Safe and Structured Use of Interrupts in Real-Time and Embedded Software

John Regehr
School of Computing
University of Utah
Salt Lake City, UT 84112
E-mail: regehr@cs.utah.edu

November 3, 2006

1 Introduction

While developing embedded and real-time systems, it is usually necessary to write code that handles interrupts, or code the interacts with interrupt handlers. Interrupts are indispensable because they use hardware support to reduce both the latency and overhead of event detection, when compared to polling. Furthermore, modern embedded processor cores can save energy by shutting down when idle, leaving only simple circuitry such as timers running. For example, a TelosB [13] sensor network node drains a pair of AA batteries in a few days by running the processor continuously, but can run a useful application for months if the duty cycle is kept appropriately low.

Interrupts have some inherent drawbacks from a software engineering point of view. First, they are relatively non-portable across compilers and hardware platforms. Second, they lend themselves to a variety of severe software errors that are difficult to track down since they manifest only rarely. These problems give interrupts a bad reputation for leading to flaky software: a significant problem where the software is part of a highly-available or safety-critical system.

The purpose of this chapter is to provide a technical introduction to interrupts and the problems that their use can introduce into an embedded system, and also to provide a set of design rules for developers of interrupt-driven software. This chapter does not address interrupts on shared-memory multiprocessors, nor does it delve deeply into concurrency correctness: the avoidance of race conditions and deadlocks. Concurrency correctness is the subject of a large body of literature. Although the vast majority of this literature addresses problems in creating correct thread-based systems, essentially all of it also applies to interrupt-driven systems.

2 Interrupt Definitions and Semantics

This section presents interrupt terminology and semantics in more detail than is typically found in an operating systems textbook or an embedded systems trade publication.

2.1 Definitions

Neither processor and operating system reference manuals nor the academic literature provides a consistent set of terminology for interrupts and their associated issues. This section provides a few definitions that are largely consistent with the prevailing usage among academics and practitioners.

The term *interrupt* has two closely related meanings. First, it is a hardware-supported asynchronous transfer of control to an interrupt vector based on the signaling of some condition external to the processor core. An *interrupt vector* is a dedicated or configurable location in memory that specifies the address to which execution should jump when an interrupt occurs. Second, an interrupt is the execution of an *interrupt handler*: code that is reachable from an interrupt vector. It is irrelevant whether the interrupting condition

originates on-chip (e.g., timer expiration) or off-chip (e.g., closure of a mechanical switch). Interrupts usually, but not always, return to the flow of control that was interrupted. Typically an interrupt changes the state of main memory and of device registers, but leaves the main processor context (registers, page tables, etc.) of the interrupted computation undisturbed.

An *interrupt controller* is a peripheral device that manages interrupts for a processor. Some embedded processor architectures (such as AVR, MSP430, and PIC) integrate a sophisticated interrupt controller, whereas others (such as ARM and x86) include only a rudimentary controller, necessitating an additional external interrupt controller, such as ARM's VIC or x86's PIC or APIC. Note that an external interrupt controller may be located on-chip, but even so it is logically distinct from its processor.

Shared interrupts are those that share a single CPU-level interrupt line, necessitating demultiplexing in either hardware or software. In some cases, software must poll all hardware devices sharing an interrupt line on every interrupt. On some platforms, interrupts must be *acknowledged* by writing to a special device register; on other platforms this is unnecessary.

An interrupt is *pending* when its firing condition has been met and noticed by the interrupt controller, but when the interrupt handler has not yet begun to execute. A *missed interrupt* occurs when an interrupt's firing condition is met, but the interrupt does not become pending. Typically an interrupt is missed when its firing condition becomes true when the interrupt is already pending. Rather than queuing interrupts, hardware platforms typically use a single bit to determine whether an interrupt is pending or not.

An interrupt is *disabled* when hardware support is used to prevent the interrupt from firing. Generally it is possible to disable all interrupts by clearing a single bit in a hardware register, the *master interrupt enable bit*. Additionally, most interrupts have dedicated interrupt enable bits.

An interrupt fires only when the following conditions are true:

1. The interrupt is pending.
2. The processor's master interrupt enable bit is set.
3. The individual enable bit for the interrupt is set.
4. The processor is in between executing instructions, or else is in the middle of executing an interruptible instruction (see below). On a pipelined processor, the designers of the architecture choose a pipeline stage that checks for pending interrupts and a strategy for dealing with architectural state stored in pipeline registers.
5. No higher-priority interrupt meets conditions 1–4.

