
ARMChair Voice Recorder Design Specification Draft

ABSTRACT

This specification describes the ARMChair Voice Recorder architecture and design. Specifically, it describes the multitasking kernel and device drivers that were developed to support this the application and how the application applies these services to deliver a fully-functional real-time record and playback device with support for bank switching and several effects.

Copyright © 2001 by Chris Lord, Nimit Sawhney, Madhur Joshi and Sajjid Salyani. All Rights Reserved.

Author:	Chris Lord
Software Version:	Proposed
Document Version:	0.0.000
Edit Number:	2
Last Revised:	5/8/2002 11:51 AM
Classification:	Internal Use Only
Location:	/afs/clord/Documents/18-349/Lab 4 Project Report.doc
Date Printed:	5/8/2002 11:51 AM

Table Of Contents

1	Introduction.....	1
1.1	Requirements.....	1
1.1.1	Application.....	1
1.1.2	Kernel.....	1
1.2	Evaluation.....	1
1.2.1	Limitations and Recommendations.....	1
1.2.1.1	Preemption and Thread State.....	1
1.2.1.2	General Interrupt Wrapper.....	2
1.2.1.3	Additional Timers and Finer Granularities.....	2
1.2.1.4	Synchronization Primitives.....	2
2	System Architecture.....	3
2.1	Multitasking Model.....	3
2.1.1	Threads.....	3
2.1.1.1	Thread Control Block.....	3
2.1.1.2	Thread Initialization.....	4
2.1.1.3	Thread State.....	4
2.1.2	Cooperative Multitasking.....	5
2.1.3	Preemptive Multitasking.....	5
2.1.3.1	Periodic Threads.....	5
2.2	Interrupt Handling.....	5
2.3	Executive and Kernel Services.....	7
2.4	Timer Service.....	8
2.4.1	Design and Implementation.....	8
2.4.2	Definitions.....	8
2.4.3	Interface and Operation.....	9
3	Application Architecture.....	10

Revision History

Date	Version	Authors	Description
28-Nov-01	0.0.001	CCL	Initial

1 Introduction

The ARMChair Voice Recorder is a digital voice recorder application designed for the ARM processor running on the 7500 Evaluation Board with the 349 Expansion Board with attached microphone and speaker. The switches provide application controls (start, stop, rewind and fast forward) with feedback through the LEDs, seven-segment display and LCD display. At the core of this application is a preemptive multitasking kernel. This specification describes the architecture and design of the kernel, support services and application.

1.1 Requirements

The project requirements are explained in detail in the lab project description. The most important requirements are included below as well as any specific deviations or reinterpretations.

1.1.1 Application

The ARMChair voice recorder must support recording, playback, rewind, and fast forward. It should sample and play back at a frequency of 8 kHz. In addition—because this was done by a four-member team—the application should support skipping to the beginning and end of the buffer, the selection of multiple sound banks using the keypad, and a DSP effects including amplification, pitch adjustment and reverse playback. Optional features such as an LED graphic equalizer are desirable, but not required.

1.1.2 Kernel

While the project specification suggested an SWI interface to the kernel, it was decided that this was not mandatory or necessary to provide the full functionality of the kernel. Specifically, there was no additional protection or isolation gained by operating in supervisor mode. It also turns out that there is measurably more overhead in managing a kernel through an SWI interface than in the exokernel model followed here. The design described here also has the advantage of simplicity and portability and is much easier to debug using the limited ARM DEMON.

1.2 Evaluation

The application meets the minimum requirements described above. Since this specification is being written at the same time as the some of the final application enhancements (DSP effects, bank switching) the submitted code may differ from the functionality described here.

1.2.1 Limitations and Recommendations

The present kernel is designed to be general but implemented to meet the needs of the application. A good example of this is the choice of constants. The number of priority levels is deliberately small (and would have been smaller had the implementation requirements allowed) because there are only a few threads needed by the application. When the number of priority levels matches the number of threads and each thread uses a unique priority, context switching is optimized. It should be clear, though, that the number of priority levels is not a limitation of the design. Different applications could define a different number to meet different requirements. There are, however, other improvements that could be done to make the kernel more generically useful.

