Can you teach an old dog new (time triggered) tricks?
Mike Smith
University of Calgary, Canada

Recently an overseas internship student, Phillipe G., joined my group for an eight week research project. It was planned that he and a local music professor would collaborate on using a new time-frequency analysis algorithm called the ultra-fast s-transform for voice training. However, they needed to demonstrate an implementation of this DSP algorithm working in real time on a low-cost processor before even thinking of proposing any commercial ideas to the algorithm’s original inventors; two colleagues of mine.

Over a period of about a month, many unusual, but reasonably musical, noises floated around my laboratory. Phillipe was moving fast down the road of getting the algorithm going on an Analog Devices Blackfin processor. This processor looked a suitable target for this project because of its DSP capabilities and ability to support both audio and video peripherals (Fig. 1). However about six weeks into the eight week project, the musical notes from that corner of the room started to turn sour. The algorithm was working well, but Phillipe was not satisfied with the resolution of the TV display that the BF533 Ez-Kit provided.

Always the kind and considerate professorial host, I offered my visitor a BF548 evaluation board (Fig. 2) which has expanded capabilities compared to the basic BF533 evaluation board. I admit not being able to find specific connections for the proverbial kitchen sink; but there was a hard-drive, this type of flash, that type of flash, UART, Ethernet and best of all – a colour LCD display.

Fig. 1. The BF533 evaluation board is capable of supporting basic audio and video demonstrations

Fig. 2. The BF548 evaluation board has significantly more capabilities than the BF533 evaluation board with many peripheral devices including a coloured touch screen.
The BF548 processor has the same basic core as the BF533; so movement of Phillipe’s existing audio analysis sub-system to the new board was straightforward. He was then able to merge this code with the existing BF548 LCD video-interrupt driven screen demonstration code. This gave him a grey-scale display of the results of his ultra-fast s-transform implementation. However, in the time he had remaining, Phillipe was never able to get a full functional colour display running. It would initially work and then, over a period of three or four minutes, colour synchronization would somehow get lost, and the display would then rotate between red and blue colour casts. We concluded that there was an occasional race condition between the audio interrupts updating a number of buffers, all the various memory DMA transfers occurring on those buffers, and the video driver’s interrupts updating the screen based on the results from the transform calculations.

Phillipe went back to Switzerland to complete his degree at the Ecole d’ingénieurs et d’architectes de Fribourg, Switzerland; and left me with a problem. Is there a reliable and systematic way of merging together two complex audio and video algorithms into a single project when the both use code sequences involving pre-emptive interrupts?

This is a standard problem in an embedded environment found many times in industry. Airplane control systems successfully merge many DSP algorithms and associated external signals. After merging, all the sub-systems reliable interact with each other, and remain responding to the external signals in a fast and reliable manner that ensures that the plane does not fall from the sky. Surely there must be a systematic way to do get the same reliability with a far less complex system?

**What choices are there?**

When merging two working sub-systems together into a single system, the developer has many choices. These include (i) the ‘super-loop’; (ii) the pre-emptive scheduler, which is what was effectively used in the DSP project described in the introduction and (iii) the co-operative scheduler.

To see the differences between these three ideas, imagine the merging of two simple subsystem examples. One subsystem is responsible for input using the processor’s general purpose input / output (GPIO) pins to read 4 switches. The other subsystem outputs signals via a series of 6 LED’s connected to a parallel port that is part of the system’s FLASH memory chip.

With a “super-loop”, Listing 1, the two GPIO and LED interfaces are first initialized using GPIO_Init() (Line 13) and LED_Init() (Line 14). Then, in a forever loop, the GPIO interface pins are read (GPIO_Read(), Line 17) and transferred to the LED output (LED_Write(), Line 18). This sort of coding pattern is totally adequate in many embedded applications where the system only requires basic functionality; and no timing issues are present.

```
12 int main(void) {
13   GPIO_Init();            // Initialize the GPIO input sub-system
14   LED_Init();            // Initialize the LED output sub-system
15
16   while (1) {            // Enter the 'Superloop'
17     int value = GPIO_Read();        // Read the input (GPIO) switch values
18     LED_Write(value);           // Echo the switches to the LEDs
19   }
20
21   return 0;               // Keep the compiler from complaining
22 }
```

