AC 2007-910: A SIMPLE MULTITASKING LIBRARY FOR STUDENT PROJECTS AND INTRODUCING EMBEDDED OPERATING SYSTEMS PRINCIPLES

Jonathan Hill, University of Hartford

Dr. Jonathan Hill is an assistant professor in the College of Engineering, Technology, and Architecture (CETA) at the University of Hartford, Connecticut (USA). Ph.D. and M.S. from Worcester Polytechnic Institute (WPI) and B.S. from Northeastern University. Previously an applications engineer with the Networks and Communications division of Digital Corporation. His interests involve embedded microprocessor based systems.

A Simple Multitasking Library for Student Projects and Introducing Embedded Operating Systems Principles

Abstract

The cxlib multitasking library is written for teaching embedded microprocessor principles to electrical and computer engineering students, serving as a stepping stone toward real time operating systems. The students also use cxlib in their projects. The library supports cooperative multitasking and a recent change allows for preemptive round-robin scheduling. The cxlib library was first written during the spring 2002 semester as the author surveyed the literature for material to use in a new course.

The intent of the cxlib library is to provide simple multitasking that students themselves can use to develop example intermediate systems. The library is intentionally Spartan, giving ample wiggle room for adding features or making significant architectural changes. The cxlib library is currently written for the Freescale 68HC12 microprocessor using the Metrowerks CodeWarrior tool chain, but it should be portable to other systems.

As a teaching tool, cxlib can be introduced in three hours of lecture time. In presenting cxlib there typically is a lively discussion regarding what is meant by context. The library is nearly transparent, helping students to see the workings of a context switch. Unlike a simple loop structure, cxlib provides a means to dynamically create and destroy tasks. Also cxlib can serve as a vehicle to introduce advanced concepts such as semaphores. The cxlib library is non-commercial open-source software.

Introduction

During the spring 2002 semester a new graduate level course in embedded microprocessor systems was developed for the Electrical and Computer Engineering Department at the University of Hartford. The type of operating system found in many small embedded systems is quite different from most desktop or modern operating systems. Rather than being an entity that is apart from the application, the operating system takes the form of a library that is part of the application. As pointed out by Silbershatz, Galvin, and Gagne[1] such operating systems are often associated with so-called real-time operating systems. Some authors including Labrosse[7] use the term executive to refer to such systems.

The cxlib library was written for the perceived a need for examples of intermediate systems using either cooperative multitasking or preemptive round-robin scheduling, sometimes called timesharing. As a teaching tool cxlib can be introduced in three hours of lecture time. The cxlib library is useful in three ways. First, it is an example that demonstrates topics present in nearly all operating systems. In particular the context switch which is demonstrated. Secondly, advanced topics such as semaphores and mailboxes are best understood in a given situation. The cxlib library allows students to study and modify such code, allowing for a deductive presentation. Finally, cxlib is a stepping stone leading toward real-time operating systems.

While some students first avoid cxlib, after an introduction they nearly all remark how easy it is to use. Students discover how using tasks helps to organize and simplify their code. Beyond classroom examples, students have used cxlib in class projects, senior capstone projects, and graduate research. Projects have included home alarm systems, trivial traffic light controllers, a simple two-phase motor controller, a simple acquisition system supporting the classic Tektronix 40xx graphics format, and a keypad scanner with Morse code sounder. Details regarding cxlib are available at the project webpage[2]. The embedded microprocessor course currently uses the Freescale 68HC12 processor along with the Metrowerks CodeWarrior software development tools. Among the references for the 68HC12 consider Doughman[4] as well as Barrett and Pack[5]. Students are provided with a tutorial[3] to the Metrowerks tools for the 68HC12.

Taxonomy and Justification for the cxlib Library

During the spring 2002 semester the author surveyed the literature for material for a new course in embedded microprocessor systems and tended to find an all-or-nothing attitude regarding embedded operating systems. This is a difficult situation as many embedded systems do not fit either extreme category. In regards to content, the University of Hartford Computer Science Department already provides a course in modern operating systems and the author already teaches an introductory course in microprocessors.