Interrupt *latency* is the length of the interval between the time at which an interrupt's firing condition is met and the time at which the first instruction of the interrupt handler begins to execute. Since an interrupt only fires when all five conditions above are met, all five can contribute to interrupt latency. For example, there is often a short hardware-induced delay between the time at which a firing condition is met and the time at which the interrupt's pending bit is set. Then, there is typically a short delay while the processor finishes executing the current instruction. If the interrupt is disabled or if a higher-priority interrupt is pending, then the latency of an interrupt can be long. The *worst-case interrupt latency* is the longest possible latency across all possible executions of a system. Although the "expected worst-case latency" of an interrupt can be determined through testing, it is generally the case that the true worst-case latency can only be determined by static analysis of an embedded system's object code.

Nested interrupts occur when one interrupt handler preempts another, whereas a *reentrant* interrupt is one where multiple invocations of a single interrupt handler are concurrently active.[1] The distinction is important: nested interrupts are common and useful, whereas reentrant interrupts are typically needed only in specialized situations. Interrupts always execute *atomically* with respect to non-interrupt code, in the sense that their internal states are invisible (remember that we are limiting the discussion to uniprocessors). If there are no nested interrupts, then interrupt handlers also execute atomically with respect to other interrupts.

[1] Note that a reentrant interrupt is different from a reentrant function. A reentrant function is one that operates properly when called concurrently from the main context and an interrupt, or from multiple interrupts.

There are a few major semantic distinctions between interrupts and threads. First, while threads can *block* waiting for some condition to become true, interrupts cannot. Blocking necessitates multiple stacks. Interrupts run to completion except when they nest, and nested interrupts always run in LIFO fashion. The second main distinction is that the thread scheduling discipline is implemented in software, whereas interrupts are scheduled by the hardware interrupt controller.

Many platforms support a *non-maskable interrupt* (NMI), which cannot be disabled. This interrupt is generally used for truly exceptional conditions such as hardware-level faults from which recovery is impossible. Therefore, the fact that an NMI preempts all computations in a system is irrelevant: those computations are about to be destroyed anyway.

2.2 Semantics

Unfortunately, there are genuine differences in the semantics of interrupts across hardware platforms and embedded compilers. This section covers the more important of these.

On most platforms, instructions execute atomically with respect to interrupts, but there are exceptions such as the x86 string-copy instructions. Non-atomic instructions are either restarted after the interrupt finishes or else their execution picks up where it left off. Typically, non-atomic instructions are slow CISC-like instructions that involve many memory operations. However, slow instructions are not always interruptible. For example, the ARM architecture can store and load multiple registers using a single non-interruptible instruction. The GNU C compiler uses these instructions in order to reduce code size. Other compilers avoid the slow instructions in order to keep interrupt latencies low. The correct tradeoff between latency and code size is, of course, application dependent.

Processor architectures vary in the amount of state that is saved when an interrupt fires. For example, the 68HC11, a CISC architecture, saves all registers automatically, whereas the AVR, a RISC, saves only the program counter. Most versions of the ARM architecture maintain register banks for interrupt mode and for "fast interrupt" mode in order to reduce the overhead of handling interrupts.

Minor details of handling interrupts can be smoothed over by the compiler. Almost all embedded C compilers support interrupt handlers that look like normal functions, but with a non-portable pragma indicating that the code generator should create an interrupt *prologue* and *epilogue* for the function, rather than following the normal calling convention. The prologue and epilogue save and restore any necessary state that is not saved in hardware, hiding these details from the programmer. On all but the smallest platforms, the top-level interrupt handler may make function calls using the standard calling conventions.

Embedded systems supporting multiple threads have multiple stacks, one per thread. The simplest and highest-performance option for mixing interrupts and threads is for each interrupt to use the current thread stack. However, this wastes RAM since every thread stack must be large enough to contain the worst-case interrupt stack requirement. The alternative is for the interrupt to push a minimal amount of state onto the current stack before switching the stack pointer to a dedicated system stack where all subsequent data is stored. On ARM hardware, each register bank has its own stack pointer, meaning that stack switching is performed automatically and transparently.

3 Problems in Interrupt-Driven Software

This section addresses several difficult problems commonly encountered during the development of interrupt-driven embedded software: avoiding the possibility of stack overflow, dealing with interrupt overload, and meeting real-time deadlines. Although interrupt overload is technically part of the overall problem of meeting real-time deadlines, these are both difficult issues and we deal with them separately.

3.1 Stack Overflow

As an embedded system executes, its call stacks grow and shrink. If a stack is ever able to grow larger than the region of memory allocated to it, RAM becomes corrupted, invariably leading to system malfunction or crash. There is a tension between overprovisioning the stack, which wastes memory that could be used for other purposes, and underprovisioning the stack, which risks creating a system that is prone to stack overflow.

