XDK Tutorial

(VERSION 9.7)

2009/07/01

Authors:

XMOS LTD.

1 Introduction

The XDK is an Event-Driven Processor development board based on the XMOS XS1-G4. It comprises a single XS1-G4, QVGA colour touch-screen, 96KHz stereo codec w/3.5mm jacks, 12 LEDs, 5 press-buttons, 10/100Mbit Ethernet, USB 2.0 PHY, JTAG and serial interfaces, and 128 I/O expansion pins.

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 through ports, details of which are available in a separate document [1].

The XMOS originated XC language [2] 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 the LEDs on your XDK and the XC language. It assumes that you are familiar with C [3]. In this tutorial you will:

- illuminate an LED on the XDK
- flash an LED connected to a single core
- implement a UART protocol and send the message “Hello World” to your host PC
- flash three LEDs connected to a single core concurrently
- flash three LEDs connected to a single core in sequence
- flash three LEDs connected to different cores in sequence
- flash all 12 LEDs on the XDK in a round robin sequence
- terminate your program when one of the XDK buttons is pressed

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 [4].

2 Illuminate an LED: `port`, `<:`

This part of the tutorial shows you how to use an XC port and an output statement to illuminate an LED on your XDK.

The XDK has 12 LEDs positioned in four LED windows. Each XCore is connected to three LED pins that control the LEDs in a single window. The LEDs are active low.

The following program illuminates a single LED on your XDK:

```
#include <platform.h>

out port led = PORT_LED_0_1;

int main(void){
  led <: 0;
  while (1);
  return 0;
}
```

The second line of this program declares a port variable `led` and initialises it with a generic port identifier `PORT_LED_0_1`. The `XDK.xn` file maps the `PORT_LED_0_1` identifier to the `XS1_PORT_1F` port which originates from the `xs1.h` header file. The `1F` refers to the pin width (1 bit) and name (F).

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:

```
led <: 0;
```

The value specified to the right of `<:` (`0`) is output to the port specified to its left (`led`). This value sets one of the LED pins low, causing the corresponding LED 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 XDK. (See the Development Tools User Guide [4] for details on compiling programs.) A single LED should illuminate in the top-left LED window.

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 on-off.

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;
```

Note that 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 an LED pin which, when executed in a loop, flashes an LED on-off. The complete program is shown below:

```
#include <platform.h>

#define PERIOD 20000000

out port led = PORT_LED_0_1;

int main(void) {
  timer tmr;
  unsigned ledOff = 1;
  unsigned t;
  tmr :> t;
  while (1) {
    led <: ledOff;
    ledOff = !ledOff;
    t += PERIOD;
    tmr when timerafter(t) :> void;
  }
  return 0;
}
```

✎ Compile and run this program on your XDK. A single LED should flash on-off in the top-left window of your XDK.

4 Flash multiple LEDs in parallel: `par`

This part of the tutorial shows you how to use the XC `par` statement to flash multiple LEDs in parallel on your XDK.

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

```
#include <platform.h>

#define PERIOD 20000000

out port led1 = PORT_LED_0_2;
out port led2 = PORT_LED_0_1;
out port led3 = PORT_LED_0_0;

void flashLED(out port led, int period);

int main(void) {
  par {
    flashLED(led1, PERIOD);
    flashLED(led2, PERIOD);
    flashLED(led3, PERIOD);
  }
  return 0;
}
```

The `flashLED` function flashes an LED connected to the specified port at the specified period.

✎ Implement the `flashLED` function (see Section 3). Compile and run this complete program on your XDK. Three LEDs should flash on-off in the top-left LED window.

5 Flash LEDs in alternating sequence: `chan`, `chanend`

This part of the tutorial shows you how to use XC channels to flash three LEDs on your XDK in round robin sequence.

An XC channel provides a synchronous, bidirectional link between two threads. A channel is declared using the `chan` keyword:

```
chan c;
```

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

```
void flashLED(chanend left, chanend right,
              out port led, int period, int m)
```

This modified declaration of the `flashLED` function is used to form a component of a token ring. The first two parameters are channel ends, the third is an LED port and the fourth is a Boolean value indicating whether or not the thread executing the function is the master thread. The master thread generates a token and inserts it into the ring; all other slave threads wait for a communication to be instigated.

A generalised token ring for all 12 LEDs is illustrated in Figure 1. Each thread receives the token, flashes its LED and then passes the token to the next thread in the ring.

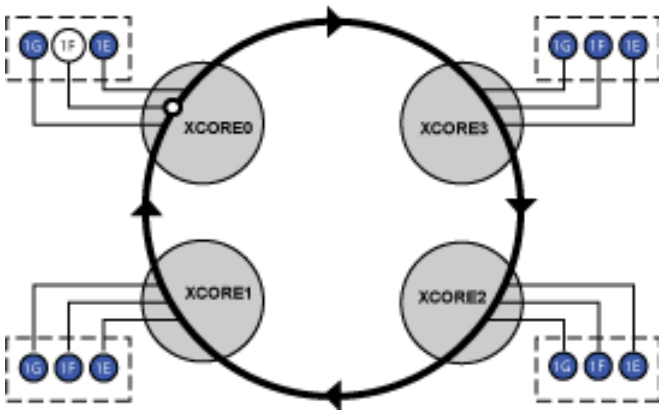

Figure 1 *LED Token Ring*

The body of the `flashLED` function is defined as follows:

```
#define TOKEN 1

void flashLED(chanend left, chanend right,
    out port led, int period, int m) {

    timer tmr;
    unsigned t;

    if (m)
        right <: TOKEN;

    while (1) {
        left :> int _;
        led <: 0;
        tmr :> t;
        tmr when timerafter(t+period) :> void;
        led <: 1;
        right <: TOKEN;
    }
    return;
}
```

This function first tests whether it is executing on the master thread and, if so, sends a token to the next thread in the ring. All threads in the token ring then repeatedly wait for the token, flash an LED and pass the token on to the next thread.

The XC input and output statements are used to pass the token between threads. Channels are synchronous (an input operation blocks until a matching output operation is ready), so exactly one thread has possession of the token at any one time.

The following `main` function declares three channels and passes them as arguments to three instances of the `flashLED` function, all of which are executed in parallel. This function is a simplification of the full ring illustrated in Figure 1, implemented on a single core.

