Design Aids - Multi-Context Switch Event Kernel

AN2392

Author: Uroš Platiše
Associated Project: Yes
Associated Part Family: All
GET FREE SAMPLES HERE
Software Version: PSoC Designer™ 4.2
Associated Application Notes: PSoC Application Notes Index

Abstract

This application note describes an event-based, multi-context switching micro-kernel for PSoC® devices. It uses very little memory and CPU time, improves system response time, and ensures that high priority code is executed quickly. It can be incorporated into existing designs very easily.

Introduction

PSoC mixed-signal architecture features tightly coupled analog and digital blocks that often require some post processing. Blocks trigger the CPU via individual interrupt vectors. The appropriate post processing code for each interrupt is provided by an Interrupt Service Routine (ISR).

If you have time-critical code, you can improve the system response time of the ISRs by selecting higher CPU clock speeds and hand coding ISRs in assembly language. However, the PSoC CPU power is limited, and this will only increase response speeds up to a point. Increasing the system clock speed also increases power consumption. If you wish to guarantee low latency system response times and reduce power consumption, you must employ a different concept of code development.

This Application Note describes an event-based, multi-context switching micro-kernel that requires 2 bytes of SRAM and minimal Flash memory. An additional byte of SRAM may be reserved for system overload error detection. The micro-kernel is created as a separate module that you can use in existing PSoC designs. To make use of the kernel, you split your current ISRs into an ISR that is as short as possible, and a series of events. Each shortened ISR does only the necessary time-dependent IO transactions with the analog or digital blocks, and then triggers an event with an assigned priority. The event is executed as soon as all ISRs and events with higher priority have completed.

Key Features

- Multi-context switching micro-kernel
- Priority-based execution
- Programmable context levels
- ISR low latency
- 2 or 3 bytes of SRAM, < 100 bytes of Flash
- System overload detection
- Stack frame check
- Dynamic memory allocation
- Easy to use

Applications

- Realtime applications
- Replacement for long ISRs
- Communication software (modems, I2C, WirelessUSB™, and so on.)
- Filters and regulators
- Library modules

[+] Feedback

Design Structure

The kernel is composed of a small, priority-based event handler (scheduler). Up to eight events are supported. Each event requires a single byte for masking and another byte representing current status. You run the event handler each time selected ISRs exit by preserving the stack memory and jumping directly to the scheduler. The event handler checks for events and executes them in the appropriate order with global interrupts enabled. Events are executed one at a time or in several context layers until there is enough stack space available.

The event handler kernel is found in three files:

- *eventhnd.asm*
- *eventhnd.inc*
- *eventhnd.h*

Include all three files in your project. You do not need to make changes in *eventhnd.asm.* You should, however, adapt the .inc and .h files to your project. There are numerous adjustable features that control performance and error checking mechanisms. You will also need to set the number of events used and the event nicknames. The 'C' header file provides macros to trigger events and user project nicknames of the events.

The following sections describe operation and use in greater detail.

Events

An event is a block of finite code that is protected by a semaphore. Each event has an assigned priority from one to eight. An event may take arguments and return values through global variables. The event is triggered by setting a dedicated bit that is reset after its execution completes. The dedicated bit is the event's global semaphore.

Given a list of events, the event handler will schedule them in a specific order and will either handle them one at a time, or allow preemptive multitasking.

Events are always executed one context layer above the main application, and one context layer below the ISRs. This results in a maximum of ten context layers for eight events. Events with a higher priority are run before events with a lower priority. If an event with a higher priority arrives when a lower priority event is executing, the higher priority event immediately preempts the lower priority event if it is able. A higher priority event may fail to preempt a lower priority event if there is not enough stack space to store the context of the lower priority event or if the maximum number of context levels has already been reached.

Event priorities play an important role in the design of your software. Events with lower priorities may take much longer to execute because events with higher priorities preempt them, creating new context layers above them.

