To have soul, you have to have rhythm!
Improved Theremin musical instrument concept
using time-triggered design techniques
Mike Smith and Lizie Dunling-Smith

Version – 17th January 2009

On seeing the initial design for our modernized version of a Theremin musical instrument (*Circuit Cellar* YYY, 2009), the first comment from the members of Mike’s 4th year University Capstone design team was “Dr. Smith, you have no soul”. The comment was unexpected as he has always seen himself as a warm and considerate professor (in a highly demanding sort of way). However the team explained that they meant the music generated by the proposed Theremin instrument had no background “beat” as an accompaniment, so the music had no rhythm. “Rather than adjusting the Theremin’s volume by calculating how much light falls on a light sensor, why don’t we build a beat stick for you?”’, they suggested.

Their idea for a beat stick was apparently the generic (home-built) equivalent of a Wii stick using an accelerometer to sense movement of the Theremin-ist’s hands. By waving the stick around you can add a beat (a sudden volume change) to the music. Currently a pulse width modulated (PWM) signal stream from a TAOS light sensor is being used to control the volume. It would be figuratively “5 minutes work” to replace this PWM stream with one from an Analog Devices ADXL213 accelerometer. A little bit of recalibration of the volume scaling factors calculated in the code; and the Theremin-ist is up and playing again!

However the second aspect of the team’s idea was a little more complicated. First they would write a DSP program to determine the basic beat from the generated Theremin music. Then a second output audio stream would be generated by changing the frequency of the beats in some music having a good basic beat to be the same as that of the Theremin music. “So the Theremin-ist will get automatic accompaniment to the music. All done in real time, of course”, was their parting comment.

Fig. 1 shows their new proposed design. The hardware was not a problem; the Blackfin BF533 evaluation board used for teaching at the University of Calgary is easily capable of handling multiple audio inputs and outputs; the BF533 has considerable DSP capability. The software needed to support the new student project would, in principle, be very easy to modify from the original Theremin code (*Circuit Cellar* YYY, 2009) already running using an operating system based around a co-operative scheduler (Listing 1).

- Switching from controlling the volume of the Theremin using a PWM pulse stream from a light sensor \( \text{LightSensorVolume()} \) to a PWM pulse stream from an accelerometer \( \text{AccelerometerSensorVolume()} \) (Line 11) is essentially a re-calibration issue.
- The old audio task \( \text{OutputAudio()} \) requires minor modification (Line 7) to handle two output channels rather than one. A second audio task \( \text{InputAudioBeat()} \) (Line 8) picks up the new input audio stream.
- Developing the DSP algorithm \( \text{DSPTasks()} \) (Line 14) to determine the “beats” present in the two audio streams would be a challenge for a 4th year University design team. However, given sufficient time, the team could make the system work even with the additional complication of having to smoothly integrate the two audio streams together.
However, as the system state history snapshot shown in Figure 2 indicates, there is a serious data acquisition problem to be overcome. The moment DSPTasks() cuts in (top line), the processor becomes fully occupied with this task. As no other task can now be executed in the co-operative scheduler, every other task block (shown as orange) and the system dies. How do you fit a long DSP task into the operation of a co-operative scheduler that is running many sampling tasks that need to be serviced at regular 20 us intervals?

### Handling lengthy hardware tasks within a co-operative scheduler

We could easily switch back to a pre-emptive scheduler to provide a solution. In that case, we’d place the DSP tasks as part of the loop in main(); and let all the audio and measuring tasks performed by the light sensor and accelerometer be fired off in the background as part of a number of interrupt service routines.

```c
int main(void) {
    SetupInterrupts();
    ActivateInterrupts();
    while (1) {
        DSPTask();
    }
}
EX_INTERRUPT_HANDLER(EverythingISR)
{
    AcknowledgeInterrupt();
    AudioInput();
    AudioOutput();
    etc.
}
```

However for us, the construction of this project was not really about making the “eerie Theremin music” that was so attractive to the 4th year project team. It had more to do with re-discovering the advantages of designing embedded systems with “highly predictable performance” present when using a co-operative scheduler.

In the last article (Circuit Cellar YYY) a similar problem arose when there were a number of scheduler tasks being used to determine the frequency of the output signals from a light sensor. While the processor was waiting for the changes in the sensor signals, these hardware tasks prevented the time critical audio-output tasks from running. The solution was to break the long running task into a large number of shorter tasks that interacted in a co-operative fashion with other tasks.

