Application Note: AN00112

eCos on xCORE

Running eCos RTOS on xCORE enables you to use your existing or any third party RTOS application stack on an xCORE. You can select from a wide range of XMOS IP libraries and add additional peripherals to suit your application needs and reduce time to market dramatically.

This application note uses a ported version of eCos RTOS running on an xCORE and demonstrates a simple eCos application that context switches between two eCos threads.

Required tools and libraries

- xTIMEcomposer Tools - Version 13.1.0

Required hardware

This application note is designed to run on an XMOS xCORE General Purpose (L-series) device.

The example code provided with the application has been implemented and tested on the xCORE L-series sliceKIT core board 1V2 (XP-SKC-L2) but there is no dependency on this board and it can be modified to run on any development board which uses an xCORE General Purpose (L-series), xCORE-USB series or xCORE-Analog series device.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the XMOS tool chain and the xC language. Documentation that is not specific to this application note is listed in the references appendix.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].
- For information on eCos RTOS please see the eCos home page[2].

[1] http://www.xmos.com/published/glossary
[2] http://ecos.sourceware.org/

1 Overview

1.1 Introduction

eCos is an open source RTOS intended for embedded applications. It is available under the terms of GNU GPL license. eCos has a highly flexible configuration system which can be tailored for the precise application needs. eCos is available for a wide range of targets. eCos RTOS is ported to xCORE in order to use your existing RTOS applications and application stack easily on xCORE.

1.2 Block diagram

In xCORE multicore microcontrollers, eCos is run as a separate task on a logical core. An eCos application is run on this eCos task, sparing remaining cores to contain other application logic as per your application needs. As an example, if you want to add Ethernet, USB or Audio support to your existing eCos application, your eCos application is run on an eCos task; using XMOS libraries, you can add Ethernet, USB and Audio functionality to run as separate tasks and share data among these tasks.

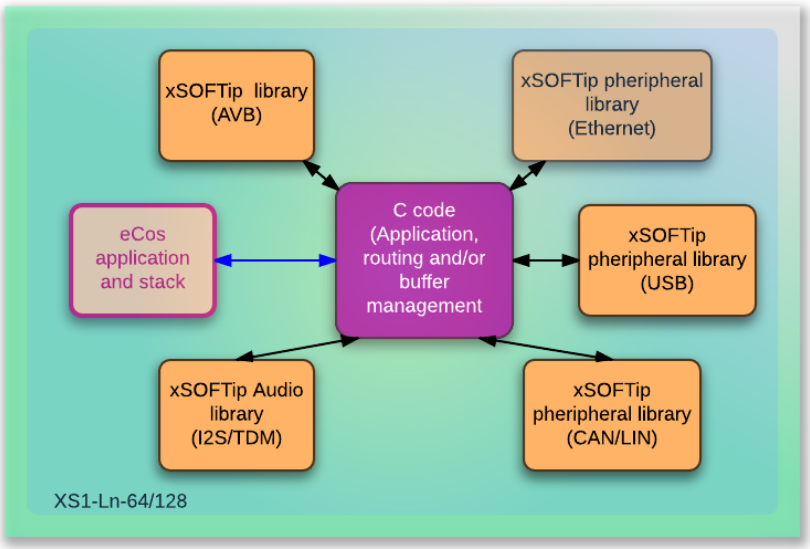

Figure 1: Block diagram of eCos application running on xCORE

1.3 Features of eCos port on xCORE

All the necessary key features in the eCos kernel are ported to xCORE. These include:

- OS Threads
- Scheduler
- Interrupt handling
- Context switching
- Mutexes

In addition to the above, HAL APIs are ported as needed to run the eCos kernel. Memory footprint is approximately 25 Kbyte of which 10 Kbyte is user stack. Only Bitmap scheduler is supported currently. More details on the eCos port and its feature is available in the section `APPENDIX C - eCos porting guide`.

