Symbian OS Internals/17. Real Time

- Symbian OS Internals Table of Contents

by Dennis May

If I could explain it to the average person, I wouldn't have been worth the Nobel Prize.
Richard Feynman

In this chapter, I will discuss real time systems, the support that operating systems must provide to enable them and how we designed EKA2 to support them.

What is real time?

The term *real time* is often misunderstood. The canonical definition of a *real time system* is the following, taken from the Usenet group comp.realtime (see FAQ posting on comp.realtime) and credited to Donald Gillies:

A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred. Essentially, a real time system has a set of tasks which it must perform, and at least one of those tasks has a deadline by which it must complete otherwise the system will fail to operate correctly. A system may perform both real time and non-real time tasks; the former are the ones with strict deadlines.

Some examples are useful at this point:

1. An engine management system has several tasks that need to be done on each revolution of the engine, such as operating the fuel injectors and triggering the ignition spark. If these tasks do not occur at the correct times, the engine will either perform badly or not function at all. To get an idea of how much accuracy is required, a typical engine has a maximum working angular velocity of 6000 rpm, or 100 revolutions per second. For optimum performance, the timing needs to be adjusted to within 1% of a revolution, so this equates to a timing margin of less than 100 µs
2. Most communication protocols include timeouts to guard against the possibility of data being lost in transit through an unreliable network or transmission medium. If a computer system running one of these protocols does not respond in time, the peer will assume the data has been lost and will retransmit it unnecessarily or possibly even terminate the connection. Typical values of the timeouts range from a few milliseconds to several minutes depending on the protocol concerned.

A common misapprehension about the term real time is that it implies that the system must operate very quickly. This is not necessarily the case - a deadline may be known about well in advance but still be critical. A well-known example here is the problem of ensuring that servers were year 2000 compliant before that date. Another example is the second one I just gave, in which the deadline may be of the order of seconds or even minutes, which is for ever for a computer!

Hard and soft real time

In Section 17.1 I give a black and white definition of real time. In reality a lot of systems have varying degrees of tolerance to missed deadlines. This leads to the following definitions:

A **hard** real time system or task is one in which a missed deadline causes complete system failure, as described in Section 17.1.

A **soft** real time system or task is one in which a missed deadline causes a reduction in system performance. Typically, the greater the margin by which the deadline is missed, the greater is the reduction in performance, and for some value of the margin the system has failed.

Examples of soft real time systems include:

1. A user interface responding to input from the user. The objective here is to provide apparently instant response. Research indicates that the user will perceive a response within 100 ms as instantaneous, and so the *deadline* is 100 ms from the user's input. However if the response takes 200 ms, no one is likely to mind. If it takes a second, the system could appear sluggish and if it takes more than a minute with no response at all, the user will probably think the device is broken
2. In most cases the communication protocol example in Section 17.1 is actually a soft real time task. If the deadline is missed, the peer entity will retransmit, which will waste bandwidth and reduce throughput but will not cause catastrophic failure

Real time operating systems

We have given definitions of real time systems and tasks. An operating system cannot be real time in the sense that I have just discussed. Instead the term *real time operating system* (usually abbreviated to RTOS) refers to an operating system that is

(C) Copyright Nokia 2014. All rights reserved.

capable of running real time applications. In this section I will discuss some of the properties that an RTOS should have.

Tasks and scheduling

A real time system will generally consist of several logically separate but interacting tasks each with their own deadline. It may also contain non-real time (or soft real time) tasks. The most important task for any RTOS is to schedule the tasks for execution on the processor in such a way that the deadlines of the real time tasks are always met. In analyzing whether this is possible, the following simplified model of task execution is often used. Consider that the system has a set of *n* tasks, denoted *t*1, . . . , t*n.* Each task is assumed to run periodically. Task *ti* is triggered every T*i* seconds, runs for a maximum time C*i* and is expected to finish within time D*i* of being triggered. In reality tasks will have a variable execution time, but to ensure that deadlines can be met the worst case execution time (WCET) is assumed. Similarly, aperiodic tasks can be incorporated by treating them as periodic with period equal to the minimum possible time between invocations of the task. We define the response time (R*i*) of a task *ti* to be the time from the task being triggered to the task completing. Figure 17.1 illustrates these task-timing parameters.

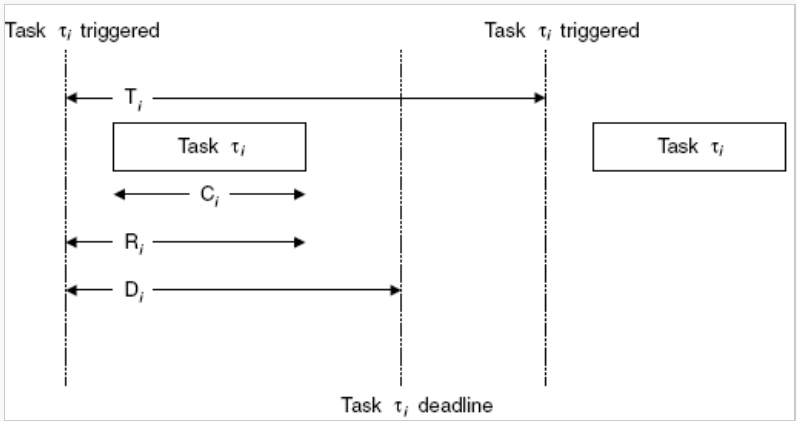

Figure 17.1 *Task timing parameters*

Cyclic scheduling

The simplest means of ensuring that tasks all run when necessary is cyclic scheduling. It does not require interrupts and is the only method available on systems without interrupts. This method is often used in simple deeply embedded systems such as engine management, and would look something like this:

```
void main()
    {
    wait_for_next_tick();
    do_task1();
    do_task2();
    do_task3();
    }
```

This shows three tasks, all with the same period. To handle tasks with periods which are multiples of the same lowest period, the scheduler loop or task functions can be modified so that they only execute the task every m ticks for some value of m. Alternatively, for more flexibility, a major/minor cycle scheme may be employed where the system period is the least common multiple of the periods of all tasks and within each period each task executes several times. In both schemes, problems will be encountered fitting tasks into the schedule. Consider the following two tasks:

- Task 1 runs for 1 ms every 2 ms, deadline 2 ms
- Task 2 runs for 1.5 ms every 4 ms, deadline 4 ms.

It is not possible to execute task 2 continuously for 1.5 ms since then task 1 would miss its deadline. Instead, task 2 must be divided into two pieces, each of which has an execution time _<_1 ms. These pieces can then be fitted into the *gaps* between invocations of task 1. So you end up with:

```
void main()
    {
```


(C) Copyright Nokia 2014. All rights reserved.

```
    wait_for_next_tick();
    ++tick_count;
    do_task1();
    if (tick_count & 1)
        do_task2_second_half();
    else
        do_task2_first_half();
    }
```

Figure 17.2 illustrates task execution under this scheduling scheme.

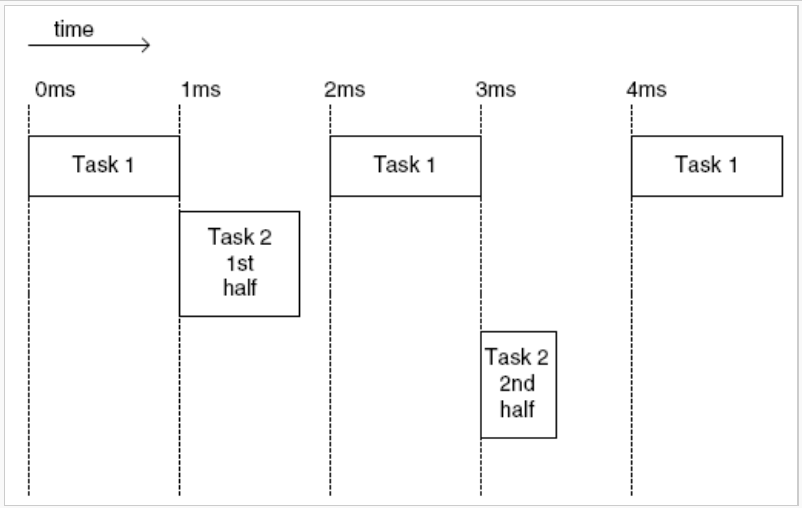

Figure 17.2

The partitioning of task 2 must be done by hand and the code must be modified to preserve any necessary state in global data, so that it can be communicated from the first to the second half. Cyclic scheduling has some advantages - it can be implemented on very small systems without interrupts, there is no preemption to worry about, and it is guaranteed to meet all deadlines provided the schedule has been correctly calculated. However it also has major disadvantages:

- It has problems dealing with periods that are not multiples of the smallest period in the system. Tasks with periods that are not simple multiples of one another will give rise to very complicated schedules. To work round this limitation, tasks often end up being run more frequently than required, which wastes processor time
- Adding a new task to the system is difficult. The new task will probably need to be manually partitioned into several pieces with lots of state communicated between the pieces using global data
- Maintenance is difficult. An increase in the execution time of any of the task pieces may require a complete rethink of the cyclic schedule, even if that increase does not actually overload the processor
- Sporadic events are handled badly. They must be polled for, and time must be budgeted for them in every cycle even though in most cases that time is not required. This leads to poor processor utilization.

Preemptive scheduling

Cyclic scheduling is a completely co-operative scheduling method - there is no preemption. Once a task begins execution, it must complete before any other task can gain control of the processor - and this is the root of the major problems with cyclic scheduling. To overcome these problems, preemptive scheduling is used. A preemptive system has the following characteristics:

- Interrupts are used to signal external events. An interrupt will have some way, direct or indirect, of triggering a task
- The tasks in the system are ordered in some way; for real time systems, they are generally ordered by how urgent they are. This ordering may be constant or it may change with time depending on circumstances
- When a task is triggered, its urgency is compared with that of the currently executing task. If the new task is more urgent, the current task is suspended and the new task begins execution immediately. If the new task is less urgent, it is simply placed on a list of pending tasks - the ready list
- When a task completes execution or waits for an external event to occur, it is suspended and control is passed to the most urgent task on the ready list.

Note that when control is passed from one task to another, the operating system automatically saves the state of the old task so that it can resume it later. Thus the operating system effectively does the partitioning of tasks into smaller pieces for you and so tasks can simply be written as a single unit. Adding a new task is simple; all that you have to do is to assign it a priority and check that the system can still be scheduled correctly with the new task included (in the sense that each task completes within its

(C) Copyright Nokia 2014. All rights reserved.

deadline). Similarly, an increase in the execution time of any task doesn't require any extra work rearranging the schedule, just a check that the system can still be scheduled correctly. Task periods don't have to be integer multiples of each other - each task simply runs when it needs to. Similarly, sporadic tasks are not a problem - they only use processor time when they actually run. Preemptive scheduling also deals with non-real time tasks; these are simply placed at the bottom of the ordering and so they run whenever no time critical tasks need the processor.

Preemptive scheduling solves the major problems of cyclic scheduling. However it brings some problems of its own:

- Resource sharing. Since tasks are started and stopped automatically by the OS it is possible that they will contend for use of a shared resource
- It is more difficult to check that a set of tasks can be scheduled correctly.

Static priority based scheduling

This is the most common form of preemptive scheduling, at least among real time operating systems. Each task is assigned a fixed priority when it is created, and these priorities determine which tasks preempt which other tasks. The actual priority values used are derived using a very simple rule from the deadlines of the various tasks. The tasks are placed in increasing deadline order - that is, the task with the shortest deadline has the highest priority and so on. This is known as *deadline monotonic scheduling*.

This result was proved in the well-known paper by Liu and Layland (Liu, C.L. and J.W. Layland, *Scheduling algorithms for multi-programming in a hard real-time environment*, *Journal of the Association of Computer Machinery* (ACM) **20**(1), Jan. 1973, pp. 46-61) for the case where $D_i = T_i$ and the more general result for $D_i < T_i$ was proved by Leung and Whitehead (Leung, J.Y.T. and J. Whitehead, *On the complexity of fixed-prior*

Predictability

The results on scheduling algorithms that I referred to previously make certain assumptions about how the system operates. Chief among these are the assumptions that task switching takes zero time and that a higher priority task is never prevented from running by a lower priority task. If the latter situation does occur, it is termed priority inversion. There is also the implicit assumption that the execution time of each task is known, or at least that it is bounded.

OS services

The time taken for task switching can be accounted for by assuming that a task switch takes place at the beginning and end of each task, so adding twice the context switch time to each task's execution time gives an estimate (actually pessimistic) for the effect of context switching time. (To see why two context switch times are added, consider tasks A and B. Suppose A is part way through running when B becomes ready and preempts A. There is a context switch from A to B and then another from B to A when B completes. So the total delay to task A is the execution time of task B plus twice the context switch time - in other words, as if task B had twice the context switch time added to its execution time. On the other hand, when the end of one task and the beginning of another coincide then there is only one context switch to be added. This is why the estimate obtained is pessimistic.)

To obtain predictable task execution times, the execution time of any operating system services used by that task must be predictable. The time taken by the operating system to switch tasks must also be predictable. Thus we arrive at our first requirement for an RTOS - task switching and OS services must have predictable, or at least well bounded, execution times. Obviously to maximize the proportion of processor time spent doing useful work these overheads should be as small as possible. However in a choice between an algorithm with a constant execution time and one with an execution time which is usually small but may occasionally be very large, the former should be chosen even though it may be slower on average.