Example 1

A long ISR may be split into three parts:

1. ISR - The new, much shorter ISR reads critical register values and triggers EVENT0.
2. EVENT0 - This event executes the most time critical post-processing tasks, feeds the appropriate hardware registers, and triggers the lowest priority event, EVENT7.
3. EVENT7 - This event executes as a background process, executing more complex computations, such as encryption.

Events between EVENT0 and EVENT7 are never blocked for long. EVENT0 has the highest priority but is very short. EVENT7 may take a long time, but can be preempted by any other event.

Event Scheduling

A new context layer may be created by an interrupt or a simulated interrupt. To simulate an interrupt, disable global interrupts and push the appropriate registers (depending on architecture) to the stack. The first method is more common. Because the ISR already runs in the highest context layer, it may create a lower context level to run events. The ISR accomplishes this by calling the event handler rather than the return from interrupt instruction.

Events can be triggered by any part of the code; application code, other events, or ISRs. However, if application code calls an event, the event will not automatically create a higher priority context layer to run the event handler. The application must either emulate the interrupt by calling the `scheduleEvents` function immediately or wait for the interrupt to be triggered normally.

Scheduler Policy

Ideally, the device would always have enough stack memory to provide a separate context layer for each event. However, in a PSoC device with 256 bytes of SRAM, this may not be possible even if the stack use per event is very small.

To handle this, you can adopt one or both of these scheduling policies:

- A fixed number of context layers
  - Compact
  - Fast
  - Requires an explicit list of events
  - Allows events to be executed with preemptive multitasking
- A dynamic number of context layers based on a stack frame check
  - Creates new context layers until stack memory is full
  - Requires a single parameter that sets the maximum stack space use
  - Stack safe but requires a few clock cycles per stack check

[+] Feedback

Error Checking

Two checks are supplied to detect system overload:

- Event throughput (CPU overload)
- Stack space (SRAM full)

Both errors are signaled in a global event error variable, `eventerr`. Event throughput errors occur when an event is called before the previous instance of the same event has completed. Each time this happens, the second instance of the event is discarded and the `eventerr` variable is incremented.

Event throughput errors signal one of the following:

- The CPU is overloaded.
- The event priorities were not set properly.
- A stack full error has occurred.

The most significant bit of the `eventerr` variable is set on a stack full error. A stack full error happens when the system does not have enough memory to run the event. You can check the most significant bit of the `eventerr` variable to determine whether a stack full error has occurred. When a stack full error is detected, no additional events can be scheduled until the stack space is freed.

This implies that the system has reached critical memory usage and you should consider the following:

- If it happens frequently, your application software may need to be redesigned to place less of a burden on resources.
- A stack overflow error may have occurred. To check this, set the last byte in SRAM to a known value and check to see if it is overwritten.
- You may wish to design a special case when stack usage is high that allows ISRs to execute, but no events. In this case, you would need to restart the event handler after stack space is released or wait for another ISR to trigger it. Note that event throughput errors may occur if you do this.

Use

Include the three *eventhnd.** files in your project, then configure *eventhnd.inc* and set the nicknames in the *eventhnd.h* file for use in 'C'.

Event Declaration

Events start and end with EVENTx declaration, where x is the event number. Events are numbered from 0 to 7. Zero represents the highest priority and seven the lowest:

```
export _EVENTx:
_EVENTx:
    ...
eventEnd EVENTx
```

When you want an event to be able to call itself, the end of the declaration changes:

```
export _EVENTx:
_EVENTx:
    ...
eventRecall EVENTx
```

Calling the Event Handler

ISRs that trigger events should start the event handler by jumping to it directly instead of returning from interrupt `(reti)`:

```
ISR_begin:
    ...
    eventHandler
```

Call higher priority events from within other events or application code by placing a call to the `scheduleEvents` function.