Most developers are familiar with the *testing-based* approach to sizing stacks, where empirical data from simulated or actual runs of the system is used to guide allocation of stack memory. Testing-based stack allocation is more of an art than a science. Or, as Jack Ganssle puts it, [7, p. 90] "With experience, one learns the standard, scientific way to compute the proper size for a stack: Pick a size at random and hope."

The stack memory allocation approach that is complementary to testing is *analysis-based*. In the simplest case, analysis is done by manually counting push, pop, and call instructions along different paths through the compiled system and adding up their effects. This approach does not scale, and so analysis must be automated. Sizing stacks through analysis can be more reliable than using testing, but analysis requires more effort unless very good tools are available.

An embedded system is *stack safe* if it is impossible for a stack to overflow into memory used for other purposes. This notion of safety is analogous to the *type safety* guaranteed by languages like Java, which ensure that, for example, an integer can never be interpreted as a pointer. In theory, the ideal system would be just barely stack safe: if any less memory were allocated to any stack, it would be possible for an overflow to happen. In practice a safety margin may be desirable as a hedge against errors in stack depth analysis.

3.1.1 Analysis vs. testing

This section discusses the two approaches to sizing the stack in more detail. To implement the testing-based approach, one just runs the system and sees how big the stack gets. Ideally the system is tested under heavy, diverse loads in order to exercise as much code as possible. Measuring the maximum extent of the stack is not hard: simulators can record this directly, and in a real system one would initialize stack memory to known values and see how many of these get overwritten. The testing-based approach treats the system, to a large extent, as a black box: it does not matter how or why stack memory is consumed, we only want to know how big each stack can get.

The second way to find the maximum size of the stack, the analysis-based approach, looks at the flow of control through an embedded system with the goal of finding the path that pushes the maximal amount of data onto the stack. This path may be complex, involving the main function plus several interrupt handlers.

Notice that the testing- and analysis-based approaches are both trying to do exactly the same thing: find the path through the program that produces worst-case stack behavior. The fundamental problem with testing is that for any complex system it is a mathematical certainty that testing will miss some paths through the code. Consider, for example, a system containing five interrupt handlers, each of which fires an average of 50 times per second and spends 200 microseconds at its worst case stack depth. Assuming that interrupts arrive independently, the interrupts will exhibit their worst-case stacking only about every 300 years. On the other hand, if 100,000 of these systems are deployed, we can expect to see one failure per day.

The analysis-based approach to bounding stack size, on the other hand, has the problem that it often overestimates the maximum stack size. Consider the extreme example of a reentrant interrupt handler, which makes it difficult to compute any bound at all. In other words, an analyzer may be forced to conclude that the worst-case stack depth is infinite, even though the system's developers know that this interrupt only arrives twice per second and cannot possibly preempt itself. Similarly, the analysis may be forced to assume that a timer interrupt may fire during a call to a certain function, when in fact this function is setting up the timer and it is known that the timer cannot fire until well after the function returns.

The true worst-case stack size for any complex system is unlikely to be computable, but it can be bracketed by combining the testing and analysis approaches. The three values are related like this:

$$\begin{aligned} \text{worst depth seen in testing} &\leq \\ \text{true worst depth} &\leq \\ \text{analytic worst depth} & \end{aligned}$$

To some extent the gaps between these numbers can be narrowed through hard work. For example, if testing is missing paths through the system, then maybe more clever tests can be devised, or maybe the tests need to be run for longer. On the other hand, if the analysis misses some crucial feature of the system such as a causal relationship between interrupt handlers, then possibly the analysis can be extended to take this relationship into account.

UU IR Author Manuscript

The advantages of the analysis-based approach are that analysis can be much faster than testing (a few seconds or less), and that analysis can produce a guaranteed upper bound on stack depth, resulting in stack-safe systems. Also, by finding the worst-case path through a program, analysis identifies functions that are good candidates for inlining or optimization to reduce stack depth. The main advantage of testing, on the other hand, is that of course all embedded systems are tested, and so there is no reason not to include stack size tests. If the results returned by analysis and testing are close together, then both testing and analysis can be considered to be successful. On the other hand, if there is a substantial gap between the two results then there are several unpleasant possibilities. First, testing may be missing important paths through the code. Second, the analysis getting confused in some way, causing it to return an overly pessimistic result. If the gap cannot be closed then picking a size for the stack becomes an economic tradeoff. If the stack is just a little larger than the worst size seen during testing, RAM consumption is minimized but there is significant risk of stack overflow after deployment. On the other hand, if the stack is set to its analyzed worst-case size, the possibility of overflow is eliminated but less RAM will be available for other parts of the system.

