Efficient Interrupts on Cortex-M Microcontrollers

Chris Shore
Training Manager
ARM Ltd
Cambridge, UK
chris.shore@arm.com

***Abstract*—The design of real-time embedded systems involves a constant trade-off between meeting real-time design goals and operating within power and energy budgets. This paper explores how the architectural features of ARM® Cortex®-M microcontrollers can be used to maximize power efficiency while ensuring that real-time goals are met.**

***Keywords*—interrupts; microcontroller; real-time; power efficiency.**

I. Introduction

A "real-time" system is one which has to react to external events within certain time constraints. Wikipedia says:

> "A system is real-time if the total correctness on an operation depends not only on its logical correctness but also upon the time in which it is performed."

We can talk about varying degrees of real-time behavior but the message is still the same – if the system does not meet its real-time design goals, it is not a functionally correct system.

So, if an event occurs – for example, the user pressing a key, or the power system registering a voltage drop, or the arrival of a message from some other system, all of which will have different reaction times associated with them – the system must do something. Most importantly, it must do that something within a certain time. If it does not do that, it has failed as a system and is not fit for purpose.

Designing systems to meet real-time constraints is hard, fixing them when those constraints are not met is also hard. It is a complex job of prioritizing the events so that those with hard and short deadlines are handled first, with great urgency, and those with softer, longer deadlines are handled later. In the end, if the system does not have enough processing power to accomplish all its goals, it will need completely redesigning and it may be impossible to fix it without changing to more powerful hardware.

So, assuming that we have a system which is powerful enough to meet all its real-time design constraints, we have a secondary problem.

That problem is making the system as efficient as possible. If we have sufficient processing power, it is relatively trivial to conceive and implement a design which meets all time-related goals but such a system might be extremely inefficient when it comes to energy consumption. The requirement to minimize energy consumption introduces another set of goals and constraints. It requires us to make use of sleep modes and low power modes in the hardware, to use the most efficient methods for switching from one event to another and to maximize "idle time" in the design to make opportunities to use low power modes.

The battle is between meeting the time constraints while using the least amount of energy in doing so. This is a constant battle and one which cannot be forgotten. The simplest and most straightforward design is unlikely to be the most energy-efficient one. Likewise, the most energy-efficient design will most likely not meet all of its real-time goals.

II. A Simple Real-Time System

So, let us imagine you are building a simple system. How do you organize your software architecture so that all these competing inputs are dealt with? Figure 1 shows an example set of inputs which you might have in a typical system.

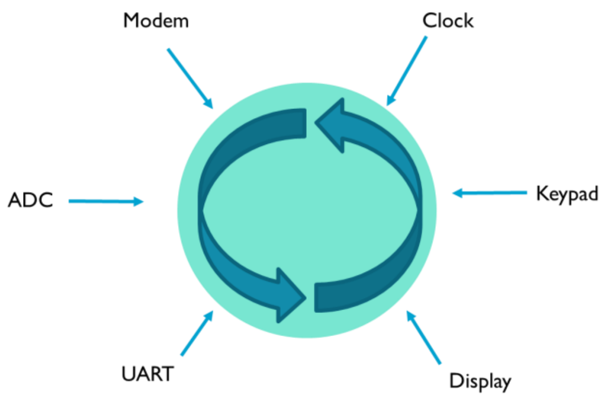


Figure 1 - A sample system

There is a clock, perhaps a repeating tick timer event, which would need servicing as quickly as possible. If it is going off once a millisecond, you only have a millisecond to service the event before it happens again so your deadline here is tight and absolute.

Then we have a keypad. This has quite different real-time requirements. Human beings react to things very slowly and when they press a key you actually have, in computing terms, quite a long time to make something happen. In any event, humans cannot press keys faster than a few times per second, so we do not need a really fast response to a keypad event.

Then we have an ADC, converting some external voltage into something we can measure and record. How quickly this needs servicing depends on how often the measurements are made. Perhaps the measurements are made every 100ms, or perhaps only once every minute. The real-time constraint is very different in each case.

There are other inputs too, each with different characteristics and different real-time requirements. The first task is to establish the real-time requirements for each input separately.

A. *Superloop*

The easiest thing to do with the software is simply to write a single loop which cycles around all of the input sources in turn and checks whether they require action. This is very easy to write in software but it is also very simplistic and potentially inefficient. It services all of the input events with the same priority, in the same order and to some extent the speed at which the loop much cycle round is dictated by the shortest, the tightest of the real-time deadlines in the system. So, if the clock needs servicing once every millisecond, the loop will check all the other inputs once every millisecond too - whether they need some action or not. Of course, most of the time, most of them will not need any action, so checking them is wasted effort. In a real-time system, that is wasted time and in an embedded, battery-powered system, that is wasted energy.

So, all in all, this is not a very good solution but it is a very easy one.

```
main()
{
 Init();
 while (1)
 {
  if (StuffToDo)
  {
   DoStuff();
  }
 }
}
```

This can be improved a little by adding a call to a “sleep” function which might introduce a delay during which the processor can go to sleep and save some energy.

```
main()
{
 Init();
 while (1)
 {
  if (StuffToDo)
  {
   DoStuff();
  }
  Sleep();
 }
}
```

However, this still treats all of the inputs with the same priority and still checks all of them when potentially no action is required. As we have observed, this represents a waste of time and a waste of energy. We can do much better than this.

