```
New Ways of Structuring Code
============================

C. Coroutines that call each other, with interrupts to handle hardware
   events and global variables used to communicate from ISR's to each coroutine,
   and between coroutines

   Advantages:
     * Code is a bit less intertwined; each coroutine has its own while loop
       that only needs to focus on one task, keeps all its local variables
       since it has its own stack and doesn't have to return to the caller).
     * No need for FLP code if we want the appearance of concurrency (but
       we do need to yield to other coroutines on a regular basis)
     * Fairly simple to implement: use setjmp/longjmp which is ANSI C
       and a few lines of custom assembly

   Disadvantages:
     * (Windows98) One coroutine, if it goes awry, can block other coroutines
       from running
     * We need to yield to other coroutines on a regular basis, which is
       error-prone (depends on code author to remember to do this), non-
       maintainable, and still a bit of an FLP problem if responsivity
       is a design goal
     * Each coroutine needs to know who to call next, which is error-prone
       and non-maintainable.
     * Each coroutine needs its own stack, which can eat into available RAM.

D. Coroutines that call a scheduler, with interrupts to handle hardware
   events and global variables used to communicate from ISR's to each coroutine,

   and between coroutines

   Advantages:
     * Code is even less intertwined, since each coroutine no longer has to
       worry about which coroutine to call next.
     * Higher level of abstraction: each coroutine is oblivious to the details
       of coroutine switching, number of coroutines in the system. It just
       has to call 'yield()' once in a while. More abstraction --> lower
       complexity for the coroutine developer.
     * All changes to the code regarding the number of coroutines, and what
       they are is done in one place (the scheduler).
     * Scheduler can now explicitly implement prioritization, coroutine
       creation/termination.
     * Fairly simple to implement: scheduler can be just another coroutine

   Disadvantages:
     * One extra coroutine (hence stack and jump buffer) are needed.
     * We need to yield to the scheduler on a regular basis, which is
       error-prone (depends on code author to remember to do this), non-
       maintainable, and still a bit of an FLP problem. Higher level of
       abstraction does not improve responsivity.
     * Each coroutine needs its own stack, which can eat into available RAM.
```

```c
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "rit128x96x4.h"
#include "scheduler.h"

#define STACK_SIZE 4096   // Amount of stack space for each thread

typedef struct {
  int active;       // non-zero means thread is allowed to run
  char *stack;      // pointer to TOP of stack (highest memory location)
  jmp_buf state;    // saved state for longjmp()
} threadStruct_t;

// thread_t is a pointer to function with no parameters and
// no return value...i.e., a user-space thread.
typedef void (*thread_t)(void);

// These are the external user-space threads. In this program, we create
// the threads statically by placing their function addresses in
// threadTable[]. A more realistic kernel will allow dynamic creation
// and termination of threads.
extern void thread1(void);
extern void thread2(void);

static thread_t threadTable[] = {
  thread1,
  thread2
};
#define NUM_THREADS (sizeof(threadTable)/sizeof(threadTable[0]))

// These static global variables are used in scheduler(), in
// the yield() function, and in threadStarter()
static jmp_buf scheduler_buf;   // saves the state of the scheduler
static threadStruct_t threads[NUM_THREADS]; // the thread table
unsigned currThread;    // The currently active thread

// This function is called from within user thread context. It executes
// a jump back to the scheduler. When the scheduler returns here, it acts
// like a standard function return back to the caller of yield().
void yield(void)
{
  if (setjmp(threads[currThread].state) == 0) {
    // yield() called from the thread, jump to scheduler context
    longjmp(scheduler_buf, 1);
  } else {
    // longjmp called from scheduler, return to thread context
    return;
  }
}

// This is the starting point for all threads. It runs in user thread
// context using the thread-specific stack. The address of this function
// is saved by createThread() in the LR field of the jump buffer so that
// the first time the scheduler() does a longjmp() to the thread, we
// start here.
void threadStarter(void)
{
```

```
  // Call the entry point for this thread. The next line returns
  // only when the thread exits.