3.1.2 Stack depth analysis

The *control flow graph* for an embedded system is at the core of any analysis-based approach to determining worst-case stack depth. The control flow graph (CFG) is simply a representation of the possible movement of the program counter through a system's code. For example, an arithmetic or logical operation always passes control to its successor and a branch may pass control either to its successor or to its target. The CFG can be extracted from either the high-level language (HLL) code or the executable code for a system. For purposes of bounding stack depth, a disassembled executable is preferable: the HLL code does not contain enough information to effectively bound stack depth.

For some systems, such as those written in the style of a cyclic executive, constructing the CFG is straightforward. For other systems, particularly those containing recursion, indirect calls, and an RTOS, either a much more sophisticated analysis or a human in the loop is required. Indirect calls and jumps come from a variety of sources, such as event loops, callbacks, switch statements, device driver interfaces, exceptions, object dispatch tables, and thread schedulers.

The CFG alone is not enough to determine the maximum stack size: the effect that each instruction has on the stack depth is also important. Assume for a moment that a system only manipulates the stack through `push` and `pop` instructions. At this point, an analyzer can compute the stack effect of different paths through the CFG: its job is to find the path that pushes the most data onto the stack. This is easily accomplished with standard graph algorithms. However, if the system contains code that loads new values directly into the stack pointer, for example to push an array onto the stack or to implement `alloca`, this simple approach is again insufficient. A stronger approach that can identify (or at least bound) the values loaded into the stack pointer is required.

A simple cyclic executive can be described by a single control flow graph. In the presence of interrupts or threads, a system contains multiple CFGs: one per entry point. Stack bounds for each of these graphs can be computed using the same algorithm. A complication that now arises is that we need to know how to put the individual results together to form a stack depth bound for the whole system. If all interrupts run without enabling interrupts, then this is easy:

$$\text{worst case depth} = \text{depth of main} + \text{maximum depth of any interrupt}$$

Since many systems enable interrupts while running interrupt handlers, the following equation is more broadly applicable:

$$\text{worst case depth} = \text{depth of main} + \text{total depth of all interrupts}$$

There are two problems with using the second equation. First, it provides an answer that is potentially too low if some interrupt handlers are reentrant. Second, it provides an answer that is too high if, for example, only one interrupt enables interrupts while running. A better approach is to determine the exact interrupt

Source	*Max. Interrupt Freq. (Hz)*
knife switch bounce	333
loose wire	500
toggle switch bounce	1 000
rocker switch bounce	1 300
serial port @115 kbps	11 500
10 Mbps Ethernet	14 880
CAN bus	15 000
I2C bus	50 000
USB	90 000
100 Mbps Ethernet	148 800
Gigabit Ethernet	1 488 000

Table 1.1: Potential sources of excessive interrupts for embedded processors. The top part of the table reflects the results of experiments and the bottom part presents numbers that we computed or found in the literature.

preemption graph—which interrupt handlers can preempt which others and when—and to use this as a basis for computing a bound.

From the point of view of stack depth analysis, reentrant interrupt handlers are a big problem. Stack depth cannot be bounded without knowing the maximum number of outstanding instances of the reentrant interrupt handler. Finding this number requires solving a global timing analysis problem that, as far as we know, has never been addressed either in industry or in academia. In general, a better solution is to avoid building systems that permit reentrant interrupts.

3.1.3 Tools for stack depth analysis

A few compilers emit stack depth information along with object code. For example, GNAT, the GNU Ada Translator, includes stack bounding support [4]. AbsInt sells a standalone stack depth analysis tool called StackAnalyzer [1]. The only tools that we know of that analyze stack depth in the presence of interrupts are ours [16] and Brylow et al.'s [5]. In related work, Barua et al. [3] investigate an approach that, upon detecting stack overflow, spills the stack data into an unused region of RAM, if one is available.

3.2 Interrupt Overload

Many interrupt-driven embedded systems are vulnerable to *interrupt overload*: the condition where external interrupts are signaled frequently enough that other activities running on a processor are starved. An interrupt overload incident occurred during the first moon landing, which was nearly aborted when a flood of radar data overloaded a CPU on the Lunar Landing Module, resulting in guidance computer resets [14, pp. 345–355]. The problem on Apollo 11 appears to have been caused by spurious signals coming from a disconnected device. Neglecting to bound maximum interrupt arrival rates creates an important gap in the chain of assumptions leading to a high-assurance embedded system.

Interrupt overload is not necessarily caused by high interrupt loads, but rather by unexpectedly high interrupt loads. For example, a fast processor running software that performs minimal work in interrupt mode can easily handle hundreds of thousands of interrupts per second. On the other hand, a slow processor running lengthy interrupt code can be overwhelmed by merely hundreds of interrupts per second.