The principle method we have available to use is interrupts.

B. *Interrupts*

Here is a system which uses only interrupts to drive everything. We assume that each input event is associated with an input to the processor which will generate an input when action is required. Each input event is then serviced when its interrupt fires and, most importantly, ONLY when its interrupt fires. There is no need to waste time and energy checking each input all the time because we know that we will receive an interrupt when something needs doing. So we just wait for the interrupts to roll in and there is nothing else to do.

Our main program is simply an empty infinite loop which sleeps all the time waiting for interrupts to happen. All of the code is in the interrupt handlers.

```
Main()
{
 Init();
 while (1)
 {
  Sleep();
 }
}
interrupt()
{
 DoStuff();
}
```

You may have heard that it is good practice to keep interrupt service routines short. This is a potential disadvantage of processing everything in interrupt handlers. If one particular event requires a lot of time-consuming processing, then this can hold up the other events and that may break some of the real-time goals in your system design. So, one option is to divide the processing between the interrupt handlers, which we often call the "background" and the main application, which we might call the "foreground". Then we can divide the processing for each event into two pieces: the part which needs to be carried out urgently as soon as the interrupt is triggered; and a second part which can wait until there is nothing else pending and which can be executed in our foreground main loop.

Here is how we might do this[1].

```
Main()
{
 Init();
 while (1)
 {
  Sleep();
  if (StuffPending)
  {
    DoForegroundStuff();
  }
 }
}
interrupt()
{
 DoStuff();
 PendForegroundStuff();
}
```

C. *Polling or Interrupts?*

In general, an interrupt-based solution will exhibit better real-time performance than a polling solution but the comparison becomes less clear when considering energy-efficiency.

This is because interrupts have an overhead associated with them – servicing an interrupt involves some overhead to do with saving system context, putting stuff on the stack and so on. If your main loop is able to make very effective use of the system's sleep capability, then perhaps a polled solution might be more energy efficient than an interrupt-based solution.

Cortex-M microcontrollers incorporate a specific set of hardware optimizations which make it possible to implement very efficient interrupt-based systems. When these are taken into consideration, an interrupt-based solution will almost always be by far the more efficient.

Before I introduce you to these features, first of all let us revise how interrupts work.

III. INTERRUPTS-101

When an interrupt occurs, the system saves the current context (a subset of the main register bank plus some status information) and jumps to a dedicated piece of software called an interrupt handler. This software handles the event and carries out whatever processing is necessary. When this is complete, the system restores the context and returns to the interrupted application. In the case of our simplest system, that would simply return to the main loop where it would immediately go to sleep if there were no other interrupts needing action.[2]

Recall also the concept of interrupt latency. This is defined as the length of time it takes from the moment the interrupt event occurs to the execution of the first instruction of the interrupt handler. Obviously we want it to be as short as possible but it depends on many things, including how long it takes to save context, how many other events are waiting to be processed at the time, whether the current routine can be interrupted or whether the new interrupt must wait until it has completed, whether the system is currently in some low power state and how long it will take to wake up, and so on. We will look at ways of minimizing and managing this latency.

A. *Interrupt Timing*

Figure 2 is a simple diagram showing the timing around the execution of a single interrupt. We assume that interrupts are enabled and the current task can be interrupted immediately so neither of those make any contribution to latency. The interrupt is detected and actioned as soon as it occurs.

You can see that there is something which we call "entry latency" - the time it takes the system to interrupt what it is doing, save context and begin executing the interrupt handler. On a Cortex-M microcontroller, this is handled completely in hardware and on a Cortex-M3 it takes 12 cycles.

Then, at the end, you can see that there is some time to wrap up and finish off the interrupt. We call that "exit latency". On a Cortex-M microcontroller this is also handled completely by the hardware and on a Cortex-M3 it takes 10 cycles.

Clearly entry latency determines how quickly we can action any particular input event. Exit latency has an effect on that too. If you have more than one interrupt, then the exit latency is part of what determines how quickly you can service the next event.

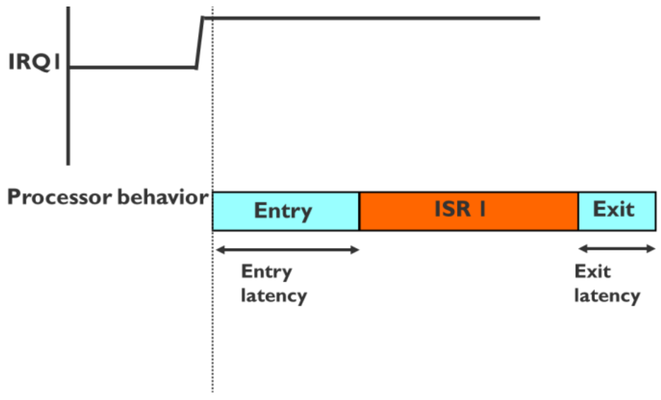


Figure 2- Interrupt Latency

Figure 3 shows two interrupts. Assuming no pre-emption, something we will get on to shortly, the second must wait for first to complete, so the total latency for the second event

[1] It is worth noting that, if you are using an RTOS, you can build any of these architectures using standard RTOS features. The main loop simply becomes a task, or a set of tasks, and interrupt handlers are installed as normal.