  (*(threadTable[currThread]))();

  // Do thread-specific cleanup tasks. Currently, this just means marking
  // the thread as inactive. Do NOT free the stack here because we're
  // still using it! Remember, this function runs in user thread context.
  threads[currThread].active = 0;

  // This yield returns to the scheduler and never returns back since
  // the scheduler identifies the thread as inactive.
  yield();
}

// This function is implemented in assembly language. It sets up the
// initial jump-buffer (as would setjmp()) but with our own values
// for the stack (passed to createThread()) and LR (always set to
// threadStarter() for each thread).
extern void createThread(jmp_buf buf, char *stack);

// This is the "main loop" of the program.
void scheduler(void)
{
  unsigned i;

  currThread = -1;

  do {
    // It's kinda inefficient to call setjmp() every time through this
    // loop, huh? I'm sure your code will be better.
    if (setjmp(scheduler_buf)==0) {

      // We saved the state of the scheduler, now find the next
      // runnable thread in round-robin fashion. The 'i' variable
      // keeps track of how many runnable threads there are. If we
      // make a pass through threads[] and all threads are inactive,
      // then 'i' will become 0 and we can exit the entire program.
      i = NUM_THREADS;
      do {
        // Round-robin scheduler
        if (++currThread == NUM_THREADS) {
          currThread = 0;
        }

        if (threads[currThread].active) {
          longjmp(threads[currThread].state, 1);
        } else {
          i--;
        }
      } while (i > 0);

      // No active threads left. Leave the scheduler, hence the program.
      return;

    } else {
      // yield() returns here. Did the thread that just yielded to us exit? If
      // so, clean up its entry in the thread table.
      if (! threads[currThread].active) {
        free(threads[currThread].stack - STACK_SIZE);
      }
    }
  } while (1);
}
```

```c

void main(void)
{
  unsigned i;

  // Set the clocking to run directly from the crystal.
  SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                 SYSCTL_XTAL_8MHZ);

  // Initialize the OLED display and write status.
  RIT128x96x4Init(1000000);
  RIT128x96x4StringDraw("Scheduler Demo",       20,  0, 15);

  // Enable the peripherals used by this example.
  SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

  // Set GPIO A0 and A1 as UART pins.
  GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

  // Configure the UART for 115,200, 8-N-1 operation.
  UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
                      (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                       UART_CONFIG_PAR_NONE));

  // Create all the threads and allocate a stack for each one
  for (i=0; i < NUM_THREADS; i++) {
    // Mark thread as runnable
    threads[i].active = 1;

    // Allocate stack
    threads[i].stack = (char *)malloc(STACK_SIZE) + STACK_SIZE;
    if (threads[i].stack == 0) {
      iprintf("Out of memory\r\n");
      exit(1);
    }

    // After createThread() executes, we can execute a longjmp()
    // to threads[i].state and the thread will begin execution
    // at threadStarter() with its own stack.
    createThread(threads[i].state, threads[i].stack);
  }

  // Start running coroutines
  scheduler();

  // If scheduler() returns, all coroutines are inactive and we return
  // from main() hence exit() should be called implicitly (according to
  // ANSI C). However, TI's startup_gcc.c code (ResetISR) does not
  // call exit() so we do it manually.
  exit(0);
}

/*
 * Compile with:
 * ${CC} -o crsched.elf -I${STELLARISWARE} -L${STELLARISWARE}/driverlib/gcc
 *     -Tlinkscript.x -Wl,-Map,crsched.map -Wl,--entry,ResetISR
 *     crsched.c create.S threads.c startup_gcc.c syscalls.c rit128x96x4.c
 *     -ldriver
 */
// vim: expandtab ts=2 sw=2 cindent
```

```c
#include <stdio.h>
#include "scheduler.h"

// These are the user-space threads. Note that they are completely oblivious
// to the technical concerns of the scheduler. The only interface to the
// scheduler is the single function yield() and the global variable
// currThread which indicates the number of the thread currently
// running.

void thread1(void)
{
  unsigned count;

  for (count = 0; count < 10; count++) {
    iprintf("In thread %u -- pass %d\r\n", currThread, count);
    yield();
  }
}

void thread2(void)
{
  unsigned count;

  for (count=0; count < 5; count++) {
    iprintf("In thread %u -- pass %d\r\n", currThread, count);
    yield();
  }
}
```

```
/*
 * Implement the thread creation task:
 *
 *   − initialize the jump buffer with appropriate values for
 *     R13 (stack) and R14 (first address to jump to)
 *   − all other registers are irrelevant upon thread creation
 *
 *   In the jump buffer, the R13 slot is set to the second parameter of this
 *   function (the top−of−stack address, passed in R1). The R14 slot is set to
 *   the address of the threadStarter() function.
