Symbian OS Internals/18. Ensuring Performance

- Symbian OS Internals Table of Contents

by Jasmine Strong with Dennis May

The best way to accelerate a PC is at 9.8 ms/-2
Jane Sales after Marcus Dolengo

In a real-time environment, ensuring acceptable performance is not as simple as one might expect. A real-time system has been defined as one in which the validity of the results depends not only on the logical correctness of the results, but also on the timeliness of their delivery. EKA2 provides a number of constructs and techniques for ensuring that code executes not only at a sufficient rate but also within an acceptable time frame.

We can, therefore, aim to improve code performance in two areas: in its flat-out speed, and in its ability to respond to real-world events that require software intervention. The provision of real-time behavior within the EKA2 operating system requires not only that code runs quickly, but also that drivers and, to a lesser extent, application software be careful to act as a good neighbor. Sometimes the fastest algorithm can compromise the real-time behavior of the rest of the system, and so best throughput must be sacrificed in order to provide better real-time performance. For example, inter-process communication in EKA2 is not quite as fast as it was under EKA1 because the kernel now pauses every 512 bytes to check if there is contention for the system lock. This allows anything that might be waiting on the system lock with higher priority than the current thread to run in good time, rather than having to be suspended for the duration of the whole operation, which could be many milliseconds. On the user-side, this problem is already solved for you. As long as an application cannot set its priority too high, the design of EKA2 prevents it from adversely affecting the real-time behavior of other software in the system.

The analysis of code performance on an interrupt-heavy system is a complex topic and quite controversial in parts. This analysis is complicated further because the typical architectures that run EKA2 have more advanced and complex memory systems than earlier platforms.

Care must be taken in the board support package, device drivers, kernel extensions, and in the use of file systems to ensure that these complex issues are sensitively handled. It is all too easy to compromise system performance with a single bit error in the configuration of a memory controller, or to allow an unforeseen interaction in a device driver to substantially increase latency.

To deliver a product with high performance requires that you understand the factors that contribute to system performance - both software factors such as context switch time and contention for exclusive locks and hardware factors such as memory access times, I/O throughput and cache architecture - and that you design the software around these. The real-time behavior of the OS is more fully described in Chapter 17, *Real Time*. In this chapter I concentrate on performance and I do not duplicate the material found there.

Before you start to modify any of the code in your system you must first profile and measure the existing performance. Only by careful testing and analysis can you truly understand how and where the performance is being consumed. Guessing at what to optimize will not solve your problems.

Writing efficient code

Writing highly efficient code for ARM processors is not as straightforward as it might at first seem. With the advent of the Thumb instruction set (see Chapter 2, *Hardware for Symbian OS*, for details), the performance gap between the most space-efficient code and the most time-efficient code has widened considerably. To write rapid code you need not write large portions of the application in assembly language. You can get 90% of the benefit in 10% of the time with care in the design phase. Of course you can use all the usual optimization strategies such as using binary searches rather than linear searches, using ASCII instead of Unicode if possible and using the most efficient available algorithm. But ARM processors have some quirks that make their performance levels vary in a way that is different from some other processors. Here are some techniques that can help.

Avoid tight loops

Tight loops (by *tight loop* I mean one that has only a very brief operation inside it) are very inefficient in ARM9 and earlier CPUs because these processors do not have branch prediction hardware. The lack of this hardware entails that the branch at the end of the loop incurs a time penalty while the pipeline is flushed and reloaded from the start of the loop. If the operation inside the loop requires only a few cycles, the branch penalty cycles can take more time than the actual operation.

XScale, ARM11 and all future cores have branch prediction hardware and do not suffer from the above looping problem. Their hardware predicts branch instructions that will be taken based on past history of the code and heuristic rules. If a branch is correctly predicted the CPU will continue execution without a penalty, unpredicted branches will incur a pipeline flush penalty.

ARM11 goes one step further with branch folding. Predicted branch instructions are removed from the pipeline and the branch target instruction is executed immediately thus saving one more cycle.

Sometimes, of course, you can't avoid writing code that does this sort of thing, and in these cases you may decide to *unroll the loop* by performing more than a single operation in each iteration. You must weigh the increased code size produced by unrolling the loop against the improved execution time it gives. For example, take this memory checksum function:

```
TInt trivial_checksum(TInt8* aBlock, TInt aLen)
    {
    TInt i=0;
    TInt8* j;
    for (j=aBlock; j<aBlock+aLen; j++)
        i+=*j;
    return i;
    }
```

Compare this to an unrolled version:

```
TInt trivial_checksum_unrolled(TInt8* aBlock, TInt aLen)
    {
    TInt i=0;
    TInt8* j;
    TInt8* endptr;
    endptr=aBlock+aLen;
    for (j=aBlock; j<endptr; ) // we unroll this to degree 5
        {
        if (j<endptr) i+=*j++;
        else
            break;
        if (j<endptr) i+=*j++;
        else
            break;
        if (j<endptr) i+=*j++;
        else
            break;
        if (j<endptr) i+=*j++;
        else
            break;
        if (j<endptr) i+=*j++;
        else
            break;
        }
    return i;
    }
```

While the unrolled version is clearly much larger and more difficult to read, because it branches much less frequently it will execute much more rapidly than the non-unrolled version. If you use this technique to unroll loops, be sure to check the assembled output from the compiler to ensure that it does what you want.

The easiest way to ensure that you're using a fast, unrolled operation is to use one provided by Symbian OS. The library-provided memory and string operations are very thoroughly unrolled, so will execute quickly without bloating your application. For example, the provided implementation of `Mem::Copy()` is very fast indeed.

Optimize the general case

Code often has a *general case* that executes very often and a *special case* that only executes much more infrequently. If you can determine which case happens most often and make this general case faster, even at the expense of a little performance in the

special case, you can improve the overall performance of your program markedly.

Don't repeatedly make small requests

If you have a list of items to get when you go shopping, do you get in your car, drive to the supermarket, take a trolley around, buy the first thing on the list, come home, look at the second thing, get in the car, drive to the supermarket, take a trolley around, buy it, come home, look at the third thing on the list, and so on? That would be crazy, and would waste a lot of time, effort and fuel.