Interrupts

Consider now the effect of hardware interrupts on the system. These may be considered as tasks with a very high priority - higher than any software-defined tasks. The tasks with the very shortest deadlines will be implemented as interrupt service routines. However, not all interrupts are equally urgent. An interrupt from a keyboard controller can probably wait 100 ms or so without any problem, whereas a receive interrupt from a 16550 UART must be serviced within 700 µs or data may be lost. In particular, some interrupts may have longer deadlines than some software defined tasks. According to the deadline monotonic scheduling rule, these interrupts should have lower priority than the tasks. However, the hardware will automatically preempt any task whenever an interrupt is signaled. There are three possible ways around this problem:

- Use a periodic task to poll for the long deadline interrupt at an appropriate priority. This has the disadvantage that the task uses processor time and consumes energy even when the interrupt is inactive
- Modify the interrupt masks of all peripherals in the system on every task switch so that certain tasks can mask certain interrupts. This approach is probably the best in principle but is complicated to implement unless the hardware is specifically designed to support it. Some processors support prioritized interrupts - typically 7 or 15 levels. It would be simple to allow the

(C) Copyright Nokia 2014. All rights reserved.

processor interrupt mask level to be modified by the scheduler on a task switch and then to assign the interrupts with long deadlines a low hardware priority. However other hardware does not have such a prioritization scheme and so would require many peripheral registers to be modified on each task switch, which would severely impact the efficiency of task switching
- Make the service routine for the long deadline interrupt as short as possible to minimize its impact on the rest of the system. In fact all it needs to do is to clear the peripheral's interrupt condition and then trigger a software task to do the real processing at an appropriate priority.

EKA2 employs the third option to solve this problem.

Mutual exclusion

In any preemptive system, there must be a means for a task to gain exclusive access to system resources for a time. Without such a mechanism, two or more tasks accessing the same resource will interact in an unpredictable manner. For example, consider two tasks both incrementing the same variable. On the ARM processor this involves executing the following sequence of instructions:

LDR R1, [R0] ; load register R1 with variable

ADD R1, R1, #1 ; add 1 to register R1

STR R1, [R0] ; update variable with new value.

If two tasks both execute this sequence and the first task executes the first instruction of the sequence before the second task gains control, the sequence of instructions executed is:

(task 1) load variable into R1

(task 2) load variable into R1

(task 2) add 1 to R1

(task 2) store R1 into variable

(task 1) add 1 to R1

(task 1) store R1 into variable.

It can be seen that the effect of this sequence is to increment the variable only once instead of twice.

The measures commonly employed to prevent such contention take various forms, listed next. They all amount to ways to specify a group of code fragments and to ensure that at any time at most one thread of execution is present in the group. This property is called *mutual exclusion*. Ways of achieving mutual exclusion include:

- Use a single atomic instruction. This is the most efficient where possible but is limited in that only certain operations are possible. On ARM architecture 4 and 5 processors, the only atomic instruction is to swap the contents of memory with a register
- Disabling hardware interrupts. On a single processor system this ensures that no other task can gain control of the processor. On a multiple processor system spin locks must be used in conjunction with disabling local interrupts to prevent a task running on another processor from entering an excluded code section. Virtually every OS disables interrupts in some places since, other than atomic instructions, it is the only way of protecting resources accessed by both interrupts and software tasks. In particular, managing the interaction of interrupts with software tasks generally requires disabling interrupts
- Disabling preemption. Similar to disabling interrupts, but only protects against other software tasks, not against hardware interrupts. Both disabling interrupts and disabling preemption are fairly drastic measures since they delay all other tasks, not just those that might contend for a particular resource
- Use an OS-provided synchronization object, such as a semaphore or mutex. These have no effect on scheduling of tasks unless the tasks contend for the protected resource.

Any method of mutual exclusion is a source of priority inversion since a high priority task may be held up waiting for a resource, which is currently locked by a lower priority task. The schedulability analysis referred to in Section 17.2.1 can be extended to account for mutual exclusion provided that the time for which priority inversion occurs is bounded. Thus we must ensure that:

- The time for which interrupts are disabled is bounded. In fact, to produce an OS of the greatest utility, this time must be minimized, since the maximum time for which interrupts are disabled gives a lower bound on the guaranteed response time for an external event
- The time for which preemption is disabled is minimized. This time, along with the time taken by all ISRs in the system, determines the fastest possible response time by a software task
- Mutual exclusion locks are held for a bounded time
- The OS provides a mutual exclusion primitive that minimizes priority inversion.

(C) Copyright Nokia 2014. All rights reserved.

The last point here requires some explanation. Consider a standard counting semaphore used to protect a resource and consider two tasks*t*1 and *t*2 which both require access to the resource. Suppose *t*1 has a higher priority than *t*2 and that *t*2 holds the semaphore. Task *t*1 is then triggered, attempts to acquire the semaphore, and is blocked, leaving *t*2 running. So far all is well, but now consider what happens if a third task *t*3 with priority intermediate between that of *t*1 and *t*2 begins running. Now *t*2 cannot run and so it cannot exit the locked section and release the semaphore at least until *t*3 finishes, and *t*1 cannot resume until *t*2 releases the semaphore. So effectively the higher priority task *t*1 cannot run until the lower priority task *t*3 finishes, even though there is no contention for resources between them. You can see that *t*1 could be delayed for a very long time through this mechanism. On a system with non-real time tasks as well as real time tasks, it could be delayed indefinitely. This scenario is the classic unbounded priority inversion scenario that caused problems with the 1997 Mars Pathfinder mission.

There are two main ways to avoid this problem. Both involve the operating system providing a special object, superficially similar to a semaphore, but designed specifically for mutual exclusion applications. Such an object is usually called a mutex.

The Mars Pathfinder lander touched down on Mars on the 4th of July 1997, and initially performed well. But a few days after landing, it started experiencing total system resets, each of which lost some of the gathered data. Pathfinder contained a shared memory area used to pass information between different components of the system. A high priority bus management task ran frequently to move data in and out of the shared area. Access to the shared area was protected by mutexes, and these did not employ priority inheritance.

A low priority task ran infrequently to gather meteorological data, and used the shared area to publish that data. This task would acquire a mutex, write to the shared area and then release the mutex. If it were preempted by the bus management task, the latter could be blocked waiting for the meteorological data task to complete.

There was also a communications task, which ran with priority between those of the tasks already mentioned. This was a relatively long running task. Occasionally it would preempt the meteorological data task just at the time when it held the mutex and the bus management task was blocked on the mutex. In that case the bus management task could not run until the communications task completed. When the bus management task was delayed too long a watchdog timer triggered a system reset.

The problem was fixed by changing the mutex so that it employed priority inheritance.

Priority inheritance

Under the priority inheritance scheme, whenever a task*t*2 holds a mutex M and another task *t*1 of higher priority is blocked on M then the priority of *t*2 is raised to that of *t*1. When the task eventually releases M, its priority is returned to normal.

If the counting semaphore in the classic unbounded priority inversion scenario is replaced by a priority inheritance mutex, you can see that the problem no longer occurs. Instead, when task *t*1 attempts to acquire the mutex, it is blocked and *t*2 resumes, but now running with the priority of *t*1. Now when task *t*3 is triggered, it does not preempt *t*2, and *t*2 continues running until it reaches the end of the protected code fragment and releases the mutex. It then returns to its normal priority and task *t*1 immediately resumes and claims the mutex. So, in this case, the delay to any task wanting to claim the mutex is limited to the maximum time for which the mutex is held by any lower priority task.

Priority ceiling protocol

Under the priority ceiling protocol, each mutex has a ceiling priority, which is the maximum priority of any task that can acquire the mutex.

The algorithm is very simple. A task acquires a mutex by saving its current priority and then raising its priority to the mutex ceiling priority. It releases the mutex by restoring its priority to the value saved while acquiring the mutex. Note that we don't need to test whether the mutex is held when acquiring it - it can't be held since, if it were, the holding task would have raised its priority to at least as high as the current task and so the current task would not be running.

This scheme is very simple and efficient and also provides the most predictable blocking times. A high priority task can be delayed by at most one low priority task holding a mutex, and that delay comes all in one block before the task actually starts running. The maximum blocking time for a task *ti* is just the maximum time that any mutex with a ceiling priority higher than that of *ti* is held by any task with priority lower than that of *ti* . Once task *ti* starts running it never waits for a mutex. The priority ceiling protocol is very good for deeply embedded, closed, hard real time systems. To use it, the following conditions must be satisfied:

- Static priority based scheduling must be used, with no time slicing for tasks at the same priority
- The set of tasks which may use a particular mutex must be known at system build time
- Tasks must not block waiting for an event while holding a mutex. This would invalidate the rule that a task never needs to wait for a mutex.

You can see that none of these conditions can be guaranteed to hold for an operating system capable of loading and executing

 (C) Copyright Nokia 2014. All rights reserved.

arbitrary code at run time.

EKA2 and real time

Now that I have explored some of the problems faced by real time operating systems and looked at some of the solutions they employ, I will look at how we have attempted to solve these problems in EKA2. EKA2 was designed with several requirements that are relevant to our problem:

- It must be an open operating system; that is it must be capable of loading and executing arbitrary code at run time. This is really a corollary of the next point but is so significant that it warrants an explicit statement
- It was to be a replacement for EKA1. Hence it should have the same functionality and should be compatible with existing application level code written for EKA1
- It was to have sufficient real time capabilities to run the frame-level and multiframe-level activities of a GSM signaling stack. This includes layers 2 and 3, and part of layer 1, depending on how the hardware is arranged.

The first point eliminates many of the schemes discussed in Section 17.2: it eliminates cyclic scheduling and pure EDF scheduling, for reasons that I've mentioned previously. So, the scheduling algorithm that we selected was static priority based scheduling, with time slicing between threads at the same priority. We chose this because it is the simplest algorithm, which gives acceptable performance for both real time and non-real time applications, and because the APIs inherited from EKA1 assumed it! We set the number of priority levels at 64, because EKA1 allowed up to 26 different priorities to be explicitly set by applications, and signaling stacks require as many as 30 priorities.

The nanokernel

Once the scheduling algorithm is decided, the other requirements for a real time OS are minimizing priority inversion times and ensuring predictability of execution times for OS services. Minimizing priority inversion times entails minimizing the time for which interrupts and preemption are disabled. This is the role of the nanokernel. The nanokernel provides the most basic services of an operating system: threads, scheduling, mutual exclusion and synchronization of threads, and managing the interaction of interrupts with threads. The nanokernel is small (around 7% of the total code volume of the kernel) but it encapsulates most of the places where interrupts or preemption are disabled. This allows us to maintain tight control over these sections.

I have discussed the nanokernel in more detail in previous chapters of this book, so I will not repeat myself here. Instead I will just describe some of the design decisions that we made for the sake of real time performance.

The ready list

To ensure that we have predictable execution times for OS services, we need to be able to perform operations on the thread ready list in predictable times. The operations we need are:

- Adding a thread to the ready list
- Removing a thread from the ready list
- Finding the highest priority ready thread.

We achieve this by using 64 separate linked lists, one for each priority level. We also have a 64-bit mask, in which bit n is set if and only if the list for priority n is non-empty. This scheme eliminates the need to scan the list for the correct insertion point when adding a thread. To find the highest priority ready thread, we do a binary search on the bit mask and the result is used to index the array of 64 lists.

This same structure, referred to as a priority list in the EKA2 documentation, is used in other parts of the kernel whenever a priority ordered list is required.

See Section 3.6 for more on this subject.

Nanokernel timers

We encountered a similar problem with timers; we wanted to allow timers to be started from interrupt service routines for efficiency reasons. We couldn't use a simple delta list, because adding a timer to such a list would involve a linear time search through the list to find the insertion point. Instead we chose to use a two-level system. If the timer is due to expire in the next 32 ticks, we place it directly on one of the 32 final queues; the system tick interrupt then inspects one of these on each tick (in a round-robin fashion) and completes any timers on it. But if the timer is due to expire after 32 ticks, then we first add it to a pending queue. A thread then sorts the timers on the pending queue into an ordered queue. When a timer at the head of the ordered queue gets within 32 ticks of its expiry time, the same thread will transfer it to the relevant final queue. In this way, the starting and stopping of timers become constant time operations.

Fast mutexes

(C) Copyright Nokia 2014. All rights reserved.

One of the main purposes of the nanokernel is to localize all the non-preemptible sections of code in the system. For this to be practical, the nanokernel has to provide a very fast mutual exclusion primitive that still allows preemption of the critical section by unrelated threads. To be useful for real time applications, this mechanism also needs to provide a means of limiting priority inversion times. Because EKA2 must support an open system, the priority ceiling protocol cannot be used, and so some form of priority inheritance is needed.

We designed fast mutexes to provide the solution to this problem. A fast mutex consists of two fields - a pointer to the nanokernel thread, which currently holds the mutex (or null if the mutex is free), and a flag (the waiting flag), which indicates that some action has been deferred because the mutex was locked. To lock a fast mutex, a thread first inspects the holding thread pointer. If this is null, the thread simply claims the mutex by writing its own address into the holding thread pointer and then proceeds. This makes fast mutexes very fast in the case where there is no contention. If the mutex is held by another thread, the nanokernel sets the waiting flag and then performs an immediate context switch directly to the holding thread (which must be of lower priority than the blocked thread). The blocked thread remains on the ready list - this is not the case when blocking on other wait objects. So, this is how we get priority inheritance - because the blocked thread is still on the ready list, a reschedule will only be triggered if a thread with a higher priority than the blocked thread is made ready. Otherwise the holding thread will continue running until it releases the mutex. At that point it checks the waiting flag and, if it is set, triggers a reschedule. This will then switch to the blocked thread.

There are two restrictions on the use of fast mutexes that we impose to guarantee predictable real time behavior:

1. Fast mutexes cannot be nested
2. A thread may not block on another wait object while holding a fast mutex.

We impose these restrictions so that the scheduler can always find the next thread to run in a bounded time, even when the preferred thread (that is, the highest priority ready thread) is currently blocked on a fast mutex. We have placed assertions in the debug build of the kernel to flag violations of these rules to those porting Symbian OS and writing device drivers.