1.2.1.1 Preemption and Thread State

There is an assumption in the submitted code that a preempted thread is running. While this is true, by definition, this does not necessarily correspond to the thread's desired state which is included in the TCB. A problem can arise if a thread sets its desired state to idle (without yielding), continues to run and is then preempted. At the point when the thread resumes, the desired state has been smashed although the thread is not aware of this. If the thread then yields without specifying a target state, it will yield to the ready state rather than the idle state. This problem is

easily addressed by preserving the target state in the TCB for preempted threads, but this change was not made prior to submission. This does problem does not affect the application because the vulnerable sequence isn't used.

1.2.1.2 General Interrupt Wrapper

The existing implementation provides a wrapper around the timer and DMA interrupt handlers that allow context switches to occur immediately following interrupt service. This makes it cumbersome for device driver writers to install new handlers for new devices that initiate event notifications in their interrupt handlers. A generic wrapper and installation service—such as that provided by the DEMON would simplify extensions.

1.2.1.3 Additional Timers and Finer Granularities

For simplicity, the timer block is embedded in the TCB. This limits the number of timer events to one for each thread using the kernel services, although it may be desirable to have any number of different timers for a single thread. Timers should be implemented as events with a blocking facility, rather than as an external service. (The kernel could still use the existing service, although that would be masked from the application.)

The kernel only supports event timers of 1ms granularity. For memory reasons described in Section 2.4, the maximum event timer is limited. It would be useful to provide a timer service using the same design that supports multiple timer granularities, perhaps transparently selecting among the various event queues.

1.2.1.4 Synchronization Primitives

Perhaps the single largest piece of functionality that would enhance the functionality of this code is the integration of synchronization primitives (such as semaphores) in the kernel with support for kernel-managed blocking. Higher-level primitives could be exposed through the Executive using a few kernel structures. In the current implementation, the cooperative multitasking functions are used to coordinate thread interactions, but this requires more complex planning and results in brittle implementations.

2 System Architecture

This chapter describes the architecture and design of the executive, kernel and interrupt components in the project. The executive provides application-level services for multitasking; the kernel implements those services; and the interrupt handlers manage hardware events that could affect multitasking.

The architecture differs slightly from the guidelines in the project specification. Most of these design decisions were made to improve performance or simplicity.

Application	
Device Drivers	Executive
	Kernel
Interrupt Handlers	

2.1 Multitasking Model

The kernel provides an event-driven, fixed-priority, preemptive dispatcher and support for cooperative multitasking. This combination yields a surprisingly flexible set of services for managing real-time application threads, but requires much more planning and coordination on the part of the application developer than a time-slicing kernel. What is sacrificed in convenience is gained in control, and it is the control that is critical in embedded systems.

2.1.1 Threads

2.1.1.1 Thread Control Block

The unit of execution is the thread. The kernel manages a fixed pool of thread control blocks (TCBs). These are allocated when a thread is created and contains all the information necessary to manage a thread's execution context.

Thread Control Block		
Field	Size	Description
listEntry	8	Forward and backward pointers for stringing the TCB on various queues.
pSelf	4	Self-pointer for simple sanity checking of supplied TCB pointers.
nUserSP	4	Thread stack pointer.
nUserPSR	4	Thread CPSR value.
nUserPC	4	Thread PC. This is for information purposes only.
nPriority	4	The priority. Currently all threads must have unique priorities, although this is likely to change.
pStack	4	A point to the top of the thread's stack. Since stacks are full descending, this is the lowest memory address of the stack.
nStackLen	4	The size of the stack in bytes. The stack pointer plus the length yields base of the stack.
nFlags	4	Flags that identify this thread and special handling it may require (for example, the main executive thread runs on the image default stack and so is immune to stack probes).
TimerBlock	n	This is an opaque timer block used to create a timer event using the timer service.
pszName	4	A descriptive thread name for diagnostic displays. The referenced string must remain valid for the lifetime of the thread.
nEnqueueTime	4	The timer value when the thread was enqueued (transitioned to a ready, blocked or idle state). This is a diagnostic field to monitor how long a thread remains in a specific state: the elapsed time from the enqueue time to the dequeue time is the wait time on a queue.
nDequeueTime	4	The timer value when the thread was dequeued. This is also used for diagnostic purposes to track the execution time of a thread: the elapsed time from when a thread was dequeued to when it is again enqueued.
Total	12	

The thread identification (TID) passed to the application is the TCB pointer recast as an unsigned integer. Often this is handled through a layer of indirection that isn't necessary within a single monolithic application. The self pointer provides sufficient and simple validation of TID values before conversion to TCB pointers.

