cores, regardless of whether they share memory.

## 7 Interface with a host using a serial link

This part of the tutorial shows you how to implement a UART transmit function that outputs characters to a console running on your host platform.

The UART protocol provides a simple way to transmit data over a serial link. Data is sent at a fixed baud rate, requiring no clock signal to be transmitted. The transmit procedure is illustrated in Figure 2. The quiescent state of the link is the high (1). A byte is sent by first asserting a start bit (0), then the data bits and finally the stop bit (1). Each of these values is asserted for an entire bit period.

The program below serialises a byte of data and transmits its individual bits over a 1-bit port.

```
#include <platform.h>

#define BIT_RATE 115200
#define BIT_TIME XS1_TIMER_HZ / BIT_RATE
```

```
void txByte(out port TXD, int byte);

out port TXD=PORT_UART_TX;

int main () {
return 0;
}

void txByte(out port TXD, int byte) {
    unsigned time;
    timer t;

    /* get initial time */
    t :> time;

    /* send start bit */
    TXD <: 0;
    time += BIT_TIME;
    t when timerafter(time) :> void;

    /* send data bits */
    for (int i=0; i<8; i++) {
        TXD <: >> byte;
        time += BIT_TIME;
        t when timerafter(time) :> void;
    }

    /* send stop bit */
    TXD <: 1;
    time += BIT_TIME;
    t when timerafter(time) :> void;
}
```

The time between consecutive bits being sent is computed by dividing the reference clock frequency by the UART baud rate.

```
#define BIT_RATE 115200
#define BIT_TIME XS1-TIMER_HZ / BIT_RATE
```

The start bit is output to a port variable `TXD` using the following statement:

```
TXD <: 0;
```

The variable `time` is incremented by the value `BIT_TIME` and the conditional input statement waits for that bit time to elapse, ensuring an entire bit period.

```
t when timerafter(time) :> time;
```

The data bits are output in the same way in a `for` loop. Finally the stop bit is also output using the same approach.

The output statement in the `for` loop

```
TXD <: >> byte;
```

includes the modifier `>>`, 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.

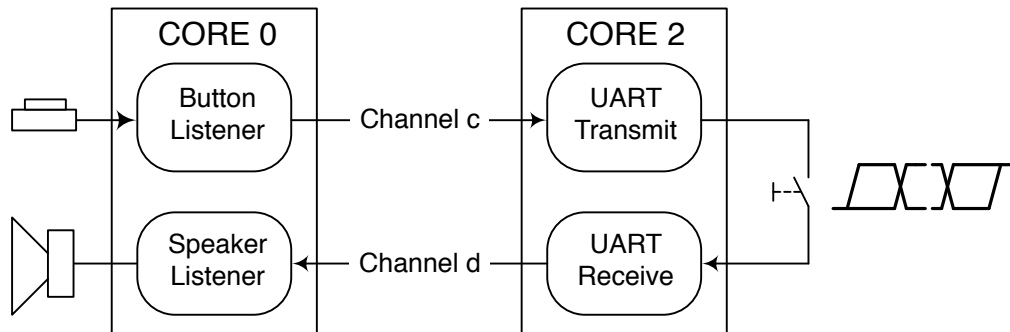
**Note:** The output-shift-right operator `<: >>` requires the output expression to be a variable of type `int`.

The XC-1 has a chip that performs a USB-to-serial conversion. When the card is connected to a PC using a USB cable, this chip presents a virtual COM port<sup>1</sup> that can be interfaced using a terminal emulator.


For your convenience, XMOS has uploaded a sample terminal listener to the Xlinkers community website<sup>2</sup>. Xlinkers has an open-source code repository of Event-Driven Processor programs, related software and reference designs.

<sup>1</sup>Currently on MAC PCs, the virtual COM port cannot be supported at the same time as the JTAG interface.

<sup>2</sup>[http://www.xlinkers.org/tag\\_search?tag=uart](http://www.xlinkers.org/tag_search?tag=uart)



**Figure 3** Thread diagram for a multicore program in which a button listener interfaces with a speaker listener using the UART protocol.

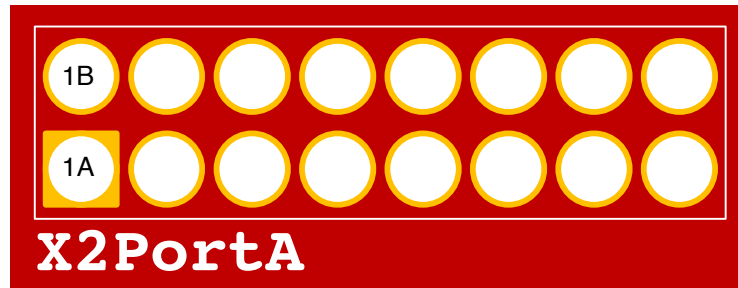
 Implement a `main` function that transmits “Hello World!” over a serial link (`PORT_UART_TX`). Compile and run this program on your XC-1 while running a terminal emulator connected to the virtual COM port installed by the XC-1. The terminal should display this text.