Nevertheless, many people release code that does things exactly like this! It's even worse in code, because it defeats many of the performance improving features of processors: the cache, which depends upon spatial and temporal locality to do its job, and burst transfer modes, which depend upon the high probability of handling more than one datum at a time.

If your program needs to open a file or any other kind of server connection, be aware that there are significant overheads in setting up a connection. If you repeatedly open a connection, perform a small operation and then close it again, you will waste huge amounts of processor time, memory bandwidth and battery power.

Similar provisos apply to image handling methods: while most of the image handling APIs provide methods for querying the values of single pixels, it is very inefficient indeed to perform single pixel operations on a large number of pixels. There is a significant overhead in every method invocation and an even bigger overhead in every IPC. It is important to make every one count.

File input and output is the same. These classes provide block methods for a reason: making a fileserver request takes a significant amount of time over and above the time required to actually fulfill the request. Use the block read and write methods if you need to read or write more than one character. In some cases, this can be literally thousands of times faster.

Memory allocation can also be slow, so you may wish to avoid repeatedly invoking memory allocation functions. I describe some approaches to improving the performance of memory allocation later in Section 18.2.9.

Performance differences between ARM platforms

Different ARM processors, even at the same clock frequencies, have different performance characteristics. With ARMv6 processors now becoming available, these differences have become very significant. If you're an application developer, you will want your code to work well on all ARM-compatible processors to maximize your audience. To achieve this, it's important to understand some of the differences between ARM platforms. Symbian OS has historically supported three-and-a-half ARM architectures. I'll ignore the ARMv3 and ARMv4 architectures because they are no longer supported by Symbian OS, and turn instead to the two that are actively supported today.

Applying optimizations

Many factors will influence the selection of optimizations that you choose to perform. If your device is short of memory, you might decide to forego time performance in order to use Thumb code, which is typically about 70% of the size of ARM code. Conversely, if your device is short of processing power, you might decide that the 20% faster execution of ARM code is a better compromise for your application. Depending on what your code does, ARM might be much faster than Thumb, particularly if you need multiword arithmetic or other things that are improved by instructions that are only available in ARM state. Your choice of optimizations will also depend on which processor you expect your code to run on. For example, there's little need to unroll loops if your code will only run on processors with branch prediction such as ARM1136JF-S and `XScale`.

In general, you have to target your code at the slowest processor it will run on. If you are aiming for the installed base of EKA2-based Symbian OS devices, these will all be ARMv5 or higher, which means that they have Harvard architecture caches (The Harvard architecture separates the instruction stream from the data stream, which potentially allows any instruction to access memory without penalty) at least 5-stage pipelines and they support the Thumb instruction set. If your performance is limited by the amount of code that can fit in the instruction cache, you may find that your performance is actually improved by compiling to a Thumb target. However, this is rare, especially on ARMv5 platforms where every process switch potentially eliminates useful content from the data cache. Performance on these devices is generally limited by the high rate of data cache misses due to these flushes.

The memory systems on mobile phone platforms are usually quite slow, and so, to alleviate the performance problems this creates, ARM processors have caches. The cache is a special area of memory that is used to keep a local copy of words that are also used in main memory.

Cache memory is very expensive, so most processors do not include very much of it. In ARM systems, the processor cache is always incorporated into the processor chip.

The influence of caches

Caches are the main source of unexpected interactions that affect program execution speed. For example, a benchmark that repeatedly performs a simple operation may show a two microsecond context switch, but this is only possible because the cache is filled early in the test and remains filled throughout. On a real system where the threads being switched do actual work, the contents of the cache are displaced, which makes context switching much slower. Also, because caches contain scattered pieces of information from throughout main memory and have to be able to find that information very quickly, they use something called *tag RAM*. Tag RAM is even more expensive than the rest of the cache, so to reduce the amount needed, spatial locality is exploited by organizing the cache into *lines*, typically of eight 32-bit words each. Each line has a single tag, and will be filled in from main memory in a single operation: an eight-word burst. It is much faster to do this than to fetch eight single words from memory: it may even be more than eight times faster, because many memory systems require a delay between requests.

When a processor with a cache is running code and data that fits entirely within the cache, it is said to be *running from cache*. Running from cache has a number of benefits: primarily, it is considerably faster than running from DRAM or, even slower, flash memory. It also generates fewer memory bus cycles, which in turn saves power. If the DRAM is not used for some time, some DRAM controllers will even close pages and place them in retention mode, reducing power consumption even further.

When the code and data in the current active set does not quite fit inside the cache, lines will have to be evicted from the cache to make room for each successive new datum or instruction that the program flow encounters. It is not uncommon for items of data to be repeatedly evicted and reloaded. This unwelcome state of affairs is known as *thrashing*. The choice of algorithm used to select lines to evict from the cache determines the processor's behavior when thrashing. Many of the processors described in this chapter support two algorithms, round robin (RR) and random replacement. RR replacement evicts lines in turn, whereas random replacement selects them at random. When an RR cache is thrashing, the performance of the processor drops off extremely quickly: it falls off a *performance cliff*. On the other hand, when a randomly replaced cache starts to thrash, its performance degrades more gracefully. For this reason, random replacement is to be preferred to round robin. On processors where the replacement algorithm is a run-time option, this is configured using a CP15 configuration register setting in the base port; it is very important to ensure that this is configured appropriately.

Alignment

Since ARM processors generally only handle word-aligned memory operations, accessing non-single-byte quantities that are not aligned to a word boundary is not as fast as accessing ones that are. The `Mem::Copy()` method is very heavily optimized, using a special *twister* algorithm to copy unaligned data using aligned memory access cycles. This makes it about three times faster than a conventional implementation of memcpy() for both unaligned copies and for copies in which the source is not aligned with the destination. Similarly, particularly on ARM926EJ platforms, the memory interface does not generate burst accesses for cache missing STM or LDM instructions that are not aligned to a 16 byte boundary. The Mem:: methods include code to alleviate these problems, and their performance is very close to the theoretical maximum.