Perhaps surprisingly, there are no general-purpose registers included in the TCB (even the PC is unnecessary and is included for diagnostics purposes only). All registers are saved on the thread's own stack prior to a context switch and restored from that stack when the execution resumes. The only registers necessary to handle this are SP (which is updated with the new thread's stack) and CPSR (which contains the thread's unique flags at the time execution was suspended).

2.1.1.2 Thread Initialization

A new thread's stack is filled with a recognizable pattern so the kernel can monitor stack usage and provide diagnostics when stack overflow is imminent. The base of the stack is initialized with the registers necessary for the first context switch as shown in Figure 2-1. Of special note is the fact the new threads are wrapped by a kernel function, `KernelStartThread`, that takes the thread context (an opaque value passed through from the thread creator to the new thread) and the actual thread function as parameters.

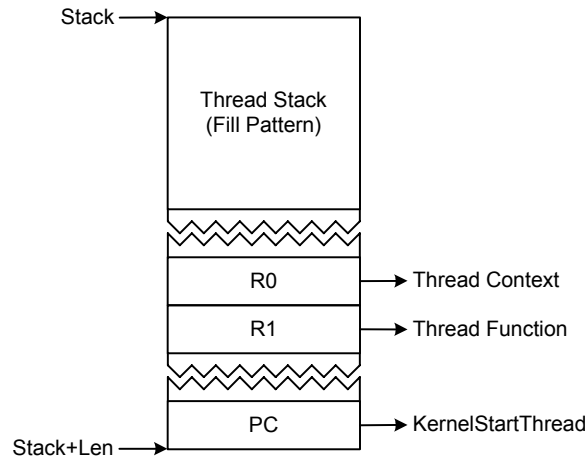


Figure 2-1: Thread Stack Initialization

The wrapper function sets the thread return address in LR as shown in the following code fragment so that threads can exit gracefully simply by returning from the thread function. This design decision was important given the desire to support threads as dynamic entities created and destroyed as necessary during the lifetime of the application. (In this and all other code samples, block comments have been removed to improve readability.)

```

KernelStartThread    adr    lr, KernelExitThread
                   mov    pc, r1

KernelExitThread    bl    ExecExitThread
                   swi    Exit                ; Should never get here

```

`ExecExitThread` is the normal thread shutdown (which can also be called directly by threads which are too deeply nested to return easily). This places the thread permanently on the idle queue and yields control. There is diagnostic code to catch threads that are woken after they've exited, a common application error.

New threads are always created in the idle state. If a thread has thread-level initialization it must perform, then the creator can mark the thread ready and the new thread can yield with an idle state following the initialization. This is actually the preferred sequence because otherwise the very first execution of a thread takes several cycles longer as the wrapper and thread function initialization is performed.

2.1.1.3 Thread State

Threads exist in one of four states: ready, blocked, idle or running. Because synchronization primitives are not part of the current kernel, the blocked state is not used. Instead, blocking on an external events is done while in the idle state. The state transitions are shown in Figure 2-2. Each state except running corresponds to an internal queue. Threads must always exist on the ready queue before they can be promoted to running. Threads may cooperatively yield to other threads (via `yield` or `sleep`), or they may be directed into a new state by external events or other threads (via `set`, `wake` and kernel preemption).

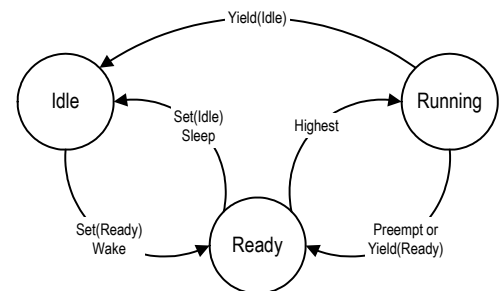


Figure 2-2: Thread State Transitions

The structure for managing the ready threads consists of a list of buckets, one for each priority level. A thread in the ready state (when it is not running) is put in the appropriate bucket based on its priority. The kernel keeps track of the highest occupied bucket and updates this when threads are added and removed. This allows the dispatcher to obtain the next thread without any calculations. The design handles the case when more than one thread has the same priority, but this is not used in the current application. Threads of the same priority level do not preempt each other: the thread running remains running until it yields or is preempted by a higher-priority thread. This is consistent with traditional fixed-priority dispatchers. The idle queue is maintained as a simple list of TCBs.