## 8 Interface with the prototyping area: on

This part of the tutorial shows you how to use the `XC on` statement to implement a multicore program. In this case, a UART transmit and receive protocol is implemented on XCore 2 (X2) over a loopback connection made between two pins on the through-hole prototyping area connected to X2.

The program consists of four separate threads, as illustrated in Figure 3. The first thread is a button listener, which must be placed on core 0 to interface with the button pins (`X0/PORT_BUTTON`). This thread outputs the values received on these pins to a channel connected to a UART transmit thread running on core 2. The UART transmit thread inputs button press values and then outputs these values to a jumper pin on the prototyping area (`X2/1A`).

The third thread is the UART receiver, placed on core 2, that waits for button values to be received on a jumper pin (`X2/1B`). The UART receiver outputs these values to a channel connected to a speaker listener thread running on core 0. The speaker listener thread inputs button values and then outputs corresponding tones to the speaker.



**Figure 4** One of the through-hole prototyping areas on the XC-1. To complete this example, you need to solder a jumper connector to pins 1A and 1B.


To complete the exercises in this section you need to solder a jumper connector through two holes on the XC-1, as illustrated in Figure 4.


To get started, you need to include the header file `platform.h`, which provides a declaration of the global variable `stdcore` for the target device, in this case an XS1-G4. Use this variable with the `on` keyword to specify the location of port declarations and parallel substatements:


```
#include <platform.h>
on stdcore[0] : in port buttons = PORT_BUTTON;
on stdcore[0] : out port speaker = PORT_SPEAKER;
on stdcore[2] : in port uartIn = XS1_PORT_1A;
on stdcore[2] : out port uartOut = XS1_PORT_1B;


int main(void) {
    chan c, d;
    par {
        on stdcore[0] : buttonListener(buttons, c);
        on stdcore[2] : transmit(uartOut, c);
        on stdcore[2] : receive(uartIn, d);
        on stdcore[0] : speakerListener(d, speaker);
    }
    return 0;
}
```


**Note:** When `main` is used with `on` it may contain only channel declarations, a single `par` statement and an optional `return` statement.


 Modify the `buttonListener` function from Section 6 so that it outputs a tone value to a channel when a button is pressed. (**Note:** Do not send the delay value directly since this value is larger than a byte.)

 Implement a `transmit` function that waits for inputs from a channel and sends them over a serial link using the `transmitByte` function from Section 7.

 Implement a `receive` function that receives values from a serial link and outputs them to a channel. This requires implementing a `receiveByte` function. (**Note:** Using the input-shift-right operator `:>>>` to input a byte into an `int` results in the byte being input to the most significant bits of the `int`. The input value must therefore be right-shifted by 24 bits after it is received.)

 Implement a `speakerListener` function that waits for inputs from a channel and then, based on the tone value received, outputs a corresponding tone to the speaker.

 Compile and run this program on your XC-1. Pressing any of the four buttons at any time should result in an audible tone. If the jumper is removed, pressing a button should not produce a sound.

 If you have two XC-1s, connect a wire between two pins, one on each XC-1. Implement two XC programs, one comprising a button listener and UART transmitter, the other comprising a UART receiver and speaker listener. Compile and run each of these programs, one on each card. Pressing any of the four buttons on one card should result in an audible tone from the speaker on the other.

## 9 XC1 Board Layout and Further Reading

A block diagram of the XC-1 is illustrated in Figure 5.

Information on pin/port mappings and the initialisers to use to access features of the XC-1 board, see the XC-1 Hardware Manual [4].