Context switching

Under the moving memory model (used on ARM architecture 4 and 5 processors), a context switch between threads belonging to different user-side processes can be a time consuming procedure. It may involve, in the worst case:

- Moving all the chunks attached to the current data section process to the home section
- Protecting all chunks attached to the last user process to run
- Moving all chunks attached to the new process from the home section to the data section
- Flushing the processor data cache.

On processors with large data caches and slow memory interfaces, this could take more than 500 µs - this is a measured value from one such system. If all this work were done directly by the scheduler, with preemption disabled, this would add 500 µs or more to the worst case thread latency. We didn't consider this to be acceptable performance - not all context switches require the full list of actions listed. Switches to kernel threads and threads in certain user processes can occur much faster and so should have lower guaranteed latency. To achieve this goal, we perform the modification of page directory entries and the flushing of the data cache with preemption enabled. Essentially, the kernel restores the registers for the new thread, so that the system is using the new thread's supervisor stack, then re-enables preemption before restoring the correct MMU configuration. The new thread then establishes its own MMU configuration. Clearly we need some protection to prevent multiple threads modifying the page tables simultaneously, and so we ensure that code holds the system lock fast mutex while performing these operations.

The Symbian OS kernel

Bounded and unbounded services

The services provided by the Symbian OS kernel are partitioned into two classes - bounded and unbounded. The bounded services obviously have bounded execution times, and are therefore suitable for real time applications. The unbounded services have no guaranteed upper limit on their execution times. The list of unbounded services includes:

- Object creation and destruction, including thread and process creation, IPC session creation and destruction
- Opening handles on existing objects
- Finding objects by name or by ID
- Symbian OS timer services `(RTimer::After()` and `RTimer::At()`.

In most cases, the unbounded services involve the allocation or freeing of memory on the kernel heap or in the global free page pool. This is because the algorithms used to manage the free page pool and the kernel heap do not have bounded execution times.

The bounded services include:

 (C) Copyright Nokia 2014. All rights reserved.

- Mutexes (DMutex and `RMutex` classes)
- Semaphores (DSemaphore and `RSemaphore` classes)
- Asynchronous request processing
- Device driver access
- Message queues (RMsgQueue and `DMsgQueue` classes)
- Client/server IPC message send and receive
- Thread control (suspend, resume, change priority, kill)
- Finding objects from a user handle.

We achieve mutual exclusion in bounded services by using fast mutexes, usually the system lock. To ensure maximum predictability for these services, the time for which the system lock is held continuously must be bounded and the bound minimized. We employ various measures to minimize this time, generally involving breaking down operations into sections between which the system lock is released.

For example, under the moving memory model, which I discussed in Section 17.3.1.4, the MMU related part of context switching could take a long time. Rather than holding the system lock for the whole duration, we make periodic checks to see if there is any contention for the system lock. We make this check every 512 bytes or every 1 KB (depending on type of cache) during the cache flush, and after moving or changing permissions of each chunk. If a higher priority thread is waiting on the system lock, we abort the context switch, release the system lock and allow the higher priority thread to run.

Similarly, during an inter-process data copy, we must hold the system lock to prevent the remote process' chunks from being moved while they are being accessed. Again, we make a check for contention after every 512 bytes are copied. If no contention is detected, the copy can continue immediately, but if contention is detected, we must release the system lock and reacquire it. In this case, we have to recheck the remote address, because the chunk involved may have been moved (or even deleted) while the system lock was released. This algorithm is illustrated in flowchart form in Figure 17.4.

Symbian OS mutexes

We designed Symbian OS mutexes for cases where mutual exclusion is needed but the restrictions on the use of fast mutexes cannot be observed. This situation arises in two main ways - either from a requirement for nested mutexes, or from a requirement for mutual exclusion in user-side code. We don't allow user-side code to hold fast mutexes since it cannot be trusted to conform to the restrictions on their use.

Symbian OS mutexes differ from fast mutexes in the following ways:

- They can only be used by Symbian OS threads (DThread) rather than by raw nanokernel threads. This has significance for personality layers, which I will describe later in this chapter
- They can be nested. A thread may hold a single mutex several times, provided that thread releases it the same number of times. A thread may hold a mutex while already holding a different mutex
- Threads can block while holding a Symbian OS mutex.

(C) Copyright Nokia 2014. All rights reserved.

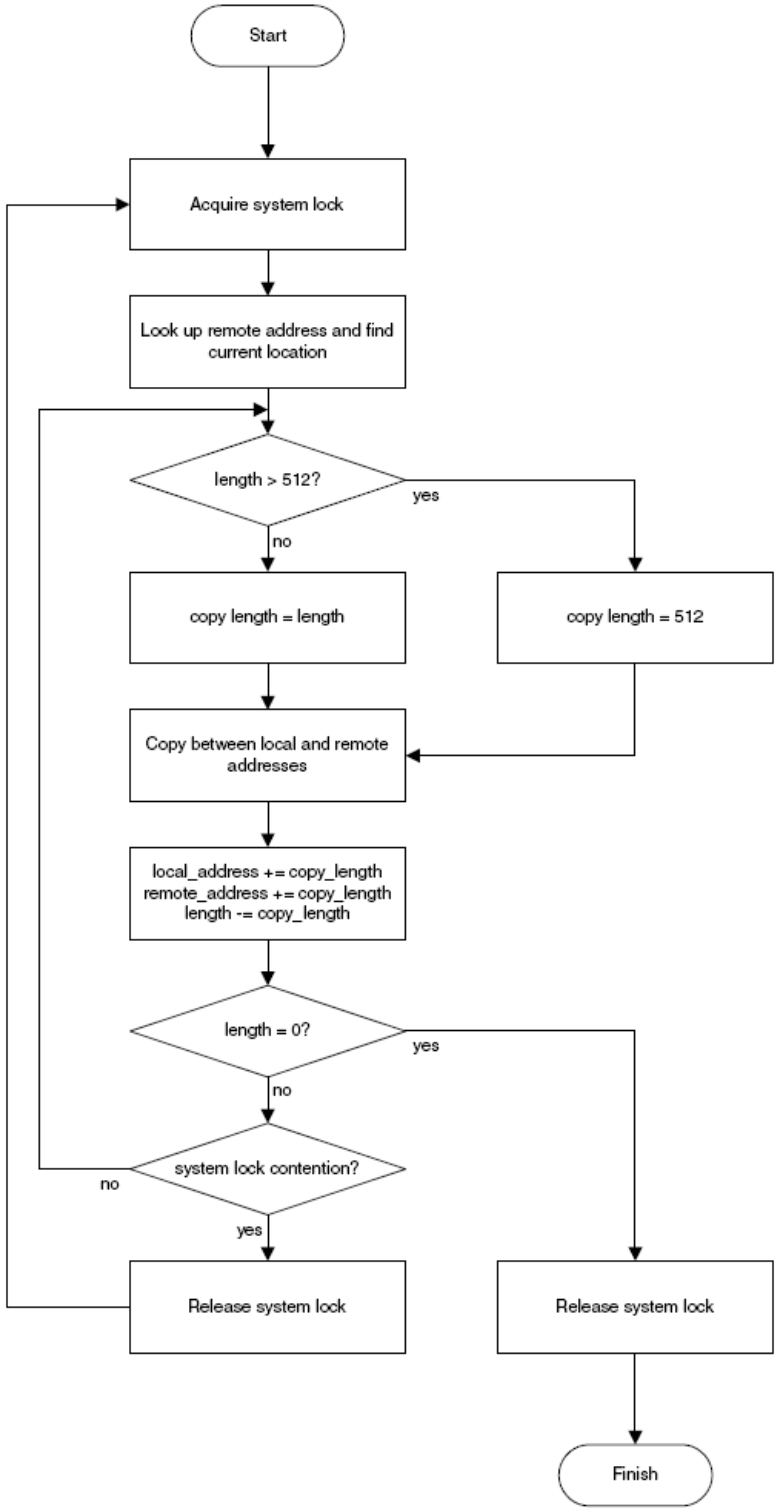

Figure 17.4 *Moving model IPC copy*

In Chapter 3, *Processes and Threads*, I covered the operation of Symbian OS mutexes in some detail. Here I will just speak from a real time perspective and say that we designed these mutexes to ensure that:

1. So far as is possible, the highest priority thread that requests a mutex holds it
2. The length of time for which point 1 is not true is minimized.

We use priority inheritance to meet this goal. In this case, priority inheritance does not come for free but has to be explicitly managed by the kernel. We use priority lists (which I described in Section 17.3.1.1) to hold the list of threads waiting to acquire a mutex and the list of mutexes held by any thread. We prioritize the latter according to the highest priority thread waiting on them, and use this to work out the required scheduling priority of the holding thread to comply with the rules of priority inheritance.

Latencies and performance

In this section I will define some key performance measurements that we use to verify the real time capabilities of EKA2. I will go on to give actual measured values for certain hardware platforms.

(C) Copyright Nokia 2014. All rights reserved.

Figure 17.5 illustrates these measurements. The scenario depicted in this figure is as follows: at time t0, a hardware interrupt occurs. It happens that interrupts are disabled at time t0 and so thread X0 continues executing for a while. When X0 re-enables interrupts, the processor's response to the interrupt begins. The interrupt preamble and dispatcher run and, at time t1, the first instruction of the service routine for the particular interrupt is executed. The elapsed time between t0 and t1 is known as the interrupt latency, denoted l**isr** in the diagram.

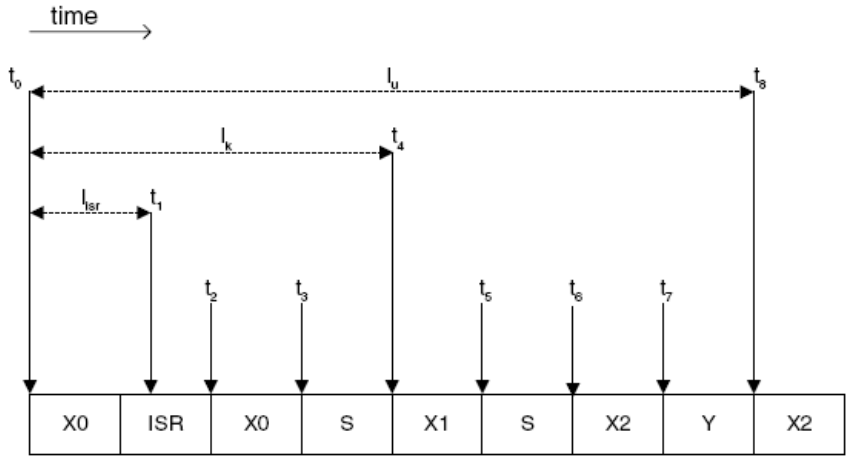

Figure 17.5 *Latencies*

The interrupt service routine finishes at time t2, having woken up a high priority kernel thread X1 (by way of an IDFC or DFC, not shown). However, the interrupted thread X0 has disabled preemption, so at time t2 thread X0 resumes execution until it re-enables preemption at time t3. The scheduler S runs at this point, and thread X1 starts running at time t4. The elapsed time between t0 and t4 is known as the kernel thread latency, denoted l**k** in the diagram.

Thread X1 finishes execution at time t5, having woken up a high priority user-side thread X2. The scheduler S runs again and schedules X2. X2 immediately tries to acquire the system lock fast mutex and is blocked by thread Y. Thread Y runs long enough to release the system lock and then thread X2 runs again, with the system lock held, at time t8. The elapsed time between t0 and t8 is known as the user thread latency, denoted l**u** in the diagram.

Worst-case interrupt latency is equal to the maximum time interrupts are disabled plus the execution time of the preamble and dispatcher. Interrupts can be disabled either by the kernel or by the base port and device drivers. When measuring the performance of the kernel only the former is of interest.

Worst-case kernel thread latency includes the maximum time for which preemption is disabled and the time taken to switch from an arbitrary thread to a kernel thread.

Worst-case user thread latency includes, in addition, the maximum time for which the system lock is held. The reason for including this is that most user threads will require the system lock to be held at some point before they can run and do useful work. This is a consequence of the following observations:

- On the moving memory model, the system lock must be held before context switching to the user thread
- If a Symbian OS asynchronous request (TRequestStatus) must be completed to make the thread run, that involves acquiring the system lock
- Most executive calls need to acquire the system lock; for example communication with a device driver will require this.

We measure the parameters l**isr**, l**k** and l**u** in the following way. We set up a hardware timer to produce a periodic interrupt. (This can simply be the timer used to produce the nanokernel tick, but it could be a separate timer.) It must be possible to read the timer count and deduce the elapsed time since the last tick. If possible, the measurement timer is set up as a high priority interrupt (FIQ on ARM), so that other peripheral interrupts do not influence the measurement. We also create two threads - one kernel thread running a DFC queue and one user thread. The priorities of these two threads are set at 63 and 62 respectively so that they preempt all the other threads in the system. Under the moving memory model, we usually ensure that the user thread belongs to a fixed process. Each periodic tick interrupt triggers a DFC on the kernel thread, and that thread then wakes up the user thread by signaling its request semaphore. We record the elapsed time from the last tick at the beginning of the ISR, the DFC, and just after the user thread wakes up. We also record maximum values for each of these times. We run stress test programs in the background to ensure that sections of kernel code that disable interrupts and preemption and that hold the system lock are exercised.

Measured performance parameters for some specific hardware platforms are given in the following table:

Parameter	Assabet	Series 5mx
Worst case interrupt latency	9 µs	25 µs

(C) Copyright Nokia 2014. All rights reserved.

Typical interrupt latency		3 µs
Worst case kernel thread latency	34 µs	120 µs
Typical kernel thread latency	13 µs	
Worst case user thread latency	63 µs	250 µs
Typical user thread latency	26 µs	
Thread switch time	2 µs	12 µs

The Assabet platform uses a 206 MHz SA1110 processor with code and data both stored in 32-bit wide SDRAM clocked at 103 MHz.