Computing a reliable maximum request rate for an interrupt source in an embedded system is difficult, often requiring reasoning about complex physical systems. For example, consider an optical shaft encoder used to measure wheel speed on a robot. The maximum interrupt rate of the encoder depends on the maximum speed of the robot and the design of the encoder wheel. However, what happens if the robot exceeds its maximum design speed, for example while going downhill? What if the encoder wheel gets dirty, causing it to deliver pulses too often?

UU IR Author Manuscript

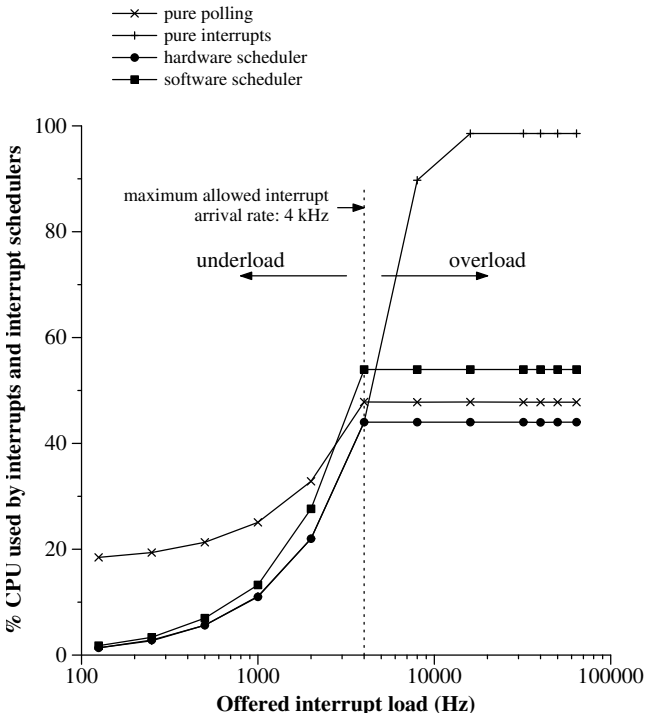

Figure 1.1: Interrupt schedulers at work

The data in Table 1.1 show some measured and computed worst-case interrupt rates. Switches that we tested can generate surprisingly high-frequency events ("bounces"), exceeding 1 kHz, during the transition between open and closed. This could easily cause problems for a system designed to handle only tens of switch transitions per second. The traditional way to debounce a switch is to implement a low-pass filter either in hardware or software. Although debouncing techniques are well-known to embedded systems designers, it is not enough just to debounce all switches: new and unforeseen "switches" can appear at run-time as a result of loose contacts or damaged wires. Both of these problems are more likely in embedded systems that operate in difficult environmental conditions with heat and vibration, and without routine maintenance. These conditions are, of course, very common—for example, in automobiles.

Network interface controllers (NICs) represent another potential source for interrupt overload. For example, consider an embedded CPU that exchanges data with other processors over 10 Mbps Ethernet using a specialized protocol that specifies 1000-byte packets. If the NIC generates an interrupt on every packet arrival, the maximum interrupt rate is 1.25 kHz. However, if a malfunctioning or malicious node sends minimum-sized (72 byte) packets, the interrupt rate increases to nearly 15 kHz [10], potentially starving important processing.

3.2.1 Preventing Interrupt Overload

The upper bound on the amount of time spent in a given interrupt handler is easy to compute: it's just the maximum execution time of the handler multiplied by the worst-case arrival rate. For example, if a network interface can deliver up to 10,000 packets per second and the interrupt handler runs for 15 μs, then at most 15% of the processor's time will be spent handling network interrupts.

There are a few basic strategies for making interrupt overload less likely or impossible:

- Keep interrupt handlers short. This is almost always a good idea anyway.
- Bound the arrival rates of interrupts. In many cases the worst-case arrival rate of an interrupt can be bounded by studying the interrupting device. Clearly, when assumptions are made (for example,

about the maximum rate of change of some physical quantity or the minimum size packet that will cross a network), these assumptions must be carefully justified.