[2] In some systems, the saving and restoring of context could be very simple, just switching a few registers; in others, it might involve putting a large number of registers on the stack. It could be quite time-consuming.

actually includes the exit latency from the first. Actually, it is worse than that. If the second event cannot pre-empt or interrupt the first before it has finished, then the potential maximum latency for the second event is made up of three things:

1) the exit latency from the first interrupt (10 cycles on a Cortex-M3);

2) the entry latency for the second interrupt (12 cycles on a Cortex-M3);

3) the entire processing time of the interrupt handler for the first event.

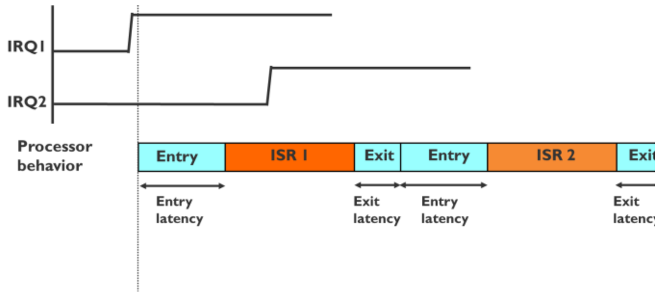


Figure 3 - Latency with Two Interrupts

Cortex-M microcontrollers support a feature called "tail-chaining" which is designed to minimize this.

B. *Tail-Chaining*

Cortex-M microcontrollers incorporate a neat feature for switching from the end of one interrupt straight to the start of another – this is called "tail-chaining". Because the context is already saved on the stack, from the first interrupt, there is no actually need to go through the entire save/restore context procedure between the end of one interrupt and the start of the next.

On a Cortex-M microcontroller, when the end of an interrupt handler is reached and there is another interrupt pending, the processor simply switches from one to the other. Rather than 22 cycles, this takes only 6 (on a Cortex-M3) as shown in Figure 4.

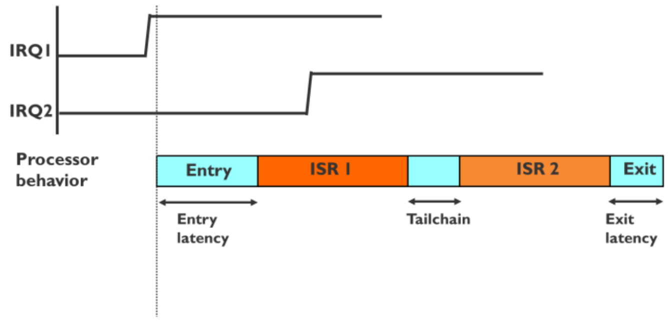


Figure 4- Tailchaining

This still does not help with the fact that the latency for the second interrupt has to include the entire processing time of the first. In fact, this problem just gets worse and worse.

Figure 5 shows a system with four interrupts. If they all go off together, they are simply processed in some arbitrary order and the latency for each successive interrupt just gets longer and longer. You can see that the latency for the fourth interrupt here is the sum of the total processing time of all the others. This is unlikely to be acceptable in the majority of real-time systems and a mechanism is required for getting round this.

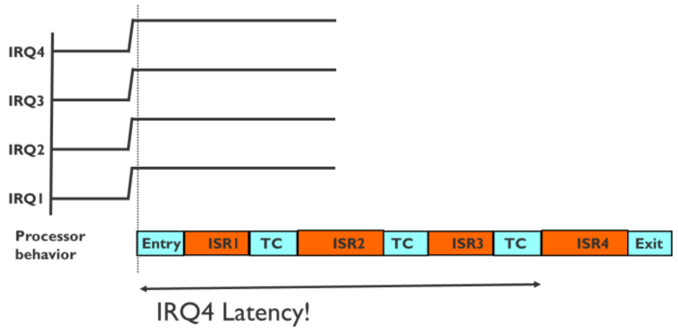


Figure 5- More Tailchaining

That mechanism is called pre-emption and it is driven by a priority system. In the Cortex-M microcontrollers, this is managed entirely by the built-in interrupt controller. You program it with a set of priorities and it takes care of the rest.

C. *Priority and Pre-emption*

The terms priority and pre-emption are separate but linked. All interrupts can be assigned a priority. This determines the order in which interrupts are serviced. If a system supports pre-emption, and all Cortex-M-based systems do, then the priority will determine whether a second interrupt can actually interrupt one which is already being serviced.

In a system without pre-emption, which is what we have just seen, priority simply defines the order in which events will be handled when they occur at the same time. So, if two interrupt events are pending, the highest priority one will be serviced first and the lower priority one will wait. Without pre-emption, we can still only handle one event at a time and that means that the latency of all events is increased as each event will simply have to wait until all higher priority events have been processed.