Listing 1. If all the embedded system has to do is something very basic, such as echo switch input values to a LED interface (Lines 17 – 18), then a basic super-loop programming format is all you need.

```
32 int main(void) {
33   GPIO_Init();            // Initialize the GPIO input sub-system
34   LED_Init();            // Initialize the LED output sub-system
35   Timer_Init();          // Provide ability to measure 'elapsed time'
36
37   int timeLED4Flash = ElapsedTime();  // Start measuring time intervals
38   int timeLED5Flash = ElapsedTime();  // used to control the flashing of LED 4 and 5
39
40   while (1) {            // Enter the 'Superloop'
41     int value = GPIO_Read();        // Read the input (GPIO) switch values
42     LED_Write(value);           // Echo the switches to the LEDs
43
44     // Has sufficient time elapsed to cause LED #4 to be flashed
45     if ((ElapsedTime() - timeLED4Flash) >= ONE_TENTH_SECOND) {
46       timeLED4Flash = ElapsedTime();  // Update timing
47       FlashLED4();               // and flash the LED
48     }
49
50     // Has sufficient time elapsed to cause LED #5 to be flashed
51     if ((ElapsedTime() - timeLED5Flash) >= ONE_TWENTIETH_SECOND) {
52       timeLED5Flash = ElapsedTime();  // Update the timing
53       FlashLED5();               // and flash the LED
54     }
55 }
56
57   return 0;               // Keep the compiler from complaining
58 }
```

Listing 2. With each task that needs to occur at a predetermined time, the super-loop code gets messier to develop and maintain. In addition, the processor is always “on”; a condition that would waste battery power in a hand-held application.
switches every time around the loop. We know that it is not humanly impossible to make changes to those switches at the speed that the processor operates.

The power problem can be solved by sending the processor to sleep with an idle( ) instruction when it is not doing anything useful (see main( ) Line 81 in Listing 3). For this to happen we needed to code up a system where a timer generates interrupts at 1 / 20 s intervals to cause the LEDs to flash (Lines 99 to 107); and enable the GPIO interface to generate interrupts whenever a switch is changed (Line ZZZ).

Such an “interrupt-driven” system works well, but there is one proviso for totally predictable operation in a safety critical environment -- the developer must ensure that no new interrupt can ever arrive during another interrupt’s service routine.

In reality, this condition can’t be met even in this simple case we are looking at here! There is no guarantee that a switch will not be pressed just as the timer interrupt occurs. In such a pre-emptive system, there is always the possibility for one task to be “paused” while a higher priority task is allowed to complete or to start. I agree that the time jitter associated with the delay of when the LEDs will be flashed is going to be small, and probably totally irrelevant in this example. However these simple tasks are intended as analogs of more complex audio and video algorithms that are triggered by interrupts; and we already suspect that the colour desynchronization issue discussed earlier is being caused by such an interrupt priority issue.

Enter the co-operative scheduler.

At the recent Embedded Systems Show held at Birmingham, UK in November 2008, I listened to a talk about time triggered technology by Michael Pont of TTE Systems Ltd., UK. In 2001, Pont had written a book Patterns for Time-Triggered Systems (now available for free download at http://www.tte-systems.com/books). I will admit that I stopped reading pretty early in that book; given it was focused towards the (old) 8051 processor and was1024 pages long. However after listening to Pont’s talk, I realized that I missed all the juicy bits after page 400+ -- the discussion on building and using a co-operative scheduler.

In Listing 4, I have re-used the switches and flashing LED light example to demonstrate the basic manner to code using such a scheduler.