The Series 5mx platform uses a 36.864 MHz ARM710T processor with code and data both stored in 32-bit wide EDO DRAM.

In general, worst-case latencies are dominated by memory access speed rather than by processor clock frequency. This is because on typical hardware used for Symbian OS, a memory access requires several processor clock cycles - for example the random access time of SDRAM (60 ns) equates to 12 processor clock cycles on Assabet. A large difference between typical and worst case execution times can also be expected on this hardware. The best case execution time occurs when all code and data involved are present in the processor caches, so no slow external memory accesses are needed. Obviously, the worst-case execution time occurs when none of the code or data is present in the caches, so a large number of slow memory accesses are required - we ensure this is the case in our tests by running the stress test programs previously mentioned.

Real time application - GSM

This section gives a brief explanation of the GSM cellular system and an outline of how it might be implemented, with particular emphasis on the real time aspects.

Introduction to GSM

Data is transmitted over the radio channel at a bit rate of 270.833 kHz (=13 MHz/48, period 3_._69 µs). The spectrum used for GSM is divided into channels spaced 200 kHz apart. These channels are time-division multiplexed into eight timeslots, numbered 0 to 7. Each timeslot lasts for 156.25 bit periods (577 µs) and eight consecutive timeslots (one of each number) make up one frame (4.615 ms). In the original GSM system (prior to GPRS and high speed circuit switched data services) each mobile receives only one timeslot per frame. If the mobile is transmitting, the transmission occurs three timeslots after the receive. Of the 156.25 bit periods in a burst, 148 actual data bits are transmitted (except for random-access bursts, which transmit only 88 bits). The other 8.25 bit times are used to allow the mobile to ramp its power up and down in such a way to avoid excessive spurious emissions.

The 4.615 ms TDMA frames are numbered modulo 2715648 (= 26 * 51 * 2048). The frame number is used to divide a single physical radio channel into a number of logical channels, which use frames on a periodic basis. The frame number is also used as a parameter by the frequency hopping and encryption algorithms. The logical channels are:

Channel name	Description
Frequency correction channel (FCCH)	Base-station to mobile only. Consists of a pure sine wave burst with a frequency 67.7 kHz above the carrier frequency, equivalent to all data bits = 1. Used to allow the mobile to find the synchronization channel and to correct its internal frequency reference sufficiently to be able to receive it. Transmitted in timeslot 0 of frames numbered 0, 10, 20, 30, 40 modulo 51.
Synchronization channel (SCH)	Base-station to mobile only. Used to enable the mobile to establish precise time and frequency synchronization, including the frame number. The current frame number between 0 and 2715647 is transmitted in the SCH burst. Transmitted in timeslot 0 of frames numbered 1, 11, 21, 31, 41 modulo 51.
Broadcast control channel (BCCH)	Base-station to mobile only. Carries information about the cell configuration (number of frequencies, number of paging slots, etc.) of a particular base station. Transmitted in timeslot 0 of four consecutive frames starting with a frame numbered 2, 12, 22, 32, 42 modulo 51.
Paging channel (PCH)	Base-station to mobile only. Used to send paging messages to mobiles; a paging message is a request for the mobile to access the network (for example, for an incoming call).
Access grant channel (AGCH)	Base-station to mobile only. Used to assign the mobile a dedicated channel in response to a mobile accessing the base station. Once a dedicated channel has been set up, a dialogue can occur between the mobile and base station.
Cell broadcast channel (CBCH)	Base-station to mobile only. Used to transmit miscellaneous information, usually of a network-specific nature, such as dialing codes to which cheaper call rates apply.
Random access	

(C) Copyright Nokia 2014. All rights reserved.

Random access channel (RACH)	Mobile to base station only. Used by the mobile to request access to the network, for example, to originate a call or in response to a paging message.
Stand-alone dedicated control channel (SDCCH)	Both directions. This is a low-rate (just under 800 bps) channel that is used for the initial phase of call setup (authentication, ciphering, call setup message, etc.), SMS transmission and for network accesses that are not user-initiated (for example, location updating).
Traffic channel (TCH)	Both directions. This is a high-rate (13 kbps) channel that is used for the duration of a call to transfer encoded speech or user data.
Slow associated control channel (SACCH)	Both directions. This is a low-rate (~400 bps) channel that is used mainly for the transmission of surrounding cell information while the mobile is using a TCH or SDCCH. It is also used to allow the mobile to send or receive SMS while a call is in progress. In the latter case, every other SACCH message is SMS-related and every other one surrounding cell-related. This channel is always associated with a TCH or SDCCH, and its bursts are time-division multiplexed with those of the TCH or SDCCH (different frame numbers).
Fast associated control channel (FACCH)	Both directions. This is a high-rate (9200 bps) channel used to send signaling messages while a call is in progress. This is mainly used to send handover messages. This channel is always associated with a TCH and its bursts replace some of the TCH bursts.

There are four different types of burst used:

Burst type	Used on	Description
Frequency	FCCH	This consists of a pure sine wave with a frequency 67.7 kHz (one quarter the bit rate) above the carrier frequency. This is equivalent to a burst where all data bits are 1.
Sync	SCH	This consists of 148 transmitted bits, and can be decoded on its own. The first and last 3 bits are 0 (called tail bits). The middle 64 bits are the midamble, which I describe next. The remaining 78 bits are encoded data bits, derived from 25 bits of user data.
Random Access	RACH	This consists of 88 transmitted bits and can be decoded on its own. The first 52 bits are the preamble, which serves the same function as the midamble for the other bursts. The remaining 36 bits are encoded data bits, derived from 8 bits of user data.
Normal	All others	This consists of 148 transmitted bits. The first and last three bits are tail bits (0). The middle 26 bits are the midamble. The two bits on either side of the midamble are called stealing flags - they differentiate between signaling and traffic channels. The remaining 114 bits are encoded data bits.

The midamble (preamble for RACH) is used to enable the receiver to get the precise timing of the burst, and to compensate for the effects of multipath distortion. By correlating the received midamble bits with the known value of the midamble, the receiver can get the timing of the burst and also estimate the multipath distortion on the radio channel. It can then compensate for the multipath distortion and make an estimate of the received data bits. This process is performed by the equalizer in the receive chain.

The stealing flags are always set to 1 except on traffic channels, where they are used to differentiate between TCH and FACCH bursts (0 = TCH, 1 = FACCH).

In general (apart from FCCH, SCH and RACH), more than one burst is required to make a meaningful data block. The data to be transmitted is first passed through a forward error correction encoder (either one or two stages) then the bits are interleaved (reordered) and divided between a number of bursts (4, 8 or 19). The encoding used is:

Channel	Encoding
SCH	Start with 25 bits of user data. Append 10-bit CRC check code. Pass through half-rate convolutional encoder. This gives (25 + 10 + 4) * 2 = 78 bits. The extra 4 bits are needed to flush out the convolutional encoder.
RACH	Start with 8 bits of user data. Append 6-bit CRC check code. Pass through half-rate convolutional encoder. This gives (8 + 6 + 4) * 2 = 36 bits.
BCCH, PCH, AGCH, CBCH, SDCCH	Start with23 bytes = 184 bits of user data. Pass through a FIRE block encoder that appends 40 parity check bits. Then pass through a half-rate onvolutional encoder. This gives a total of (184 + 40 + 4) * 2 = 456 bits. These bits are divided between the transmitted bursts in four consecutive frames.
TCH	Start with 260 bits of compressed speech data. Split these into 50 class Ia, 132 class Ib and 78 class II bits. Calculate and append a 3-bit CRC check code to the class Ia bits. Pass the class Ia and class Ib bits through a half-rate

(C) Copyright Nokia 2014. All rights reserved.

TCH speech	convolutional encoder. Append the class II bits, unencoded. This gives a total of (50 + 3 + 132 + 4) * 2+78 = 456 bits. Divide these into eight sets of 57 bits and spread them over eight half-bursts, in combination with the previous and successive speech blocks.
TCH data	For 9600 bits per second data, start with 240 bits of user data. Pass through a half-rate convolutional encoder to give (240 + 4) * 2 = 488 bits. Puncture the code by omitting 32 specified bits to give 456 bits of encoded data. Divide these into 19 sets of 24 bits and spread them over 19 consecutive traffic bursts, in combination with other data blocks.
FACCH	Channel encoding as for BCCH, but divide resulting 456 bits into eight sets of 57 and spread over eight half-bursts as for full-rate speech TCH.
SACCH	Channel encoding as for BCCH, but the four bursts are transmitted with a 26-frame gap between them instead of in consecutive frames, thus giving a throughput of 23 bytes per 104 frames.

On the receive side, the process is reversed, with bits from various equalized bursts being stored until there are enough bursts to make a decodable block. The bits are then de-interleaved (reordered into an order which the error correction decoder can make sense of) and passed through a Viterbi convolutional decoder. If necessary, the output of the Viterbi decoder is then passed through a FIRE decoder or a CRC check.

Encryption is performed on SDCCH, TCH, FACCH and SACCH. A special algorithm is used (A5.1 or A5.2) which takes the frame number and the encryption key as parameters and produces two 114-bit blocks of encryption data, one for receive and one for transmit. The receive block is exclusive-ORed with the output of the equalizer prior to de-interleaving. The transmit block is exclusive-ORed with the output of the interleaver prior to addition of the midamble and stealing flags. The key used for encryption is generated by the SIM as part of the authentication process. Whenever a mobile tries to connect to the network, the base station will perform a challenge/response authentication procedure. The base station sends a random number to the mobile and this is passed to the SIM where it is used as input to the A3 algorithm in conjunction with a secret key (K_i) which never leaves the SIM, but which is also known to the network. The A3 algorithm produces two outputs - SRES and Kc. The SRES is sent back to the base station in response to the authentication request, and the Kc is used as the encryption key for subsequent traffic.

Idle mode

When a mobile phone is not actively being used it spends most of its time in idle mode. In this mode the phone performs receive operations only. The following operations are performed by a phone in idle mode:

1. Periodically receive the BCCH for the cell on which the mobile phone is camped. This tells the mobile when it should listen for paging requests and which frequencies it should monitor for neighbor cells
2. Periodically receive the paging channel as indicated by the BCCH message. Paging messages are transmitted in four consecutive frames and any particular mobile must listen every 102*N frames, where N is between 2 and 9 inclusive, so approximately every 1 to 4.5 seconds
3. Monitor the signal strength of neighbor cells as indicated in the BCCH message. If a neighbor cell is received consistently better than the current cell for a period of time the mobile will move to the neighbor cell
4. Send location update messages to the network both periodically (around every 30 minutes) and following a change of cell where the old and new cells are in different location areas. Base stations are grouped into location areas and an incoming call to a mobile will result in paging messages being transmitted by all base stations in the last reported location area. Of course to send a message to the network the mobile must briefly leave idle mode.

Traffic mode

When a call is in progress, the mobile phone is in traffic mode. The phone receives a burst in 25 out of every 26 frames and transmits in around 60% of frames. The GSM traffic channel operates on a period of 26 frames, used as follows:

Frame mod 26	Description
0 to 11	Encoded speech bursts or FACCH bursts
12	SACCH burst or idle slot
13 to 24	Encoded speech bursts or FACCH bursts
25	Idle slot or SACCH burst

A single 20 ms speech frame is spread between eight consecutive frames, overlapped with the preceding and following speech frame. Four SACCH bursts are required to make up a meaningful data block, so one block is received and transmitted every 104 frames. This is too slow for seamless handover between cells so when necessary some bursts normally used for speech are *stolen* for signaling messages. The SACCH bursts are staggered between timeslots - the first of a group of four occurs when frame number modulo 104 equals 13*TN, where TN is the timeslot number between 0 and 7. This was done so that base stations (which must obviously process all timeslots) only need to perform one SACCH encode and decode every 13 frames rather than

(C) Copyright Nokia 2014. All rights reserved.

eight all at once.

While a call is in progress the mobile phone performs the following operations:

1. Receive traffic bursts, pass them through the GSM speech decoder and out to the speaker
2. Accept audio from the microphone and pass it through the GSM speech encoder. If someone is actually speaking, transmit the encoded speech in traffic bursts. During periods of silence no transmission occurs to save battery power
3. Receive SACCH bursts and decode them. They contain instructions as to precise transmission timing (to compensate for the propagation times of the signals between the mobile phone and base station), the transmit power that the phone should use and the neighboring cells that the phone should monitor
4. Monitor the indicated neighboring cells. The signal level from each one should be measured and additionally FCCH and SCH bursts are received from each one to get the precise frequency and timing in case a handover is required. All this activity occurs in the *idle* slot. The 26 frame period of the traffic channel is deliberately chosen to be coprime to the 51 frame period of the control channels so that the FCCH and SCH bursts of the neighbor cell will eventually occur during the idle slot
5. Transmit the measured signal levels on the neighboring cells back to the base station in the SACCH bursts
6. Receive and act on handover commands transmitted on the FACCH.

GSM hardware

A typical GSM mobile implementation consists of the following functional blocks:

- RF stages
- Baseband converters
- Timing controller
- Encryption unit
- DSP
- Microcontroller
- SIM interface
- Miscellaneous (LCD, keyboard, earpiece, microphone, battery charging circuitry).

The original GSM frequency band has 124 channels, spaced 200 kHz apart. The mobile receive frequency is 935+0.*2n* MHz, the mobile transmit frequency is 890+0.*2n* MHz where n is the channel number between 1 and 124. On the receive side, the signal is amplified and downconverted using a frequency synthesizer for channel selection. A quadrature mixer is used, producing two baseband signal outputs I and Q (in-phase and quadrature components - the result of mixing the received signal with two different carriers 90º out-of-phase). On the transmit side, the I and Q signals from the modulator are mixed up using another quadrature mixer to produce the final RF frequency for feeding to the PA (power amplifier). The PA output level is adjustable by a separate DAC output.