2 Simple eCos application example

This application demonstrates how to:

- run eCos RTOS on a logical core
- develop a simple multi-threaded eCos RTOS application using eCos APIs

eCos is run as a task and an RTOS application runs on top of eCos. This application spawns two RTOS threads. Each thread wakes up after a random sleep period, prints a message and goes back to sleep. This application demonstartes the usage of thread creation, schdeuler context switching, interrupts, mutex objects.

Figure 2 shows the task and communication structure for a typical eCos RTOS application communicating with other xCORE tasks. c_task_x are xCONNECT channels to communicate between different xCORE tasks[3].

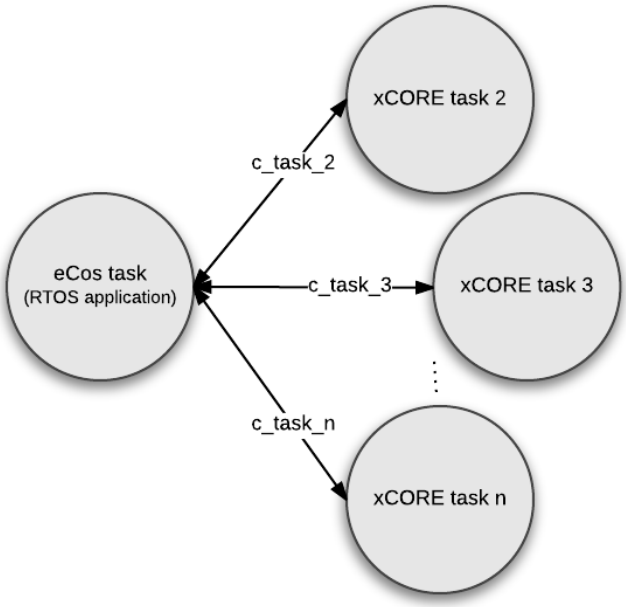

Figure 2: A typical task diagram of an eCos RTOS application co-existing with other xCORE tasks

[3]This is an illustrative block diagram depicting eCos task communicating with other tasks; however the application example uses only the eCos task

2.1 The application main() function

Below is the source code for the main function of this application, which is taken from the source file `main.xc`

```
int main(void)
{

  par
  {
    on tile[0]: cyg_start();
  }
  return 0;
}
```

`main` function has a par statement in which the eCos task is run using the `cyg_start` function. This function has the application logic to create two eCos threads and start the scheduler.

2.2 Two threads application example

You will find this application available in the examples directory of the standard eCos distribution. Below is the description of the source code from the source file `twothreads.c`:

You can access the eCos APIs in your source code using the kapi.h header file:

```
#include <cyg/kernel/kapi.h>
```

Each eCos thread requires a stack and an entry point function to operate. Each thread is assigned a 4Kbyte stack and `simple_program` is used as an entry point function.

```
cyg_thread thread_s[2];         /* space for two thread objects */

char stack[2][4096];            /* space for two 4K stacks */

/* now the handles for the threads */
cyg_handle_t simple_threadA, simple_threadB;

/* and now variables for the procedure which is the thread */
cyg_thread_entry_t simple_program;
```

`cyg_start` is the application start-up routine in which a mutex object (cliblock) and two threads (Thread A and Thread B) are created using eCos APIs. `cyg_thread_resume` is the API call to move an eCos thread into a runnable state. eCos scheduler is started using *cyg_scheduler_start* once the threads are ready.

```
void cyg_start(void)
{
  hal_hardware_init();
  printstr("Entering twothreads' cyg_user_start() function\n");

  cyg_mutex_init(&cliblock);

  cyg_thread_create(4, simple_program, (cyg_addrword_t) 0,
                    "Thread A", (void *) stack[0], 4096,
                    &simple_threadA, &thread_s[0]);

  cyg_thread_resume(simple_threadA);

  cyg_thread_create(5, simple_program, (cyg_addrword_t) 1,
                    "Thread B", (void *) stack[1], 4096,
                    &simple_threadB, &thread_s[1]);

  cyg_thread_resume(simple_threadB);

  cyg_scheduler_start();
```