- First you initialize the system using TTCOS_Init() (Line 123). This function sets up an array of structures to use as the co-operative scheduler’s “to-do” list. It also prepares the TTCOS timer which will be used to generate the only interrupt present in the system.
- Next you add each new task to this “run-list” using a series of TTCOS_AddTask() calls (Lines 129 to 135). The TTCOS_AddTask() function requires three parameters – the first is a pointer to the task (function) you want to have run. The second parameter is the initial task delay describing when you want the task to be run for the first time. The final parameter is the task period detailing how often you want the task to run. Setting the period of the sub-system initialization tasks to RUN_ONCE (Lines 129, 130) causes those tasks to be deleted from the “to-do” list once they have been run.
- The TTCOS timer is activated by TTCOS_Start() (Line 137)

Whenever the time interrupt service routine (Lines 148 to 151) is activated, the co-operative scheduler’s TTCOS_Update() code is run. This code checks through the “to do” list and, based on the current time and the delay and period of each task, determines whether a given task is to be run or left asleep. A key point here is that the Run-Me-Now variables for each task are incremented rather than just being set to 1. This guarantees that each occurrence of every task will be eventually be run even if the co-operative scheduler gets side tracked handling an error condition.

```c
int main(void) { 
    GPIO_Init();         // Initialize the GPIO input subsystem
    LED_Init();          // Initialize the LED output subsystem
    Timer_Init();        // Provide ability to measure elapsed time
    TTCOS_Start();

    Activate_GPIO_Interrupts(); // Activate GPIO interrupts to monitor switch operations
    Activate_TimerInterrupts(ONE_TWENTIETH_SECOND);

    while (1) {
        idle();      // Send processor to sleep when not in ISR
        
        return 0;    // Keep the compiler from complaining
    }

    EX_INTERRUPT_HANDLER(GPIO_Interrupt) { // ISR for when switch press occurs
        Acknowledge_GPIOInterrupt();

        int value = GPIO_Read(); // Read the input (GPIO) switch values
        Acknowledge_TimerInterrupt();

        LED_Write(value);       // Echo the switches to the LEDs

        if ((numberInterrupts & 1) == 0) {
            FlashLED4(); // LED 4 flashes every second interrupt
            numberInterrupts++;
        } else {
            FlashLED5(); // Flash LED 5 every 1/20th s interrupt
        }

    }

    LED_Init();                      // Initialize the LED output sub-system
    GPIO_Init();              // Initialize the GPIO input sub-system
    Activate_GPIO_Interrupts(); // Activate GPIO interrupts to monitor switch operations
    Activate_TimerInterrupts(ONE_TWENTIETH_SECOND);

    numberInterrupts = 0;

    EX_INTERRUPT_HANDLER(Timer_Interrupt) { // ISR to handle timer interrupts
        if (numberInterrupts & 1) { 
            FlashLED( );     // Flash LED 5 every 1/20th s interrupt
            numberInterrupts++;
        } else {
            FlashLED4();     // LED 4 flashes every second interrupt
        }
    }

    // Provide ability to measure elapsed time
    Timer_Init();
    GPIO_Init();

    // Read the input (GPIO) switch values
    int value = GPIO_Read();
    Acknowledge_GPIOInterrupt();
    return 0; // Keep the compiler from complaining
}
```
The scheduler now falls into an infinite loop consisting of

\[TTCOS\_DispatchTask()\]

(Line 140) and

\[TTCOS\_Sleep()\]

(Line 141) functions.

The \[TTCOS\_DispatchTask()\] function sequentially searches through the \[TTCOS\] task list for all tasks with a non-zero \[Run-Me-Now\] variable. runs that task and decrements the \[Run-Me-Now\] variable. Finally, the processor is sent into a low power mode (\[TTCOS\_Sleep()\]).

When the next timer interrupt reactivates the system, the \[Run-Me-Now\] flags are updated, the system wakes from the \[TTCOS\_Sleep()\] mode and the \[TTCOS\_DispatchTask()\] is called again.

### Building and using a basic co-operative scheduler

Coding a co-operative scheduler is straightforward. You need an array or linked “to-do” list containing information on required tasks and information of when to run them (the delays and periods set by \[TTCOS\_AddTask()\]) in Listing 4). To identify tasks ready to run, a timer driven interrupt service routine is used to search through the list and compare the current system time with timing information stored with each task in the to-do list.

Finally there is a routine that activates all tasks that have been identified as ready-to-run.

Given the fact that I have got a little rusty in setting up prototypes of subroutines that manipulate pointers to functions (Line 120), I took the easy route. I used ideas from the example code from Pont’s website rather than completely developing my own.

Using a co-operative scheduler for the tasks discussed above seems fairly intuitive. Each task that must be run is simply added to the scheduler’s “to-do list”. With each task independent of the

other tasks it is not complicated to test each task in terms of functionality and execution time.