In some situations you may choose to modify your code to place important members of structures in aligned locations. Packing structs can help to save memory but may significantly reduce the available memory bandwidth on these processors. ARM926EJ is a very popular core at the time of writing and can be found in many mobile phones. ARM926EJ cache lines are eight words long, so to be aligned, an address must be divisible by 32.

Pragmatism, not idealism

Unfortunately, there is only a finite amount of work that any processor can do in one second, and that amount depends on the nature of the work. While the processors available to mobile phone applications now would have been astonishingly powerful as recently as ten years ago, they aren't that powerful compared to workstation processors. Mobile phone manufacturers have to choose processors based on their benchmark scores. That is, they use figures published by manufacturers, measured using programs designed to advertise how fast their processors are. The problem with this is that the performance of a processor is entirely dependent on what it's being asked to process. The application binary environment, compiler and even the instruction set may have been tweaked to improve benchmark performance! That's all very well as long as your application does the same sort of thing as the benchmark, and only the same sort of thing - rather than a mixture of different operations. To make a safer estimate, it's important to have a set of representative, mixed operation benchmarks, timing how long it takes to perform actual user operations, using real applications. The user interface can be as important as the application engine itself in these tests.

It's not a simple operation to take a set of benchmark results, perform an examination of a very different problem, and say with certainty that problem x can be solved with processor y in time *t* . It is, therefore, very important to define the scope of your application with this in mind. The manufacturer's benchmarks, such as Dhrystone, running on the *bare metal* of the machine, may demonstrate performance far in excess of that actually available in a real-time, multithreaded, interrupt rich environment. Treat these benchmarks as you do all other forms of advertising!

Memory throughput

ARM systems have fairly fast memory systems. They consist of a large DRAM, accessed by the processor through a cache, prefetch buffer and a write buffer. The cache usually consists of around 16 KB of very fast memory, which is usually run at the same speed as the processor core itself. This is necessary in the ARM architecture because almost every cycle will issue an instruction.

All ARMv5 and later processors have a write buffer, which is a simple FIFO queue that accepts data to be written to memory. It serves to decouple the processor pipeline from the core's data port so that if the cache misses on the data to be written, the processor pipeline does not stall. The size of the FIFO is usually eight words, and the words are usually not required to be ordered in any way. ARM systems today use a wide variety of different interconnects and memory types, but most current systems use synchronous DRAM, similar to SDRAM or DDR RAM, attached to essentially transparent interconnects. These clock at around half the processor core speed, though with the trend towards increased core speeds this is likely to change in the near future. Power consumption and processor package pin count constrain mobile memory systems more than component cost, which means that most mobile platforms use 16-bit wide memory interfaces. This essentially limits the maximum sustained bandwidth of their memory interfaces to one word every four core cycles. This variable remains remarkably consistent across new and old processors.

However, in practice this limit can never be achieved. SDRAM is line-oriented, that is, it is designed in such a way that a *row address* is applied, taking between two and seven cycles depending on the type of RAM, then a *column address* is applied, returning data usually between two and three cycles later. Subsequent words from the same row can then be fetched in subsequent cycles without delay. This *burst mode* is very useful in cached architectures like ARM, where whole lines of eight words are generally fetched at any time, and can make memory access several times faster.

The exact management of RAM parameters is beyond the scope of this book, but is critical to the performance of the system as a whole. It is very important to ensure that systems have their RAM configured correctly.

Maintaining real-time performance

Techniques and limitations

This section explores some techniques for writing code with real-time performance requirements and also for writing code that does not compromise the real-time requirements of other code. It also explains some of the limitations of EKA2 for real-time systems.

Task partitioning

The first thing you should consider when writing any real-time software is how to partition the desired functionality into separate tasks, according to the deadlines involved. Some of these tasks may run as interrupt service routines, others will run in threads with varying priority levels. You should only do the minimum amount of work needed to meet a deadline at any given priority level. Work that is less urgent should be postponed to lower priority tasks, where it will not hold up more urgent activities. This is especially important for tasks which run as ISRs, since these run in preference to threads. Interrupts should always be kept as short as possible.

The following table gives a rough guide to the thread priority ranges that would be used for tasks with certain deadlines:

Priority	Deadline range	Comments
0	-	Null (idle) thread.
1-15	-	Normal application priorities.
16	-	Kernel cleanup activities.
16-24	-	System servers (file server, window server and so on).
25-26	>100 ms	Media drivers.
27	>20 ms	General device drivers. The *default* DFC thread (DfcThread()) runs at this priority.
28-31	2-20 ms	*Real-time* priority Symbian OS user processes.
28-47	2-20 ms	High priority kernel-side threads and personality layer threads. Priorities above 31 are not directly accessible user-side and must be set using a device driver.
48	2-20 ms	Nanokernel timer thread.
49-63	100µs-10 ms	Personality layer threads.
IDFC	100µs-	Personality layer routines to enable ISRs to wakeup threads

IDFC 1ms Personality layer routines to enable ISRs to wakeup threads.

ISR 10µs-1ms

Threads that run in normal Symbian OS user processes, with priorities of `EPriorityForeground` or lower, have absolute priorities between 1 and 15. Various Symbian OS servers with high legacy code content, which were not written with real-time performance in mind, inhabit priorities 16-24 and so it is not possible to give any meaningful real-time guarantees for code running at these priorities. At the high end of the spectrum, the deadlines possible at each priority depend on the speed of the hardware and how the base port is written. On very fast hardware it should be possible to service deadlines of 100 µs with a priority 63 kernel thread, provided the base port does not have long ISRs. On slower hardware, tasks with deadlines of 1 ms will need to be ISRs.