- Reduce the worst-case arrival rate of interrupts. For example, a smart NIC need not generate an interrupt for each arriving packet and a serial port need not generate an interrupt for each arriving byte. The general pattern is that smart devices can greatly reduce the interrupt load on a processor.
- Poll for events rather than using interrupts. The problem with polling is that it adds processor overhead even when there are no events to process. Some existing hardware devices, such as NICs, already adaptively switch between interrupts and polling depending on system load, but generally they do not do so in a timely fashion: there is a period of overload before the mode switch occurs. This would be harmful in the context of a real-time system.
- Use an *interrupt scheduler*: a technique that we developed in previous work [15]. An interrupt scheduler is a small piece of hardware or software that limits the maximum arrival rate of an interrupt source. The software implementation works by switching between interrupt behavior when arrival rates are low and polling-like behavior when arrival rates are high. Figure 1.1 illustrates the effect of an interrupt scheduler attached to an Atmel AVR processor running at 4 MHz. In this scenario, the interrupt handler performs 250 cycles (62.5 μs) of work and the developers would like to handle at most 4000 interrupts per second. Pure polling performs well during overload, but incurs close to 20% CPU overhead even when no interrupts are arriving. In contrast, the pure interrupt-driven solution performs well when few interrupts arrive, but saturates the CPU during overload. The interrupt schedulers support good performance during both underload and overload. The software scheduler, which runs on an unmodified CPU, incurs some overhead, whereas the hardware scheduler essentially offers perfect performance characteristics.

3.3 Real-Time Analysis

Section 3.2 addressed the problem of bounding the arrival rate of interrupts in order to prevent them from starving non-interrupt work. In reality, preventing interrupt overload is just one subproblem of the overall problem of *real-time schedulability analysis*: ensuring that the time constraints of all computations can be met. The literature on real-time scheduling is vast. In this section we deal only with issues specific to interrupt-driven software, and provide references to the wider literature.

Every embedded system has a discipline for scheduling interrupts: for deciding which interrupt handler gets to execute, and when. Some hardware platforms, such as the x86 with PIC, provide *prioritized, preemptive scheduling* by default. A prioritized interrupt scheduler permits a high-priority interrupt to preempt a lower-priority interrupt. On the other hand, if a low-priority interrupt becomes pending while a higher-priority interrupt is executing, the low-priority interrupt cannot execute until it is the highest-priority pending interrupt. Prioritized interrupts can be analyzed using *rate monotonic analysis* (RMA). In some cases interrupt priorities are fixed at hardware design time, limiting the flexibility of a system to accommodate, for example, a new latency-sensitive peripheral. Embedded platforms with sophisticated interrupt controllers permit priorities to be configured dynamically.

To apply RMA to an interrupt-driven system, it is necessary to know the *worst-case execution time* (WCET) C and the *minimum interarrival time* T for each of the N interrupt handlers in a system. The minimum interarrival time of an interrupt is the inverse of its maximum arrival rate. Then, the overall CPU utilization U of interrupts is computed as:

$$U = \sum_{i=1..N} \frac{C_i}{T_i}$$

As long as U is less than 0.69, it is guaranteed that every interrupt will complete before the end of its period. On the other hand, if interrupts have deadlines shorter than their interarrival times, or if interrupts are disabled for non-negligible amounts of time, then the situation is more complex. However, solutions exist to many different variations of the scheduling problem. Good references on prioritized, preemptive scheduling include Audsley et al. [2] and Klein et al. [11]. Also, products for analyzing fixed-priority systems, such as RapidRMA [20] and the RTA-OSEK Planner [6], are available.

An important benefit of prioritized, preemptive interrupt scheduling is that it composes naturally with the predominant scheduling discipline provided by real-time operating systems (RTOSs): prioritized, preemptive thread scheduling. In this case, an overall schedulability analysis is performed by considering interrupts to have higher priority than threads.

Some platforms, such as Atmel's AVR, provide prioritized scheduling among pending interrupts, but not among concurrently executing interrupts. Thus, by default, the AVR does not provide prioritized preemptive interrupt scheduling: the distinction is subtle but important. For example, if a low-priority AVR interrupt becomes pending while a high-priority interrupt is running with interrupts enabled, then the low-priority interrupt preempts the high-priority interrupt and begins to execute. To implement prioritized preemptive scheduling on the AVR, it is necessary for each interrupt handler to clear the individual enable bits for all lower-priority interrupts before enabling interrupts. Then, these bits must be reset to their previous values before the interrupt handler finishes executing. This bit-twiddling is cumbersome and inefficient. On the other hand, the AVR naturally supports *prioritized, non-preemptive interrupt handling* if interrupt handlers avoid enabling interrupts.

The real-time behavior of a system supporting prioritized, non-preemptive interrupt handling can be analyzed using non-preemptive static priority analysis, for which George et al. have worked out the scheduling equations [9]. To analyze non-preemptive interrupts composed with preemptive threads, an analysis supporting mixed preemption modes must be used. Saksena and Wang's analysis for preemption threshold scheduling is suitable [17, 21]. In this case, the priority of each interrupt and thread is set as in the fully preemptive case, while the preemption threshold of each interrupt handler is set to the maximum priority, and the preemption threshold of each thread is set to the priority of the thread. The overall effect is that interrupts may preempt threads but not interrupts, while threads can preempt each other as expected.