One of the stated advantages of co-operative scheduler is that, since no tasks can pre-empt another, there are never any race conditions with two tasks fighting to access shared memory. That’s not important with this simple GPIO / LED tasks of Listing 4, but with the audio / video code discussed earlier, that may be a significant advantage in providing the route to avoid the data race issues we think are causing the problems within the LCD display code.

There is a famous saying – \textit{but the devil is in the details}. Is using a co-operative scheduler just that simple? The answer is yes -- if you are just want the co-operative scheduler as an organized form of “super-loop”. In that case, you will probably be okay with a little bit of uncertainty on when a task will run (time jitter).

To understand time jitter, consider a simple example of a co-operative scheduler handling two tasks. Using the \[TTCOS\_AddTasks(name, delay, period)\] syntax, one task is set to flash LED #4 for \(1/4\) second every three seconds after an initial delay of three seconds

\[TTCOS\_AddTask(FlashLED4, 3, 3)\]

while the other task flashes LED #5 every second after a zero initial delay.

\[TTCOS\_AddTask(FlashLED5, 0, 1)\]

If, for simplicity, we code both tasks using the form

\[void FlashLED\_N(void)\]

\[
\begin{align*}
&TTCOS\_AddTask(FlashLED\_N, 0, 1) \\
&while(1) \\
&\{ \\
&  &TTCOS\_DispatchTasks(); \\
&  &TTCOS\_Sleep(); \\
&  &\} \\
&return 0; \\
&\}
\end{align*}
\]

then the processor should easily be able to handle both tasks in the time available on the system. We can minimize the battery power used by setting the scheduler \[TTCOS\] timer to interrupt every second rather than more frequently.

We can test each task by making it the only task present in the scheduler’s to-do list. It is will be easy to determine that each code section flashes the right LED as required.

However, upon activating both tasks within the scheduler, we find that LED #4 is flashed every 3 seconds as required and LED #5 does start off flashing at one
second intervals. However after just three flashes, the time that LED #5 turns on starts fluctuating (time jitter). Sometimes LED #5 flashes one second after the last time it was turned on but sometimes after only 0.75 s and at other times the task is delayed to flash after 1.25 s.

Looking at the timing graph (Fig. 3), it is easy to see why this is happening. The TTCOS_Update code, driven by the TTCOS timer interrupt, will cause both the Run-Me –Now variables of the FlashLED4 and FlashLED5 tasks to be incremented every 3 s and every 1 s respectively.

There are no problems until the timer interrupt at time = 3 s when both tasks have been prepared to run. Since the processor can handle only one task at any one time, the first task in the “to-do” list runs (FlashLED4) and the second task (FlashLED5) is delayed.

For this example, there is the simple solution; change the order in which the scheduler runs the tasks by changing the order in which they are added to the scheduler’s to-do list

\[
\begin{align*}
&\text{TTCOS_AddTask(FlashLED5, 0, 1);} \\
&\text{TTCOS_AddTask(FlashLED4, 3, 3);} \\
&\text{TTCOS_Init();}
\end{align*}
\]

As seen in Fig. 4, we now have predictable operations with LED #5 now flashing at 1 second intervals with LED #4 flashing reliable at 3 second intervals. However, this reordering is not a real fix as adding a third task to flash LED #6 every 4 seconds will show.

\[
\begin{align*}
&\text{TTCOS_AddTask(FlashLED5, 0, 1);} \\
&\text{TTCOS_AddTask(FlashLED4, 3, 3);} \\
&\text{TTCOS_AddTask(FlashLED6, 4, 4);} \\
&\text{TTCOS_Init();}
\end{align*}
\]

The time jitter re-appears – now at 12 second intervals (Fig. 5)

**Solving the time jitter problem**

As seen above, rearranging tasks within the scheduler list will not ensure that tasks run with the exact period specified in the system requirements. Currently the way we are using the co-operative scheduler ensures (i) that during each one second period LED #5 will flash, (ii) that during every three second period LED #4 will flash and (iii) that during each four second period that LED #6 will flash etc.