Avoid priority inversion

When choosing a mutual exclusion method for use in software that must meet real-time deadlines, you should take care to minimize the time for which priority inversion exists. The following methods are available:

1. Lock-free algorithms Lock-free algorithms for updating data structures make use of atomic memory access instructions such as SWP on ARMv5, LDREX/STREX on ARMv6 and locked instructions on IA32. The ARM SWP instruction atomically interchanges the contents of a register with the contents of a memory location. The instruction is atomic with respect to other code running on the same processor (simply by virtue of being a single instruction, so not interruptible between the read and write) and also with respect to accesses by other bus masters. No other bus master in the system can access the memory location referred to in between the read and write part of the SWP. The ARM LDREX and STREX instructions make use of an exclusive access monitor, which on systems with only a single processor is just a single bit hardware flag. The flag is usually clear, or, in ARM's terminology, in the Open Access state. Executing an LDREX instruction loads the contents of a memory location into a register and sets the flag (puts it into the *Exclusive Access* state). Executing a STREX instruction checks the state of the flag. If it is in the *Exclusive Access* state, the contents of the register is written into a memory location and a value of 0 is written into the result register. If the flag is in the *Open Access* state, no memory write occurs and the result register is set to 1. In either case, the flag is reset to the *Open Access* state by a STREX instruction. The usage of these instructions is illustrated here:

Use of LDREX and STREX

```
; Atomically increment the memory location at [R1]
; Return the original memory contents in R0
1 LockedInc:
2 LDREX R0, [R1]
3 ADD R2, R0, #1
4 STREX R3, R2, [R1]
5 CMP R3, #0
6 BNE LockedInc
```

Line 2 reads the original value of the memory counter and places the monitor into the *Exclusive Access* state. Line 3 increments the value. Line 4 writes the new value to memory, provided the monitor is still in the *Exclusive Access* state. Lines 5 and 6 retry the entire operation if the store at line 4 found the monitor in the *Open Access* state. To ensure the correct functioning of such routines, the operating system must perform an STREX as part of the interrupt preamble. This will cause the monitor to be reset to the Open Access state if any interrupt occurs between the load and store, which in turn will cause the load-increment-store to be retried. For more details on the operation of the LDREX and STREX instructions, refer to the *ARM Architecture Reference Manual* (by Dave Seal. Addison-Wesley Professional). The IA32 architecture supports many instructions that both read and write a memory location. Since interrupts cannot occur within a single instruction these instructions are automatically atomic with respect to code executing on the same processor. If atomicity with respect to other bus masters is required - for example on a system with more than one CPU sharing memory - you can apply the LOCK prefix to the instruction. This causes the CPU to output a lock signal which prevents any other bus master getting in between the read and write parts of the instruction. Lock-free algorithms are recommended where possible for both kernel and user-side use, since they are usually very fast, but can only be used for certain tasks. The tasks for which they can be used depend on the set of atomic instructions available on the processor in question. As an example of a lock-free algorithm, consider adding elements to the end of a singly linked list. The following example shows some code to do this.

Lock-free singly linked list add

```
; Add the element pointed to by R0 to a singly linked
; list pointed to by R1. The first word of
```

```
; each element is the link pointer, which points
; to the next element in the list or is zero if this
; is the last element.
; The list itself consists of two pointers – the first
; points to the last element on the list (zero if the
; list is empty), and the second points to the first
; element in the list (again zero if the list is
; empty).


1 MOV R2, #0
2 STR R2, [R0]
3 SWP R2, R0, [R1]
4 STR R0, [R2]
5 CMP R2, #0
6 STREQ R0, [R1, #4]
```

Lines 1 and 2 set the link pointer of the element being added to zero, since it is the last element. Line 3 atomically gets the address of the current last element and sets the element being added as the last element. Line 4 sets the link pointer of the previous last element to the one just added. Lines 5 and 6 set the element just added as the first element if the list was initially empty. It can be seen that this algorithm works even if the function is called simultaneously in multiple threads.

2. Disabling interrupts This method is only available kernel side, and it must be used if any of the contending tasks runs in an ISR, unless a lock-free algorithm can be used. The duration for which interrupts are disabled should not exceed the maximum time for which the kernel disables interrupts. A good rule of thumb is that it should not exceed 10 µs.

3. Disabling preemption This method is only available kernel side and it must be used if any of the contending tasks runs in an IDFC, unless a lock-free algorithm can be used. The duration for which preemption is disabled should not exceed the maximum time for which it is disabled by the kernel. A good rule of thumb is that it should not exceed 30 µs.

4. Fast mutexes Use of a fast mutex is the recommended method for mutual exclusion between kernel-side threads. Both Symbian OS threads and nanothreads can use it. If the system lock is used, it should not be held for longer than the kernel holds it. A reasonable rule of thumb for this is that it should not be held for more than 100 µs.

5. Symbian OS mutexes Symbian OS mutexes are available to both kernel- and user-side code. They are slower than methods 2-4, and so we only recommend them for critical sections that are entered relatively infrequently (periods in the millisecond range) and for critical sections that contain other nested critical sections protected by fast mutexes - because fast mutexes cannot nest. Obviously these mutexes are only available to Symbian OS threads.

6. Threads Our final method is the use of a single thread to run multiple tasks, one at a time. This can be the simplest method and should work well provided that the tasks have similar deadlines and execution times. The nanokernel's DFC queues make this method particularly simple to use, and the different DFC priorities within the same queue can be used to cope with whatever differences there are in task deadlines. This method is the slowest since it involves two context switches, instead of a mutex wait and signal.

For user-side code, only methods 1, 5 and 6 are available. The `RFastLock` primitive available to user code is faster than `RMutex` but it does not implement any form of priority inheritance so will generally be unsuitable for applications requiring real-time guarantees.

Algorithm selection

For an application with real-time constraints, the algorithm that you would choose for a particular function is often different from the one that you would choose in the absence of those constraints. This is because you need an algorithm with predictable execution time. This may not be the fastest algorithm in the average case, and it may be less memory efficient than other algorithms.

For example, when you are managing a list of N elements, you should use constant time or logarithmic time algorithms rather than algorithms that are linear or higher order, unless the value of N is well bounded. So, you would generally use doubly linked lists rather than singly linked lists if you need to insert and/or remove items from the middle of the list. If the list must be ordered according to some key, then you can use a priority list structure (as in Section 17.3.1.1) if the number of possible key values is small, or you can use a tree-based structure.

