Managing High-Rate Interrupts in Multitasking Applications on Low-Cost Embedded Systems

speaker:

Andrew E. Kalman, Ph.D.
President, Pumpkin, Inc.

Part I

Why Use Interrupts?

Synchronous applications

* Embedded applications that do not use interrupts are inherently synchronous and deterministic. Program flow is very tightly-coupled. Outputs are controlled via highly sequential code. Inputs are explicitly sampled at defined times, therefore polling is required.
* State machines, jump tables, etc. are often used to control program flow.
* Performance is often characterized by an overall processing rate – e.g. all functionality (input-to-output) is repeated at 100Hz.
* All of an application's code operates in a single layer.
* Example: 1200bps serial-to-AX.25 amateur packet-radio link-layer protocol converter (implemented on a PIC12).
* Benefits: Easy to write, understand and analyze. Often used in situations where application functionality is narrowly defined.
* Disadvantages: Inefficient use of processor power, poor responsiveness, not well-suited to low-power (/sleep) applications, applications are not easily expanded.

Asynchronous applications

* The real world is inherently asynchronous – i.e. you usually cannot predict when an external event will occur.
* An application must respond quickly to events or else they may be lost. The more efficient the response (in terms of CPU cycles, etc.), the higher the number / frequency of events that the application can handle.
* Since asynchronous events are unpredictable, interrupt capabilities were added to µC designs in order to efficiently handle asynchronous events.
* When using interrupts, an application's code operates in two layers: background / `main()` code and foreground / ISR code.
* Code can be very tightly coupled (e.g. synchronous main loop, which samples outputs from ISRs) or more loosely coupled (e.g. most processing performed in individual ISRs).
* Example: Solar-powered low-earth-orbit CubeSat Kit picosatellite. Operates at low power levels (including sleep) until it receives commands from ground station, whereupon it wakes up and begins transmitting data.
* Benefits: Rapid response to asynchronous events, more efficient use of processor power because no polling is required, well-suited to low-power / sleep modes, efficient coding possible.
* Disadvantages: Much more difficult to analyze because application behavior is a function of external events. Requires more RAM (for stack).

A simple example: 1200 baud RS-232 reception on GPIO pin

* Without interrupts, code must sample[1] the Rx signal on the GPIO pin every 1/(1200 x 2) = 416µs, and begin assembling incoming data into a received character via a state machine once the start bit is detected. The programmer must ensure the application always samples the GPIO pin every 416µs regardless of the state of the receiver state machine and any other parts of the application. Careful hand-coding and cycle-counting is often required. Any other application functionality must be carefully "time-sliced" into the 416µs window of available processing time. This very tightly-coupled code is often non-trivial to write.

* With interrupts, an interrupt is made to occur every 416µs, e.g. via a simple timer interrupt. Inside the ISR, the state machine receives the start, data and stop bits to assemble a received character. The only requirement on the rest of the application is that interrupts not be disabled for more than 416µs, else Rx data might be lost. The rest of the application is very loosely-coupled.

* In both cases, assuming 1µs/instruction cycle and 80 instruction cycles for the Rx state machine + overhead,[2] (80 inst x 2400 /s) / (1,000,000 inst/s) = 20% of CPU power is spent on RS-232 reception.

Another example: 1200 baud RS-232 reception via built-in USART

* The addition of a real USART reduces the load on the processor even further. Assuming 1 start bit, 8 data bits and one stop bit, the USART receives a new character every 1/(1200 bps / 10 bits) = 8.33ms.
* Without using interrupts, the code must read the incoming Rx data every 8.33ms or faster. The code's `main()` must ensure that this rate is always met, regardless of any other processing (e.g. library calls, long delays, etc.).
* By using interrupts, the `main()` is interrupted every 8.33ms with the arrival of a new character. The only requirement on the rest of the application is that interrupts not be disabled for more than 8.33ms, else Rx data might be lost. Again, the rest of the application is very loosely-coupled, and can tolerate library calls, long delays, etc.
* The USART has the additional advantage that the CPU need not poll the start bit. Assuming 20 instruction cycles to move the Rx data to the main application, the worst-case load drops by a factor of 80 to (20 inst x 120 /s) / (1,000,000 inst/s) = 0.24%.[3]

We use interrupts because …

* It's much more efficient to handle asynchronous / unpredictable events via interrupts. Interrupts remove the need for polling. Polling for changes on inputs is a waste of cycles when no changes occur.
* Code in `main()` is much simpler to write – no major time constraints are imposed upon it. No need to "schedule" event polling. Code can be written without regard for the ISRs as long as a few basic requirements are met.