This observation led to a discovery. Without developing a history course, to aid student learning there is need for a way to crystallize and present students with a number of shades of gray in fairly broad strokes; this suggests taxonomy. To construct taxonomy of embedded operating system types consider the following mechanisms and characteristics:

- Presence or lack of elementary constructs to mimic multiprogramming or multitasking
- Use of a context switch – The context is a list of all the data structures swapped to change from one running task to another. In a number of small processors, performing a context switch or swap is difficult or impossible.
- Use of interrupts – Invoking an interrupt in most system creates a temporary new context that exists as long as the interrupt service routine is executing
- Context switch initiation – Systems in which only a task itself can yield the CPU are called cooperative. Systems using an interrupt to force a context switch are preemptive.
- Scheduling policy – Given a context switch, some policy decides what task will run. Such a policy can be trivial as in a list, a strict priority of the tasks, or other rules.

The following taxonomy is made in broad brush strokes, to aid students learning about embedded microprocessor operating systems, and to foster open discussion of embedded microprocessor operating systems.

- Type 0 – There is no recognizable operating system or structure present. In such systems spin locks are widely used as the synchronization mechanism.

- Type 1 – A single context is used to execute a series of functions. This is what Peatman[1] and Labrosse[7] refer to as a super-loop and Simon[8] refers to as Round-Robin. For processors not capable of performing a context switch, this is the highest system type that can be implemented.

- Type 2 – A context is created for each task. Only cooperative context switching is performed and a list scheduling policy is used. This is cooperative multitasking.

- Type 2x – This is a degenerate case. A non-trivial scheduling policy is used with cooperative context switching. Without a means to enforce the scheduling policy, the task response time cannot be guaranteed.

- Type 3 – A periodic interrupt preempts the current task, dividing time into slices. With list type scheduling policy, all tasks execute with the same priority.

- Type 4 – Preemptive context switching and a non-trivial scheduling policy. When used with strict priority based scheduling this is usually what is meant by the phrase real-time operating system. The response time for the highest priority task can be bounded.

- Higher types – Just keeping adding features.

There are numerous examples of type 0 and type 1 systems. The use of the super-loop program structure is common for very small systems. Likewise, there are plenty of type 4 and higher type systems. Numerous commercial and open-source real time operating systems exist. Likewise, some embedded systems use a modern operating system such as Linux or Windows.

Despite historical examples of type 2 and type 3 systems, the number in class-room ready form is rare. After forming the taxonomy, came the realization of the need for example intermediate systems. This is the justification for cxlib. Such a library is educational and provides a stepping stone to higher system types. At least one project is assigned in the embedded systems course involving the use of cxlib to write a cooperative multitasking system.

The cxlib library was initially inspired by a type 2 system written by the author to service the correlators in a Global Positioning System (GPS) receiver where each correlator is represented by its own task. A simple scheduler gives each task an opportunity to execute in a periodic fashion. The GPS receiver was developed in an 80486 based PC along with the uC/OS real-time kernel[7]. Software development was performed with the Borland C Compiler and Borland Turbo Assembler. In contrast, cxlib was developed for the 68HC12 processor using Metrowerks CodeWarrior, but should be portable to other systems. Early versions of cxlib only supported type 2 cooperative multitasking. As of version 0.3, cxlib can be used with type 2 or type 3 systems.

The cxlib Structure

Figure 1 below is a view of the cxlib structure. Following the curved line starting at the entry point, control is passed to a first task which in turn passes control to the next. From the outside, cxlib looks similar to a super-loop structure. However in looking inside an individual task, the view is quite different. Each task calls a function to yield the CPU and is relatively unaware of the other tasks.

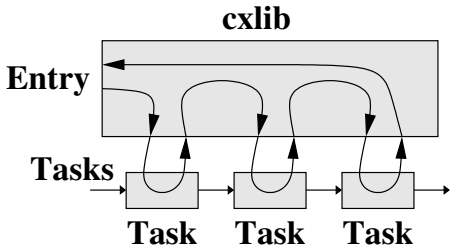

Figure 1: Outside view of cxlib

The difference in views arises because each task has its own unique context, which at least includes its own stack and CPU register values. Rather than calling, a task is actually entered by returning to the task context. To a task, control does not appear to return to a main program loop. In a manner like changing hats, the context switch allows a CPU to switch between running tasks.