2.1.2 Cooperative Multitasking

Cooperative multitasking is based on the transfer of control from one thread to another by deliberately yielding. The transfer is directed by manipulating thread states and by selecting thread priorities that reflect the importance of various threads. In the current kernel, cooperative multitasking can be used to augment preemptive multitasking. This is particularly true when one thread (a consumer) wakes another (a producer) and has no useful work it can do until the second thread runs. There are two possible cases that both benefit from cooperative multitasking: the consumer priority is higher than the producer and the producer is higher than the consumer.

Consider the case where the consumer priority is higher. In this case, it is specifically necessary for the consumer to yield to an idle state so the lower-priority thread can run. This is implicitly done when a thread sleeps. Now consider the case where the producer priority is higher. The producer thread will automatically be dispatched (assuming there are no other ready higher priority threads) on the next interrupt event. Since this is not predictable, cycles will be wasted waiting for this to occur. By explicitly yielding, the producer can run immediately. In this case, it is possible for the consumer to yield but remain ready. This ensures it will automatically be dispatched as soon as the producer thread completes.

2.1.3 Preemptive Multitasking

The interrupt handlers for the timer and device interrupts are wrapped to allow context switching upon completion of the interrupt handler if there is a ready thread with a higher priority than the thread currently executing. This check is synchronized with the handlers to only occur if the interrupt handler performed an operation that could have altered the state of ready threads (such as waking a previously idle thread).

When threads directly transition the state of other higher-priority threads to ready, the point at which these threads execute is at most the base period of the timer interrupt (25 or 125us) away, but may be much sooner.

2.1.3.1 Periodic Threads

The kernel provides a general service to assist in preemptive multitasking known as periodic threads. These are threads that are woken on a periodic basis from within the timer interrupt. Unlike the timer services which are described in Section 2.4, periodics operate in the microsecond level of granularity rather than the millisecond level. The minimum period is fixed (currently 25us, but it may be 125us by the time the project is submitted) and all application-specified values are truncated to an integral number of the base period. Periodic threads can easily be enabled and disabled as necessary by other tasks.

The priority of periodics is expected to be high (often the highest), but that is not required by the kernel and is entirely an application decision. If a periodic thread does not have a high priority, then it will not preempt existing threads despite being woken by the timer.

2.2 Interrupt Handling

The style of context switching in this kernel is unlike that described in the lab project specification and bears further discussion. The following description is based on the timer interrupt, but the same wrapper (actually, the same code with different labels) supports other interrupts. The crux of the context switching lies in preserving the LR register and then redirecting where the interrupt returns in user mode.