 *
 *   The C prototype for this function call is:
 *       createThread(threads[i].state, threads[i].stack)
 *   thus:
 *       R0 <−− state (a setjmp()−style jump buffer)
 *       R1 <−− stack (address of top−of−stack)
 */
    .syntax unified
    .text
    .align 2
    .thumb
    .thumb_func
    .type createThread,function
    .global createThread
createThread:

  /* Save registers in the jump buffer. Their values are
     not important when the thread is first created. This line is the same as
     the first two lines of setjmp(), except we don’t save SP and LR since we
     want to set these to our own values. Really, the only point of this
     instruction is to advance R0 to the right location in the jump buffer for
     pointing to SP (without having to do any math :−) */

    stmea  r0!, { r4−r10, r11 }

  /* Now we save SP and LR in that order. SP is the R1 parameter, and we have
   * to get the address of threadStarter() into a higher register (so they are
   * placed in the jump buffer in the right order). */

    ldr     R2, .L0
    stmea   R0!, { R1, R2 }   @ Store "SP" and "LR" for the new thread

    bx      lr

.L0:
    .word   threadStarter
```

```
New Ways of Structuring Code
============================

D.
   Disadvantages:
     * We need to yield to the scheduler on a regular basis, which is
       error-prone (depends on code author to remember to do this), non-
       maintainable, and still a bit of an FLP problem. Higher level of
       abstraction does not improve responsivity.

E. Time-sliced threads, with interrupts to handle hardware events and
   global variables used to communicate from ISR's to each thread,
   and between threads

   Advantages:
     * No need to explicitly yield to a scheduler...the scheduler forcibly
       interrupts (i.e., pre-empts) a thread when its "time slice" is up.
       This is a natural use for the SysTick timer.
     * The code is minimally intertwined and there are only well-defined
       interfaces between threads<-->scheduler and thread<-->thread.
     * If time slices are small enough, the system provides the "appearance
       of concurrency" and has very good latency and responsivity.

   Disadvantages:
     * Complex: interrupt-driven system is easy to get wrong, debugging is
       difficult.
     * Each thread needs its own stack.
     * The "concurrency problem". When program state is saved/restored at
       an explicit function call boundary (i.e., yield) then behavior is
       (mostly) deterministic. But when program state is saved/restored
       at any assembly-language instruction boundary, then behavior is
       non-deterministic and previously-true assumptions no longer hold.
```

```c
#include <stdio.h>
#include "scheduler.h"

// This is "the next level": threads that are pre-emptively interrupted
// in order to return control to a central scheduler, and are mostly
// entirely unconcerned with (a) other threads, and (b) the scheduler itself.
//
// NOTE: A thread can be interrupted AT ANY TIME, even in the middle of
// a function call, on ANY assembly language boundary.

void thread1(void)
{
  unsigned count;

  for (count = 0; count < 10; count++) {
    iprintf("In thread %u -- pass %d\r\n", currThread, count);
  }
}

void thread2(void)
{
  unsigned count;

  for (count=0; count < 5; count++) {
    iprintf("In thread %u -- pass %d\r\n", currThread, count);
  }
}
```

```
So what do you need to do?
=-=-=-=-=-=-=-=-=-=-=-=-=-

1. Program SysTick for periodic interrupts:
    - why are shorter periods better?
    - why are shorter periods worse?

2. On a SysTick interrupt, do a "forced setjmp", i.e., save the state of the pro
cessor as for any exception.

3. Determine which thread should run next (round-robin? priorities?)

4. Restore the state of the processor, not of the thread that was interrupted, b
ut of the thread that is to run.

Things that need to change
=-=-=-=-=-=-=-=-=-=-=-=-=-

* threadStruct_t: jmp_buf is not enough to save exception state
        typedef struct {
          int active;       // non-zero means thread is allowed to run
          char *stack;      // pointer to TOP of stack
          jmp_buf state;    // saved state for longjmp()
        } threadStruct_t;

* yield(): must simulate a SysTick exception, i.e., cause a context switch

NOTE: Simply sitting in a loop waiting until the next SysTick interrupt is NOT O
K!!!

* createThread: must simulate saved exception state, instead of simulating setjm
p state
   - What should exception return address be?
   - What should R14 be?
   - What should R13 be?
   - What should R0-R12 be?

* scheduler(): goes away...scheduling functions are done in the SysTick ISR

* main(): how do you kickstart the process? change privilege levels? change to p
rocess stack?
```