However there is no guarantee that the LED #5 task will run exactly at one second intervals, or that the LED #6 task will run exactly at four second intervals. For that to occur, we need to pre-arrange the time when every task runs rather than re-arrange when each task runs.

This “un-predictability” issue can be solved in the following manner. Since, by design, there are no interrupts other than the scheduler interrupt permitted, then we can accurately predetermine the length of time that each task will run. We can then arrange that the initial delay parameter used when adding the task to the scheduler’s to-do list was “just enough” to ensure that no task is ready to run until after a previous task has completed. For the 3 LED example we would need to add the tasks to the scheduler with the following initial delays and periods:

\[
\begin{align*}
&\text{TTCOS_AddTask(FlashLED5, 0, 1);} \\
&\text{TTCOS_AddTask(FlashLED4, 3.25, 3);} \\
&\text{TTCOS_AddTask(FlashLED6, 4.5, 4);} \\
&\text{TTCOS_Init();}
\end{align*}
\]

However, just changing the task delay is not sufficient to stop the time jitter. With a one second timer interrupt, the TTCOS_Update() code will prepare the LED #4 task to run after a 4 second delay, not after the desired 3.25 s delay. Similarly the LED #6 task will be prepared to run after 5 seconds and not after the 4.5 s needed to solve the time jitter problem. Thus the time jitter will simply be re-introduced elsewhere in the project.

This problem can be solved if we speed up the timer interrupts so that the scheduler is capable of handling “a quarter second delay” as shown in Fig. 6. The final time triggered schedule code for the 3 LEDs flashing at predictable intervals is shown in Listing 5.

In this listing, the TTCOS_Init function prototype (Line 168) has been changed so that the optimum interrupt time for maximum power saving can be set for each co-operative scheduler project; 1/4 s interrupts for this project. The tasks delays (Lines 157 to 159) and task periods (Lines 161 to 164) are not defined as specific time intervals but as multiples of the timer interrupt period. Describing the delay or period times using a float variable would not lead to any greater precision as the Run-Me–Now variables for each task are only updated by the TTCOS_Update routine each time the timer interrupt occurs.

**Where to next?**

In this article, I explained my own re-awakening to the advantages of co-operative schedulers. According to reports, such schedulers have the potential to offer an advantage that pre-emptive schedulers don’t offer – predictability of performance. After detailing with some syntax issues, the simplicity and code maintainability of using a co-operative scheduler for handling multiple tasks was demonstrated. However, it was then shown that this “bull-at-the-gate” approach of using co-operative schedulers is acceptable only when the embedded project requirements can be satisfied with a predictability of the form – this task must run sometime during a period of X seconds. Using this predictability definition is often acceptable when the project functionality will not be compromised by a certain level of “time jitter” (random delay) in the execution time of the tasks. However it is different than meeting the stricter requirement often required in safety critical system that the task must run precisely every X seconds.

A simple approach was introduced to overcome the time jitter problem. This involved speeding up the interrupts and pre-calculating task delays to ensure that no two tasks were ever prepared by the co-operative scheduler to be ready to run at the same time.

However many practical embedded issues remain unresolved in the use of a co-operative scheduler. For example, suppose there are tasks that take 3 seconds to complete (e.g. output of a buffer) while other tasks must be executed every 1/4 second (immediate echo of a value in an input channel to an output channel as shown in Listing 5, Line 182). Direct application of the co-operative scheduler appears to suggest that the reliable operation of the shorter running task will be compromised (blocked) by the longer running task. It actually sounds as if the project would be better handled by a
scheduler where the faster task pre-empts the slower task. I plan to look at these issues in other articles examining the use of time triggered systems. There I will look at these, and other issues, and show that they can be overcome to recover the predictability stated as being available using co-operative schedulers.

Mike Smith is a long time contributor to Circuit Cellar magazine, who disappears into the woodwork for a couple of years until he finds something ‘new’ that really sparks his curiosity! 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 multicore and multiple-processor embedded systems in a systematic and reliable fashion. Mike has recently become a convert to the application of Agile Methodologies in the embedded environment. Mention “test driven development and his eyes light up. In 2008, Mike had his Analog Devices University Ambassadorship renewed for the eighth straight year. He can be contacted at Mike.Smith@ucalgary.ca.