Each task is represented with a task control block (TCB) and the TCB list is called the task pool. Currently in cxlib, the TCB list is statically allocated hence a fixed maximum number of tasks can be in use at any one moment. The following summarizes the behavior of functions in cxlib. The 'C' calling convention is used so a user need not be concerned with whether the function is written in 'C' or assembly language.

cx_task_init( )
This initializes data structures in cxlib and should be part of system initialization. This must be called before any tasks are created or multitasking is attempted

cx_task_new( )
This removes one TCB from the unused list, initializes a new context, and inserts the TCB at end of the list of tasks ready to execute, ready list. A function named by the caller serves as the starting point for the task. A return address is also inserted so that returning past starting point terminates the task, removing the TCB from the ready list.

cx_task_head( )
This is the entry point which passes control to the first task in the ready list. When used with a type 2 system it is assumed that each task passes control of the CPU to the next in polite fashion.

cx_task_switch( )
This passes control of the CPU to the next task in the ready list. It is important that this function only be called by tasks actually running under the control of cxlib.

cx_task_delay( )

This provides a delay by calling the task switch function 'N' times. Fairly reliable timing can be provided if cxlib is entered in a periodic fashion.

cx_task_end( )

A task is terminated either by returning past its entry point of by calling this function. In the current version of cxlib a task can only be terminated by its own action. Tasks cannot terminate other tasks.

An example cooperative multitasking system is presented below. To outline the use of cxlib, the library is first initialized with cx_task_init. A first task is created with cx_task_new. Each call to cx_task_head gives each task one opportunity to execute code. Each task yields the CPU by calling cx_task_switch. Fairly reliable timing can be provided by calling cx_task_head in a periodic fashion. A call to cx_task_head can be followed by a timing mechanism like that in Peatman's[1] discussion of hardware timers with super-loops.

Implementing the cxlib Library

The cxlib library is constructed with the following files. The file cxlib.h summarizes the programmer interface to cxlib, giving the prototypes for the entry points to cxlib, listed above. An example program using cxlib is also given later. The assembly language file includes a bare minimum of code. Each assembly language procedure follows 'C' parameter passing conventions, so that the higher level routines written in 'C' are fairly platform independent. To port cxlib to a different platform, it is necessary to write new system dependent routines in assembly language and fine tune values in the header file and linker parameter file.

cxlib.c	- Higher level routines written in 'C'
cxliba.asm	- Lower level routines written in assembly language
cxlib.h	- Header file for cxlib users

The two ideas that make up the core of cxlib are the linked list and the context switch. The task control block (TCB) structure is defined in cxlib.h. As in Figure 2, each TCB is a node in a singly linked list. Extra elements can be inserted at the end of the TCB structure. The tasks are each assigned a stack. The stacks are all the same size, defined in cxlib.h. Each task has a unique task identifier or tid among tasks. Currently the TCB pool is an array and the tid is simply the corresponding array index value.

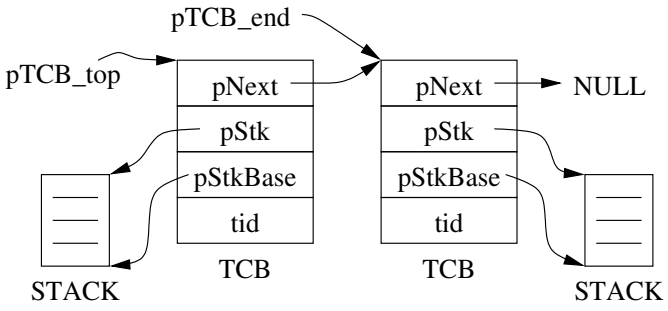

Figure 2: TCB list structure

A context switch is deceptively simple and is performed by pushing the current context onto the stack, changing stacks, and then popping values off the new stack. The moment the program counter value is replaced, the context switch is complete. In version 0.3 the cx_task_switch code invokes a trap to return to the next task. With this change an entry must be added to the linker script to configure the interrupt vector table. Prior versions simply pushed the current context onto the stack, before returning to the next task. Another change appeared in version 0.3 in the optional use of the real time clock interrupt to force a context switch. This second change allows cxlib to be used in type 2 or type 3 systems.