Triggering and Status

Three functions are provided to trigger events and provide event status.

event() - This function triggers one or more events passed as arguments:

```
// assembly language
event <EVENTx | EVENTy | ... | EVENTz>
/* C language */
event(EVENTx | EVENTy | ... | EVENTz)
```

If error checking is enabled, the `eventerr` variable is incremented if at least one of the events in the list has a current instance that has not completed processing.

isEventPending() - This function returns a nonzero value if the event is pending. It returns zero if the event is not pending. In assembly, the function sets the Z flag. This function and isEventProcessed() may be used to protect shared variables used by the event.

```
// assembly language
    isEventPending <eventNames>
/* C language */
    isEventPending(eventNames)
```

isEventProcessed() - This function returns a nonzero value if the event process is complete. It returns zero if the event processing is not complete. In assembly, the function sets the Z flag.

```
// assembly language
   isEventProcessed <eventNames>
/* C language */
   isEventProcessed(eventNames)
```

Configuration

The number of events, event nicknames, and scheduling policy is configured in *eventhnd.inc*.

EVENT_ERROR_CHECK - When enabled (set to 1) the error handling code and the `eventerr` variable are compiled into the project.

EVENT_CPU_MNG_ENTER - When enabled (set to 1) it sets the CPU frequency to maximum (24 MHz) immediately after the event handler is called. This feature would be more useful if it were executed at the beginning of the ISRs generated by the PSoC code generator.

EVENT_CPU_MNG_LEAVE - When enabled (set to 1) it restores the CPU frequency to system default after the last event has completed.

EVENT_MML eventName1|eventName2|... - The multi-context mask level (MML) sets which events may be preempted by higher priority events. For example:

```
EVENT_MML EVENT3|EVENT4
```

Setting the variable as shown in the example allows events 3 and 4 to be preempted by any event with a higher priority. Event 3 can be preempted by events 0, 1, and 2. Setting the list to all available events permits all events to be preempted by higher level events, removes the MML logic, and saves a little processing time.

Setting it to 0 does not allow any preemption. In this case, the scheduler always runs the highest priority event first, but if a higher priority event is called by a lower one, the lower priority event must finish before the higher priority event starts.

EVENT_STACK_CHECK - When enabled (set to 1) it checks for stack space before creating a new context layer. When using this stack check the MML feature still helps to ensure that higher priority events are executed first.

EVENT_STACK_MIN bytes - Defines the minimum stack size that must be available before the event handler schedules an event. It calculates the minimum stack size as the sum of the maximum stack space required by the ISR plus the stack space required by the event that requires the maximum amount of stack space. This gives you a worst-case stack use for an event. The call to the ISR consumes 3 bytes of stack memory.

EVENTn value - When set to 0, disables an event. To enable EVENTn, set it to 2^n. For example, to enable events 0 through 4:

```
Event0 0x01
Event1 0x02
Event2 0x04
Event3 0x08
Event4 0x10
Event5 0x00
Event6 0x00
Event7 0x00
```

nickname value - You can provide nicknames for events by equating nicknames with the same values used when enabling the events. For example:

```
I2C 0x01
MyEvent 0x02
```

This allows you to call the events with nicknames, which makes your code easier to read.

```
Event I2C
```