Listing 5. By analyzing the system ahead of time, it is possible to schedule all the tasks so that no task is ever waiting for another task to complete before being run by the scheduler. With this approach, each task operates with complete predictability and reliability. However, if the GPIO_LED_Echo() task to echo the switch input to the LED output was activated (Line 184), then this task would suffer from “time jitter” as it is continually blocked and then unblocked by the other tasks.

```c
155  #define QUARTER_SECOND_INTERRUPTS (250 * 500 * 1000)  // Processor clock cycles per 1/4 second
156
157  #define NO_DELAY       (0)                              // 0 interrupt delay
158  #define QUARTER_SECOND_DELAY   (1)                  // 1 interrupt delay
159  #define HALF_SECOND_DELAY       (2 * QUARTER_SECOND_DELAY)  // 2 interrupt delay
160
161  #define RUN_ONCE        (0)             // Delete task if not periodic
162  #define EVERY_SECOND        (4)                               // 1 second task period = Every 4th interrupt
163  #define EVERY_THREE_SECONDS       (3 * EVERY_SECOND)          // 3 second task period
164  #define EVERY_FOUR_SECONDS       (4 * EVERY_SECOND)          // 4 second task period
165
166  int main(void) {
167    TTCOS_Init(QUARTER_SECOND_INTERRUPTS);  // Initialize the co-operative scheduler 'to-do' list and timer
168
169    // Add Tasks
170    // TTCOS_AddTask(FunctionPointer, taskDelay, taskPeriod);
171
172    // The sub-system initialization tasks run with NO_DELAY and RUN_ONCE only
173    TTCOS_AddTask(GPIO_Init, NO_DELAY, RUN_ONCE);       // Initialize the GPIO input sub-system
174    TTCOS_AddTask(LED_Init, NO_DELAY, RUN_ONCE);      // Initialize the LED output sub-system
175
176    // Specify delay and period of other tasks
177    TTCOS_AddTask(FlashLED5, NO_DELAY, EVERY_SECOND);  // Every second for 1/4 s
178    TTCOS_AddTask(FlashLED4, QUARTER_SECOND_DELAY, EVERY_THREE_SECONDS); // Every 3 seconds for 1/4 s
179    TTCOS_AddTask(FlashLED6, HALF_SECOND_DELAY, EVERY_FOUR_SECONDS);  // Every 4 seconds for 1s/ 4 s
180
181    // Task to echo switch input to LED output realistically needs to be run every 1/4 s
182    // ***** If activated, this task would have "considerable" time jitter as it is "blocked" by other tasks
183    //***** TTCOS_AddTask(GPIO_LED_Echo, NO_DELAY, RUN_ALWAYS);
184
185    TTCOS_Start();  // Activate the co-operative scheduler's timer interrupt
186
187    while (1) {  
188      TTCOS_DispatchTasks(); // Run any tasks with a Run-Me-Now variable that is not zero
189      TTCOS_Sleep();  // and send the processor back to sleep until next timer interrupt
190    }
191
192    return 0;    // Keep the compiler from complaining
193  }
```
Adding the tasks `TTCOS_AddTask(FlashLED4, 3, 3)` and `TTCOS_AddTask(FlashLED5, 0, 1)` to the co-operative scheduler task list leaves two tasks ready to run every 3rd second. The second task is forced to wait until the processor completes the first task. The second task is said to suffer from “time jitter” – shown in red.

Adding the tasks to the scheduler list in a different order `TTCOS_AddTask(FlashLED5, 0, 1)` and `TTCOS_AddTask(FlashLED4, 3, 3)` still leaves two tasks ready to run every 3rd second. As the second task is consistently forced to wait until the processor completes the first task, the system acts in the required manner, with LED 5 flashing at one 1 second intervals and LED 4 flashing every 3rd second with no time jitter present.

No re-ordering of the tasks will avoid the time jitter (shown in red) when a third task `TTCOS_AddTask(FlashLED6, 0, 4)` is added to the system. This task flashes LED 6 every 4th second.

By speeding up the scheduler interrupts four fold, and predetermining appropriate initial delays for each task, no task is forced to wait until another task completes. All tasks complete at their designated time intervals without time jitter.