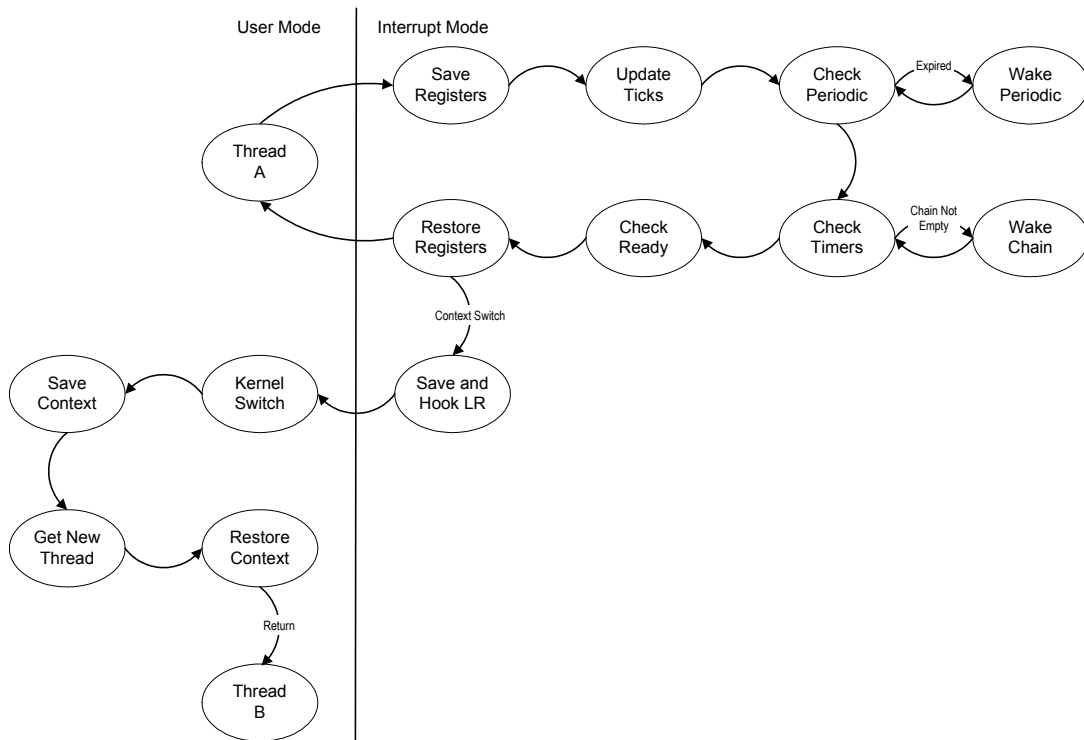


Figure 2-3: Interrupt Handling with Context Switching

The timer interrupt is wrapped by the following code. As a precaution, the preemption of the foreground only occurs if the thread is executing at an address within the current image. If this case is not satisfied, the preemption is deferred until the next interrupt (TimerDispatcher returns 0).

```

TimerHandlerWrapper
    stmfd    sp!, {r0-r4, r12, lr} ; Just save what we hose
    mov     r4, lr                ; This gets preserved by TimerDispatcher
    mov     r0, lr                ; Pass into TimerDispatcher for display
    bl     TimerDispatcher
    mov     lr, r4                ; Get the original LR
    cmp     r0, #0                ; TRUE to switch context, FALSE otherwise
    bne    TimerSwitchContext    ; Handle a context switch
    ldmfd   sp!, {r0-r4, r12, lr} ; No context switch, return
    subs   pc, lr, #4            ; Need to adjust due to pipelining

```

If the TimerDispatcher function reported that a higher priority thread was ready, then we don't return as normal and instead branch to the common context switch routine that follows. This sets the interrupt flag in the SPSR (which is really the user mode CPSR) to keep interrupts disabled through the return to user mode (interrupts were previously enabled or this code wouldn't execute in the first place, a fact used to advantage later).

```

TimerSwitchContext

    mrs     r0, spsr
    orr     r0, r0, #PSR_I_Bit
    msr     spsr, r0

    adr     r0, InterruptedPC    ; Global storage location
    sub     lr, lr, #4           ; Adjust so we can treat it like a user-mode LR
    str     lr, [r0]            ; Save it for later

    adr     lr, TimerStealContext
    ldmfd   sp!, {r0-r4, r12}    ; Restore all but LR
    add     sp, sp, #4           ; Trash saved LR on stack
    movs   pc, lr                ; Returns back to user mode at next instruction

```

The above code preserves the LR in IRQ mode which is the address of the execution that was interrupted when the interrupt occurred and replaces it with a new address. This allows the kernel to regain control following the return from the interrupt and preserve a critical section since the user mode interrupts are now off. Execution resumes below. All user registers are restored except for the PC and the IRQ registers are unbanked so this code operates on the user's own stack. The users LR is saved in the context switch stack frame along with the PC value which indicates where this thread must return. Note that none of the instructions up to reading the CPSR alter the user flags. This is critical since any change to the user flags will result in erroneous execution when this thread runs.

```

str    lr, [sp, #-8]      ; Push it

adr    lr, InterruptedPC ; Recover original PC
ldr    lr, [lr]
str    lr, [sp, #-4]!    ; Push it
sub    sp, sp, #4

stmfd  sp!, {r0-r12}     ; Save all other users registers

mov    r0, sp
mrs    r1, cpsr
mov    r2, lr
bl     KernelSwitchThread

```

KernelSwitchThread will move the current thread to the ready queue (since it was running at the time it was preempted) and return the next highest priority thread ready to run. This routine also performs diagnostics on the supplied SP (is it in the range of the thread, how much stack space is remaining), PC (is it within program image bounds), and CPSR (are interrupts disabled). The actual context switch takes place simply by switching to the new thread's stack and setting the thread's flags..

```

mov    sp, r0             ; Change the user-mode SP
msr    cpsr, r1          ; Update with our task's flags

swi    EnableInts
ldmfd  sp!, {r0-r12,lr,pc} ; And back to the new task
swi    Exit               ; We should never get here...