To sort a list of N elements, where N is known, you should prefer heap sort to quick sort, even though quick sort has lower

average execution time. This is because quick sort has a pathological worst-case execution time proportional to N^2.

Lists come in intrusive and non-intrusive varieties. Intrusive lists require that the objects on the list include reserved space used for management of the list, so that the object must *know* what lists it will potentially be added to. Non-intrusive lists do not make such demands; instead any memory required for management of the list is allocated dynamically when the object is added to the list and freed when the object is removed from the list. For real-time applications you should clearly prefer intrusive lists, since non-intrusive lists require memory allocation and freeing when adding and removing elements, which spoils, or at least makes it more difficult to achieve, real-time behavior.

Hardware issues

The design of the hardware places fundamental limits on how well real-time applications will run on that hardware. I have already mentioned that memory bandwidth has a large effect on achievable interrupt and thread latencies.

There can also be other more subtle problems, one example being the effect of DMA on memory bandwidth. Generally DMA controllers have higher bus arbitration priority than the CPU, so the CPU is stalled while DMA accesses occur. This problem is most apparent with high-resolution, high-color displays which use main memory for the frame buffer.

Certain peripheral designs can also seriously damage real-time performance. Peripherals that generate very frequent interrupts to the processor, generally as a result of inadequate internal buffering or lack of DMA capability, can be a particular problem. The archetypal example of this is a USB controller with no buffering or DMA. Running flat out at 12 Mbit/sec this will generate an interrupt every 40 µs (one per 64-byte USB packet). Such a high interrupt rate will use a large proportion of the processor bandwidth. Even trying to shorten the ISRs by deferring work to a thread will cause problems, unless the processor is very fast, because the two thread switches (to and from the thread handling USB) every 40 µs will use an unacceptable percentage of processor time and will result in degradation of the USB data transfer rate. Ideally a peripheral delivering data at this rate would either buffer the data internally or (more likely) use DMA to transfer data to or from memory without needing processor intervention. In either case the rate of interrupts would be greatly reduced.

Software limitations

The most obvious limitation on real-time software under Symbian OS is that, with the exception of the bounded kernel services, most Symbian APIs do not have bounded execution times. Non-kernel APIs such as file system access and other system servers generally do not provide any real-time guarantees. One (but not the only) reason for this is that most of them rely on unbounded kernel services, most notably memory allocation and freeing.

When you are writing real-time software to run under EKA2, the first thing to bear in mind is that the standard dynamic memory allocation primitives do not provide any real-time guarantees. There are two main reasons for this:

Firstly, the default algorithm used to manage an application's heap memory (the `RHeap` class) is address-ordered first fit, using a simple linked list of free cells. It may need to search many free cells to find one that is capable of satisfying an allocation request, or to find the correct place in which to insert a cell that is being freed.

Secondly, if there is no free cell large enough to satisfy a request, the algorithm requests more memory from the global free page pool. Even though a request for a single page can be completed in a known time, the pool is protected by a mutex which could be held for a long period if another thread is also performing memory allocation or freeing. This means that accesses to the global pool cannot be performed in a known time.

There are two main techniques for avoiding this problem. The first is to avoid dynamic memory management altogether in time-critical sections of code. Instead you should allocate all memory blocks at initialization time and free them when the application or operation terminates.

The second technique is to replace the standard `RHeap` allocator with a custom allocator designed to offer predictable execution times - for example an allocator based on a small number of fixed block sizes. The allocator will need to be seeded with a fixed amount of memory from the global pool when the real-time application initializes. Why didn't Symbian provide such an allocator? The address-ordered first-fit algorithm used by the standard Symbian OS `RHeap` class is a good general purpose algorithm that provides low memory overhead and acceptable performance for most applications without making any assumptions about the size and pattern of allocations made by the application. Custom allocators can take advantage of their knowledge of the particular application involved, especially of the sizes of allocations made, and can give real-time guarantees without compromising on space efficiency. Alternatively, they may trade space efficiency for bounded execution time. The custom allocator approach has the advantage that any standard library functions used in the same thread or process also use the predictable algorithms. Of course this doesn't help with allocations occurring in the kernel or in server processes; these must simply be avoided in time critical code.

Another fundamental limitation of Symbian OS for real-time software is that it is an open OS. Subject to the restrictions of platform

security, any code, even that which is written long after the mobile phone was designed and built, may be loaded onto the phone and executed. There is no way for programs to declare real-time deadlines to the OS, and no way for the OS to make use of such information. The user could run many applications, all requiring real-time guarantees, and there is no way for the OS to indicate that it cannot provide the required guarantees to all the applications. In the strictest sense, real-time guarantees can only be given to code supplied with the mobile phone by the manufacturer, and even then only if aftermarket applications are restricted to the lower levels of the thread priority spectrum.

As well as ensuring that your code runs quickly enough to deliver acceptable results without consuming too much power, it's also essential to ensure that it doesn't damage system response times. There are several places where latency can be measured, and at all of these it can prove a problem. These locations are in interrupt service routines (ISRs), delayed function calls (DFCs and IDFCs) and user threads, in order of increasing response time (illustrated in Figure 18.1). The vast bulk of code in the system runs in one of these three environments, so it's very important not to slow down any of them unduly. The basic problem is how to enforce and maintain scheduling priority over short timescales? This is the *priority inversion* problem and hinges on one question: how long can an urgent process, triggered by some event that is not directly associated with program execution, be delayed because the system is busy with a less urgent task?