Part II

Using Interrupts in Embedded Systems

When using interrupts:

* We must ensure that interrupts are never disabled for longer than the critical period of the fastest ISR. Interrupt latency must be kept to a minimum. Missed interrupts = lost data.
* ISR overhead should be kept to a minimum. Unnecessary overhead = wasted CPU cycles = reduced performance.

Interrupts can come from internal and/or external sources

* Internal interrupts: timers, DMA, Tx communications, etc.
* External interrupts: captures, change-on-pin, Rx communications, etc.

Interrupts often have priorities and/or vectors associated with them

* PIC18: two separate programmable priority levels (2 vectors)
* MSP430: multiple interrupt vectors, fixed priority scheme

Proper use and control of interrupts is often one of the hardest skills for embedded programmers to master.

Interrupt control

* µC's usually have a global interrupt enable bit (`GIE`).
* µC peripherals usually have individual interrupt enable bits (`pIE`).
* For a particular ISR to be active, both the `GIE` and the associated `pIE`(s) must be enabled.
* Applications normally run with global interrupts enabled.
* When accessing a global variable shared between `main()` and ISR code, access to the variable from `main()` must occur with interrupts disabled, else corruption may occur.

Example: MSP430 USART transmit routines

*** User calls `MSP430PutcharTx0(char)` from main(). Data is added to the outgoing / Tx user ring buffer via `tx0InP` (not shared). After data is added to buffer, interrupts are disabled while Tx interrupt enable bit is set and `txCount` (shared) is incremented. Note that global interrupts are disabled for a very short time (3 instructions).**

*** On the MSP430, the act of enabling Tx interrupts causes a Tx interrupt to occur (USART's Tx buffer is empty and ready to accept a new character). Therefore Tx0 ISR calls `MSP430SendcharTx0()`, which pulls a char out of the ring buffer via `tx0OutP` (not shared), sends it out via the USART, decrements `txCount` and disables further interrupts if the ring buffer is now empty. Note that no interrupt control around `txCount` is required, because interrupts are already disabled inside of an ISR.**

Example: MSP430 USART transmit routines

```
unsigned char MSP430PutcharTx0(unsigned char data)
{
   if (tx0Count < TX0_BUFF_SIZE)      /* room in buffer?       */
   {
       tx0Buff[tx0InP++] = data;      /* yes, add to buffer    */

       if (tx0InP > TX0_BUFF_SIZE-1)  /* wrap ptr if req'd     */
           tx0InP = 0;

       _DINT();                       /* shared access to      */
       IE1 |= UTXIE0;                 /*  global vars          */
       tx0Count++;                    /*  ""                   */
       _EINT();                       /*  ""                   */

       return TRUE;
   }
   else
   {
       return FALSE;                  /* buffer full           */
   }
}
```

Example: MSP430 USART transmit routines (cont'd)

```
void MSP430SendcharTx0(void)          /* call from ISR          */
{
    TXBUF0 = tx0Buff[tx0OutP++];      /* send data out          */

    if (tx0OutP > TX0_BUFF_SIZE-1)    /* wrap ptr if req'd      */
        tx0OutP = 0;

    tx0Count--;                       /* that char is gone      */

    if (tx0Count == 0)                /* no need for more ints */
        IE1 &= ~UTXIE0;               /*  if no more chars      */
}
```

Part III

How Other People's Code Affects Interrupts in your Application

When you control interrupts

* You know (or at least you can analyze) how long interrupts are disabled in your own code.