An Example Program

The so-called blink-n-beep example is a classic cooperative multitasking program for cxlib and is included with the distribution. The program blinks an LED at a one Hertz rate, that is one half second on and one half second off. Each time the LED is turned on or off, a speaker produces a beep for approximately one tenth of a second. The beep is actually produced by toggling the speaker approximately every one millisecond. The LED and speaker are connected as shown below in Figure 3. Using two pins to drive the speaker is optional.

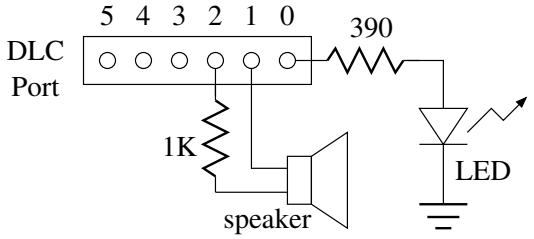

Figure 3: LED and speaker connection

While such a program can certainly be written without using any form of multitasking, it should be evident in the following how using tasks helps to organize and simplify things. The following code first initializes cxlib and creates a first task before entering the main loop. In creating a new task the number zero is a dummy value indicating that there is no extra initialization data. After each task is given an opportunity to execute, the code waits for the real-time tick event. Given that the real time clock is a peripheral device, the clock flag is set in a periodic fashion, in this case every 1.024ms. Once detected, the program clears the clock flag and repeats the loop. In this way, if the tasks return in less than one real-time clock period, the program provides reliable timing, which means the LED and speaker behave correctly.

```
  // initialize and make a new task
  cx_task_init();
  tid = cx_task_new(LEDtask, 0);

  // main loop
  while(!done) {       // not-done

    // give each task a time slice
    cx_task_head( );

    // wait for and clear real time clock flag
    while (RTIFLG == 0) ;
    RTIFLG = 0x80;
  }
```

While in the LEDtask task which controls the LED, we have no use for the 'data' pointer, the assignment to itself is enough to prevent a compiler warning that the variable is never used. The delay function is called to produce a one half second delay and the new task function is called to create a function that will generate the beep sound. The XOR operator is used to toggle the LED state before the task repeats.

```
/*************************************************
 * LEDtask()
 * Every half second toggle LED and start a beep
 *************************************************/
void LEDtask(void *data)
{
  int done = 0;
  data = data;

  while(!done) {       // not-done
    cx_task_delay(LED_DELAY_MAX);
    cx_task_new(BeepTask,0);
    PORTDLC  ^= 0x01;  // toggle LED
  }
}
```

The speaker task activates the speaker by making one terminal high and then toggles the speaker terminals one hundred times before turning it off by making both terminals low and exiting. Each context switch returns approximately one millisecond later. Once a function returns past its starting point the task is terminated.

```
/*************************************************
 * BeepTask()
 * Make a beep for a short time period
 *************************************************/
void BeepTask(void *data)
{
  int BeepCount = BEEP_COUNT_MAX;
  data = data;

  PORTDLC  |= 0x04;          // activate speaker

  while (BeepCount != 0) {   // done yet?
    BeepCount--;
    cx_task_switch();
    PORTDLC ^= 0x06;         // toggle speaker
  }
  PORTDLC  &= 0xF9;          // silence speaker
}
```

Introducing Advanced Topics

Once introduced, cxlib can be used as a springboard to introduce further topics. The idea of task states and task management is important to operating systems. While cxlib does not provide task management per-se, each task can be thought of as being in states. In the blink-n-beep example LEDtask is ready to execute, waiting, or is executing. Likewise, BeepTask is either ready to execute, is executing, or is terminated. This experience can be drawn upon when introducing students to real time systems

While in concept a semaphore is a counter, in practice the use of a semaphore involves an operating system. If a requested service is not available, a mechanism causes the requestor to wait and other tasks to proceed. In one regard cxlib is convincing enough so that students can see such a relationship. Yet cxlib is also simple enough so that students can be presented with of what a semaphore is, the idea can be generalized and used with more advanced systems.