In a system with pre-emption, simultaneous events will still be handled in priority order but, and here is the difference, a higher priority event can interrupt the processing of a lower priority event. This means that higher priority events can be handled much more quickly and the system can be made much more efficient and manageable.

All Cortex-M-based systems support pre-emption.

Figure 6 shows a similar system to the one we saw earlier but adds pre-emption. In the earlier diagram, ISR2 simply had to wait until ISR1 had completed. Now you can see that pre-emption means that ISR2 can start immediately IRQ2 is triggered and it does this by interrupting ISR1. So, the overall completion time of IRQ1 is increased, by the execution time of IRQ2, but the latency of IRQ2 is potentially greatly reduced.

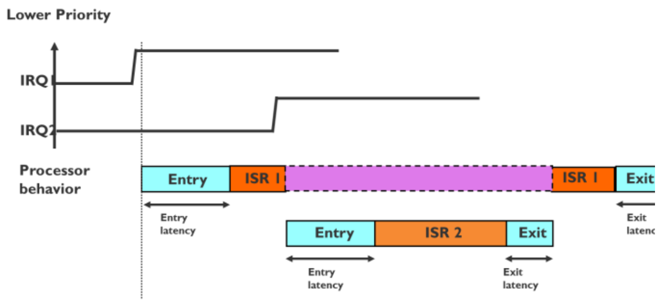


Figure 6 - Pre-emption

Here is a slightly more complicated example.

Figure 7 shows an example system which has three interrupt sources: a timer, a UART and a keypad. In this diagram, they are all set to equal priority. You can see that they effectively get serviced one at a time, each one having to wait until all others have completed before it can be serviced.

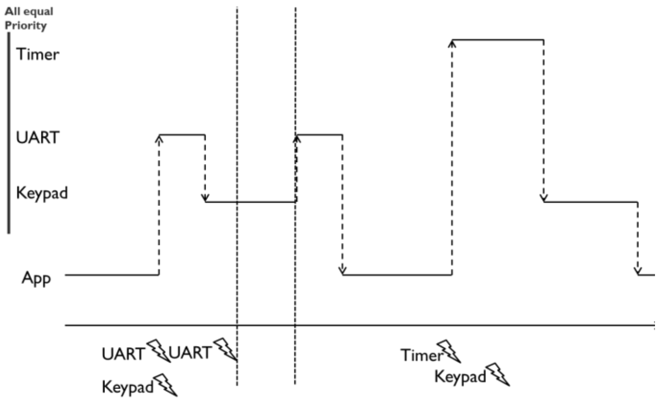


Figure 7- Example System Without Pre-emption

This results in significant variability in latency.

Consider the first UART interrupt: because nothing else is waiting, this one is handled immediately. Then consider the second UART interrupt: the latency is much longer because it occurs when a keypad interrupt is already being processed.

Now this might be OK but most likely it will not. A peripheral like a UART will be receiving incoming data over a serial data link and that data might be coming in very fast. If we delay processing it, we might miss data. So, actually, it is likely to be very important indeed that we process UART interrupts as soon as they happen.

We can define a priority scheme, as shown in Figure 8 to ensure that this is what happens.

The UART interrupt is assigned a higher priority than the Keypad interrupt. With pre-emption, the UART event interrupts the Keypad event while it is still being processed and the latency for the UART is maintained at a low value. But this has a cost.

In this case, the cost of enabling low latency for the UART is accepting that the Keypad event takes a bit longer to be processed. But, as we have remarked, human beings pressing keys are very slow compared to the rate at which a computer system might receive data over a UART. So, we do not mind that.

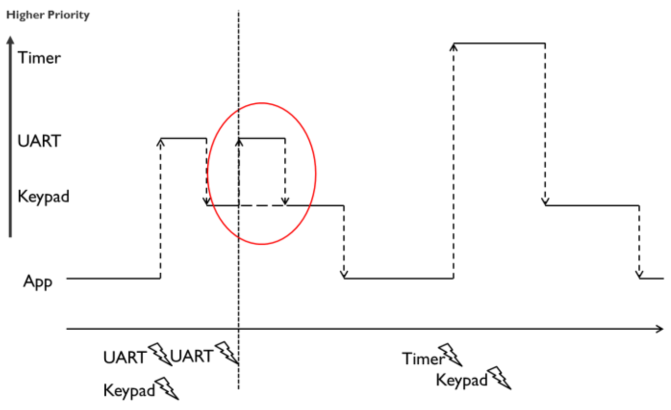


Figure 8 - Example System With pre-emption

So, pre-emption sounds allows us to manage the latency of individual interrupt events closely. But it has another cost…

Look a little more closely at our system with four interrupts. If we give them priorities and enable pre-emption, then you can see in Figure 9 how they are all able to interrupt each other and latencies no longer accumulate. Completion times increase - look particularly at IRQ1 which gets interrupted three times before completing - but latency is kept short for the higher priority events.