### Fig. 2. The moment that the DSPTasks() task starts to execute, all the other tasks start to block (shown in orange). When the DSPTask() completes execution, all the blocked tasks start to “play catch-up”; executing with every scheduler tick rather than at their proper times. This would first block and then distort the audio signal.

### Handling lengthy software tasks within a co-operative scheduler

The original Theremin code design had tasks “waiting” for an external hardware signal to change its state. This required tasks with durations between 10 us to 30 ms depending on the signal’s frequency. Such a hardware-oriented task can be easily broken up into a series of “tiny” tasks, of 300 ns duration and period 10 us,
that “poll”, rather than “wait” for the external signal changes.

Imagine trying the same approach with a lengthy software-oriented task. For the code in Listing 1 to actually work, the Capstone project team would have to write `DSPTasks()` as a series of code overlays that were swapped in and out of memory between the 10 us timer interrupts. That sounds like a really practical, time efficient, solution!

If this rediscovery of using co-operative schedulers in embedded systems is not to die a sudden death, a number of things need to be tackled to make the revised Theremin design work:

1) What code development process will make it possible to simultaneously service a number of highly time critical “fast” data acquisition tasks without these tasks being blocked by an equally important, but long- running, `DSPTasks()` routine?

2) There are now multiple tasks that interact with each other. How do you modify the operation of a co-operative scheduler to ensure that you can recover from one of the tasks failing? Sure the Theremin does not need to meet safety critical requirements, but we would like to be able to continue in a default mode if a cable was temporarily disconnected.

**Finding the time to do some DSP**

Executing a “long DSP task” using a co-operative scheduler is obviously a pretty standard thing to do. Rather than re-inventing the wheel, we went back to Pont’s Patterns for Time-Triggered Systems to see what he suggests to handle these issues. His book is available as a free download from [http://www.tte-systems.com/books](http://www.tte-systems.com/books).

Pont recommends two solution approaches. The first is to avoid frequent software polling. The second is to allow the scheduler to execute a single “pre-emptive task”.

**Hardware polling:** Currently the `LightSensorFrequency()` task examines (polls) the input light sensor signal by monitoring a general purpose I/O (GPIO) pin. Each transition of signal causes the task to either move between one of four states (`HIGH1, LOW1, HIGH2, LOW2`) or return from the task and free up the processor for other tasks to run.

This software polling task needs to be executed at 100 us intervals to ensure that the frequency is determined with a high enough accuracy to ensure that the Theremin does not produce sour notes.

```c
enum {HIGH1, LOW1, HIGH2, LOW2};
void LightSensorTask( ) { // Set system state the first time this task runs
    static int currentState = HIGH1;
    int newState = currentState;
    // Determine level of sensor input
    // either HIGH or LOW
    int pinState = GPIOPin();
    switch (currentState) { // SOFTWARE poll
    case HIGH1: // Test for sensor going high
        if (pinState == LOW) {
            newState = LOW1;
            break;
        }
    case LOW1: // Test for sensor going high
        if (pinState == HIGH) {
            newState = HIGH2;
            time1 = CurrentTime( );
            break;
        }
    case HIGH2: // Test for sensor going high
        if (pinState == LOW) {
            newState = LOW2;
            break;
        }
    case LOW2: // Test for sensor going high
        if (pinState == HIGH) {
            newState = HIGH1;
            sensorPeriod = CurrentTime( ) – time1;
            break;
        }
    // Update the task state
    currentState = newState;
}
```

Switching from software polling to hardware polling using the general purpose timer offers a number of advantages. First we have switched from a frequent time critical task to an infrequent non-time critical task. The time constraint on the duration of `DSPTasks()` has shifted from hundreds of micro-seconds to tens or hundreds of milli-seconds; that’s a load of cycles on the DSP capable Blackfin processor.

The second advantage is that we are getting much higher precision time measurements of the sensor signals even though we are performing less frequent time-measuring tasks. In this Theremin application, poor timing accuracy is best avoided as it translates into terrible tonal quality of the music!
Permitting a single “pre-emptive task”

The Blackfin ADSP-BF533 processor has three general purpose timers that can be used in the Theremin project—one for monitoring the accelerometer signal (volume control), one for monitoring the light sensor signal (frequency control) and an extra one for a “future feature”.