simple_program is used as the common entry point function for both the threads. When any of the Thread A or Thread B is scheduled, it picks a random delay (in order to represent sleep duration), prints its value and goes into sleep for this clock ticks. You will also notice that the printf calls are protected by the mutex lock that we created earlier.

```
void simple_program(cyg_addrword_t data)
{
  int message = (int) data;
  int delay;

  printstr("Beginning execution; thread data is ");
  printintln(message);
#define SIM_RUN 1
#ifndef SIM_RUN
  cyg_thread_delay(200);
#else
  cyg_thread_delay(10);
#endif //SIM_RUN

  for (;;) {
#ifndef SIM_RUN
    delay = 200 + (rand() % 50);
#else
    do {
        delay = rand() % 5;
    } while (delay == 0);

#endif //SIM_RUN
    /* note: printf() must be protected by a
       call to cyg_mutex_lock() */
    cyg_mutex_lock(&cliblock); {
    printstr("Thread ");
    printint(message);
    printstr(": and now a delay of ");
    printint(delay);
    printstrln(" clock ticks");

      //printf("Thread %d: and now a delay of %d clock ticks\n",
           //  message, delay);
    }
    cyg_mutex_unlock(&cliblock);

   cyg_thread_delay(delay);
  }
} //end: simple_program
```

APPENDIX A - Demo hardware setup

To run the demo, connect the xTAG-2 debug adapter to the sliceKIT core board. Connect xTAG-2 debug adapter to your development PC.

On the sliceKIT core board ensure that the xCONNECT LINK switch is set to ON, as per the image, to allow xSCOPE to function. The use of xSCOPE is required in this application so that the print messages that are generated on the device as part of the demo does not disturb the real-time behaviour of the device application.