```

It is known that every thread has interrupts enabled at the time the preemption occurred or prior to a cooperative multitasking call. These were deliberately disabled to guard the context switching in user mode, so the last thing before restoring all registers and returning to the thread is to reenale interrupts.

2.3 Executive and Kernel Services

There are parallels between most executive and kernel services (ExecSetThread and KernelSetThread, for example) that at first glance seem redundant. They aren't. The executive services are specifically designed to be used by an application. The interface exposed is less trusting of the caller and uses higher-level abstractions (such as TIDs and thread handles). Executive services also manage the interrupt flag to ensure atomic operations on kernel structures. In most cases, executive calls eventually use the lower-level kernel equivalent.

Executive Service Summary	
Name	Description
ExecInitialize	Initialize the executive. This sets up initial data structures and pools and calls KernelInitialize.
ExecUninitialize	Cleanup and shutdown the executive after it stops. This calls KernelUninitialize.
ExecStart	Start the executive which begins thread dispatching. There should be at least one thread to run when this is called. This will return when the executive is directed to shutdown or when all other threads have exited. Note that the executive itself runs as an idle-time thread when nothing else is running.
ExecShutdown	Stop the executive. Since the executive runs at the lowest possible thread priority, the time of the shutdown is based on the executive being dispatched.
ExecCreateThread	
ExecSetThread	Marks a thread as ready, idle or blocks. If the thread is already in the desired state, then this has no affect. If this is the current thread, then it simply sets the state in the TCB but does not perform any queue exchanges since the current thread is not on a dispatcher queue. If this isn't the current thread, then the TCB is unspliced from existing queue and added to the new queue. For

	the ready queue, this is in priority order; for all other queues threads are always added to the tail.
ExecYieldThread	Relinquish control to the next highest priority thread ready to run. The desired thread state can optionally be specified in this call.
ExecSleepThread	Create a timer for the specified period and yield to the idle state.
ExecExitThread	Gracefully terminate the current thread.
ExecEnablePeriodicThread	Create a periodic (and typically high-frequency) event notification to wake a specified thread.
ExecDisablePeriodicThread	Disable a periodic timer event. Care must be used in disabling periodic threads since a thread may be woken (but not run) just before the disable applies.

The kernel interface is entirely trusting of the caller, both in the case of parameters and in the required interrupt state. It must be used anywhere the interrupt state must be preserved (such as in the context of an interrupt handler). Since most of the services match those provided by the Executive, they are not summarized here.

2.4 Timer Service

The timer service provides the ability to schedule an event to occur at a relative point in the future by creating a timer and associating it with a notification function.. It provides the services to create and cancel timers and also provides a global capability to start, stop, pause and resume all timeouts. It does this in a manner that is efficient and scalable.

2.4.1 Design and Implementation

It is anticipated that thousands of events could be active at any given moment, each with an associated timer. Traditional timer queues are inefficient for managing many events. Instead, the timeout service provides a ring of $n+1$ entries where n is largest timeout, in milliseconds, necessary (100 milliseconds, for example). Each entry serves as the head of a list of timer events.

The process of starting a timer adds the event to the list attached to entry $m+1$ from the head where m is the timeout in seconds. This guarantees timeouts are always greater than or equal to their desired value. When the timer service wakes up every millisecond (in interrupt context), every event in the list at the current head of the ring has expired and is handled as described below. The head is then advanced. The process of canceling a timer is as simple as unlinking the event from the list based on a supplied timer block.

The ring is managed as a fixed-length array so that adding a timer involves little more than computing the array index and inserting the timer block on the chain. Timer block contents can be pre-initialized to further reduce overhead.

The fixed granularity of this scheme makes this approach unsuitable for some situations, but it is ideal for managing the timer events required by threads because of its very low overhead during interrupt context. If a large range of timeouts is required, then this scheme does required additional memory for each timer list head (eight bytes for every period the maximum timeout is extended). It is perfectly scalable: efficiency is not tied to the size of the ring.