However switching from software polling to hardware polling does not really solve anything. There are still the frequent (10us) tasks needed to handle the audio input and output samples; these will still be prevented from running by DSPTasks(). To solve this problem, Pont suggests making a minor modification to the co-operative scheduler’s interrupt service routine (ISR) to allow two types of scheduler tasks. These are discussed in the next two paragraphs.

Standard co-operative tasks (Listing 2, Lines 24 to 29) are run by TTCOS_DispatchTasks() which is part of the Theremin’s main loop (Line 37). These “standard tasks” can be interrupted by a single “pre-emptive task” (Line 22) that is set up to run as part of the co-operative scheduler’s ISR (Listing 3, Lines 60 to 68).

As the audio manipulation tasks involve moving one or two words to and from the Blackfin’s audio CODECs, they take very few processor cycles. Thus moving all these tasks into a single pre-emptive task should have negligible impact on the overall performance of the system.

Thus the short PreEmptiveTask() will meet the restrictions on the pre-emptive task used in a co-operative scheduler; restrictions that are stronger than those placed on the tasks (threads) present in a pre-emptive scheduler.

```
#pragma always_inline
inline void ModifiedOutputAudio() { }
// Unchanged Task Code;
}

// New code to handle a pre-emptive task
if ([Tasks][taskIndex].pre_emptiveTask == true) {  // Check to see if pre-emptive task
    (*[Tasks][taskIndex].pointerToTask)();        // If pre-emptive task then run the task
    [Tasks][taskIndex].RunMeNow--;                // simply decrement the remaining delay time
} else {    // If the task is READY_TO_RUN
    [Tasks][taskIndex].RunMeNow++;                // then increment the RunMeNow flag
}

EX_INTERRUPT_HANDLER(TTCOS_Interrupt) {       // Only ISR operative in the co-operative scheduler
    TTCOS_Update( );                          // Part of scheduler interrupt service routine
}

void TTCOS_Update(void) {                     // Part of scheduler interrupt service routine
    int taskIndex;

    for(taskIndex = 0; taskIndex < NUMBERTASKS; taskIndex++) {
        if ([Tasks][taskIndex].taskDelay != 0) // If taskDelay is non-zero then task is not ready to be run
            [Tasks][taskIndex].RunMeNow++;  //    then increment the RunMeNow flag
        else {     //  If the task is READY_TO_RUN
            [Tasks][taskIndex].RunMeNow++;    // then increment the RunMeNow flag
        }

        if ([Tasks][taskIndex].taskPeriod != RUN_ONCE) // Prepare periodic tasks to run again
            TTCOS_DeleteTask(taskIndex);
    }

    if ([Tasks][taskIndex].taskPeriod == RUN_ONCE) // Remove 'RUN_ONCE' pre-emptive tasks from todoList
        TTCOS_DeleteTask(taskIndex);
}
```

Listing 2: Combining a pre-emptive task (Line 24) with hardware polling (Lines 26 and 27) removes the need for the signal manipulation code DSPTasks() to meet very strict time constraints.
• There must be only one pre-emptive task in order to maintain the predictability of the co-operative scheduler.

• It is not necessary that the pre-emptive task be executed every time the timer interrupt occurs. However it must be guaranteed that the pre-emptive task will complete in a shorter timer than that of the scheduler’s timer interrupt.

The Theremin pre-emptive audio manipulation task will probably meet these requirements without any re-coding. However, given the expected small size of these tasks (Listing 2, Lines 1 to 14), attempts to inline this task to further improve system performance is unlikely to cause any significant code bloat that would over tax the embedded system’s memory.

Permitting this single pre-emptive task has given back to the co-operative scheduler much of the flexibility present in handling tasks with a pre-emptive scheduler. However the total predictability of the co-operative scheduler is retained.

Recovering from “soft” data-acquisition errors

In a typical embedded project, minor data acquisition errors could be easily introduced by a sensor signal disappearing as a cable is temporarily unplugged.

In this situation, we want to make a soft recovery. For example, if the volume sensor has been temporarily removed, then we want to set the system to a safe volume. When the signal re-appears, the system should automatically resume normal operation. This situation can be handled by introducing software watchdog code which permits a soft system recovery. This is illustrated in Listing 4 where the LightSensorFrequency() task code has been modified to not have interruptions; meaning the code will not produce unbearable high pitched sounds if the light sensor becomes unplugged.