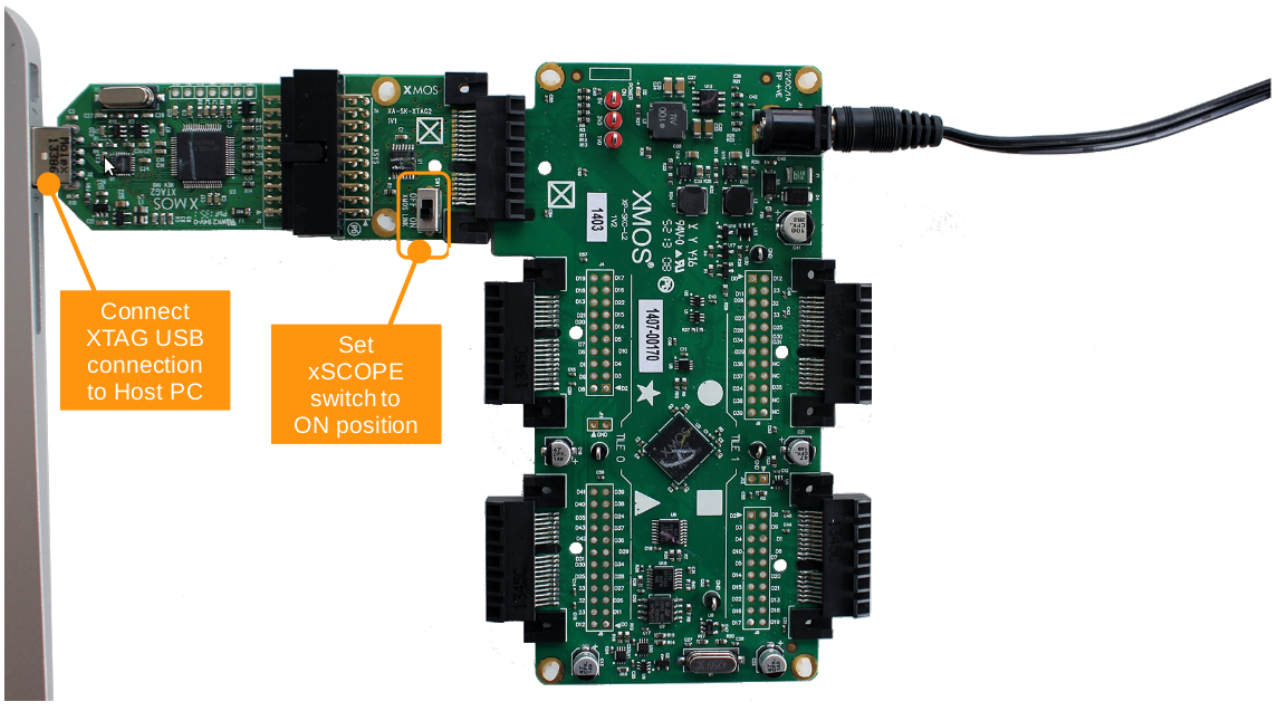

Figure 3: XMOS xCORE-L16 sliceKIT set up for eCos demo application

APPENDIX B - Launching the demo device

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

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.

B.1 Launching from the command line

From the command line, use the `xrun` tool to download code to the xCORE device. Navigate to the bin directory of the project and execute the code on the xCORE microcontroller as follows:

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

Once this command is executed, you will see the following text in the console window:

```
Entering twothreads' cyg_user_start() function
Beginning execution; thread data is 0
Beginning execution; thread data is 1
Thread 0: and now a delay of 3 clock ticks
Thread 1: and now a delay of 3 clock ticks
Thread 0: and now a delay of 2 clock ticks
Thread 1: and now a delay of 4 clock ticks
Thread 0: and now a delay of 3 clock ticks
Thread 1: and now a delay of 2 clock ticks
```

B.2 Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio, use the run mechanism to download code to xCORE device. Select the xCORE binary from the bin directory, right click and then follow the instructions below:

- Select **Run Configuration**.
- Enable xSCOPE in Target I/O options:

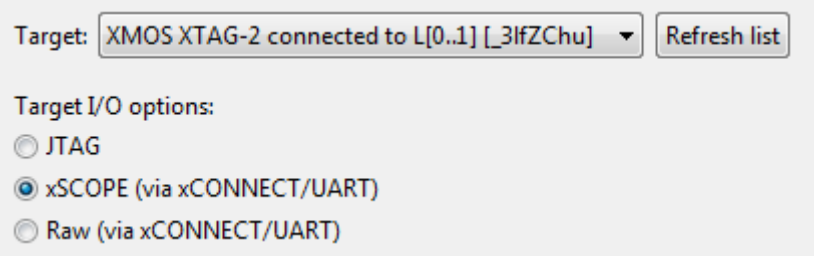

Figure 4: xTIMEcomposer xSCOPE configuration

- Navigate to **XScope** tab and then select **Offline [XScope] Mode**
- Click **Apply** and then **Run**.

When the processor has finished booting you will see the following text in the xTIMEcomposer console window:

```
Entering twothreads' cyg_user_start() function
Beginning execution; thread data is 0
Beginning execution; thread data is 1
Thread 0: and now a delay of 3 clock ticks
Thread 1: and now a delay of 3 clock ticks
Thread 0: and now a delay of 2 clock ticks
Thread 1: and now a delay of 4 clock ticks
Thread 0: and now a delay of 3 clock ticks
Thread 1: and now a delay of 2 clock ticks
```

APPENDIX C - eCos porting guide

C.1 Building eCos for xCORE

eCos is based on component architecture. It contains a number of packages which can be enabled or disabled using `ecosconfig` tool. This tool checks for appropriate dependencies and makes available a configuration for a specified hardware. ecosconfig tool is not yet supported to build for xCORE architecture. Instead eCos configuration information is made available as ecos_xcore.ecc file which can be used in a manual build. This file is used as a baseline to create source package used in `module_ecos`. eCos is made available as a software module which can be built by the XMOS tool chain. Any xCORE application can use this eCOS module in a similar manner as detailed in this application example.

C.2 Customization of eCos port on xCORE

This section describes some details about the adaptations done to eCos source code in order to use eCos on xCORE.

C.2.1 eCos boot sequence

The eCos boot sequence is modified to suit xCORE. Traditional eCos way set up the reset vector, handles exceptions and includes C libraries in addition to HAL and kernel related features. In xCORE, CRT (C-Runtime Library) handles the start up code, exceptions and C libraries. Hence the boot sequence is modified to utilize all the features already available in xCORE.