Considering Student Feedback

To obtain feedback pertinent to cxlib, a questionnaire was recently e-mailed to students who completed the course and from a typically small course, a small response was received. In total seven former students replied. Some questions asked students to reply with a numerical answer and others asked for a statement. Students were asked to make any comments they wished.

0 Disagree Strongly	1 Disagree	2 Neutral or Indifferent	3 Agree	4 Agree Strongly

The questions are listed below along with the average values. Question 1 asks for the overall satisfaction with learning cxlib, the response is medium strong agreement. One student

commented that some sort of multi-tasking or real-time operating system example should be part of every embedded systems course. Question 2 is more focused regarding satisfaction and the response is also medium strong agreement. One student commented that cxlib helped in understanding the basics but that there is really much more to learn in real-time operating systems to learn. Topics in which students felt cxlib helped include multi-tasking and switching between tasks.

	Question	Average
1.	Overall, the cxlib multitasking library was worthwhile and worth my time to learn	3.571
2.	The cxlib library helped me to better understand real-time operating systems principles. Also list a topic that cxlib helped your understanding in.	3.429
3.	Do you use or have you used a real-time system in any applications other than cxlib? Please describe.	– –
4.	In comparing a real-time operating system to a super-loop application, a real-time operating system is superior for my use	3.429
5.	Is there another topic that cxlib can be used to better introduce?	– –
6.	Can you think of a way that cxlib could be improved?	– –

Given that cxlib is meant to be a stepping stone, question 3 asks if students have experience with a real-time system. Of those who responded, several including uC/OS, eCos, RT-kernel, and a task scheduler simpler than cxlib were outlined.

Question 4 asks if students feel that use of a real-time operating system is superior to a simple super-loop application. The response was medium strong agreement. One student echoed a point made in class that a real-time operating system affords more freedom and benefits but that on a rudimentary system, may not be possible or may be inappropriate. In such cases a super-loop is the better choice, thus the lesson was learned.

In regards to question 5, topics include the handling of a shell interface and interrupt handling. Lastly, in regards to question 6, topics include priority systems, queues, task scheduling, and interrupt handling. With the addition of pre-emption to cxlib, the topic of interrupts will have move discussion. Given the availability of priority based real-time operating systems, there is not much need however to expand cxlib beyond type 2 and type 3 systems to include priority based structures.

To summarize, my experience with cxlib has been positive and the students who replied to the questionnaire agree. The cxlib project is worth the effort in helping students to learn about multitasking in embedded systems.

Conclusion

The cxlib multitasking library is written for use in electrical and computer engineering student projects and in teaching embedded microprocessor principles. The library supports cooperative multitasking and a recent change allows for preemptive round-robin scheduling. As a teaching tool, cxlib can be introduced in three hours of lecture time. The library is intentionally Spartan, giving ample wiggle room for adding features or making significant architectural changes. The cxlib library is currently written for the Freescale 68HC12 microprocessor using the Metrowerks CodeWarrior tool chain, but it should be portable to other systems.

Bibliography

1. Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, Operating System Concepts, copyright 2005 by John Wiley & Sons, Inc.
2. cxlib website, http://uhaweb.hartford.edu/jmhill/projects/cxlib/index.htm
3. CodeWarrior Tutorial, http://uhaweb.hartford.edu/jmhill/suppnotes/CodeW12/index.htm
4. Gordon Doughman, Programming the M68HC12 Family, copyright 2000 by Gordon Doughman, published by Annabooks.
5. Steven F. Barrett and Daniel J. Pack, Embedded Systems Design and Applications with the 68HC12 and HCS12, copyright 2005 by Pearson Education, Inc.
6. John B. Peatman, *Design with PIC Microcontrollers*, copyright 1998 by Prentice-Hall, Inc, a division of Simon & Schuster.
7. Jean J. Labrosse, MicroC/OS-II, The Real-Time Kernel, copyright 1999 by Miller Freeman, Inc. Lawrence Kansas.
8. David E. Simon, An Embedded Software Primer, copyright 1999 by Addison-Wesley