But, every time we pre-empt, there is a cost. Put simply, all of those shaded boxes in Figure 9 cost energy. Simply tail-chaining from one interrupt to another takes only six cycles and does not involve any stack access. In a pre-emptive system, we can no longer tail-chain through the series of interrupts. Instead, the system has to continually save and restore context to allow interrupts to pre-empt each other.

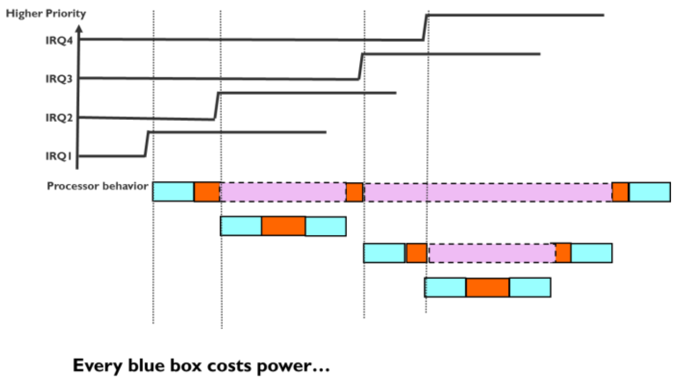


Figure 9 - Extensive Pre-emption

Reducing and managing latency has an energy cost. What we need is a way of achieving a balance between the desire to manage latency and the energy cost of doing that. Cortex-M microcontrollers, again, provide features which allow just that. First, we need to look carefully at how they handle priority.

IV. Defining a Priority Scheme

A. *Simple Priority*

In a Cortex-M microcontroller, each interrupt can be assigned a priority, configured by registers in the interrupt controller. The architecture allows up to 8-bits of priority, so you could have 256 priority levels as shown in Figure 10. But you only have that many levels if the chip designer has actually implemented all of the bits.[3]

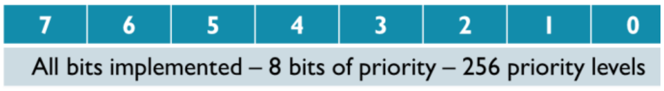

7
6
5
4
3
2
1
0
All bits implemented – 8 bits of priority – 256 priority levels

Figure 10 - A simple priority scheme

Figure 11 shows an example in which only 3 bits are implemented, giving 8 priority levels. The interrupt controller architecture is defined so that a lower priority value actually gives an interrupt a higher relative priority - an interrupt configured with priority level 3 is a higher priority interrupt than an interrupt with priority level 8, and so on.[4]

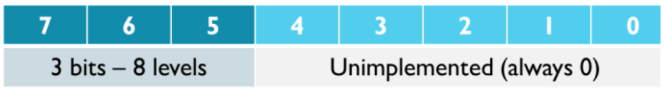


Figure 11 - Restricted priority levels

B. *Sub-Priority*

The priority register can be split, under software control, into two fields, called “pre-empt priority” and “sub-priority”.

Figure 12 shows an example where all 8 priority bits are implemented and the register is split into a 3-bit pre-empt priority field and a 5-bit sub-priority field. That allows 8 levels of priority which control pre-emption and then 32 levels of sub-priority.

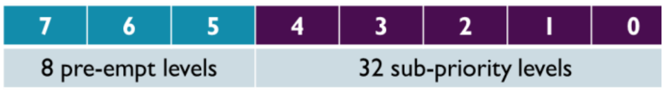


Figure 12 - Pre-emption priority and sub-priority

Figure 13 shows another example where not all the bits are implemented – in this case we have 6 bits of priority which are split into a 2-bit pre-empt priority and a 4-bit sub-priority.

[3] This is one of the elements of the processor which the chip designer can configure in order to reduce the gate count of the core. Implementing fewer bits of priority reduces the complexity of the interrupt controller quite considerably and reduces the overall size and power consumption of the processor. So, a chip designer will usually implement fewer than 8 bits.

[4] Note that any unimplemented bits are always omitted from the bottom end of the priority value and they simply read as zero. This is useful to know that when you are programming a device which has fewer than 8 bits in the priority registers.

In Figure 10, priorities start from zero and increase in steps of one; in Figure 11, priority starts from 0 and increases in steps of 0x20, or 32 in decimal. So the priorities are 0, 32, 64, 96 and so on.

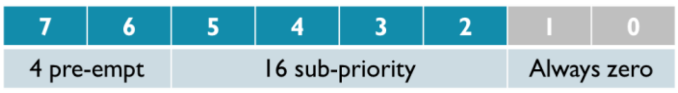


Figure 13 - Reduced number of priority levels

The key thing to realize here is that only the pre-empt priority affects pre-emption, so we can group interrupts together in such a way that only certain interrupts can actually interrupt others. This gives us flexibility to manage latency while keeping control of the amount of pre-emption which goes on. And since pre-emption, as we have seen, has an energy cost, it is important to eliminate unnecessary pre-emption.

Next we will look at an example of using these two priority fields to configure interrupts into groups.

C. *Interrupt Groups*

In Figure 14 we have grouped our interrupts into four sets. The top two are timers and are in groups on their own - the top one is assumed to be the highest priority interrupt in the system. It will pre-empt anything else and will have the lowest and most consistent latency.