```
int main(void) {
  chan c0, c1, c2;
  par {
    flashLED(c0, c1, led1, PERIOD, 1);
    flashLED(c1, c2, led2, PERIOD, 0);
    flashLED(c2, c0, led3, PERIOD, 0);
  }
  return 0;
}
```

✎ Compile and run this program on your XDK. Three LEDs should flash on-off in alternating sequence in the top-left LED window.

6 Flash LEDs connected to different cores: `on`

This part of the tutorial shows you how to use the XC `on` statement to implement a multicore program, in this case to flash LEDs in any of the four LED windows on your XDK. The LEDs in each of these four windows are connected to different cores on the XS1-G4.

The header file `platform.h` provides a declaration of the global variable `stdcore` for the target device, in this case an XS1-G4. This variable can be used with the `on` keyword to specify the location of port declarations, for example:

```
#include <platform.h>

on stdcore[0] : out port led1 = PORT_LED_0_2;
on stdcore[3] : out port led2 = PORT_LED_3_1;
on stdcore[2] : out port led3 = PORT_LED_2_0;
```

The `on` statement can also be used to specify the core on which each substatement in a parallel statement is placed. The example in Section 5 is easily modified so that each LED port and corresponding `flashLED` function is placed on a different core:

```
int main(void) {
  chan c0, c1, c2;
  par {
    on stdcore[0] :
      flashLED(c0, c1, led1, PERIOD, 1);
    on stdcore[3] :
      flashLED(c1, c2, led2, PERIOD, 0);
    on stdcore[2] :
      flashLED(c2, c0, led3, PERIOD, 0);
  }
  return 0;
}
```

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

✎ Compile and run this program on your XDK. Three LEDs should flash in alternating sequence in three separate windows.

✎ Modify this program so that each of the 12 LEDs on the XDK is flashed in a clockwise cycle. (See Figure 1.)

7 Terminate the program: `select`

This part of the tutorial shows you how to use the XC `select` statement to respond to a button on the XDK, in this case to safely terminate the program.

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 proceeded by the keyword `case` and its body must be terminated with a `break` or `return` statement.

The first task is to define a button listener thread and integrate it into the token ring. A `select` statement is used to wait for either a button to be pressed or the token to be received (highlighted in the example below). If a button is pressed then a check is made to determine whether it is the black button and, if so, a terminal token is inserted into the ring. After inserting the terminal token into the ring, the listener waits for it to cycle around the ring and then returns. The code for the button listener thread is shown below:

```
#define TERM  0
void listen(chanend left, chanend right) {
  int token = TOKEN;
  while(token == TOKEN) {
    select {
      case left :> token:
        break;
      case button when pinsneq(0xf) :> int x :
        /* check which button pressed */
        if ((x & 0x4) == 0) { // black button
          token = TERM;
          left :> int _; // wait for TOKEN
        }
        break;
    }
    right <: token; // pass token on
  }
  left :> int _; // wait for TERM
}
```

✎ Modify the `flashLED` function so that upon receiving a `TERM` token it passes this token to its neigbour and returns.

✎ Add a declaration for the `button` port 4F (PORT_BUTTON_0_3) and modify `main` by adding the button listener into the token ring. Both the button port and the button listener must be placed on core 0.

✎ Compile and run this program on your XDK. Pressing the black button should cause the program to terminate upon completion of the current flash cycle.

✎ Modify the button listener so that pressing one button changes the direction of the flash cycle.

✎ Modify the button listener to that pressing one button increases the speed of the flash cycle and pressing another reduces its speed.

8 Further Reading

Further information on the XC language can be found in the *Programming XC on XCore XS1 Devices* document [5].

Information on the XS1 architecture is available in the XS1 Architecturee [6], Instruction Set [7], System [8] and Assembly [9] documents.

See also:

- `http://www.xmos.com`
- `http://www.xlinkers.org`

References

[1] XMOS Ltd. XMOS XS1-G Port Map. Website, 2008. http://www.xmos.com/published/xs1-portmap.

[2] Douglas Watt and Richard Osborne and David May. XC Reference Manual (8.7). Website, 2008. http://www.xmos.com/published/xc87.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[4] Huw Geddes and Matt Fyles and Mike Wrighton and Douglas Watt. XMOS Tools User Guide. Website, 2009. http://www.xmos.com/published/xtools.

[5] Douglas Watt. Programming XC on XCore XS1 Devices. Website, 2009. http://www.xmos.com/published/xcxs1.

[6] David May. XMOS XS1 Architecture. Website, 2008. http://www.xmos.com/published/xs1-87.

[7] David May and Henk Muller. XMOS XS1 Instruction Set Architecture. Website, 2008. http://www.xmos.com/published/xs1inst87.

[8] David May and Ali Dixon and Ayewin Oung and Henk Muller. XS1-G System Specification. Website, 2008. http://www.xmos.com/published/xsystem.

[9] Douglas Watt. XS1 Assembly Language Manual (8.7). Website, 2008. http://www.xmos.com/published/xas87.