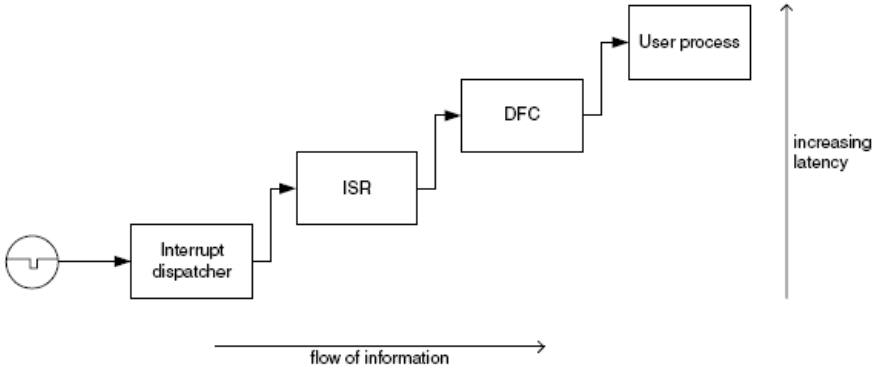

Figure 18.1 *Order of events*

Various locks, mutexes and processes in the system contribute to these delays. The *worst-case latency* is determined by several variables:

- The length of time for which preemption is disabled
- The length of time for which interrupts are disabled
- The length of time for which each mutex in the system is held
- The length of time it takes to change to the higher-priority process.

For a detailed explanation of latencies please refer to Chapter 17, *Real Time.*

Reducing ISR latency

The interrupt service routine is the second piece of code to run after a hardware interrupt has been asserted. The first is the interrupt vector handler, which has one job - to call the appropriate ISR as quickly as possible. During the ISR, you can do a little work, or you can queue a DFC to perform a larger job later when the kernel is less busy, or you can do both. ISRs must execute quickly because, in systems not supporting nested interrupts, no interrupt can occur during their execution; the longer they run, the higher the system's overall latency. If you need to do a lot of work in response to an interrupt, you should use a DFC if at all possible. Since ISRs are the first code that executes in response to a hardware event, ISR latency is critical to the system's real-time performance. It must be kept as low as possible. To ensure that interrupt latency is kept low, you must not mask interrupts for more than a very few instructions if you can possibly avoid doing so. Code that was acceptable in a non-real-time EKA1 environment often masked interrupts for long periods of time while servicing peripherals. This is no longer acceptable; you must find a better method.

Some systems support *nested interrupts*. Nested interrupts allow interrupts to occur during the execution of an ISR, by switching out of IRQ mode while the ISR runs. This can help to reduce latency at the ISR level. However, since non-ISR code cannot execute until the last ISR completes, using nested interrupts will not help to improve DFC or user thread latency. I discuss nested interrupts in more detail in Chapter 6, *Interrupts and Exceptions*.

If your system needs interrupt latency to be even lower than the IRQ mechanism can provide, you may be able to use an FIQ (Fast Interrupt reQuest) instead, to provide extremely rapid service for a very few events. FIQs take advantage of a processor-supported very fast context switch mechanism to provide very low assertion-to-ISR latency. EKA2 disables both FIQ and IRQ interrupts in some places, however, so the shortest possible worst-case latency dictated by the hardware can never be realized. You can only use FIQs if your hardware supports them. FIQ interrupts take priority over IRQ interrupts, and can execute their service routines

during ISR execution. FIQs should only be used in cases of dire necessity and their service routines must complete as quickly as possible, as they cannot be interrupted by anything else. The FIQ dispatcher is part of the ASSP, and should always be written in assembly language. Because FIQs are supported by a large number of banked registers, FIQ service routines have a lower overhead than IRQ service routines. They do not need to use the stack as much as an IRQ service routine, which means they may execute more quickly and pollute the data cache less. The FIQ banked registers are useful as work registers, but they can't be used for FIQ-to-FIQ persistence unless FIQs never queue IDFCs or DFCs directly, only doing it via an IRQ. The FIQ postamble uses the banked registers as work registers.

I discuss interrupts in general, nested interrupts and FIQs more fully in Chapter 6, *Interrupts and Exceptions.*

Reducing DFC latency

DFCs are the basic tool for doing anything in response to an interrupt. In a real-time environment, there is no time for the interrupt preamble to prepare the system to offer a full and flexible execution environment for the interrupt service routine to run in. Because at the time of an interrupt the kernel could be in almost any state, it's not possible to access kernel data structures freely during an ISR. Instead, EKA2 offers DFCs and IDFCs. These provide a method for your interrupt handler to get work done after the system exits interrupt mode. In addition to allowing interrupt handlers to actually get work done, DFCs also encourage you to keep interrupt routines short, which is a very good thing indeed to do.

Since so much work has to be done in DFCs, their latency is as important as interrupt latency. While the similarity in names between IDFCs and DFCs might lead you to expect that they are different varieties of the same thing, they are actually completely different. They are usually treated as quite separate entities, because they behave in very different ways. Because IDFCs are invoked with the kernel lock equal to 1, there are some restrictions on which kernel functions they may interact with. IDFCs are queued by ISRs and will run immediately after the kernel finishes interrupt dispatch. If you need to do some work to finish servicing an interrupt, then an IDFC is an obvious place to do it. However, you should be warned that time spent in an IDFC will increase DFC and user thread latency. For this reason, you should use IDFCs sparingly and with care. in fact, we only recommend the use of IDFCs for the implementation of personality layers and when an ISR needs to make a thread other than a DFC thread ready. IDFCs run in the order in which they were queued, so there can be a delay if other IDFCs are pending. Of course, ISRs take priority over IDFCs and will run before them.

So, if we don't recommend IDFCs, how about DFCs? Although they do run later, a high-priority DFC will run almost as soon as the last IDFC finishes. The only difference between a priority 63 DFC and an IDFC is the time taken for the scheduler to run and switch to the DFC thread, which should be of the order of two microseconds. No MMU manipulation or cache reload penalty is involved, since DFCs run in a kernel thread.

DMA and its impact on latency

DMA has both advantages and disadvantages over programmed I/O. A DMA controller is an essential part of most phone platforms, as nearly all of them include DMA-driven display controllers. These display controllers perform a regular burst of memory accesses every few microseconds in order to feed a line buffer in the LCD panel driver. Because the display hardware's need for pixel data is constrained by the physics of the LCD panel, the DMA that fills it must have a very high priority if the display is not to be disrupted by other system activity. Because the display DMA will have higher priority than the microprocessor core, it is likely that on occasion the processor will be effectively stalled while the display DMA completes.