- xCORE specific HAL hardware initialization includes enabling interrupts, setting up a handler for hardware timer and setting up a separate stack for handling interrupts
- Clock initialization: this involves setting the hardware timer to interrupt at a specified period. Using current setting, eCos scheduler will be interrupted once every 10000 timer ticks
- cyg_start is the kernel entry point function. This is where the HAL to kernel transition takes place
- cyg_user_start is the beginning of user application
- cyg_scheduler_start: the scheduler is invoked at this point. The scheduler that is started is a bitmap scheduler. The job of the scheduler is to select the appropriate thread for execution and to control the effects of interrupts on thread execution

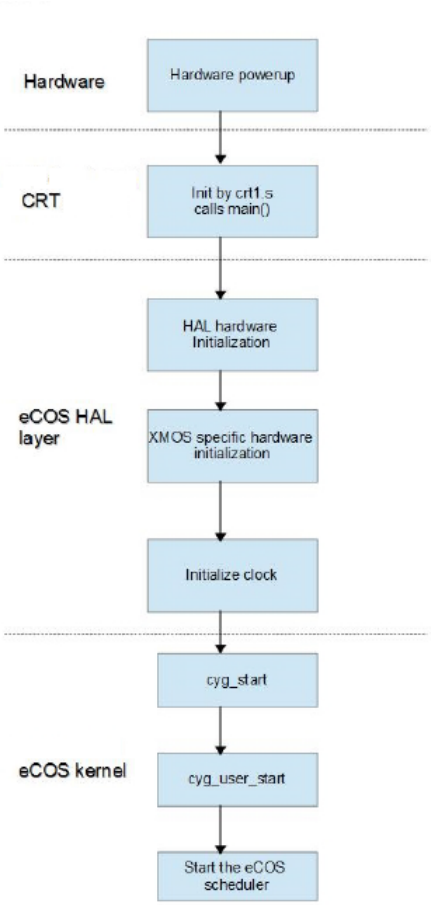

Figure 5: eCos boot sequence for xCORE

C.2.2 Interrupts

The interrupt controller in the current port can handle only one timer interrupt in order to support the RTOS scheduling. Figure 6 shows the execution flow when an interrupt occurs. The sequence of execution is explained below:

1. A thread is currently in execution
2. External hardware interrupt occurs
3. At the hardware level the handler for the specific timer interrupt is fetched
4. Timer interrupt handler starts execution
5. The context of the thread that was executing when the interrupt happened, is saved
6. The installed timer ISR (in this case RTC ISR) is serviced. When the ISR is invoked, the scheduler is locked so that rescheduling does not happen
7. The interrupt handler calls `interrupt_end` in the kernel
8. If there exists a Delayed service routine (DSR) for this interrupt, it is posted, so that it gets called later
9. The scheduler gets unlocked so that DSR and thread execution can resume
10. DSR gets executed. In case of timer interrupt, the eCos ticks are incremented in case of DSR. If an alarm object needs to be invoked at a particular tick, it happens here. It might be possible that the

thread that was interrupted when the interrupt happened might not be the thread that is restored by the scheduler

11. At the end of the interrupt, the scheduler decides which thread should execute and the context of that thread is restored
12. The thread chosen by the scheduler resumes execution