4 Guidelines for Interrupt-Driven Embedded Software

This section synthesizes the information from this chapter into a set of design guidelines. These guidelines are in the spirit of MISRA C [12]: a subset of the C language that is popular among embedded software developers. For embedded systems that are not safety critical, many of these rules can be relaxed. For example, a full WCET and schedulability analysis is probably unnecessary for a system where the only consequence of missed real-time deadlines is degraded performance (e.g., dropped packets, missed data samples, etc.). On the other hand, it is seldom a good idea to relax the rules for stack correctness or concurrency correctness, since stack overflows and race conditions generally lead to undefined behavior.

For each of these rules, developers creating embedded software should be able to justify—to themselves, to other team members, to managers, or to external design reviewers—either that the rule has been followed, or else that it does not apply. Obviously these rules should be considered to be over and above any in-house software design guidelines that already exist. Chapters 4 and 5, and Appendix A, of Ganssle's book [8] are also good sources for design guidance for interrupt-driven software.

A difference between these guidelines and MISRA C is that tools for statically checking many of the interrupt guidelines either do not exist, or else are not in common use. A reasonable agenda for applied research in embedded software would be to create software tools that can soundly and completely check for conformance to these rules.

4.1 Part 1: Scheduling

This set of guidelines invites developers to quantify the scheduling discipline for interrupts and to figure out if it is the right one.

1. The scheduling discipline for interrupts must be specified. This is straightforward if scheduling is either preemptive or non-preemptive priority-based scheduling. On the other hand, if the scheduling discipline mixes preemptive and non-preemptive behavior, or is otherwise customized, then it must be precisely defined. Furthermore, response-time equations for the scheduling discipline must either be found in the literature or derived.

UU IR Author Manuscript

2. The scheduling discipline must never permit two different interrupt handlers to each preempt the other. This is never beneficial: it can only increase the worst-case interrupt latency and worst-case stack depth.

3. When one interrupt is permitted to preempt another, the reason for permitting the preemption should be precisely identified. For example, "the timer interrupt is permitted to preempt the Ethernet interrupt because the timer has a real-time deadline of 50 μs while the Ethernet interrupt may run for as long as 250 μs." System designers should strive to eliminate any preemption relation for which no strong reason exists. Useless preemption increases overhead, increases stack memory usage, and increases the likelihood of race conditions.

4.2 Part 2: Callgraph

Once the callgraphs for a system have been identified, many nontrivial program properties can be computed.

1. The callgraphs for the system must be identified. There is one callgraph per interrupt, one for the main (non-interrupt) context, and one for each thread (if any). Most callgraph construction can be performed automatically; human intervention may be required to deal with function pointers, computed gotos, longjumps, and exceptions.

2. Every cycle in the callgraph indicates the presence of a direct or indirect recursive loop. The worst-case depth of each recursive loop must be bounded manually. Recursive loops should be scrutinized and avoided is possible. It is particularly important to identify (or show the absence of) unintentional recursive loops. Unintentional recursion commonly occurs in systems that use callbacks.

4.3 Part 3: Time Correctness

1. The maximum arrival rate of each interrupt source must be determined. This could be a simple rate or else it could be bursty. The reason that the interrupt rate cannot exceed the maximum must be specified. This reason can either be external (i.e., there is some physical limit on the interrupt rate) or internal (i.e., an interrupt scheduler has been implemented).

2. The deadline for each interrupt must be determined. This is maximum amount of time that may elapse between the interrupt's firing condition becoming true and the interrupt's handler running. Furthermore, the cost of missing the deadline must be quantified. Some interrupts can be missed with very little cost, whereas others may be mission critical. In general only part of an interrupt handler (e.g., the part that reads a byte out of a FIFO) must complete by the deadline. The final instruction of the time-sensitive part of the interrupt handler should be identified.

3. The WCET of each interrupt must be determined. This can be computed using a WCET tool, by measurement if the interrupt code is straight-line or nearly so, or even by manual inspection. Gross estimates of WCET may be valid if the consequences of missing real-time deadlines are not catastrophic.

4. The WCET of the non-time-sensitive part of each interrupt handler should be considered. If these entrain a substantial amount of code, then the non-time-sensitive code should be run in a delayed context such as a deferred procedure call [19, pp. 107–111] or a thread in order to avoid unduly delaying lower-priority interrupts.

5. The longest amount of time for which interrupts are disabled must be determined, this is simply the maximum of the WCETs of all code segments that run with interrupts disabled. This time is used as the *blocking term* [18] in the schedulability equations.