While there is nothing that can be done to alleviate this particular problem, it's important to bear it in mind when designing systems that use DMA while relying upon the low latency of the rest of the system. If your DMA controller supports multiple priority levels, you should select the lowest priority level that allows your driver to work appropriately. If your DMA needs to be of a very high priority, or your DMA controller does not support multiple priority levels, make sure the transfers are short or, at least, that the latency they will introduce is acceptable to your system.

On the other hand, DMA controllers can transfer data very quickly without polluting the CPU cache. By leaving the CPU cache populated, latency to the ISRs and exception vectors can be significantly improved compared with a software copy - improvements of the order of hundreds of cycles are not uncommon.

Priority inversion

Another source of DFC and user thread latency is priority inversion caused by disabling preemption. Anything that disables interrupts or holds certain locks (for example, the preemption lock) too long will prevent the scheduler from running when it ought to, causing a priority inversion (that is, a thread that should have been suspended will continue to run while the higher priority thread that should run in its place will remain suspended). IDFCs and DFCs will not run until the scheduler is able to run. From the point of view of any thread that needs the processor, this appears as if the scheduler was slow in returning control. If you can possibly avoid it, do not claim any system locks. If you must claim them, use them as briefly as possible to reduce priority inversion time. The preemption lock (also known as the kernel lock) and the system lock are particularly important in this respect,

as they stop important scheduler processes from proceeding. Other kernel locks are sometimes held for long periods in other places, but they should still be used judiciously.

Unless you are implementing a personality layer, there are no points where the board support package will be entered with the kernel locked. The personality layer N-state extension hooks are the only place where the kernel calls out to the BSP while it is locked. If you are implementing such functions, be sure they run quickly.

As I explained earlier, IDFCs also run with the kernel locked.

DFCs and DFC threads

As I explained in Chapter 6, *Interrupts and Exceptions*, DFCs have one of eight priorities (between 0 and 7) and belong to a DFC thread. Higher priority threads obviously run their DFCs before lower priority threads, and within one thread, the kernel runs higher priority DFCs before lower priority ones. This means that you need to choose your DFC thread and priority carefully to ensure that your DFC does not prevent more important DFCs from running when they need to. Since many services will use DFCs, there is the potential for wide-ranging performance impact. If you have created your own DFC thread, it may be acceptable for your DFCs to block, run indefinitely or otherwise not terminate for some time. But be aware that while a DFC is running, no other DFCs in the same thread will receive a share of the processor. It is completely unacceptable to place blocking or slow DFCs in system DFC threads as this will severely damage performance. If you have multiple drivers provided as a unit, for example in a base port, it may be desirable to create a DFC thread and share it between all the drivers. Be aware that if any of your DFCs block, there could be interactions between otherwise separate drivers that may not be predictable and will probably not be desirable.

Reducing user thread latency

Neither DFCs nor user threads can be scheduled while the preemption (kernel) lock is held. The system lock prevents user-side memory from being remapped, so must be claimed before memory copies between user-side processes, or between kernel-side and user-side memory take place; otherwise an interrupt or a reschedule could occur during the copy and cause problems by removing memory that was allocated and available at the start of the copy. For this reason, it's essential to avoid large memory copy operations. In fact, IPC copies are paused every 512 bytes to allow the system lock to be checked for contention. If contention is detected, the system lock is released and relocked, allowing the contending thread to claim the system lock and run in the intervening period. If you need to copy large areas of user-side memory around from kernel-side routines, it is important to bear the duration of the system lock in mind. It might even be worth thinking about redesigning your architecture to use a shared chunk instead of copying.

Mutual exclusion techniques

When code runs in an environment in which it may be interrupted, some sections of that code may need to be *protected* from being executed out of order, from being re-entered, or from their data structures being modified *behind their backs*. The most common way of doing this is with a mutex. Obviously, since mutexes revolve around preventing code from running until it is convenient for it to do so, this can have some impact on latency, and so care is needed to avoid adverse consequences.

Disabling interrupts: Since multitasking in the system is based on interrupts, a very lightweight way to achieve exclusion from re-entry for a very short sequence of code is to disable interrupts before it, and re-enable them afterwards. This is almost never an appropriate thing to do for long pieces of code, and may break in multiprocessor environments (for example, environments like OMAP where the DSP may be running concurrently to the ARM processor) or in cases where a call during that code causes a reschedule. Obviously, since no interrupts can be serviced while interrupts are disabled, this strategy has the potential to significantly increase worst-case interrupt latency. You should only ever disable interrupts for a very few instructions, and you should consider 30 µs with interrupts disabled the absolute maximum. For example, the kernel queue handling functions disable interrupts only for the innermost critical portion of their execution. Some kernel functions require interrupts to be enabled on entry, so this strategy is not suitable for driver operations that may interact with them.

Writing your code so that it doesn't need to be locked (a lockless implementation) is another possibility: Lockless implementation isn't really a mutual exclusion method, but I've included it here because it falls into the same general category of techniques for ensuring consistency in concurrent code.

Disabling preemption: If your code does not need protecting from the effects of interrupt service routines themselves, it may not be necessary to disable all interrupts. Merely holding the kernel lock is enough to prevent rescheduling from taking place. This is enough to prevent another thread from interfering with your protected code, and has the advantage that it does not increase interrupt service latency at all. However, DFCs will not run until the kernel lock is released, and, as we noted earlier, holding the kernel lock constitutes a potential priority inversion, so you must not hold it for long: 30 µs should be considered a maximum.

Using a fast mutex: The nanokernel provides a very fast special case of mutex. The interface to fast mutexes is the class `NFastMutex`, this class is exposed via `NKern::` methods. Fast mutexes have a few limitations on their use. For example, waiting on

a fast mutex constitutes blocking, so you cannot do this in an IDFC or ISR. You also must release fast mutexes before your thread attempts to terminate or block. Fast mutexes can't be nested, either. These limitations are more fully explored in Chapter 3, *Threads, Processes* and Libraries. The nanokernel will ensure that threads are not suspended while they hold a fast mutex. It will treat threads that hold a fast mutex as if they are in a critical section and will not suspend them while they hold the mutex.