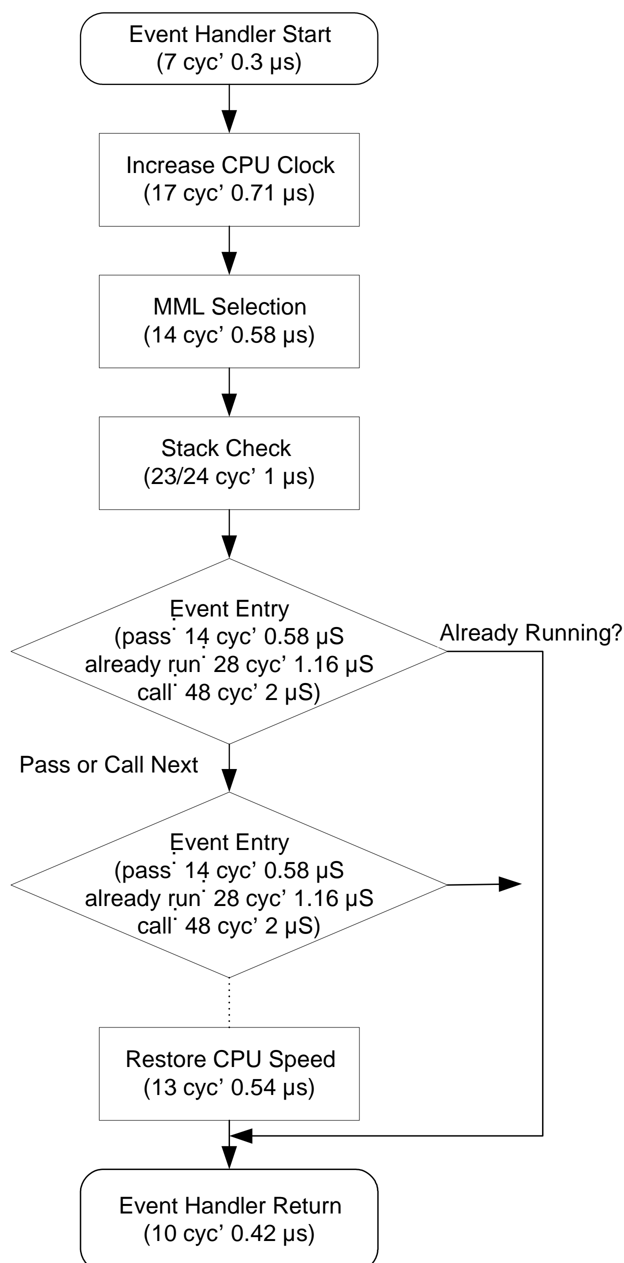
To use the nicknames in 'C' the same nicknames should be placed in the *eventhnd.h* file.

System Performance

Figure 1 on page 5 represents the program flow chart and CPU use in cycles and µs. The time calculations assume the CPU is running at its maximum speed of 24 MHz. The minimum possible latency time for an ISR is computed as the time interval during which the global interrupt flag is disabled. The worst case (maximum) latency time is calculated as the length of time that it would take all available code in the ISR to execute once.

[+] Feedback

Figure 1. Event Handler Flow Chart



With CPU clock management disabled, MML and stack frame check enabled, eight events total, and the last event called but not yet scheduled, it takes 191 cycles (8 µs) before global interrupts are re-enabled and the last event is scheduled. If we use the delta sigma A/D conversion ISR as an example, the ISR takes 190 cycles (another 8 µs) before calling the event handler. This results in a worst case latency of 16 µs.

Interrupt latency may be further increased in the event handler by re-enabling global interrupts (after the MML check, after the stack frame check, and after each event entry) for a short time. The time increase in the event handler for this is 2 µs, resulting in a minimum latency of 10 µs.

To ensure higher priority events are always executed, the EVENT_MML value should list only those events with large computation times (as it makes no sense to interrupt short events). Short, high priority events should be executed sequentially in the same context level.

[+] Feedback

Summary

The multi-context switching micro-kernel event described in this Application Note is compact and efficient. It increases system performance with minimal changes in the existing software design structure and offers a multi-threaded environment on PSoC devices with minimal system requirements, dynamic context switch levels, and error tracking.

Events should be written in assembly language as they may be executed in parallel with application main() software and other events, as assembly language is significantly more efficient than 'C' in time-critical applications.

Events are ordered according to their priorities. Applications requiring more than eight events may utilize event groups with a second layer event scheduler.

About the Author

Name:	Uroš Platiše
Title:	Research & Development
Background:	Project Manager, mixed-signal hardware and software design, sensors, measurement and control, mission critical solutions, and parallel and distributed systems.
Contact:	Uros Platise Seljakovo naselje 45 SI-4000 Kranj Slovenia uros@andeuros.org http://www.andeuros.org

In March of 2007, Cypress re-cataloged all of its Application Notes using a new documentation number and revision code. This new documentation number and revision code (001-xxxxx, beginning with rev. **), located in the footer of the document, will be used in all subsequent revisions.

PSoC is a registered trademark of Cypress Semiconductor Corp. "Programmable System-on-Chip," PSoC Designer and PSoC Express are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com

© Cypress Semiconductor Corporation, 2007. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

[+] Feedback