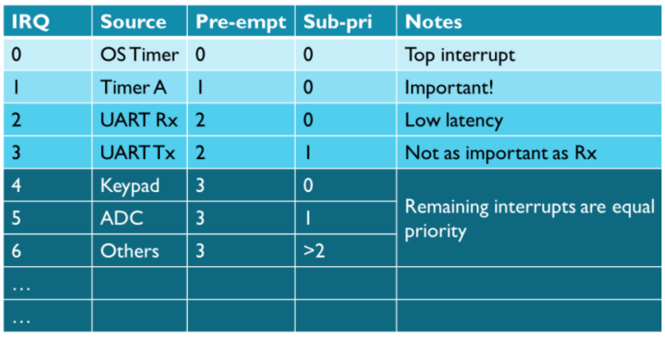
The next one down is another timer but we assume it is not quite so important so it goes into the next level down. It will pre-empt everything except the main OS timer.

Then we group together two interrupts associated with the UART, one for receive and one for transmit. We make reception more important than transmission as we do not want to risk losing data.[5]

All the rest of the interrupts are grouped at the lowest level in the system. They cannot pre-empt each other, nor can they pre-empt any of the higher priority events. Their latency will be the longest in the system but these events are such that this does not matter. We do not need to expend energy getting to them more quickly.

But notice that we can actually give them a priority relative to each other. This is a priority which only applies within this group but it is important. This sub-priority determines the order in which they will be handled if more than one of them occur simultaneously. So, if we finish processing all the higher priority events and there are a number of these lower ones pending, then they will be serviced in the order determined by the sub-priority. This mechanism is very powerful and allows us to manage the relative latency of all events at the same priority. In this case, since we want to respond promptly to the user pressing a key, we specify that the Keypad interrupt should be handled first, then the ADC, then others.

[5] If the two events occur simultaneously, it would be important to read the incoming data out (before it is potentially over-written by further incoming data) before writing outgoing data.



IRQ	Source	Pre-empt	Sub-pri	Notes
0	OS Timer	0	0	Top interrupt
1	Timer A	1	0	Important!
2	UART Rx	2	0	Low latency
3	UART Tx	2	1	Not as important as Rx
4	Keypad	3	0	Remaining interrupts are equal priority
5	ADC	3	1	
6	Others	3	>2	
…				
…				

Figure 14 - An example priority scheme

Most importantly, if more than one of these needs processing, then they will tail-chain into each other. And remember that tail-chaining is much more energy-efficient way of switching from one interrupt context to another.

The division of the priority field gives us the ability to allow pre-emption when absolutely necessary for low latency events, while managing the ordering of other events using the sub-priority, maximizing tail-chaining when they are handled.

This allows us to manage the balance between TIME and ENERGY very effectively. We manage latency using pre-empt priority where necessary and manage energy by maximizing tail-chaining everywhere else.

To make best use of this, consider carefully the latency requirements of all the interrupts in the system and then configure the pre-emption priorities to make sure that all the real-time goals are met. Then carefully consider the relative priority of the other events which can be kept at the same priority and use the sub-priority to control the order of the tail-chaining.

But we are nowhere near finished yet! We have managed the interrupt events in our system to minimize latency where necessary and maximize tail-chaining everywhere else. Now, we need to rewind all the way to the beginning and consider what we should do when there are no events pending.

Clearly, we need to maximize the opportunties to go to sleep.

V. When There Is Nothing To Do

Recall our simple system from earlier.

```
Main()
{
 Init();
 while (1)
 {
  Sleep();
 }
}
interrupt()
{
 DoStuff();
}
```

Almost every Operating System has some kind of sleep facility and this will be doubly true of any system which is designed to run on the kind of deeply embedded system for which Cortex-M microcontrollers are very often used.

What happens when you go to sleep depends on the processor you are using. Cortex-M microcontrollers have a pretty standard set of power saving modes which are illustrated in Figure 15.

You can see that they range from Power Off on the left, to Active on the right. There are various steps in between and it is important to understand that exactly what they do is determined by the designer of the particular chip you are using. The difference between Sleep mode and Deep Sleep mode, for instance, depends on the configuration of power domains within the chip and which particular components the chip designer decided should be powered down at this transition.[6]

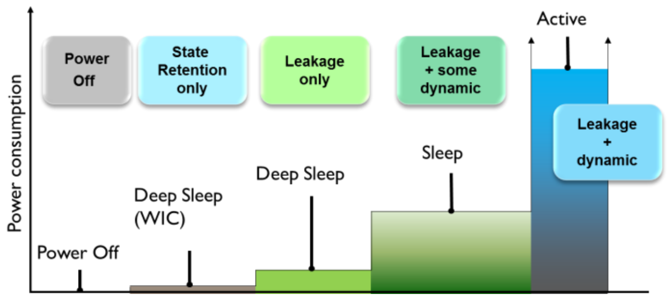


Figure 15 - Typical Cortex-M power saving modes

Using these power modes is easy and they interact with interrupts in a standard and sensible way. They can be entered in two ways, one under software control and one automatic.