The baseband receive ADC samples both the I and Q receive channels. It may also contain some of the channel filtering (in which case it oversamples the I and Q signals and filters them digitally). The output of the baseband receive ADC is one I and one Q sample, of ~10 bits each, every GSM bit period (3.69 µs). Each receive burst will produce around 160 sample pairs (must receive more samples than there are bits to account for timing errors).

The transmit modulator converts the bits to be transmitted into varying I and Q voltages according to the specification in the GSM standard (GMSK modulation). Essentially, a 1 bit causes the phase of the carrier to advance by 90º and a 0 bit causes the phase to be retarded by 90º; however some low-pass filtering is applied so that the phase changes smoothly, with the previous 3 bits making a significant contribution to the carrier phase at any time. This is done to reduce the bandwidth of the transmitted signal. The modulation is done digitally and the output converted to I and Q signals by the baseband transmit DAC.

At least three other baseband converters are required. A DAC is required to control the output power level while transmitting. A FIFO store is required to feed this DAC so that the power can be ramped up and down at the beginning and end of the transmit burst (one output sample per half bit-period). An ADC and DAC are required for the microphone and earpiece. These both work at a sampling rate of 8 kHz and resolution of 13 bits.

A versatile timing controller is required, which can be programmed to a resolution of one-quarter bit period. This is used to switch various parts of the hardware on and off at the correct time, and to initiate receive and transmit operations at the correct time. The timing controller is synchronized with the received signal from the base station after FCCH and SCH receive.

The DSP usually performs the following functions:

- Reading received data from the baseband receive ADC
- Frequency burst detection
- Sync and normal burst equalization

(C) Copyright Nokia 2014. All rights reserved.

- Channel encoding
- Channel decoding
- Speech encoding
- Speech decoding
- Assembly of bursts for transmission
- Buffering of audio samples
- Generation of sidetone and miscellaneous GSM-specified tones.

The DSP may also perform a certain amount of the low-level control and scheduling functions. The split of these between the DSP and microcontroller varies between implementations. It is implementation dependent whether the timing controller is configured by the DSP or the microcontroller.

The microcontroller performs the following tasks:

- Some of the layer 1 control functions (implementation-dependent split with DSP)
- Layers 2 and 3 of the GSM protocol stack
- Control of screen and keyboard, battery monitoring, and possibly charging
- User interface functions
- Control of the SIM interface
- Extra processing for data traffic channels.

The SIM interface connects the SIM (subscriber identity module - smart card) to the microcontroller. It is essentially a UART, but operates in a half-duplex mode. The SIM has only one data line and it is used for both reading and writing to the SIM. There is a specified protocol for this interface, which is usually implemented in hardware.

A GSM stack on EKA2

As an illustration of a real time application running on EKA2, I will consider the implementation of a GSM protocol stack. Obviously, only an outline will be given and not all scenarios will be considered - a real stack requires many man-years of development! I will assume that all processing is done on the main ARM processor and there is no DSP - while, in principle, this could be achieved on high end ARM devices such as ARM1136, in practice no-one would actually do so, because it would be inefficient in terms of power consumption, cost and the number of cycles available for application processing.

I will assume the following hardware configuration:

- Timing controller has free running counter at four times the GSM bit rate which wraps to zero at 5000 quarter bits, that is, one GSM frame period
- There are several match registers that can trigger an event. This event can be an interrupt to the processor, switching on or off some piece of hardware such as the frequency synthesizer, the receive chain or the transmit chain, the start of a receive or the start of a transmission
- Once a receive starts, the processor is interrupted whenever 16 sample pairs are available and these must be read before the next interrupt occurs
- Before a transmission starts the data to be transmitted must be loaded into a TX buffer and the power ramp up and ramp down masks must be loaded into a power control buffer
- Frequency synthesizer needs 500 µs to lock before beginning RX or TX
- Equalizing a received burst, channel encoding or decoding and speech encoding or decoding takes 250 µs maximum
- Scanning for a FCCH burst takes 10 µs maximum per 16 sample pairs.

A processor interrupt is triggered once every frame in the same place so that a frame count can be maintained.

Frequency burst receive

The main activities related to FCCH burst reception are shown in Figure 17.6.

(C) Copyright Nokia 2014. All rights reserved.

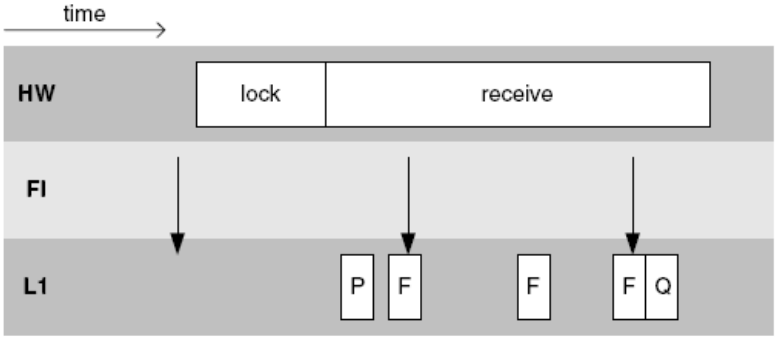

Figure 17.6 *Frequency burst search*

In Figure 17.6, the HW line shows what the hardware is doing, the FI line indicates the time of the once-per-frame interrupt and the *L1 Task* line shows which software tasks are in progress.

The tasks involved are:

(i) Programming the hardware with the frequency to receive and the time at which to start the receive. This is shown as task P. This task is triggered by the main layer 1 state machine when a frequency burst search is required

(ii) Reading blocks of 16 sample pairs from HW buffer to main memory buffer. This task is not shown since it is too frequent. It is triggered by the baseband receive ADC interrupt

(iii) Processing the received samples to search for a frequency burst. This is shown as task F, although it occurs more frequently than is shown. It is triggered by task (ii)

(iv) On either finding a frequency burst or timing out, shut down the hardware and start the next operation if any. This is shown as task Q. It may be necessary to search for up to 11.25 frame times - the worst case occurs when you start the receive just after the beginning of the FCCH burst and the next burst starts in 11 frames. This task is triggered by the per-frame interrupt following either the detection of a frequency burst or a timeout.

Task (ii) has the shortest deadline (16 bit periods or 59 µs), so I will implement this as an ISR. In fact with a hard deadline as short as this, it would be best to use the ARM processor's FIQ interrupt. The other tasks must run in a thread context. It would be inefficient to schedule a thread at this frequency - it would use around 10% of the processor time context switching. It is also unnecessary. Instead, task (iii) will be triggered every 10 receive interrupts. The deadline for task (iii) depends on how much buffering is used. If there is enough buffering for 480 sample pairs (around 2 K) the deadline would be 320*3_._69 µs = 1180 µs, since the task is triggered when there are 160 sample pairs available and must complete before the buffer fills.

Tasks (i) and (iv) are triggered by the per-frame interrupt and need to complete quickly enough to set up hardware for the following frame. A convenient time for the per-frame interrupt is two timeslots into the frame, since it then does not clash with either the receive or transmit windows and allows six timeslots for any setup to complete before the next frame. This is 3.460 ms. However, 500 µs is required for the synthesizer to lock before the next receive and so the deadline for tasks P and Q is 2.9 ms. So task P and Q should have lower priority than task F. In fact in this case the same thread could be used for all three tasks since these tasks never execute simultaneously anyway.

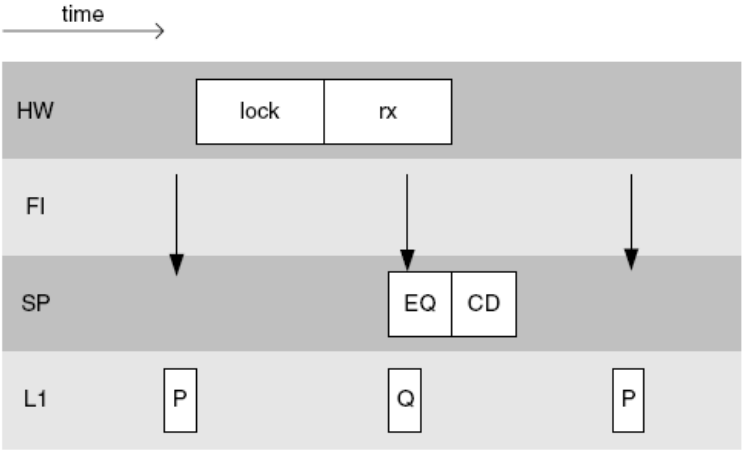

Figure 17.7 *Sync burst receive*

Sync burst receive

The activities involved in receiving an SCH burst are shown in Figure 17.7. The tasks involved are:

(C) Copyright Nokia 2014. All rights reserved.

(i) Programming the hardware with the frequency to receive and the time at which to start the receive. This is shown as task P. This task is triggered by the main layer 1 state machine when an SCH burst receive is required

(ii) Reading blocks of 16 sample pairs from HW buffer to main memory buffer. This task is not shown since it is too frequent. After 11 blocks have been received, task (iii) is triggered. 176 sample pairs are used instead of the usual 160 since there is generally more uncertainty about the precise timing of a sync burst, since the only timing information may have come from the previous FCCH receive. Non-FCCH bursts contain a known sequence of bits (the midamble) which can be used to estimate the burst timing accurately

(iii) Pass the 176 received sample pairs through the sync burst equalizer (EQ). This calculates the precise timing of the burst and demodulates it

(iv) Pass the demodulated burst through the channel decoder (CD). This corrects any errors in the received burst. This task is triggered by the end of the EQ task

(v) Reprogram the hardware to stop the receive occurring again. This is shown as task Q. It is triggered by the per-frame interrupt following the receive period.

As before, task (ii) has the shortest deadline, at 59 µs, and I implement it directly in the FIQ. The EQ and CD tasks together have to meet a deadline of ten timeslots -176 bit periods, or 5.1 ms, in order for the decoded data to be available at the following frame interrupt. As before task (i) has a deadline of 2.9 ms. Task (v) must complete before the receive would start again, so the deadline is six timeslots minus 500 µs, or 2.9 ms.

Control channel receive

Figure 17.8 shows the reception of a control channel block. This could be a BCCH, PCH, AGCH, SDCCH or SACCH accompanying SDCCH.

The tasks marked RX also include waiting for the frequency synthesizer to lock. The tasks involved are:

(i) Programming the hardware with the frequency to receive and the time at which to start the receive. This is shown as task P. The main layer 1 state machine triggers it when an SCH burst receive is required

(ii) Reading blocks of 16 sample pairs from HW buffer to main memory buffer. After 10 blocks have been received, task (iii) is triggered. 160 sample pairs give sufficient leeway to cope with small errors in the burst timing. There is no need to reprogram the hardware after a burst receive since the next receive will automatically be initiated when the timer wraps round

(iii) Pass the 160 received sample pairs through the normal burst equalizer (EQ). This calculates the precise timing of the burst and demodulates it

(iv) Pass the four demodulated bursts through the de-interleaver and channel decoder (CD). This corrects any errors in the received data. This task is triggered by the end of the fourth EQ task

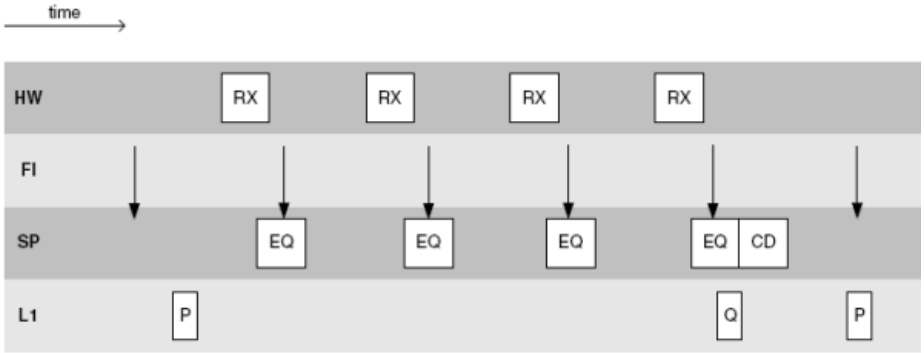

Figure 17.8 *Control channel receive*

(v) Reprogram the hardware after the final burst receive so that another receive does not occur. This is shown as task Q. It is triggered by the frame interrupt following the final burst receive.

Task deadlines are the same as for the sync burst receive, apart from the EQ task on its own. The deadline for this is that it should finish before the next receive starts, which is a time lapse of seven timeslots, or 3.9 ms.

Control channel transmit

Figure 17.9 shows the transmission of a control channel block. This could be either an SDCCH or SACCH accompanying SDCCH.

Tasks marked TX include waiting for the frequency synthesizer to lock.

The tasks involved are:

(C) Copyright Nokia 2014. All rights reserved.

(i) Passing the data block to be transmitted through the channel encoder and interleaver and adding the midambles, stealing flags and tail bits to produce four 148-bit bursts for transmission. This is shown as task CE in the figure. The main layer 1 state machine triggers it when an SDCCH or accompanying SACCH transmission is required

(ii) Programming the hardware with the frequency to transmit and time at which to start transmission. This is shown as task P and is triggered by the end of the CE task or by an interrupt at the end of the previous transmit burst if there is one

(iii) Transferring each burst to the transmit data buffer. This is shown as task T on the diagram. Task T is triggered by the end of task P or by the end of a transmit burst

(iv) Reprogram the hardware after the final burst transmission so that another transmission does not occur. This is shown as task Q. It is triggered by the end of the final transmit burst.