Further information on the XC language can be found in Programming XC on

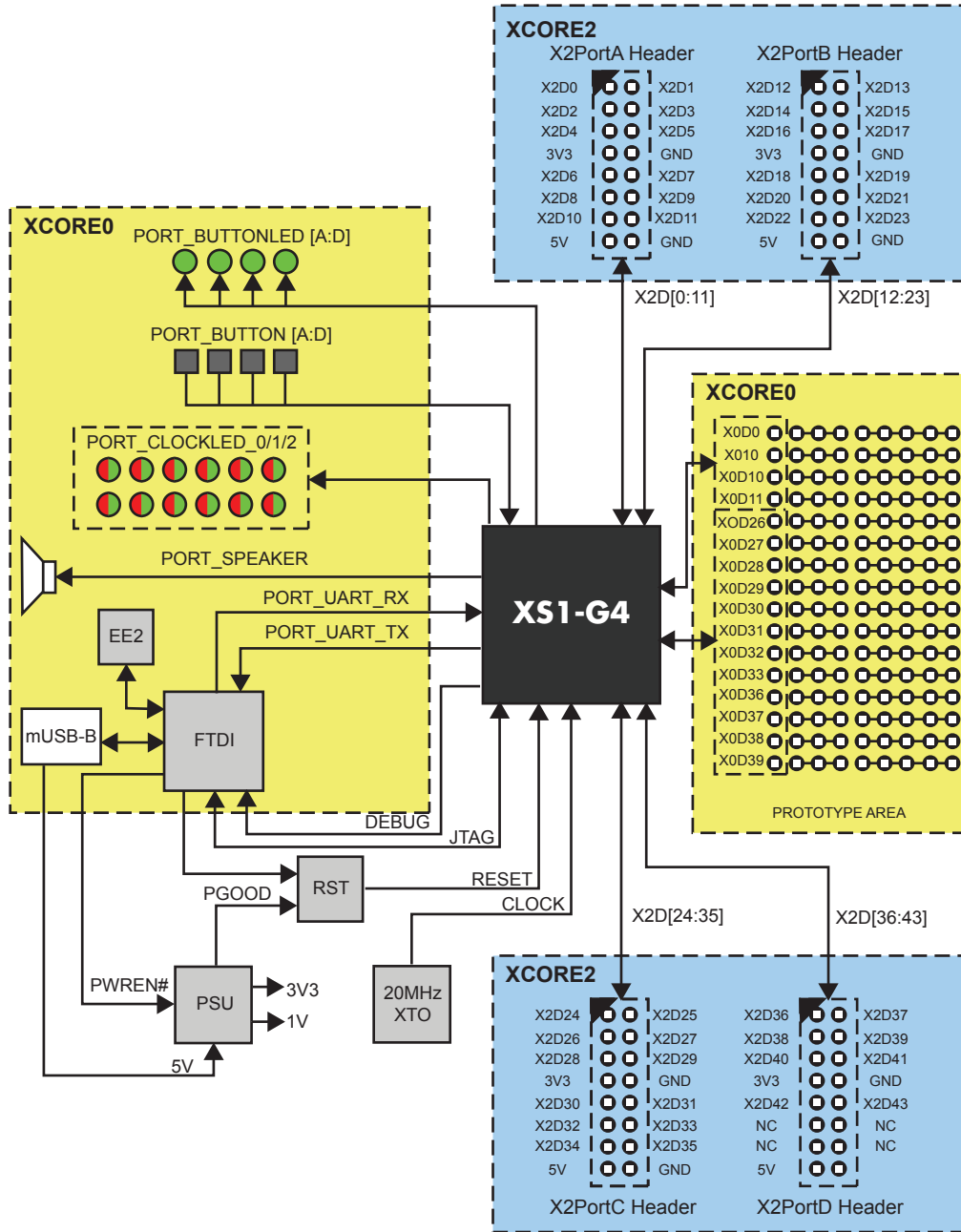


Figure 5 XC-1 Block Diagram

XCore XS1 Devices [1].

Information on the tools is available in the Development Tools User Guide [3].

Information on the XS1 architecture is available in the XS1 Architecture [5], Instruction Set [6], System [7] and Assembly [8] documents.

See also:

- <http://www.xmos.com>
- <http://www.xlinkers.org>

## References

- [1] Douglas Watt. Programming XC on XCore XS1 Devices. Website, 2009. <http://www.xmos.com/published/xcxsl>.
- [2] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.
- [3] Huw Geddes. Tools User Guide. Website, 2009. <http://www.xmos.com/published/xtools>.
- [4] XMOS Ltd. XC-1 Hardware Manual. Website, 2009. <http://www.xmos.com/published/xclhw>.
- [5] David May. XMOS XS1 Architecture. Website, 2008. <http://www.xmos.com/published/xs1-87>.
- [6] David May and Henk Muller. XMOS XS1 Instruction Set Architecture. Website, 2008. <http://www.xmos.com/published/xslinst87>.
- [7] David May and Ali Dixon and Ayewin Oung and Henk Muller. XS1-G System Specification. Website, 2008. <http://www.xmos.com/published/xsystem>.
- [8] Douglas Watt. XS1 Assembly Language Manual (8.7). Website, 2008. <http://www.xmos.com/published/xas87>.

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.

(c) 2009 XMOS Limited - All Rights Reserved