To enter low power state under software control, use one of these two instructions: WFI (Wait for Interrupt) or WFE (Wait for Event). They behave in a very similar way: if no interrupts are pending, they will place the system immediately into a low power state. When an interrupt occurs, the processor will automatically wake and restart execution from the next instruction. Wakeup will also be triggered by a debug event or, in the case of WFE, by a signal on the external Event input signal.[7]

Typically, a WFI or WFE instruction would be included in the main loop.

Low power state can also be entered automatically by setting the SLEEPONEXIT bit. When this bit is set, the processor will automatically sleep when no interrupts are pending.

There is a configuration bit called DEEPSLEEP which controls whether you enter Sleep or Deep Sleep state. And the

[6] WIC, here stands for Wakeup Interrupt Controller - a small piece of logic which can be used to wake up the system when an interrupt occurs. It is optional but if the chip designer has included it, then almost the entire device except the WIC can be powered down, allowing an incredibly low power state.

[7] This event input can be connected by the chip designer to any event which might be used to wake the system. It is frequently used, for instance, to provide a wake-up signal between two cores in a multi-processing system.

system simply wakes up automatically when the next interrupt happens.

The ARM C compiler has an intrinsic function, __WFI(), which can be used directly from C to avoid having to write any assembler.

```
Main()
{
 Init();
 while (1)
 {
  __WFI();
 }
}
interrupt()
{
 DoStuff();
}
```

Most RTOSs will provide an API for the sleep function. If one is available, it should be used in preference to calling WFI directly, as the RTOS may need to carry out some internal configuration actions before activating low power state.

VI. Everything in Interrupts

A. An Interrupt-Only System

In a system which does everything in interrupts, you might do something like this.

```
Main()
{
 Init();
 Set SLEEPONEXIT;

 while (1)
 {
  __WFI();
  // will never get here;
 }
}
interrupt()
{
 DoStuff();
}
```

This system will never actually execute its main loop - it just sleeps. Everything is handled in interrupts and the system automatically enters low power state when there are no more interrupts pending. You would observe a power consumption profile like the one shown in Figure 16.

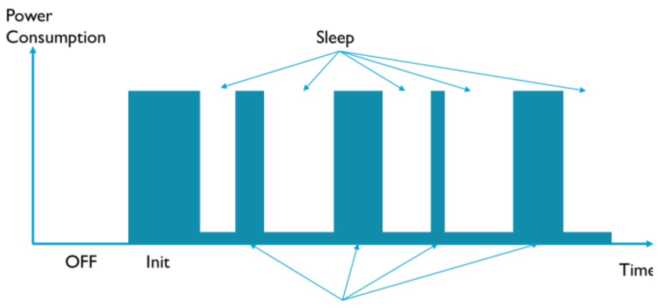


Figure 16 - Example power consumption profile

Once the system is initialized, it is active only when interrupts are being processed and it sleeps automatically whenever there are no interrupts pending. Setting up a system like this is really easy to do and results in a very energy-efficient design.

Because the SLEEPONEXIT bit is set, the system sleeps automatically at the end of the last interrupt. Since the event which will wake it up will be another interrupt, it is able to leave the context on the stack. When the next interrupt is triggered and the system wakes up, it does not need to save context and can jump immediately to the interrupt handler. This takes only a few cycles and avoids all the time and energy cost of saving and restoring context.

As we have seen, these features are very easy to use from a software point of view.

But it gets better!

B. Managing Latency

Recall from Figure 6 how to maintain the low latency requirements of a particular interrupt by giving it a higher priority. The penalty we pay for that is that the time to complete the low priority interrupt is extended. In this case, by the entire time it takes to process the high priority interrupt.

That may be a problem and to fix it we need to split the processing of the high priority interrupt into two parts: a short part which needs to be carried out urgently, followed by a longer part which can wait a little.

One method of doing this is to set a flag in the interrupt handler and then drop back into the main, or foreground, loop when interrupts have completed. This would be done by unsetting the SLEEPONEXIT bit so that the system would not sleep when the last interrupt handler exits. Instead, the system would fall back into the main loop and carry out any processing necessary before setting SLEEPONEXIT and going to sleep again to wait for the next event.

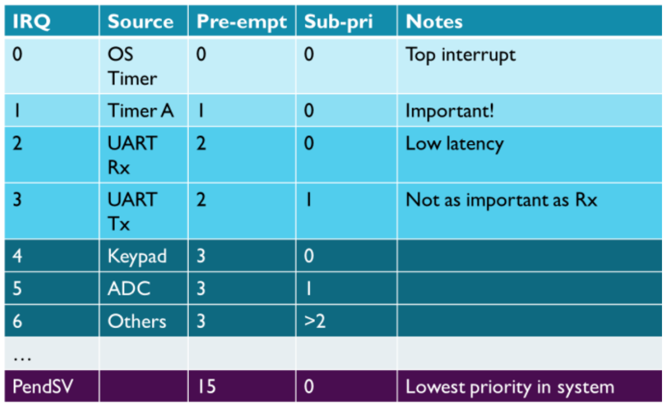
The disadvantage of this method is that it requires continual management of the SLEEPONEXIT bit. It also requires frequent entry to the foreground task and we have remarked that there is a penalty for doing that – the system must restore context again (taking all the registers off the stack) and then save context again (putting all the registers back on the stack) when returning to the interrupt context. If the foreground processing lasts some while, then we would do this potentially many times, and the potential energy cost is actually quite high.