What happens when you add these “black boxes” to your application?

* Standard libraries (e.g. floating-point math)
* Other “canned” code (e.g. peripheral libraries / routines)
* Multitasking kernels / RTOSes

Interrupt latency, response and recovery times become critical

Interrupt control may be beyond your control

* To be fully reentrant, a function may only operate on data on the stack. Often due to architectural limitations, libraries may include non-reentrant functions that must be protected against reentrancy via interrupt control. This occurs because of the need for temporary global variables to hold intermediate results. Examples include software multiply (e.g. for array element lookup) and 32x32 bit multiply on an 8-bit PICmicro®.

* Similarly, RTOSes usually disable interrupts to protect shared RTOS objects, the stack, etc.

Solving the reentrancy problem

* Reentrancy can be avoided simply by ensuring that there is only a single call tree for any function(s) that are not reentrant. This usually means that these functions can only be called from `main()` or a single, non-nested ISR. Satisfying this requirement may require changes to your code.
* A brute-force solution to the reentrancy problem is to duplicate the required function and use separate instances for `main()` and interrupt-level code.
* Another solution is for the compiler to automatically protect shared globals as part of interrupt save / restore.
* Non-reentrant functions can prevent reentrancy via interrupt control inside the functions themselves (critical sections).

Interrupt control

* Library functions that include global interrupt control (e.g. for EEPROM writes) must be analyzed for their impact on interrupts. If they disable interrupts for too long, they need to be re-coded or avoided altogether.

Multitasking and interrupt control

* Broadly speaking, multitasking requires some sort of context switching as the scheduler switches program execution from one task to another. Context switching involves the stack and changes to the stack pointer (SP). Context switchers are often written in assembly. Since the SP is a shared resource, it is likely that interrupts must be disabled globally during context switches.[4] Hopefully, the kernel / RTOS you are using disables interrupts globally for a minimum number of instruction cycles.

RTOSes and critical sections

* RTOSes also have objects (e.g. task control blocks) that are global in nature. RTOS services that access these objects are not normally reentrant. Access to these objects must be protected to avoid corruption when RTOS functions are called from the interrupt level. By default, most RTOSes protect critical sections by disabling interrupts globally. Note, however, that all that is required is that the critical sections be protected against reentrancy caused by those interrupt sources that call RTOS services.
* Figure 1 shows the number of instruction cycles for which (global) interrupts are disabled by kernel services in the popular µC/OS RTOS. Figures for other RTOSes are likely to be similar.

Table 6.1

80186/80188 (Small Memory Model)				
KERNEL SERVICE	INT. DIS.	MINIMUM	MAXIMUM	NEW TASK
	(CPU Clock Cycles)	(CPU Clock Cycles)	(CPU Clock Cycles)	(CPU Clock Cycles)
OSIntEnter()	25	50	50	-
OSIntExit()	375	275	450	400
OSMboxCreate()	75	250	475	-
OSMboxPend()	400	200	1,300	850
OSMboxPost()	400	125	1,050	550
OSQCreate()	75	250	800	-
OSQPend()	400	250	1,250	850
OSQPost()	400	150	1,150	1,050
OSSchedLock()	25	50	50	-
OSSchedUnlock()	400	100	575	500
OSSemCreate()	75	400	400	-
OSSemPend()	400	125	1,225	650
OSSemPost()	400	150	1,050	1,000
OSTaskChangePrio()	500	1,425	1,650	1,600
OSTaskCreate()	400	850	1,500	-
OSTaskDel()	400	725	950	900
OSTimeDly()	400	425	650	625
OSTimeGet()	50	125	125	-
OSTimeSet()	50	100	100	-
OSTimeTick()	150	4,510	10,810	-

Kernel Services Execution Times
in CPU Clock Cycles

Figure 1: For µC/OS

The effect of critical sections

* The µC/OS kernel services in Figure 1 show a maximum of 500 instruction cycles during which global interrupts are disabled during a critical section ($t_{\_int\_disabled}$). Therefore this application using this RTOS cannot support any interrupts whose critical period is shorter than 500 instruction cycles.

