Project 3: Pre-Emptive Multi-Threaded Kernel for the LM3S6965

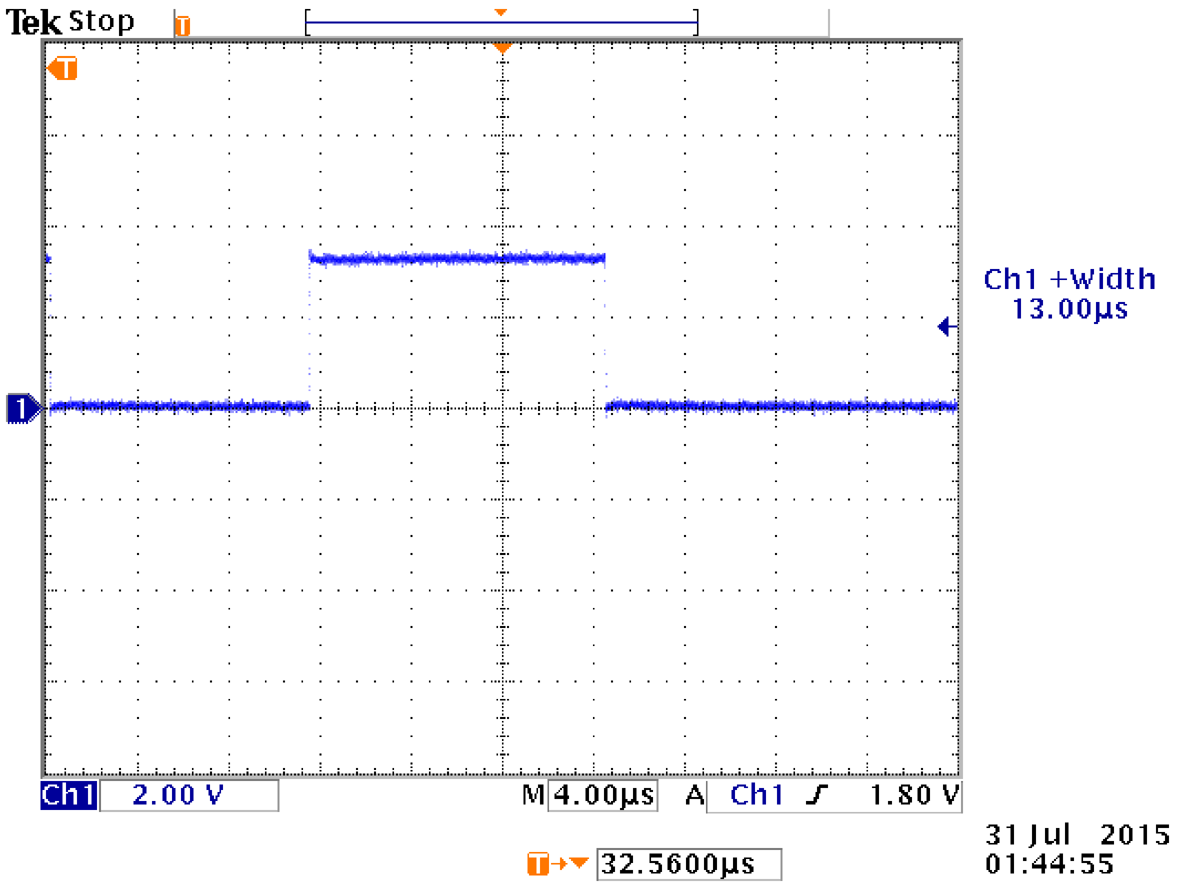
Greg Lepley and Bill Neuson

EGR424

Dr. Parikh

Context Switch Time Measurement and Steps

In order to measure the context switch time of our kernel, we utilized the GPIO output capabilities of Port F Pin 1. When the reg_save function is called, not only does it save the state of all of the registers being used by the current thread, but it also sets Port F Pin 1 high. In contrast, when the reg_restore function is called, in addition to restoring all of the registers used by the next desired thread, it turns the output of Port F Pin 1 off. Since the reg_save and reg_restore functions are called at the beginning and end of the scheduler_Handler, this causes a pulse of positive voltage that matches the length of the context switching time. We can then probe Port F Pin 1 with an oscilloscope in order to measure the context switch time. For our project, we have a measured context switch time of 13.00µs. Here is a screenshot of a measurement taken with a Tektronix TDS 3012B oscilloscope.



Description of Modified and Custom Functions

void createThread(int *buf, char *stack); (create.S)

The buffer is filled with data that will be restored to registers when reg_restore is called. The process stack must be prepared to include the variables that will be restored when an interrupt return is faked.

void reg_save(int *state);

Saves r4->r12, PSP into the state buffer for a thread that has just been interrupted and turns on Port F Pin 1 in order to mark the start of the context switch in the kernel.

void reg_restore(int *state);

Restores r4->r12, PSP from the state buffer for a thread that will begin after being interrupted and turns off Port F Pin 1 in order to mark the end of the context switch.

void lock_init(unsigned *lock);

Initializes a locking variable to the 1 state, denoting an initially unlocked state.

unsigned lock_aquire(unsigned *lock);

Implement locking by storing an exclusive 0 to the lock variable, denoting a locked state.

void lock_release(unsigned *lock);

Releases a locking variable to the 1 state, denoting an unlocked state.

int get_currThread(void);

Access the currthread variable to replace a global variable implementation.

void yield(void);

This function is called from within user thread context. It executes a svc call to jump to the scheduler. When the scheduler returns here, it acts like a standard function return back to the caller of yield().

unsigned lock(unsigned *threadlock);

> Locks a resource by upcount Locking. Checks to see if the desired resource is already locked. If it is already locked by the current thread, then increment the lock count. If it is unlocked then it calls lock_acquire to try and lock the resource.

void unlock(unsigned *threadlock);

> Unlocks a resource by downcounting until 0. Checks to see if the desired resource is locked by the current thread. If it is, then it decrements the lock count. If the lock count reaches zero, then it executes the lock_release function.

void unlock_force(unsigned *threadlock);

> Forces an unlock of the resource in the event of an issue, regardless of which thread currently has a lock on the resource.

void thread0(void);

> Thread 0 is an idle thread for the system to fall into when there are no other active threads.

void thread1(void);

> Thread 1 is a UART thread demonstrating peripheral locking as well as calls to Yield().

void thread2(void);

> Thread 2 is a UART thread demonstrating peripheral locking as well as calls to Yield().

void thread3(void);

> This thread contains LED flashing commands and scheduler pre-emption

void thread4(void);

> This thread contains OLED commands that cause a bar to move back and forth across the OLED screen.

void unpriv(void);

Set the privilege value of thread mode to unprivileged. Only needs to be called at the beginning, because this sets the state of thread mode execution for the duration of execution.

void scheduler_Handler(void);

This is the handler for the systic timer/SVC interrupt that handles the scheduling of the threads. It saves the current state of all necessary registers for the current thread, finds the next active thread, and loads all of the necessary registers for the new current thread. If there are no more active threads, then the scheduler jumps to the idle thread.

void threadStarter(void);

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 buffer so that the first time the scheduler() does a reg_restore to the thread, we start here.

void main(void);

Main function of the program, sets up peripherals, creates threads, and starts the scheduler.

void printFault(void);

Prints out a fault message to the OLED screen.