Application Note: AN00176

A startKIT tic-tac-toe demo

This application demonstrates several aspects of I/O on the startKIT and programming XMOS devices by allowing the user to play a game of tic-tac-toe on the startKIT.

The LED grid is used as the game board and the buttons and capacitive sensors are used for input.

Required tools and libraries

- xTIMEcomposer Tools - Version 14.0
- startKIT support library (lib_startkit_support) - Version 1.0.0

Required hardware

This application note is designed to run on the XMOS startKIT.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the XMOS GPIO library, the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the references appendix.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].

[1] http://www.xmos.com/published/glossary

1 Overview

1.1 Introduction

startKIT is a low-cost development board for the configurable xCORE multicore microcontroller products from XMOS. It's easy to use and provides lots of advanced features on a small, extremely low cost platform.

xCORE lets you software-configure the interfaces that you need for your system; so with startKIT you can configure the board to your match your exact requirements. Its 500MIPS xCORE multicore microcontroller has eight 32bit logical cores that perform deterministically, making startKIT an ideal platform for functions ranging from robotics and motion control to networking and digital audio.

startKIT also connects easily to your Raspberry Pi, allowing you to add real-time I/O and communication features to this popular computing platform, and to try out advanced applications for xCORE.

2 A startKIT tic-tac-toe game

The tic-tac-toe demo is a program that plays tic-tac-toe (also known as noughts and crosses) on the XMOS startKIT development board. It is provided as a demonstation program of how to program the device. The 3x3 display of LEDs shows the board status:

- Full LEDs: user player (marking a 0)
- Dimmed LEDs: computer player (marking a 1)

When it is the user's move, one of the LEDs flashes - this is a cursor and it can be moved by swiping the sliders. Pressing the button makes a move, and the computer player will make the next move.

The application consists of four tasks:

- The `startkit_gpio_driver` task drives the LEDs on the device (using PWM to make the lights glow at different levels of intensity), the capacitive sensors on the sliders and the button. It has three interface connections connected to it - one for the button, one for the LEDs and one for the slider.
- The game task which controls the game state. It is connected to the two player tasks and to the gpio task to drive the LEDs to display the game state.
- The `user_player` task which receives notifications from and sends commands to the game task. It also connects to the gpio task to read the sliders and buttons when the user player makes a move.
- The `cpu_player` task which receives notifications from and send commands to the game task. It uses an internal AI algorithm to determine what move to make.

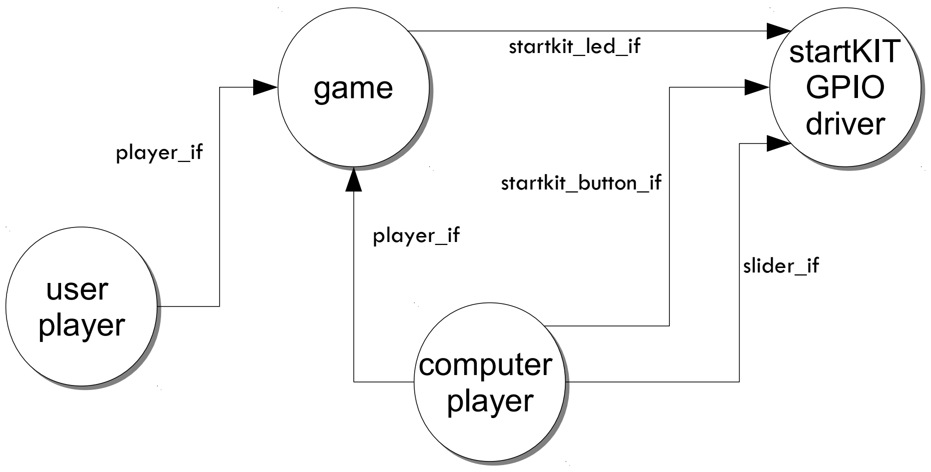

Figure 1: Application task diagram

The four tasks are spread across two logical cores. One logical core runs the gpio driver which needs to be responsive to the I/O pins. The other core runs the other three tasks which do not have real-time constraints and share the core via co-operative multitasking.

The main program consists of a par statement to run all the tasks in parallel with three tasks placed on the same core. The declarations are typedefs of interface types to connect the tasks together:

```
// The port structure required for the GPIO task
startkit_gpio_ports gpio_ports =
  {XS1_PORT_32A, XS1_PORT_4A, XS1_PORT_4B, XS1_CLKBLK_1};

int main() {
  startkit_button_if i_button;
  startkit_led_if i_led;
  slider_if i_slider_x, i_slider_y;
  player_if i_game[2];
  par {
    on tile[0].core[0]: game(i_game, i_led);
    on tile[0].core[0]: user_player(i_game[0],
                                    i_slider_x, i_slider_y, i_button);
    on tile[0].core[0]: computer_player(i_game[1]);
    on tile[0]: startkit_gpio_driver(i_led, i_button,
                                     i_slider_x, i_slider_y,
                                     gpio_ports);

  }
  return 0;
}
```

2.1 The game task and player interface

The game task controls the board state and blinking cursor, displaying them via the LED connection to the GPIO driver task. The key interface is between the game task and the two player tasks. This includes functions for getting and updating the current game state.