2.4.2 Definitions

Components provide the necessary memory for any event that needs a timeout in the form of a timer block that may, for example, be part of the thread control block. The block is opaque to the caller and is as follows:

Timer Control Block		
Field	Size	Description
listEntry	8	Forward and backward pointers for stringing the timer block on the appropriate bucket list.
pFunction	4	Function to call when the timeout expires. For thread wake ups, this is ExecWakeThread.
nTimeout	4	Context to pass the function. For thread wakeups, this is the thread handle.
nTimeout	4	The timeout value, in milliseconds. This isn't actually used after creation since the timeout is set when the timer is created by inserting the block in the correct bucket.
Total	20	

2.4.3 Interface and Operation

The timer service is independent of the multitasking model, but can be used effectively with it. In the most common case of a thread that needs to sleep for a fixed period of time, it must first create a timer block. This is automatically included as part of the TCB (which does limit threads to a single timer event, but greatly simplifies allocation). The thread must then create the timer and then yield to an idle state. When it wakes up, it must check to see if the wakeup was the result of the timer or some other event. If it wasn't the timer, then the thread must cancel the timer event. Partially for convenience but more so to maintain atomicity, all of this functionality is combined in the ExecSleepThread function as follows (block comments removed):

```
PUBLIC void ExecSleepThread( UINT32 nDelay )
{
    HANDLE hTimeout;

    KernelDisableInterrupts();
    pKernelData->pCurrentThread->nState = THREAD_K_STATE_IDLE;
    hTimeout = TimerCreateTimeout(&pKernelData->pCurrentThread->listTimeout,
        nDelay, ExecWakeupThread, (UINT32)pKernelData->pCurrentThread);
    ExecSwitchThread();
    TimerCancelTimeout(hTimeout);
    return;
}
```

Atomicity is critical to these and other functions that manipulate thread state. In this case, it is necessary to prevent short event times from triggering and waking the current thread before it has idled and therefore causing the thread to miss the event. This can result in dead threads: threads that are idle even though the application thinks they should be ready. This is also one of the reasons that diagnostics were included to track the amount of time threads remain in any given state.

Timer Service Summary	
Name	Description
TimerCreateTimeout	Create a timer and return a handle to it so it can be canceled.
TimerCancelTimeout	Cancel an existing timer. If the specified timer isn't active, this has no effect.
TimerPauseTimeouts	Temporarily stop all timeouts. This is a diagnostic function that defers all timeouts by an amount equal to the time between the pause and the resume functions.
TimerResumeTimeouts	Resume timer events. During the paused state, timers do not advance so there is no barrage of expirations when timeouts are resumed.

3 Application Architecture

Primarily three threads are used: control, record and playback. The record and playback threads are of high priority. The control thread is assigned a relatively low priority. When record is selected, every 125 microseconds (or 8KHz), the Timer Interrupt wakes the Record thread to read a sample from the microphone input. For this reason, the Record thread has to be made real-time. The record thread reads a sample from the A/D converter, does the format conversion (8 bit unsigned to 16 bit signed), stores this value in a buffer, and then relinquishes control to the Control Thread.

The control thread wakes up the playback thread when the appropriate switch is toggled. The playback thread sets up the sound card and depending on the bank which is selected, outputs sound to the DMA buffer. The playback thread yields after filling the DMA buffers. Every time a DMA buffer is emptied (once the buffer has been played) the DMA raises an interrupt – thus waking up the Playback thread to fill in the buffer. Since there are two DMA buffers, the DMA plays one while the other is being filled and the output is uninterrupted.

The control thread continuously runs and checks if any switches are toggled. If this is the case, the appropriate thread (Record or Play) is woken up. The control thread also determines which banks are to be played or recorded. This is done by checking for user input from the keypad. The control thread ensures that only one of the other two thread are active at any given time.

Even when a higher priority thread is active, it yields control for a certain period of time, thus allowing the control thread to run and check for any user input. The control thread is also responsible for shutting down the kernel and exiting the program if the user toggles the appropriate switch.

For bank switching, the control thread sets a global variable to select the bank the user wishes to use. The playback and record threads use this variable to perform their operation on the corresponding buffer.