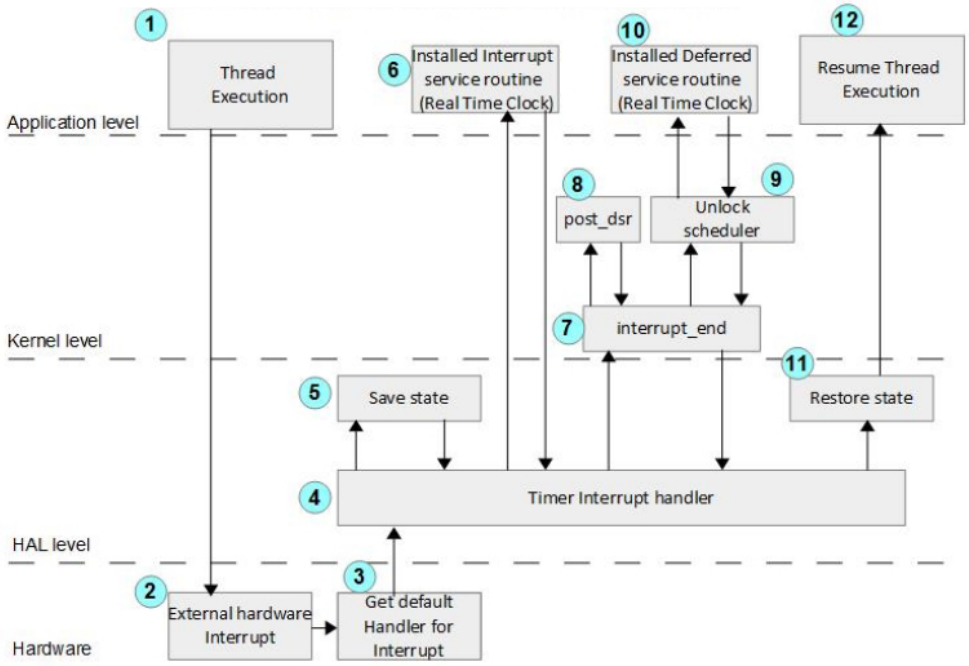

Figure 6: eCos interrupt handling

C.2.3 OS threads

A thread is a single flow of execution in a program. Each thread defined in an eCos system contains its own context or workspace to perform its actions and a priority level to execute. The scheduler decides which thread to run at any given point of time. The stack allocated to threads is passed as a parameter at the time of creation of thread. HAL_THREAD_INIT_CONTEXT is the HAL API used to initialize the context. A thread's context is stored at the bottom of its own stack so that it can be loaded later. HAL_THREAD_LOAD_CONTEXT is the API used to load the context of the thread that is ready to be executed. These APIs are supported to suit xCORE.

C.2.4 Context Switching

Context switching is the process of storing the state of an active process and resuming the state of a process that is to be executed. Context switching is performed by the scheduler. Context switching happens in the following cases:

- A thread whose priority is greater than the current active thread is ready to be executed
- A thread exits from the system, hence the scheduler needs to find the next thread to execute
- A thread goes into sleep by calling cyg_thread_delay. Hence the next thread needs to be invoked
- When interrupt occurs, tick counter increments and finds that an alarm objects (a thread) needs to be invoked

When interrupt occurs, the context of the current thread is saved in hal_saved_registers, a global data structure of type HAL_SavedRegisters. This is added in this port. While servicing the interrupt if it is found that a higher priority thread needs to be invoked, the status of the current thread saved in hal_saved_registers is flushed to the bottom of its stack. The context of new thread is loaded in hal_saved_registers. If not, contents of hal_saved_registers remains unaltered. While returning from interrupt, the context of the thread stored in hal_saved_registers is restored.

C.2.5 Interrupt stack

eCos allows to use a separate stack for handling ISR. Changing the stack, during interrupts the eCos-way was very complicated and involves multiple instructions. Instead this port uses the kernel stack feature in xCORE to handle interrupts. It is done using single instructions: kentsp (to switch to kernel stack) and krestsp (to return from kernel stack). The complete interrupt handling is done on the kernel stack.

APPENDIX D - 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

eCos User Guide

http://ecos.sourceware.org/docs-latest/user-guide/ecos-user-guide.html

APPENDIX E - Source code listing

E.1 Source code for main.xc

```
// Copyright (c) 2014, XMOS Ltd., All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

/*
 ================================================================================
 Name        : OS test - threads
 Description : Test
 ================================================================================
 */
#include <xs1.h>
#include <print.h>
#include <platform.h>
#include <stdio.h>
extern void cyg_start(void);
/**
 * Top level main that spawns the OS scheduler
 */

int main(void)
{

  par
  {
    on tile[0]: cyg_start();
  }
  return 0;
}
```