The game task uses notifications to inform the player tasks that a move is required:

```
typedef enum board_val_t {
  BOARD_EMPTY,
  BOARD_X,
  BOARD_O,
} board_val_t;

typedef interface player_if {
  // This function will fill in the supplied board array with the
  // current game state.
  void get_board(char board[3][3]);

  // Set the user cursor to the specified position.
  void set_cursor(unsigned row, unsigned col);

  // Clear the user cursor from the board.
  void clear_cursor();

  // This function can be called by players to determine whether they
  // are the X piece or the O piece.
  board_val_t get_my_val();

  // This notification will be signalled by the game to the player when
  // a move is required from the player.
  [[notification]] slave void move_required();

  // This function is called by the player to make a move in the specified
  // position.
  [[clears_notification]] void play(unsigned row, unsigned col);
} player_if;

// This task controls the game state providing two connections to the
// two players of the game.
[[combinable]]
void game(server player_if players[2], client startkit_led_if i_led);
```

Note that game task is *combinable* allowing it to share processing time with other combinable tasks in between reacting to events.

2.2 The user player task

The user player task connects to the game task and the gpio task. It is either in a playing state or idle state. When it gets a move request notification from the game task, it moves into the playing state and sets up the cursor in the game task. Whilst in the playing state it reacts to slider and button events to move the cursor and complete the game move.

```
[[combinable]]
void user_player(client player_if i_game,
                 client slider_if i_slider_x,
                 client slider_if i_slider_y,
                 client startkit_button_if i_button)
{
```

The task has some local state - a variable to determine whether it is in a playing state or not, x and y variables to store the current position of the cursor and a local copy of the board state:

```
int playing = 0;
int x = 0, y = 0;
char board[3][3];
```

The main body of the task consists of a `while (1) select` loop:

```
while (1) {
  select {
```

The first case in the select reacts when the game tasks requests a move is played. This causes the player task to enter the playing state. At this point the task takes a copy of the board state and sets up the cursor by interacting with the game task:

```
case i_game.move_required():
  // Get a local copy of the board state
  i_game.get_board(board);
  // Find an empty place to place the cursor
  int found = 0;
  for (int i = 0; i < 3 && !found; i++) {
    for (int j = 0 ; j < 3 && !found; j++) {
      if (board[i][j] == BOARD_EMPTY) {
        x = i;
        y = j;
        found = 1;
      }
    }
  }
  i_game.set_cursor(x, y);
  playing = 1;
  break;
```

If the button is pressed it causes an event on the connection to the gpio driver. The following case reacts to this event and if the task is in the playing state and the cursor is at an empty space on the board, it calls the `play` function over the connection to the game task to play the move, and then leave the playing state:

```
case i_button.changed():
  button_val_t val = i_button.get_value();
  if (playing && val == BUTTON_DOWN) {
    if (board[x][y] == BOARD_EMPTY) {
      // Make the move
      i_game.clear_cursor();
      i_game.play(x, y);
      playing = 0;
    }
  }
  break;
```

The task also reacts to changes in the slider. In this case it moves the cursor if the slider notifies the task of a LEFTING or RIGHTING event (indicating that the user has swiped left or right):

```
case i_slider_x.changed_state():
  sliderstate state = i_slider_x.get_slider_state();
  if (!playing)
    break;
  if (state != LEFTING && state != RIGHTING)
    break;
  int dir = state == LEFTING ? 1 : -1;
  int new_x = x + dir;
  if (new_x >= 0 && new_x < 3) {
    x = new_x;
    i_game.set_cursor(x, y);
  }
  break;
```

The case to handle the vertical slider is similar. Handling move requests, slider swipes and button presses completes the player task.