Cortex-M microcontrollers provide a features which allows us to do this while remaining in the interrupt context.

C. The PendSV Exception

The PendSV exception is often used by operating systems to handle context switches but we can use it here to delay processing of low latency interrupts without having to drop into the foreground.

PendSV is an architecturally-defined interrupt and is typically configured with the lowest priority in the system. So, in Figure 17, it is added it to the list we had before with a pre-empt priority of 15, the lowest available in a system with only 4 bits of priority. Because it has the lowest priority in the system, it never pre-empts anything but gets handled when all other interrupts have been processed and no others are pending. We can trigger it when we need to under software control.



IRQ	Source	Pre-empt	Sub-pri	Notes
0	OS Timer	0	0	Top interrupt
1	Timer A	1	0	Important!
2	UART Rx	2	0	Low latency
3	UART Tx	2	1	Not as important as Rx
4	Keypad	3	0	
5	ADC	3	1	
6	Others	3	>2	
…				
PendSV		15	0	Lowest priority in system

Figure 17 - Priority scheme including PendSV

Here are some details about PendSV.

1) It is an internal exception - that means it is built in to the system.

2) It is imprecise – that means that it does not occur when you trigger it but happens at some later time.

3) It can be triggered under software control by setting a bit in the Interrupt Control and State Register.

4) It can be triggered by any interrupt handler.

And because we will configure it with the lowest priority in the system, it gets taken only when all other interrupts have completed.

The effect can be seen in Figure 18.

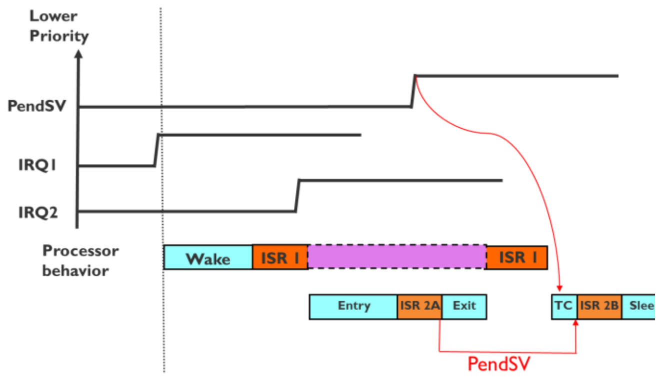


Figure 18 - Using PendSV

– IRQ1 wakes the system up and starts to execute ISR1.

– ISR2 still pre-empts ISR1 so the low latency requirement of IRQ2 is satisfied but ISR2 only does what it needs to do urgently, sets PendSV and then exits, allowing ISR1 to finish earlier.

– Once ISR1 has finished, assuming nothing else is pending, the processor automatically tail-chains straight into the PendSV handler.

– It then completes the processing of IRQ2.

No context switch is required, other than a tail-chain from the final interrupt into the PendSV handler. When the PendSV handler completes, the system automatically goes back to sleep.

VII. Conclusions

For the vast majority of systems, a properly implemented interrupt-based solution is more efficient and exhibits better real-time characteristics than a polling solution - certainly on a Cortex-M microcontroller which has all these interrupt system optimizations.

Polling creates potential problems with managing latency and power; interrupts, except in the simplest systems, always have some small overhead in saving and restoring context. But this can be managed and minimized in a way which polling does not allow.

The features of the Cortex-M microcontroller architecture which make this possible are:

1) a hardware-supported interrupt entry and exit sequence;

2) tail-chaining support in hardware which minimizes the time and energy cost of switching from one interrupt to the next;

3) a configurable priority scheme which separates pre-emption priority and sub-priority;

4) the ability to enable a low power state automatically when there are no interrupts pending;

5) low power states which can entered and exited with almost no penalty at all;

6) the PendSV mechanism which allows us to split an interrupt handler into an urgent part and a part which can safely be deferred.

How best to use these features depends, as usual, on each individual system, its real-time requirements and its energy constraints.

– For those interrupts for which latency is most important, prioritize them carefully to allow pre-emption where necessary.

– For those which take a long time to process, consider making use of the PendSV feature.

– To minimize energy use, maximize tail-chaining - in order to do that, group as many interrupts as possible at the same pre-empt priority and use sub-priority to control the order in which they are processed.

As ever, there is a spectrum of possibilities. It is always worth doing some modelling and, if possible, some measurements on a real prototype system to determine which approach is best for your system.

References

[1] ARM, “ARMv7-M Architecture Reference Manual”, ARM Ltd, ARM DDI0403

[2] ARM, “Cortex-M3 Devices Generic User Guide”, ARM Ltd, ARM DUI 0552A

[3] Joseph Yiu, “The Definitive Guide to ARM Cortex-M4 and Cortex-M4 Processors”, Newnes