Interrupt control in critical sections

* Most RTOSes are compiled with global interrupt control in critical sections, e.g.

```
#define OSEnterCritical  _DINT() /* disable interrupts globally    */
#define OSExitCritical   _EINT() /* enable interrupts globally     */
```

These definitions guarantee that regardless of which interrupts involve calls to the RTOS services, no corruption of the shared RTOS objects is possible. I.e. they prevent reentrancy through any ISR. The disadvantage of this approach is that all interrupt sources are affected by the RTOS's maximum $t_{\_int\_disabled}$ time.

Tailoring interrupt control to your application

*** If instead we redefine the critical section code to be e.g.[5]**

```
#define OSEnterCritical  (TMR0IE = 0) /* disable Timer0 interrupts */
#define OSExitCritical   (TMR0IE = 1) /* enable Timer0 interrupts */
```

Now the RTOS no longer affects interrupts globally in its critical sections. $t_{\_int\_disabled}$ is effectively 0 (!) for all interrupts other than Timer0's interrupt. Therefore high-rate interrupts can coexist with the RTOS on all interrupt sources other than Timer0. With this specific critical section code, RTOS services (e.g. `OSTimer()`) can only be called from those interrupts that are disabled during critical sections – e.g. from the Timer0 ISR.

*** Interrupt control in critical sections can be expanded to handle more than just a single interrupt enable.**

*** Access to the RTOS source code is required to implement changes to the critical section macros.**

Example: Salvo RTOS + I2C on 4MHz PIC18 PICmicro® MCU

*** The default clock speed for I2C is 100kHz. An interrupt is generated (`SSPIF`) every time a 9-bit I2C packet is received, i.e. every 90µs. At 4MHz, the PIC's instruction cycle is 1µs. Therefore I2C interrupts must be serviced every 90 instruction cycles to avoid errors.**

*** Salvo's context switcher for the PIC18 (using HI-TECH PICC-18 compiler) does not involve the stack, etc. The standard critical-section protection is used.**

*** Many of Salvo's critical sections (e.g. priority queue resolution, etc.) will take longer than 90 instructions. Therefore the default Salvo configuration for protecting critical sections:**

```
#define OSDi()  do { GIEH = 0; GIEL = 0; } while (0)
#define OSEi()  do { GIEH = 1; GIEL = 1; } while (0)
```

is inadequate in this situation, because all interrupts (`GIE/GIEH` and `PEIE/GIEL`) are disabled globally during context switching and critical sections. The interrupt latency for the I2C interrupt will exceed 90 instruction cycles due to interrupt control by the RTOS.

* The solution is to configure the PIC18 for mixed-priority interrupts, elevate the I2C subsystem to high-priority interrupts, and configure Salvo to only disable low-priority interrupts in critical sections.

```
RCON:IPEN   = 1; /* enable priority levels on interrupts  */
PIE1:SSPIE  = 1; /* enable MSSP (I2C) interrupts          */
IPR1:SSPIP  = 1; /* MSSP interrupt priority = high        */
INTCON:GIEL = 1; /* enable low-priority interrupts        */
INTCON:GIEH = 1; /* enable high-priority interrupts       */
…
#define OSDi()  do { GIEL = 0; } while (0)
#define OSEi()  do { GIEL = 1; } while (0)
```

* With this configuration, your application can be in the middle of a Salvo critical section with low-priority interrupts disabled, and an I2C interrupt will be serviced immediately.
* If you use more than one high-priority ISR, high-priority interrupt response times will be based solely on PIC hardware and software-induced priorities.[6] I.e. they are not affected by the RTOS control of critical sections.
* With Salvo disabling low-priority interrupts in its critical sections, Salvo services (e.g. `OSTimer()`) can be called from any low-priority ISR. Calling Salvo services from a high-priority ISR in this configuration will result in data corruption sooner or later.