2.3 The computer player task

The computer player uses an auxiliary function to find a move for the computer to play. It fills the best_i and best_j reference parameters with the position of the best move based on an AI algorithm that searches possible future move combinations. The parameter board is the current board state and the parameter me indicates which type of piece the computer is playing.

```
static void find_best_move(char board[3][3],
                           int &best_i,
                           int &best_j,
                           board_val_t me);
```

With this function, the computer player task is quite simple. It just waits for the game tasks to request a move, gets a copy of the board state, determines the best move to play and then communicates back with the game state playing the move.

```
[[combinable]]
void computer_player(client player_if game)
{
  while (1) {
    select {
    case game.move_required():
      char board[3][3];
      int i, j;
      game.get_board(board);
      find_best_move(board, i, j, game.get_my_val());
      game.play(i, j);
      break;
    }
  }
}
```

APPENDIX A - Launching the demo application

Once the demo example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer studio we can execute the application on the startKIT.

Once built there will be a `bin` directory within the project which contains the binary for the xCORE device. The xCORE binary has a XMOS standard .xe extension.

A.1 Launching from the command line

From the command line we use the `xrun` tool to download code to both the xCORE devices. If we change into the bin directory of the project we can execute the code on the xCORE microcontroller as follows:

```
> xrun AN00176_startKIT_tic_tac_toe.xe          <-- Download and execute the xCORE code
```

Once this command has executed the application will be running on the startKIT and the game should start.

A.2 Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio we use the run mechanism to download code to xCORE device. Select the xCORE binary from the bin directory, right click and go to Run `Configurations`. Double click on xCORE application to create a new run configuration and then select Run.

Once this command has executed the application will be running on the startKIT and the game should start.

APPENDIX B - References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

APPENDIX C - Full source code listing

C.1 Source code for main.xc

```
// Copyright (c) 2015, XMOS Ltd, All rights reserved
#include <xs1.h>
#include <platform.h>
#include "game.h"
#include "user_player.h"
#include "computer_player.h"

/**
  The tic-tac-toe demo is a program that plays tic-tac-toe (also known as
  noughts and crosses) on a XMOS startKIT development board. It is provided
  as a demonstation program of how to program the device.
  The 3x3 display of LEDs shows the board status:

    - Full LEDs: user player (marking a 0)
    - Dimmed LEDs: computer player (marking a 1)

  When it is the user's move, one of the LEDs flashes - this is a cursor and
  it can be moved by swiping the sliders. Pressing the button makes a move,
  and the computer player will make the next move.

  The application consists of four tasks:

    * The ``startkit_gpio_driver`` task drives the LEDs on the device (using
      PWM to make the lights glow at different levels of intensity), the
      capacitive sensors on the sliders and the button. It has three
      interface connections connected to it - one for the button, one for the
      LEDs and one for the slider.
    * The ``game`` task which controls the game state. It is connected to the
      two player tasks and to the gpio task to drive the LEDs to display the
      game state.
    * The ``user_player`` task which receives notifications from
      and sends commands to the game task. It also connects to the
      gpio task to read the sliders and buttons when the user player makes
      a move.
    * The ``cpu_player`` task which receives notifications from and send
      commands to the game task. It uses an internal AI algorithm to determine
      what move to make.

  .. aafig::

                `gpio_driver`    `      `user player task` `cpu player task`
                +--------+              +-------+          +-------+
                |        | `button_if`  |       |          |       |
                |        +<-------------+       |          |       |
          I/O<--+        | `slider_if`  |       |          |       |
                |        +<-------------+       +--+    +--+       |
                |        |              |       |  |    |  |       |
                +--+-----+              +-------+  |    |  |       |
                   ^                               |    |  +-------+
                   |           `game task`         |    |
                   |           +-------+           |    |
                   |           |       |           |    |
                   +-----------+       +<----------+    |
                   `led_if`    |       |  `game_if`     |
                               |       +<---------------+
                               |       |
                               +-------+

  The four tasks are spread across two logical cores. One
  logical core runs the gpio driver which needs to be responsive to the I/O
  pins. The other core runs the other three tasks which do not have real-time
  constraints and share the core via co-operative multitasking.

**/

/**
  The main program consists of a ``par`` statement to run all the tasks in
  parallel with three tasks placed on the same core. The declarations
  are typedefs of interface types to connect the tasks together.
*/

// The port structure required for the GPIO task
startkit_gpio_ports gpio_ports =
  {XS1_PORT_32A, XS1_PORT_4A, XS1_PORT_4B, XS1_CLKBLK_1};

int main() {
  startkit_button_if i_button;
  startkit_led_if i_led;
  slider_if i_slider_x, i_slider_y;
  player_if i_game[2];
  par {
    on tile[0].core[0]: game(i_game, i_led);
    on tile[0].core[0]: user_player(i_game[0],
                                    i_slider_x, i_slider_y, i_button);
    on tile[0].core[0]: computer_player(i_game[1]);
    on tile[0]: startkit_gpio_driver(i_led, i_button,
                                     i_slider_x, i_slider_y,
                                     gpio_ports);

  }
  return 0;
}
```