• The SOFTWARE_WATCHDOG_SETUP macro (Line 84) establishes a number of static variables. In this example, the software watchdog time out condition is set to be a QUARTER_SECOND and the watchdog is directed to record LIGHSENSOR_ERR type errors. Finally the system is instructed to move onto recovery task state RECOVERY (Line 102, which in this case leaves the note unchanged) if the software watchdog ever times out because the task can’t be completed.

• Upon a watchdog timeout, the error condition is recorded in the global errorRecorder variable, the task is directed to move to the recoveryState and the watchdog timer is reset (see Listing 5 for macro details).

  • In Line 79, the task’s state variable is initialized so that the first time the task is run it will immediately call the new FEED_SOFTWATCHDOG case statement (Line 87). There the macro SOFTWARE_WATCHDOG_HANDLER (Listing 5) resets the software watchdog and cause a move to the NOT_READY state which was the originalFirstTaskState for the LightSensorFrequency task.

  • The final state in the original task (Line 99) is now directed to call FEED_SOFTWATCHDOG to reset the watchdog timer each time the task has run to completion.

This intuitive approach of initializing a software timer and adding a new

FEED_WATCHDOG case is applicable for adding a software watch-dog timer to any data acquisition task in the system.

Allowing the co-operative scheduler to recover in the presence of “hard errors”

A major error might occur if one co-operative scheduler is being driven as a slave by an interrupt signal from a second processor (shared clock scheduling). Suppose the master processor suffers a hard error so that the shared interrupt signals stop arriving. The slave co-operative scheduler will complete its current task, and then “hangs”. In this situation, we must be able to activate a “back-up” interrupt that would allow the slave co-operative scheduler to run tasks and put the system into a safe or shutdown mode.

This is can be handled by activating a hardware watchdog timer; and adding a FeedHardwareWatchTimer() task to the system’s to-do List. Provided the shared interrupt keeps arriving, the co-operative scheduler would be able to run
FeedHardwareWatchTimer() which ensures that the watchdog timer interrupt does not occur. If the external signal stops, then the watchdog timer interrupts start occurring and can be used to allow the co-operative scheduler to produce a system safe condition.

The next project design problem to overcome

In this article we have looked at how to allow a co-operative scheduler to recover from hard and soft data acquisition issues – missing interrupts or external sensor signals. We have also shown that using hardware, rather than software, polling, combined with a single pre-emptive task gives the co-operative scheduler a great deal of flexibility.

The capstone design project team has indicated that they plan to prototype their system by pairing the Theremin music stream with a recording of a snare drum. The repetitive beat in the snare drum music should be identifiable by using a correlation algorithm. They will then use this beat time, and a sample of the beat note itself, to establish a new music stream whose tempo can be manipulated (Fig. 4).

Sounds like they will need to use multiple buffering techniques where they gather audio signals into one buffer, process a second and place the DSP results into a third buffer, while outputting previous calculations from a fourth. That means they will need the Blackfin’s direct memory access (DMA) capability to move the large amount of audio data stored in the evaluation board’s external memory into the core for processing while not disrupting the basic DSPTasks() operation. How all this is handled within a co-operative scheduler environment will be the subject of an DSP article that the team plans to submit.

Listing 5: These macros provide the functionality to place a software watch timer inside every task to ensure that the lack of input signal does not cause the system to end up in an unsafe state. The variable errorRecorder uses a bit to signify errors associated with each individual task. Error can be identified, and recovery procedure initiated as part of the per-emptive task that is run at every tick of the scheduler.

Figure 4: Schematic of the proposed “adaptive drum tempo” DSP algorithm.

Mike Smith has been contributing to Circuit Cellar magazine since the ’80s. He is a professor in computer engineering at the University of Calgary, Canada. His main interests are in developing new biomedical engineering algorithms and moving them onto multi-core and multiprocessor embedded systems in a systematic and reliable fashion. Mike has recently become a convert to the application of Agile Methodologies in the embedded environment. He is Analog Devices University Ambassador (2001 – 2009). He can be contacted at Mike.Smith@ucalgary.ca.

Lizie Dunling-Smith is a 4th undergraduate student at the University of Alberta, Canada, and will graduate in 2009 with a B. Sc. in Engineering Physics (specializing in Electrical Engineering).