6. The ability of each interrupt to meet its deadlines must be verified. This is accomplished by plugging the system's constants into the appropriate schedulability equation, such as the rate-monotonic rule shown in Section 3.3.

7. The impact of interrupts on any non-interrupt work with real-time deadlines must be considered; the details are beyond the scope of this chapter.

4.4 Part 4: Stack Correctness

1. A *stack model* for the system should be developed, identifying the effect of interrupts on each stack in the system (see Section 2.2) and also the way that individual interrupt stack depths combine (see Section 3.1.2). Any assumptions made by the stack model should be explicitly justified.
2. The stack budget for interrupts must be determined. This is just the worst-case (smallest) amount of RAM that may be available for interrupts to use under any circumstances.
3. The worst-case stack memory usage of each interrupt must be determined. This could be done using a tool, by manual inspection of the object code, or else by measurement if the interrupt code is straight-line or nearly so.
4. The overall worst-case stack depth for interrupts must be computed by plugging the system's constants into the stack model.

4.5 Part 5: Concurrency Correctness

1. Reachability analysis must be performed using the callgraph to determine which data structures are reachable from which interrupt handlers, and from the program's main context. A data structure is any variable or collection of variables whose state is governed by a single set of invariants. A data structure is *shared* if it may be accessed by more than one interrupt handler, if it may be accessed by at least one interrupt and by the non-interrupt context, or if it is accessed by any reentrant interrupt. Any data structures not meeting these criteria are *unshared*. Note that the reachability analysis is complicated by pointers, whose worst-case points-to sets must be taken into account.
2. All automatic (stack allocated) data structures must be unshared.
3. A data structure is *protected* if its invariants are broken only in an *atomic* context. In an interrupt-driven system, atomicity is achieved by disabling one or more interrupts. All shared variables should be protected.
4. For every unprotected shared variable, a detailed description must be provided explaining why the unprotected access does not compromise system correctness.
5. For every reentrant interrupt handler, a detailed explanation must be provided that explains why the reentrancy is correct and what benefit is derived from it.
6. Every function that is reachable from an interrupt handler must be reentrant.

5 Conclusions

This chapter has summarized some problems associated with creating correct interrupt-driven software, and it has proposed a set of design rules that can help developers avoid common (and uncommon) traps and pitfalls. It is important for developers to gain a strong understanding of interrupts and their associated software development issues. First, these issues occur across a wide range of embedded software systems. Second, many of the same issues recur in the kernels of general-purpose operating systems. Third, it is empirically known that interrupts are hard to get right. Finally, embedded systems are often used in safety-critical applications where the consequences of a bug can be severe.

References

[1] AbsInt. StackAnalyzer, 2004. `http://www.absint.com/stackanalyzer`.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sept. 1993.

[3] S. Biswas, M. Simpson, and R. Barua. Memory overflow protection for embedded systems using run-time checks, reuse and compression. In *Proc. of the ACM Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Washington, DC, Sept. 2004.

[4] E. Botcazou, C. Comar, and O. Hainque. Compile-time stack requirements analysis with GCC. In *Proc. of the 2005 GCC Developers Summit*, Ottawa, Canada, June 2005.

[5] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, pages 47–56, Toronto, Canada, May 2001.

[6] ETAS Inc. OSEK-RTA Planner, 2006. `http://en.etasgroup.com/products/rta`.

[7] J. Ganssle. *The Art of Designing Embedded Systems*. Newnes, 1999.

[8] J. Ganssle. Cheap changes. *Embedded.com*, July 2004. `http://www.embedded.com/showArticle.jhtml?articleID=22103322`.

[9] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Technical Report 2966, INRIA, Rocquencourt, France, Sept. 1996.

[10] S. Karlin and L. Peterson. Maximum packet rates for full-duplex Ethernet. Technical Report TR-645-02, Princeton University, Feb. 2002.

[11] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[12] MISRA. MISRA-C:2004—Guidelines for the use of the C language in critical systems, 2004. `http://www.misra.org.uk`.

[13] Moteiv. Telos rev. B datasheet, 2005. `http://www.moteiv.com`.

[14] C. Murray and C. B. Cox. *Apollo: The Race to the Moon*. Simon and Schuster, 1989.

[15] J. Regehr and U. Duongsaa. Preventing interrupt overload. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, IL, June 2005.

[16] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, Oct. 2003.

[17] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symp. (RTSS)*, Orlando, FL, Nov. 2000.

[18] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[19] D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.

[20] TriPacific Inc. RapidRMA, 2006. `http://www.tripac.com`.

[21] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. of the 6th Intl. Workshop on Real-Time Computing Systems and Applications*, Hong Kong, Dec. 1999.

Index

UU IR Author Manuscript