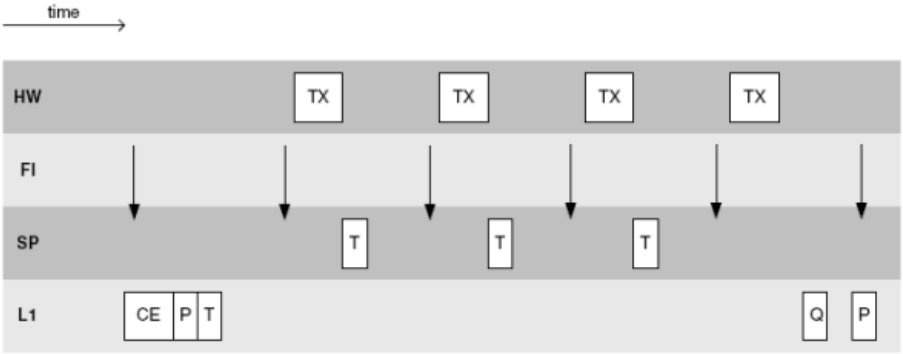

Figure 17.9 *Control channel transmit*

The deadline for task CE, P and the first task T is 500 µs before transmission is due to start, so approximately 1 frame period, since transmission starts three timeslots into the next frame. If a previous transmit operation is in progress the deadline for CE is unchanged, but the deadline for P and T is reduced to six timeslots or 3.4 ms. The subsequent task T invocations are triggered by the end of the transmission and must complete before the start of the next transmission. Hence the deadline there is seven timeslots (3.9 ms).

Control channel simultaneous receive and transmit

Figure 17.10 shows the simultaneous transmission and reception of a control channel block. This only occurs with an SDCCH in one direction and an SACCH accompanying SDCCH in the other.

The tasks involved are:

(i) Pass the transmit data block through the channel encoder and interleaver (CE). Add midambles, stealing flags and tail bits to produce bursts for transmission. This task is triggered by the main layer 1 state machine when a simultaneous SDCCH receive and transmit is required

(ii) Programming the hardware with the frequency to receive/transmit and the time at which to start the receive and transmission. This is shown as task P. It is triggered by the end of the CE task or the end of the previous transmit burst if there is one

(iii) Reading blocks of 16 sample pairs from HW buffer to main memory buffer. This task is triggered by an interrupt from the receive ADC when 16 sample pairs are available. After 10 blocks have been received, task (ii) is triggered

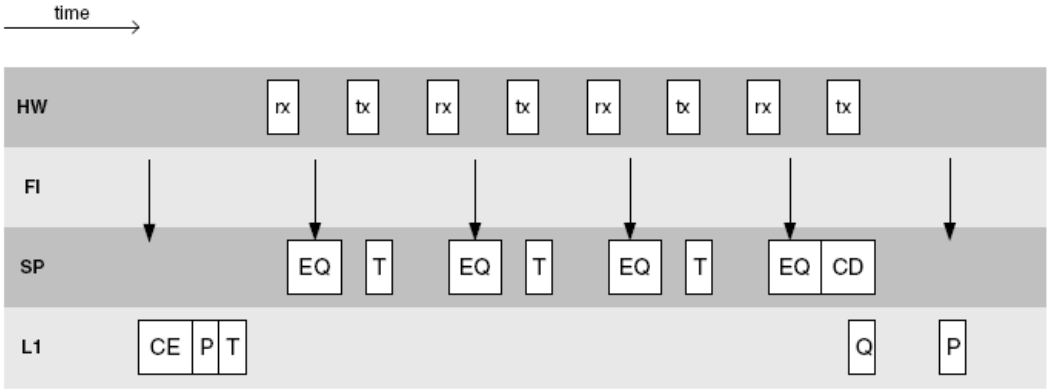

Figure 17.10 *Control channel TX + RX*

(iv) Pass the 160 received sample pairs through the normal burst equalizer (EQ). This calculates the precise timing of the burst and demodulates it. This task is triggered by the receive ISR when 160 sample pairs have been received

(v) Pass four demodulated bursts through the de-interleaver and channel decoder (CD). This corrects any errors in the received

(C) Copyright Nokia 2014. All rights reserved.

data. This task is triggered by the end of the last EQ task

(vi) Transfer each traffic burst to the transmit data buffer. This is shown as task T on the diagram. This task is triggered either by the end of the P task or by the end of the previous transmit burst

(vii) Reprogram the hardware after the final burst transmission so that another receive/transmit does not occur. This is shown as the task Q. It is triggered by the end of the final transmit burst.

Task deadlines are as for the separate receive and transmit operations already described.

Traffic channel

Figure 17.11 shows the operation of a full rate traffic channel (TCH). The line marked FN shows the frame number modulo 26. Frames 21 to 1 are shown since they illustrate both the normal reception and transmission of traffic bursts and activity in the idle slot.

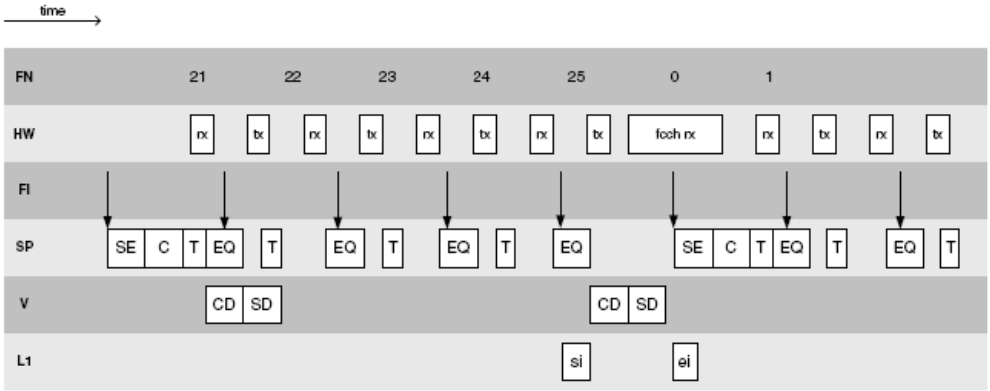

Figure 17.11 *Traffic channel*

The tasks involved are:

(i) Reading blocks of 16 sample pairs from HW buffer to main memory buffer. This task is triggered by an interrupt from the receive ADC when 16 sample pairs are available. After 10 blocks have been received, task (ii) is triggered. In most cases there is no need to reprogram the hardware after a burst receive since the next receive will automatically be initiated when the timer wraps round. The exception is the case shown where a different activity may be performed in the idle slot

(ii) Pass the 160 received sample pairs through the normal burst equalizer (EQ). This calculates the precise timing of the burst and demodulates it. This task is triggered by the receive ISR when 160 sample pairs have been received

(iii) Pass eight half-bursts from the previous 8 frames through the de-interleaver and channel decoder (CD). This corrects any errors in the received data. This task is triggered by the end of the EQ task in frames 3, 7, 11, 16, 20, 24 modulo 26

(iv) Pass the decoded data through the speech decoder (SD) and output the resulting audio to the speaker. This task is triggered by an interrupt from the audio DAC. Note that owing to slight errors in the locally generated audio sampling frequency relative to the base station's frame timing (generated from a precise frequency source) the task needs to handle buffer underflows and overflows. Underflow would normally be handled by repeating the previous sample and overflow by discarding a sample

(v) Read audio data from the microphone and, when 20 ms worth of samples are available, pass them through the speech encoder (SE) to produce a traffic data block. This task is triggered by the per-frame interrupt in frames 3, 7, 12, 16, 20, 25 modulo 26. The audio data itself will be read in by an interrupt from the audio ADC. As with task (iv), buffer underflow and overflow must be handled

(vi) Pass the traffic data blocks containing encoded speech through the channel encoder and interleaver (CE). Add midambles, stealing flags and tail bits to produce traffic bursts for transmission. This task is triggered by the end of the SE task

(vii) Transfer each traffic burst to the transmit data buffer. This is shown as task T on the diagram. This task is triggered either by the end of the CE task or by the end of the previous transmit burst

(viii) Decode an SACCH data block after every 4 SACCH bursts. This task is not shown on the diagram. It is triggered by the end of the EQ task for frames 13*TN+78 modulo 104

(ix) Encode an SACCH data block prior to the beginning of each group of four SACCH bursts. Triggered by a layer 2 SACCH block becoming available in frames between 13*TN-12 and 13*TN-1 modulo 104

(x) Reprogram the hardware to perform the required activity in the idle slot (task marked si on the diagram). This could either be a neighbor cell power measurement, which requires a 160 bit receive, a neighbor cell FCCH burst search, which requires a 1408-bit receive, or a neighbor cell SCH burst receive, which a requires a 176-bit receive starting at an arbitrary offset within a frame.

(C) Copyright Nokia 2014. All rights reserved.

This task is triggered either by the frame interrupt for the frame before the idle slot or by an extra interrupt at timer wraparound if the idle slot activity commences more than two timeslots into the idle frame

(xi) Reprogram the hardware for normal TCH activity following the idle slot (task marked ei on the diagram). Triggered either by the start of an idle slot power measurement or SCH receive, or by the frame interrupt during an idle slot FCCH receive

(xii) Process the samples received during the idle slot. This consists of either a simple power measurement, a frequency burst search or an SCH equalization and channel decode. This task is triggered by the end of the idle slot activity

As ever the deadline for task (i) is 59 µs, and this task runs directly in the ISR. Task (ii) should complete before the next burst receive, so the deadline is seven timeslots, or 3.9 ms. Tasks (v), (vi) and the first (vii) of each four form a group and their deadline is nine timeslots (5.1 ms) since the transmitted data must be available before the next transmit slot. Tasks (iii) and (iv) form a pair and their deadline is governed by the maximum allowed round trip delay time for GSM of 100 ms. After taking account of the fact that it takes eight frames to receive a speech block and eight frames to transmit one, we find that a deadline of four frame periods (18 ms) is permissible for tasks (iii) and (iv).

Tasks (viii) and (ix) only occur once every 104 frames. Their deadlines, in combination with the layer 2 and layer 3 processing of SACCH messages, are determined by the fact that a received SACCH message should have been processed in time for the next idle slot, and that an SACCH message for transmission must be assembled in the time between the previous idle slot and the first SACCH transmit frame. Thus a deadline of 12 frames (55 ms) is acceptable for each direction of SACCH processing.

The worst case for task (x) is an SCH receive starting immediately after the TX burst finishes (RX starts timeslot 5), which is two timeslots after the frame interrupt. Thus the deadline for task (x) is 1.1 ms. The worst case for task (xi) is an SCH receive starting just before timeslot 5 in the idle slot. The task must have completed by the start of timeslot 7 to set up for the next normal receive. Hence again the deadline is two timeslots, or 1.1 ms.

The result of task (xii) will not be needed before the next SACCH transmission, so this shares the deadline of 55 ms with other SACCH processing.

Layer 1 threads

From my earlier discussion, you can see that the real time deadlines involved in GSM layer 1 fall into five clusters. I list these in the following table:

Deadline	Context	Tasks
59 µs	FIQ	Reading 16 sample pairs from baseband receive ADC into main memory.
1 ms	L1 Thread	Main layer 1 state machine, idle slot setup and teardown during traffic mode, control channel encode.
3-5 ms	SP Thread	Receive equalization, control channel decode, traffic channel encode, speech encode.
18 ms	V Thread	Speech decode, traffic channel decode.
55 ms	SA Thread	Idle slot data processing, SACCH channel encode and decode.

Each of the tasks that I've mentioned in the preceding sections (apart from the baseband receive ISR) would be implemented as a DFC running on one of the four threads in the table. The priorities of these threads would be set in the order L1, SP, V, SA (highest to lowest).

Layers 2 and 3

Messages received on BCCH and PCH during idle mode are passed to layer 3 of the GSM protocol stack. Signaling messages received or transmitted on either SDCCH, SACCH or FACCH go through a small layer 2 as well, which discriminates between messages meant for the different parts of layer 3.

Layer 3 of GSM is split into three main pieces:

- **RR (Radio Resources).** This is responsible for allocation and management of radio frequencies and timeslots. Allocation of a channel (SDCCH or TCH) to a mobile, change of channel or handover between base stations are handled by RR, as is the reporting of measurements on neighboring cells, which is used to determine when to hand over a call
- **MM (Mobility Management).** This is responsible for keeping the network informed of each mobile's approximate position via location update messages
- **CC (Call Control).** This is responsible for setting up and closing down voice calls and data calls and for sending and receiving SMS messages.

Deadlines for processing messages in these layers range from four frames (18 ms) to seconds. Typically MM runs in a single thread and CC runs in two threads - one for calls and one for SMS. SMS can be sent and received while a call is in progress; every other SACCH message is used for SMS in this case. RR typically uses several threads (up to 10).

If you were writing a GSM stack for EKA2 from scratch, I would recommend putting layer 3 in a user-side process. This is because

(C) Copyright Nokia 2014. All rights reserved.

there would be no problem achieving latencies of 18 ms there, and running user side makes debugging easier.

Personality layers

17.5.1 Introduction

Every Symbian OS product will need to incorporate some type of mobile telephony stack, and usually also a Bluetooth stack too. These two items have the following features in common:

- They are large complex pieces of software in which the phone manufacturer has made a considerable investment
- They have significant real time requirements
- They generally run over some type of RTOS, either a proprietary one or a standard commercial RTOS such as Nucleus Plus, VRTX or OSE.

In the rest of this chapter, I will refer to any such software block as a legacy real time application (LRTA).

One way in which you could incorporate an LRTA into a mobile phone is by running it on its own CPU, separate from the one that runs Symbian OS. There are some advantages to this solution - the LRTA need not be modified and it is completely isolated from the Symbian OS software, reducing the integration burden. However, there are also disadvantages - mainly the cost of the extra processor and the accompanying memory. So, let us assume that the separate processor solution is too expensive, and that the LRTA must run on the same CPU as Symbian OS. There are essentially three ways of achieving this:

1. Modify the source code (and possibly design) of the LRTA to run directly over Symbian OS - either as a purely kernel-mode device driver, or as a combination of kernel and user mode components
2. Implement a system in which both Symbian OS and the LRTA RTOS run concurrently. You could do this either by placing hooks into the Symbian OS kernel at strategic places (interrupt and possibly other exception vectors) to allow the RTOS to run, or by implementing some kind of *hypervisor* that performs context switches between the two operating systems. This would require modifications to both operating systems to make calls to the hypervisor to indicate thread switches, priority changes and so on
3. Implement a personality layer over the EKA2 kernel, which provides the same API as the RTOS, or at least as much of it as is required by the LRTA. The RTOS itself can then be dispensed with and the LRTA can run using EKA2 as the underlying real time kernel.