*** The careful reader will wonder "How do I pass information from a high-priority ISR up to the RTOS, if I can't call RTOS services from the high-priority ISR?" A common solution is to use a semaphore with explicit interrupt control to avoid corruption, e.g.**

```
while (1)
{
  if (HighPrioISRDataReady == 1)  /* simple user semaphore */
  {
    GIEH = 0;
    HighPrioISRDataReady = 0;
    GIEH = 1;
    OSSignalBinSem(HIGH_PRIO_ISR_DATA_READY_P);  /* wake task */
  }
  OSSched();  /* run highest-priority eligible task */
}
```

The length of time that global interrupts are disabled should be minimized when using this approach.[7] Additionally, the ISR should be coded to reduce the load on the main application as much as possible. E.g. signal a task only when a complete multi-byte packet has arrived, etc.

Part IV

Conclusion

By handling asynchronous events in embedded systems via interrupt handlers, we make most efficient use of CPU cycles.

Data will be lost if interrupt latency for high-rate ISRs is too high.

Interrupt control is often a combination of global and peripheral-specific enable/disable mechanisms.

Global shared variables require protection against corruption from interrupts and unwanted reentrancy.

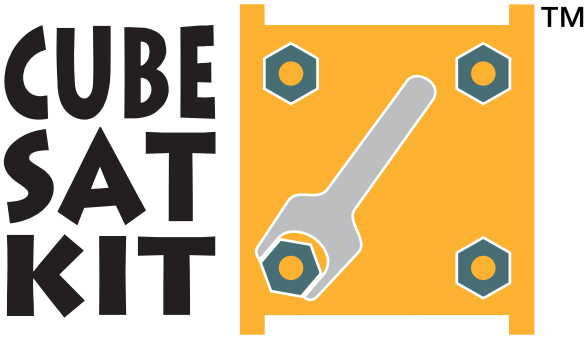
Complex programs – including multitasking schedulers – control interrupts to protect global shared variables.

By explicitly controlling individual interrupt sources, interrupt latency can be minimized or even eliminated on a per-interrupt-source basis.

Salvo™

The RTOS that runs in tiny places.™

and the



are products of

PUMPKIN™

INCORPORATED

750 Naples Street

San Francisco, CA 94112 USA

tel: (415) 584-6360

fax: (415) 585-7948

web: http://www.pumpkininc.com/

web: http://www.cubesatkit.com/

email: info@pumpkininc.com

PUMPKIN

First presented at the University of Applied Sciences of Southern Switzerland (SUPSI) on May 24, 2005.

Speaker information

Dr. Kalman is Pumpkin's president and chief software architect. He entered the embedded programming world in the mid-1980's. After co-founding a successful Silicon Valley high-tech startup, he founded Pumpkin with an emphasis on software quality and applicability to a wide range of microcontroller-based applications. He is also involved in a variety of other hardware and software projects, including http://www.cubesatkit.com/.

Acknowledgements

Figure 1 from Labrosse, Jean J., *µC/OS*, *The Real-Time Kernel*, R&D Publications, Lawrence, Kansas, 1992, ISBN 0-87930-444-8.

Copyright notice

End Notes

1 2x oversampling is employed. 3x oversampling would be more robust.

2 In the case of the non-interrupt-based code, overhead would include synchronization code. For the interrupt-based code, overhead would include interrupt context save and restore.

3 Assumes a constant stream of incoming Rx data.

4 This is especially true when the native data size of the embedded processor is much smaller than the native stack pointer size. E.g. an 8-bit processor with a 16-bit SP must disable interrupts while redefining the SP or else SP corruption will occur should an interrupt be serviced while the SP is being redefined over the course of several instructions.

5 For Microchip PIC18, e.g. PIC18F452.

6 Since the PIC18 has only a single high-priority interrupt vector, it's up to the programmer to implement a priority scheme in software.

7 In those cases where an instruction set provides the means to e.g. clear a single bit of information with a single, atomic instruction, disabling and re-enabling interrupts around access to the semaphore may be superfluous. Interrupt control is shown for the general case.