Using a conventional mutex: The Symbian OS layer provides various sophisticated synchronization objects, including a fully featured mutex, a condition variable object and a counting semaphore. Symbian OS semaphores support multiple waiting threads (unlike nanokernel fast semaphores) and release waiting threads in priority order. As I discussed in Chapter 3, Threads, Processes and Libraries, Symbian OS mutexes are fully nestable and they support priority inheritance. Symbian OS mutexes are covered in Chapter 3, *Threads, Processes and Libraries*. The kernel and memory model use Symbian OS mutexes extensively to protect long-running critical code sections. However, these wait objects are comparatively heavyweight and the overhead of using them may not be worth the convenience. What's more, since they are implemented by the Symbian OS layer, they can't be used in code that runs at nanokernel level. If you need to block or wait while holding a mutex, or you may have more than one waiting thread, you must use a Symbian OS mutex. If you will hold the mutex for longer than 20 ms, you should consider using a Symbian OS mutex in any case, as the overall overhead will be small when the frequency with which the mutex code will run is considered.

Using a DFC queue thread: Because the DFCs in a DFC queue thread will be called in priority order, they can provide a useful measure of mutual exclusion. The DFCs in a single queue will never execute concurrently. This means that the queuing mechanism guarantees that any one DFC can never interrupt another in the same thread. This can provide all the protection you need to manage some forms of buffer.

Memory allocation

Nanokernel threads, IDFCs and interrupt service routines cannot use the standard Symbian OS memory allocation APIs, since they obviously contain internal state, and this may be inconsistent when the relevant mutex is not held. ISRs and IDFCs cannot block, as described in Chapter 6, and therefore they cannot wait on a mutex. Since this means that they cannot claim the mutex, they can't guarantee allocator consistency and therefore cannot claim memory. As we've seen, the nanokernel does not contain any code relating to memory allocation and depends on the Symbian OS layer to allocate memory for it.

The `RHeap` allocator conducts a linear search through the array describing the cells in the arena it allocates from, looking for free cells; when it finds a free cell it then has to check if the cell is large enough to contain the request. This process is memory-efficient but not particularly fast and may not complete in a reasonable length of time. Also, if interrupts could allocate memory, the allocator would have to be re-entrant. The design of a re-entrant allocator is extremely difficult and would significantly complicate the memory system's overall design. Therefore these methods are not available to any real-time process.

If you're trying to write code to run quickly, you may choose to avoid allocating memory during sections that must run quickly. There are several approaches to this problem.

Avoid allocating memory at all: Running entirely from stack is fairly efficient, and is an option available everywhere in the system. Even interrupt routines have stack, but stack space under Symbian OS is very limited, and of course stack-based (*automatic*) variables are local, so they may not be useful as structures to be passed around by reference. Different contexts have different stacks. Be aware that the stack limit under Windows (that is, the emulator) is not the same as it is in a real device. Windows has a larger stack limit than the real devices do, so code may fail to run on real devices even if it works on the emulator.

Allocate statically: If you allocate a single block of heap memory when your program starts, you can sub-allocate parts of it by defining a data structure containing all the necessary objects. This approach is extremely efficient and has almost no overhead, as no allocation process happens at run time. Many video applications use this approach, and it works well, but it is very inflexible. While it can easily become unmanageable, it has the advantage that it makes it easy to integrate C++ with assembly language routines, since all the offsets from the start of the allocated block are known at compile time and can be included statically in the source.

Write your own allocator: This is an approach similar to static allocation, but more flexible. You allocate a large block of memory at startup time and then use a simple allocator routine to distribute it. There are many high-performance memory allocation routines discussed in the literature; some work by using a hash table, thereby reducing the general case time needed for an allocation, and others use fixed-length cells, simplifying the search. Some high-performance network protocol implementations use both fixed-length cells and hash tables. These methods are extremely effective in reducing time taken to allocate memory. In Symbian OS you may even choose to replace `RHeap`'s allocator for your application, so that all allocations will benefit from your faster algorithm. This is discussed in Section 3.3.1.3.

Performance in the emulator

The EKA2 emulator provides a few ways to help application developers simulate the performance of a real mobile phone. Only one of these, the emulation LFFS media driver, is of production quality, but the others are there to try if you think that they might

help.

Media performance

We provide a way to change the general performance of the file system. You can inject small, not-too-noticeable delays into the file system code, by setting the *DiskRead* and *DiskWrite* values in the emulator initialization file, epoc.ini. These values specify the read and write speed to emulate in units of microseconds/KB, and we act on them by injecting delays into file system calls using the millisecond timer. This is still at the experimental stage, and so is rather approximate.

You can change the performance of the LFFS media driver in a more robust way. The [emulation] LFFS media driver, which uses a memory mapped file for the flash image, can be parameterized by settings in epoc.ini to have performance similar to most NOR flash hardware. Coupled with the LFFS file system, this gives quite realistic performance for this file system on a PC.

CPU performance

The HalData::ECpuSpeed HAL item is read/write on the emulator, which means that by writing a low value to it, you can slow down the speed of the emulated CPU. We act on this item by making the emulator timer system consume some fraction of the CPU time when the emulator is busy.

Summary

In this chapter I have discussed techniques that will help you to improve the performance of your software. However, all the tweaks and tricks in the world are no substitute for a good, efficient design. In a deeply object-oriented environment it is easy to lose sight of what is really happening to your processor and memory. I hope that this chapter will have made the underlying processes a little more familiar to you.

Techniques such as unrolling and flattening recursive routines into iterative ones can improve the performance of an algorithm, but if that algorithm is inherently suboptimal, it's better to avoid it. A little care during the design phase of your project can save a lot of scraping for cycles later on.

Symbian OS mobile phones are getting faster and faster every year, and there is no end to this scaling in sight. The software stacks that run on them are getting larger and more powerful, and users are expecting more and more from their phones. EKA2 was designed to provide capabilities for the next generation of devices, smaller, faster and cheaper than ever. In the rich, real-time environment it provides, performance is more important than ever. Every cycle consumes a few nanojoules, and every nanojoule has to come out of a battery that is more compact than the one fitted to the last model. Cycles are an endangered species; treat them like the precious commodity they are.