I alluded to the first of these options in Section 17.4.5. Nevertheless, this option is unlikely to be viable because of the time it would take to modify the LRTA, the risk involved in doing so and the problem of creating a second distinct version of the LRTA that then increases the phone manufacturer's maintenance burden.

The second option suffers from the following problems:

- Performance is degraded because of the hooks that are called on every interrupt and every executive call, even if they are not related to the LRTA. The hypervisor system will degrade performance even more due to the presence of more hooks and a whole extra layer of processing on interrupts
- The hooks add additional complication and risk of defects to particularly sensitive areas of code
- Inserting hooks into the Symbian OS kernel to allow the RTOS to run whenever it wants to destroys the real time performance of EKA2 since a low priority thread in the LRTA will take precedence over a high priority thread in Symbian OS. The hypervisor system would not necessarily suffer from this problem but would be considerably more complicated and incur a larger performance penalty
- The hooks become extremely complicated and hard to manage if more than one RTOS needs to run, for example if both a GSM signaling stack and a Bluetooth stack are required and each uses a different RTOS.

For these reasons, Symbian prefers option 3 as a solution to this problem. In the rest of this section I will describe how such a personality layer may be implemented.

The RTOS environment

Even the most minimal RTOS provides the following features:

- Threads, usually scheduled by static priority-based scheduling with a fixed number of priorities. Round robin scheduling of equal priority threads may be available but is usually not used in real time applications. Dynamic creation and destruction of threads may or may not be possible
- At least one mechanism for thread synchronization and communication. Typical examples would be semaphores, message queues and event flags. There is wide variation between systems as to which primitives are provided and what features they support. Again, dynamic creation and destruction of such synchronization and communication objects may or may not be supported. Mutual exclusion is often achieved by simply disabling interrupts, or occasionally by disabling rescheduling

(C) Copyright Nokia 2014. All rights reserved.

- A way for hardware interrupts to cause a thread to be scheduled. This is usually achieved by allowing ISRs to make system calls which perform operations such as signaling a semaphore, posting a message to a queue or setting event flags, which would cause a thread waiting on the semaphore, message queue or event flag to run. Some systems don't allow ISRs to perform these operations directly, but require them to queue some kind of deferred function call. This is a function that runs at a lower priority than hardware interrupts (that is, with interrupts enabled) but at a higher priority than any thread - for example a Nucleus Plus HISR (High-level Interrupt Service Routine). The deferred function call then performs the operation that causes thread rescheduling.

Most RTOSes also provide a timer management function, allowing several software timers to be driven from a single hardware timer. On expiry, a software timer may call a supplied timer handler, post a message to a queue or set an event flag.

RTOSes may provide other features, such as memory management. This is usually in the form of fixed size block management, since that allows real time allocation and freeing. Some RTOSes may also support full variable size block management. However most RTOSes do not support the use of a hardware MMU. Even if the RTOS does supports it (OSE does), then the real time applications generally do not make use of such support, since they are written to be portable to hardware without an MMU.

Mapping RTOS to EKA2

I will now assume that the real time application expects a flat address space with no protection, as would be the case on hardware with no MMU. To get this behavior under EKA2, the application must run in supervisor mode in the kernel address space. The obvious way to do this is to make the real time application plus personality layer a kernel extension; this will also ensure that it is started automatically, early on in the boot process.

In general, a real time application will have its own memory management strategy and will not wish to use the standard Symbian OS memory management system. To this end, at boot time the personality layer will allocate a certain fixed amount of RAM for use by the real time application. For a telephony stack this will be around 128 KB-256 KB.

The personality layer can do this either by including the memory in the kernel extension's. bss section or by making a one-time allocation on the kernel heap at boot time. Depending on the LRTA's requirements, the personality layer may manage this area of RAM (if the RTOS being emulated provides memory management primitives) or the LRTA may manage it.

The personality layer will use a nanokernel thread for each RTOS thread. It will need a priority-mapping scheme to map RTOS priorities, of which there are typically 64 to 256 distinct values, to the 64 nanokernel priorities. As long as the real time application does not have more than 35 threads running simultaneously (which is usually the case) it should be possible to produce a mapping scheme that allows each thread to have a distinct priority. If you do need to exceed this limit, you will have to fold some priorities together.

The nanokernel does not support most of the synchronization and communication primitives provided by standard RTOSes. You will have to implement any such primitives required by the LRTA in the personality layer. This basically means that the personality layer has to define new types of object on which threads may wait. This in turn means that new N-states (discussed in Chapter 3, *Processes and Threads*) must be defined to signify that a thread is waiting on an object of a new type; generally each new type of wait-object will require an accompanying new N-state. To make a thread actually block on a new type of wait object, you would use the following nanokernel function:

```
void NKern::NanoBlock(TUint32 aTimeout, TUint aState, TAny* aWaitObj);
```

You should call this function with preemption disabled since it removes the current thread from the scheduler ready list. The parameters are as follows:

- aTimeout is the maximum time for which the thread should block, in nanokernel timer ticks; a zero value means wait for ever. If the thread is still blocked when the timeout expires the state handler (which I discuss next) will be called
- aState is the new N-state corresponding to the wait object. This value will be written into the `NThreadBase::iNState` field
- aWaitObj is a pointer to the new wait object. This value will be written into the `NThreadBase::iWaitObj` field.

You can use the `TPriListLink` base class of `NThreadBase` to attach the thread to a list of threads waiting on the object; note that you must do this after calling the `NanoBlock()` function. Since preemption is disabled at this point, a reschedule will not occur immediately but will be deferred to the next point at which preemption is re-enabled.

Every thread that wants to use a new type of wait object must have a nanokernel state handler installed to handle operations on that thread when it is waiting on the new type of object. A nanokernel state handler is a function with the following signature:

```
void StateHandler(NThread* aThread, TInt aOp, TInt aParam);
```


(C) Copyright Nokia 2014. All rights reserved.

The parameters are as follows:

- aThread is a pointer to the thread involved
- aOp indicates which operation is being performed on the thread (a value from enum `NThreadBase::NThreadOperation`)
- aParam is a parameter that depends on aOp.

The state handler is always called with preemption disabled. The possible values of aOp are described in the following table:

aOp	Description
`ESuspend`	Called if the thread is suspended while not in a critical section and not holding a fast mutex. Called in whichever context `NThreadBase::Suspend()` was called from. Requested suspension count is passed as aParam.
`EResume`	Called if the thread is resumed while actually suspended and the last suspension has been removed. Called in whichever context `NThreadBase::Resume()` was called. No parameter.
`EForceResume`	Called if the thread has all suspensions cancelled while actually suspended. Called in whichever context `NThreadBase::ForceResume()` was called. No parameter.
`ERelease`	Called if the thread is released from its wait. This call should make the thread ready if necessary. Called in whichever context `NThreadBase::Release()` was called. aParam is the value passed into `NThreadBase::Release()` to be used as a return code. If aParam is nonnegative this indicates normal termination of the wait condition. If it is negative it indicates early or abnormal termination of the wait; in this case the wait object should be rolled back as if the wait had never occurred. For example a semaphore's count needs to be incremented if aParam is negative since in that case the waiting thread never acquired the semaphore.
`EChangePriority`	Called if the thread's priority is changed. Called in whichever context `NThreadBase::SetPriority()` is called. This function should set the `iPriority` field of the thread, after doing any necessary priority queue manipulations. The new priority is passed as aParam.
`ELeaveCS`	Called in the context of the thread concerned if the thread executes `NKern::ThreadLeaveCS()` with an unknown `iCsFunction`, that is negative but not equal to `ECsExitPending`. The value of `iCsFunction` is passed as aParam.
`ETimeout`	Called if the thread's wait timeout expires and no timeout handler is defined for that thread. Called in the context of the nanokernel timer thread (DfcThread1). No parameter. This should cancel the wait and arrange for an appropriate error code to be returned. The handler for this condition will usually do the same thing as the handler for `ERelease` with a parameter of `KErrTimedOut`.

When a thread's wait condition is resolved, you should call the following nanokernel method:

```
void NThreadBase::Release(TInt aReturnCode);
```

The parameter is usually `KErrNone` if the wait condition is resolved normally (for example the semaphore on which it is waiting is signaled). A negative parameter value is used for an abnormal termination - in this case the wait object may need to be rolled back. You should call the Release method with preemption disabled. It performs the following actions:

- Calls the state handler with `ERelease` and the return code. If the return code is negative this should remove the thread from any wait queues and roll back the state of the wait object. In any case it should call `NThreadBase::CheckSuspendThenReady()` to make the thread ready again if necessary
- Sets the `NThreadBase::iWaitObj` field to NULL and cancels the wait timer if it is still running
- Stores the supplied return code in `NThreadBase::iReturnCode`.

The final piece of the puzzle for simple personality layers is the mechanism by which ISRs cause threads to be scheduled. Most RTOSes allow ISRs to directly perform operations such as semaphore signal, queue post and set event flag - usually using the same API as would be used in a thread context. The EKA2 nanokernel does not allow this - ISRs may only queue an IDFC or DFC. The way to get round this limitation is to incorporate an IDFC into each personality layer wait object. The personality layer API involved then needs to check whether it is being invoked from an ISR or a thread, and in the first case it will queue the IDFC. The API also might need to save some other information for use by the IDFC; for example it may need to maintain a list of messages queued from ISRs, a count of semaphore signals from ISRs or a bit mask of event flags set by ISRs. Checking for invocation from an ISR can be done using the `NKern::CurrentContext()` API.

```
class NKern
    {
```


(C) Copyright Nokia 2014. All rights reserved.

```
    enum TContext
        {
        EThread=0, // execution in thread context
        EIDFC=1, // execution in IDFC context
        EInterrupt=2, // execution in ISR context
        EEscaped=KMaxTInt // emulator only
        };

    // Return a value indicating the current execution
    // context. One of the NKern::TContext enumeration
    // values is returned.
    IMPORT_C static TInt CurrentContext();
    };
```

Hardware interrupts serviced by the LRTA need to conform to the same pattern as those serviced by Symbian OS extensions or device drivers. This means that the standard preamble must run before the actual service routine, and the nanokernel interrupt postamble must run after the service routine to enable reschedules to occur if necessary. You can do this by calling the standard `Interrupt::Bind()` provided by the base port during LRTA initialization (possibly via a personality layer call if it must be called from C code).

I will illustrate these points with an example. Consider the implementation of a simple counting semaphore, with the following properties:

1. Semaphores are created at system startup, with an initial count of zero
2. Any personality layer thread may wait on any semaphore. The wait operation causes the count to be decremented; if it becomes negative the thread blocks
3. Any personality layer thread may signal any semaphore. The signal operation causes the count to be incremented; if it was originally negative the highest priority waiting thread is released
4. Semaphores may be signaled directly by ISRs.

Since we have a new wait object, we need a new N-state and a state handler to deal with threads in the new state. We make the following declaration for personality layer threads:

```
class PThread : public NThread
    {
public:
    enum PThreadState
        {
        EWaitSemaphore = NThreadBase::ENumNStates,
        };
public:
    static void StateHandler(NThread* aThread, TInt aOp, TInt aParam);
    void HandleSuspend();
    void HandleResume();
    void HandleRelease(TInt aReturnCode);
    void HandlePriorityChange(TInt aNewPriority);
    void HandleTimeout();
    };
```

The semaphore object itself must have a count and a list of waiting threads. This list must be priority ordered since threads are woken up in priority order. Also, threads that are suspended while waiting must be kept on a separate list since they are no longer eligible for execution when the semaphore is signaled. We make the following definition for our semaphore class:

```
class PSemaphore
    {
public:
    static void CreateAll();
```


(C) Copyright Nokia 2014. All rights reserved.

```
    PSemaphore();
    void WaitCancel(PThread* aThread);
    void SuspendWaitingThread(PThread* aThread);
    void ResumeWaitingThread(PThread* aThread);
    void ChangeWaitingThreadPriority(PThread* aThread,

    TInt aNewPriority);
    void Signal();
    void ISRSignal();
    static void IDfcFn(TAny*);
public:
    TInt iCount; // semaphore count
    TInt iISRCount; // number of pending ISR signals
    TDfc iIDfc; // IDFC to signal semaphore from ISR
    SDblQue iSuspendedQ; // suspended waiting threads
    // list of waiting threads
    TPriList<PThread, KNumPriorities> iWaitQ;
    static TInt NumSemaphores;
    static PSemaphore* SemaphoreTable;
    };
```

I will first examine semaphore creation:

```
void PSemaphore::CreateAll()
    {
    NumSemaphores = semaphore_count;
    SemaphoreTable = new PSemaphore[semaphore_count];
    __NK_ASSERT_ALWAYS(SemaphoreTable != NULL);
    }

PSemaphore::PSemaphore()
: iCount(0),
iISRCount(0),
iIDfc(iDfcFn, this)
    {
    }
```

The `PSemaphore::CreateAll()` method is called when the personality layer initializes. It allocates and initializes an array of semaphore_count (this is a configuration parameter) semaphore objects on the kernel heap. Since personality layer initialization takes place in an extension entry point, it occurs in the context of the supervisor thread (see Chapter 16, *Boot Processes*) and so it can make use of the kernel heap.

The second method, the semaphore's constructor, initializes the count and ISR count to zero, the wait queues to empty and sets up the callback function for the IDFC.

Now let's consider a thread waiting on a semaphore:

```
extern "C" int semaphore_wait(int sem_id, int time_ticks)
    {
    if (time_ticks < WAIT_FOREVER)
        return BAD_TIME_INTERVAL;
    if (TUint(sem_id) >= TUint(PSemaphore::NumSemaphores))
        return BAD_SEM_ID;
    PSemaphore* s = PSemaphore::SemaphoreTable + sem_id;
    PThread* t = (PThread*)NKern::CurrentThread();
    TInt r =OK;
```


(C) Copyright Nokia 2014. All rights reserved.

```
    NKern::Lock();
    if (time_ticks == NO_WAIT)
        {
        if (s->iCount <= 0)
            r = TIMED_OUT;
        else
            --s->iCount;
        NKern::Unlock();
        return r;
        }
    if (--s->iCount < 0)
        {
        TInt waitp;
        waitp = (time_ticks == WAIT_FOREVER) ? 0 : time_ticks;
        NKern::NanoBlock(waitp, PThread::EWaitSemaphore, s);
        s->iWaitQ.Add(t);
        NKern::PreemptionPoint();
        if (t->iReturnValue == KErrTimedOut)
            r = TIMED_OUT;
        }
    NKern::Unlock();
    return r;
    }
```

We have declared the public API provided by the personality layer with C linkage since most LRTA code is written in C. The function first validates the semaphore ID and timeout parameters, and then looks up the semaphore control block from the ID (just a simple array index in this case). If a non-blocking call is required (NO_WAIT) the semaphore count is checked; if positive the count is decremented and OK returned, otherwise the count is left alone and TIMED_OUT returned. If blocking is permitted the count is decremented. If it becomes negative the current thread is blocked for a maximum of time ticks nanokernel ticks (for ever if WAIT_FOREVER). The blocked thread is placed in the new N-state `PThread::EWaitSemaphore`. When the thread eventually wakes up the wake up reason (iReturnValue) is checked. If this is `KErrTimedOut`, the wait was timed out so TIMED_OUT is returned to the caller, otherwise the wait ended normally with the semaphore being signaled, so OK is returned.

The code to signal a semaphore is complicated by the fact that the API must work whether it was called from a thread or an ISR:

```
extern "C" int semaphore_signal(int sem_id)
    {
    if (TUint(sem_id) >= TUint(PSemaphore::NumSemaphores))
        return BAD_SEM_ID;
    PSemaphore* s = PSemaphore::SemaphoreTable + sem_id;
    TInt c = NKern::CurrentContext();
    if (c == NKern::EInterrupt)
        {
        s->ISRSignal();
        return OK;
        }
    NKern::Lock();
    s->Signal();
    NKern::Unlock();
    return OK;
    }

void PSemaphore::Signal()
    {
    if (++iCount <= 0)
        {
```


(C) Copyright Nokia 2014. All rights reserved.

```
        // must wake up next thread
        PThread* t = iWaitQ.First();
        iWaitQ.Remove(t);
        t->Release(KErrNone);
        }
    }

void PSemaphore::ISRSignal()
    {
    if (NKern::LockedInc(iISRCount)==0)
        iIDfc.Add();
    }

void PSemaphore::IDfcFn(TAny* aPtr)
    {
    PSemaphore* s = (PSemaphore*)aPtr;
    TInt count;
    count = (TInt)NKern::SafeSwap(0, (TAny*&)s->iISRCount);
    while (count--)
        s->Signal();
    }
```

Again we declare the public API with C linkage. The function begins by validating the semaphore ID argument and looking up the `PSemaphore` object from the ID. Then it checks the current execution context. If it was called from within an ISR, it calls ISRSignal(), otherwise it calls `Signal()` with preemption disabled.

`Signal()` increments the semaphore's count; if the count was originally negative, it takes the first thread off the wait queue (which is priority ordered, so it gets the highest priority waiting thread) and releases it. Note that the wait queue cannot be empty since a semaphore count of-N implies that there are N threads on the wait queue.

If the semaphore is signaled from an ISR, we can't wake up the first waiting thread immediately, since neither the semaphore wait queue nor the thread ready list is guaranteed to be consistent during ISRs. Instead, we atomically increment the `iISRCount` field. If `iISRCount` was initially zero the semaphore's IDFC is queued. The IDFC atomically reads `iISRCount` and zeros it again, then signals the semaphore the required number of times.

Finally, let's examine the state handler used for personality layer threads, and its associated methods.

```
void PThread::StateHandler(NThread* aThread, TInt aOp, TInt aParam)
    {
    PThread* t = (PThread*)aThread;
    switch (aOp)
        {
        case NThreadBase::ESuspend:
            t->HandleSuspend();
            break;
        case NThreadBase::EResume:
        case NThreadBase::EForceResume:
            t->HandleResume();
            break;
        case NThreadBase::ERelease:
            t->HandleRelease(aParam);
            break;
        case NThreadBase::EChangePriority:
            t->HandlePriorityChange(aParam);
            break;
        case NThreadBase::ETimeout:
            t->HandleTimeout();
```


(C) Copyright Nokia 2014. All rights reserved.

```
            break;
        case NThreadBase::ELeaveCS:
        default:
            __NK_ASSERT_ALWAYS(0);
        }
    }

void PThread::HandleSuspend()
    {
    switch(iNState)
        {
        case EWaitSemaphore:
            ((PSemaphore*)iWaitObj)->SuspendWaitingThread(this);
            break;
        default:
            __NK_ASSERT_ALWAYS(0);
        }
    }

void PThread::HandleResume()
    {
    switch(iNState)
        {
        case EWaitSemaphore:
            ((PSemaphore*)iWaitObj)->ResumeWaitingThread(this);
            break;
        default:
            __NK_ASSERT_ALWAYS(0);
        }
    }

void PThread::HandleRelease(TInt aReturnCode)
    {
    switch(iNState)
        {
        case EWaitSemaphore:
            if (aReturnCode<0)
                ((PSemaphore*)iWaitObj)->WaitCancel(this);
            else
                CheckSuspendThenReady();
            break;
        default:
            __NK_ASSERT_ALWAYS(0);
        }
    }

void PThread::HandlePriorityChange(TInt aNewPriority)
    {
    switch(iNState)
        {
        case EWaitSemaphore:
            ((PSemaphore*)iWaitObj)->ChangeWaitingThreadPriority(this, aNewPriority);
            break;
        default:
        __NK_ASSERT_ALWAYS(0);
        }
    }
```


(C) Copyright Nokia 2014. All rights reserved.

```
void PThread::HandleTimeout()
    {
    switch(iNState)
        {
        case EWaitSemaphore:
            ((PSemaphore*)iWaitObj)->WaitCancel(this);
            break;
        default:
            __NK_ASSERT_ALWAYS(0);
        }
    }

void PSemaphore::WaitCancel(PThread* aThread)
    {
    if (aThread->iSuspendCount == 0)
        {
        iWaitQ.Remove(aThread);
        ++iCount;
        }
    else
        aThread->Deque();
    aThread->CheckSuspendThenReady();
    }

void PSemaphore::SuspendWaitingThread(PThread* aThread)
    {
    // do nothing if already suspended
    if (aThread->iSuspendCount == 0)
        {
        iWaitQ.Remove(aThread);
        ++iCount;
        iSuspendedQ.Add(aThread);
        }
    }

void PSemaphore::ResumeWaitingThread(PThread* aThread)
    {
    aThread->Deque();
    if (--iCount<0)
        iWaitQ.Add(aThread);
    else
        {
        aThread->iWaitObj=NULL;
        aThread->Ready();
        }
    }

void PSemaphore::ChangeWaitingThreadPriority(PThread* aThread, TInt aNewPriority)
    {
    if (aThread->iSuspendCount == 0)
        iWaitQ.ChangePriority(aThread, aNewPriority);
    else
        aThread->iPriority = (TUint8)aNewPriority;
    }
```


(C) Copyright Nokia 2014. All rights reserved.

The state handler calls different `PThread` methods according to which operation is being performed on the thread. Each of these `PThread` methods then performs the appropriate operation on the object on which the thread is waiting, depending on the N-state.

If a thread is suspended while waiting on a semaphore, the code first checks the thread's suspend count. The suspend count is zero if the thread is not suspended and-N if it has been suspended N times. If the thread was already suspended, no action is required. Otherwise we move the thread from the semaphore's wait queue to its suspended queue. We then increment the semaphore count to preserve the invariant that (if negative) it equals minus the number of threads on the wait queue.

If a thread is resumed while waiting on a semaphore, we remove it from the semaphore's suspended queue, where it was placed when it was first suspended. Then we decrement the semaphore's count, balancing the increment when the thread was suspended. If the count becomes negative, we add the thread to the semaphore's wait queue and it remains in the `PThread::EWaitSemaphore` state. If the count is non-negative, we make the thread ready. Note that the state handler is only called when a thread is resumed if all suspensions have been cancelled, so there is no need for us to check the thread's suspend count.

If a thread is released while waiting on a semaphore, we check the reason code for the release. If this reason is `KErrNone`, this is a normal release event - in other words the semaphore has been signalled and the thread's wait condition has been resolved normally. In this case we call `CheckSuspendThenReady()` on the thread, which makes it ready, provided that it is not also explicitly suspended. If the reason code is not `KErrNone`, the wait has been cancelled - for example because the thread was terminated. In this case, we detach the thread from the semaphore - either from the wait queue or suspended queue depending on whether the thread is also suspended. If we detached it from the wait queue, we increment the count, balancing the decrement when the thread waited on the semaphore. Finally, we call `CheckSuspendThenReady()` on the thread. The effect is to reverse the actions taken when the thread waited on the semaphore.

If a timeout occurs on a thread waiting on a semaphore, we take the same action as a release with reason code `KErrTimedOut`, so the semaphore wait is cancelled. If a thread's priority is changed while it is waiting on a semaphore, the thread's position on the semaphore wait queue needs to be adjusted. If the thread is also suspended, its position on the suspended queue needn't be changed, but we do need to modify the thread's `iPriority` field.

Symbian-LRTA communication

If the functionality of the LRTA is to be available to Symbian OS applications, we need a mechanism by which Symbian OS code and the LRTA may communicate with each other. In practice this means:

1. It must be possible for a Symbian OS thread to cause an RTOS thread to be scheduled and vice-versa
2. It must be possible for data to be transferred between Symbian OS and RTOS threads in both directions.

It is usually possible for a Symbian OS thread to make standard personality layer calls (the same calls that RTOS threads would make) to cause an RTOS thread to be scheduled. This is because the nanokernel underlies both types of thread and most *signal* type operations (that is, those that make threads ready rather than blocking them) can be implemented using operations which make no reference to the calling thread, and which are therefore not sensitive to the type of thread they are called from. The semaphore-signal operation in the previous example code falls into this category - a Symbian OS thread could use this to signal a personality layer semaphore.

In the other direction, it is not possible for a personality layer thread to signal a personality layer wait object and have a Symbian OS thread wait on that object. The most straightforward way for RTOS threads to trigger the scheduling of a Symbian OS thread is to enque a DFC on a queue operated by a Symbian OS thread. Another possibility is for the Symbian OS thread to wait on a fast semaphore, which could then be signaled by the RTOS thread; however the DFC method has a better fit with the way device drivers are generally written. A device driver is needed to mediate communication between Symbian OS user mode processes and the LRTA, since the latter runs kernel side.

Data transfer between the two environments must either occur kernelside or via shared chunks. It is not possible for any RTOS thread to access user-side memory via the IPC copy APIs, since these access parts of the Symbian OS `DThread` structure representing the calling thread (for example to perform exception trapping). If you want to use shared chunks, they must be created by a Symbian OS thread, not by a personality layer thread. This is because creating Symbian OS objects such as chunks requires waiting on `DMutex` objects, which only Symbian OS threads may do. The chunks would either be created at personality layer initialization or by the device driver used to interface Symbian OS with the LRTA. Shared chunks can be useful as a way to reduce copying overhead if bulk data transfer is necessary between the two domains.

Some possibilities for the data transfer mechanism are:

- A fairly common architecture for real time applications involves a fixed block size memory manager and message queues for inter-thread communication. The memory manager supports allocation and freeing of memory in constant time. The sending thread allocates a memory block, places data in it and posts it to the receiving thread's message queue. The receiving thread

 (C) Copyright Nokia 2014. All rights reserved.

then processes the data and frees the memory block, or possibly passes the block to yet another thread. It would be a simple proposition to produce such a system in which the memory manager could be used by any thread. In that case, a Symbian OS thread could pass messages to RTOS threads in the same way as other RTOS threads do. Passing data back would involve a special type of message queue implemented in the personality layer. When the personality layer wanted to send a message to a Symbian OS thread, it would enque a DFC. That DFC would then process the message data and free the memory block as usual. This scheme combines the data transfer and scheduling aspects of communication

- You could use any standard buffering arrangement, for example circular buffers, between the LRTA and the device driver. You could prevent contention between threads by the use of nanokernel fast mutexes, on which any thread can wait, or by the simpler means of disabling preemption or interrupts.

Summary

In this chapter I have explored real time systems and the challenges they present to an operating system. I have looked at several solutions to these challenges and examined their pluses and minuses. Then I presented the solutions that we chose for EKA2, looked at the measurements typically made on real time `OSes` and gave the results for EKA2. After that, I discussed the hardware and software issues that need to be considered when building real time applications for EKA2. In the second half of the chapter, I gave a quick overview of the GSM system, while considering how it might be implemented under EKA2. Finally, I showed how personality layers enable existing real time software to be ported to EKA2 with minimum effort, and I gave an example of how you might implement such a personality layer. In the next chapter, I will look at how you can ensure the best performance when using EKA2.

© 2010 Symbian Foundation Limited. This document is licensed under the Creative Commons Attribution-Share Alike 2.0 license. See http://creativecommons.org/licenses/by-sa/2.0/legalcode for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

(C) Copyright Nokia 2014. All rights reserved.