Build a super simple tasker

By Miro Samek
Founder and CTO
Quantum Leaps, LLC

Robert Ward
Founder
The C Users Journal
Principal Software Engineer
Netopia Inc.

Almost all embedded systems are event-driven; most of the time they wait for some event such as a time tick, a button press, a mouse click, or the arrival of a data packet. After recognising the event, the systems react by performing the appropriate computation. This reaction might include manipulating the hardware or generating secondary, "soft" events that trigger other internal software components. Once the event-handling action is complete, such reactive systems enter a dormant state in anticipation of the next event.1

Ironically, most real-time kernels or RTOSes for embedded systems force programmers to model these simple, discrete event reactions using tasks structured as continuous endless loops. To us this seems a serious mismatch—a disparity that’s responsible for much of the familiar complexity of the traditional real-time kernels.

In this article we’ll show how matching the discrete event nature typical of most embedded systems with a simple run-to-completion (RTC) kernel or "tasker" can produce a cleaner, smaller, faster, and more natural execution environment. In fact, we’ll show you how (if you model a task as a discrete, run-to-completion action) you can create a prioritised, fully preemptive, deterministic-real-time kernel, which we call Super Simple Tasker (SST), with only a few dozen lines of portable C code.2

Such a real-time kernel is not new; similar kernels are widely used in the industry. Even so, simple RTC schedulers are seldom described in the trade press. We hope that this article provides a convenient reference for those interested in such a lightweight scheduler. But more importantly, we hope to explain why a simple RTC kernel like SST is a perfect match for execution systems built around state machines including those based on advanced UML statecharts. Because state machines universally assume RTC execution semantics, it seems only natural that they should be coupled with a scheduler that expects and exploits the RTC execution model.

We begin with a description of how SST works and explain why it needs only a single stack for all tasks and interrupts. We then contrast this approach with the traditional real-time kernels, which gives us an opportunity to re-examine some basic real-time concepts. Next, we describe a minimal SST implementation in portable ANSI C and back it up with an executable example that you can run on any x86-based PC. We conclude with references to an industrial-strength single-stack kernel combined with an open-source state-machine-based framework, which together provide a deterministic execution environment for UML state machines. We’ll assume that you’re familiar with basic real-time concepts, such as interrupt processing, context switches, mutual exclusion and blocking, event queues, and finite state machines.

Preemptive multi-tasking with a single stack
Conventional real-time kernels maintain relatively complex execution contexts (including separate stack spaces) for each running thread or task, as explained in the excellent book MicroC/OS-II: The Real-Time Kernel by Jean Labrosse.3 Keeping track of the details of these contexts and switching among them requires lots of bookkeeping and sophisticated mechanisms to implement the context switch magic. The kernel we’ll describe can be ultra simple because it doesn’t need to manage multiple stacks and all of their associated bookkeeping.

By requiring that all tasks run to completion and enforcing fixed-priority scheduling, we can instead manage all context information using the machine’s natural stack protocol. Whenever a task is preempted by a higher-priority task, the SST scheduler uses a regular C-function call to build the higher-priority task context on top of the preempted-task context. Whenever an interrupt preempts a task, SST uses the already established interrupt stack frame on top of which to build the higher-priority-task context, again using a regular C-function call. This simple form of context management is adequate because we’re insisting that every task, just like every interrupt, runs to completion. Because the preempting task must also run to completion, the lower-priority context will never be needed until the preempting task (and any higher-priority tasks that might preempt it) has completed—and at which time the preempted task will naturally be at the top of the stack, ready to be resumed.

The first consequence of this execution profile is that an SST task cannot be an endless loop, unlike most tasks in traditional kernels.3 Instead, an SST task is a regular C-function that runs to completion and returns, thus removing itself from the execution stack as it completes. An SST task can be activated only by an ordinary C-function call from the SST scheduler and always returns to the scheduler upon completion. The SST scheduler itself is also an ordinary C-function that’s called each time a preemptive event occurs and that runs to completion and returns once all tasks with priorities higher than the preempted task have completed.

The second consequence of the SST execution profile is that events associated with lower-priority tasks must be queued until the higher-priority tasks complete. The scheduling algorithm will work with a wide range of prioritisation mechanisms; here we assume the simplest. For the sake of this discussion we’ll assume a priority numbering scheme in which SST tasks have priorities numbered from 1 to SST_MAX_PRIO, inclusive, and a higher number represents a higher urgency. We reserve the priority level 0 for the SST idle loop, which is the only part of SST structured as an endless loop.

Simply calling a task function for every preemption level may seem too naive to work, but the
The high-priority task, which continues (9).

Obviously, synchronous pre-emptions are not limited to only one level. If the high-priority task were to post an event to a still higher-priority task at point (5) of Figure 2, the high-priority task would be synchronously preempted and the scenario would recursively repeat itself at a higher level of priority.

Figure 3 illustrates the asynchronous preemption scenario caused by an interrupt. Initially a low-priority task is executing and interrupts are unlocked (1). An asynchronous event interrupts the processor (2). The processor immediately preempts any executing task and starts executing the ISR (interrupt service routine). The ISR executes the SST-specific entry (3), which saves the priority of the interrupted task on the stack and raises the current priority to the ISR level. The ISR performs its work (4) which includes posting an event to the high-priority task (5). Posting an event engages the SST scheduler, but the current priority (that of the ISR) is so high that the scheduler returns immediately, not finding any task with priority higher than an interrupt. The ISR continues (6) and finally executes the SST-specific exit (7). The exit code sends the end-of-interrupt (EOI) instruction to the interrupt controller, restores the saved priority of the interrupted task, and invokes the SST scheduler (8). Now, the scheduler detects that the high-priority task is ready to run, so it enables interrupts and calls the newly ready, high-priority task (9). Note that the SST scheduler doesn’t return. The high-priority task runs to completion (10) unless it also gets interrupted. After completion, the high-priority task naturally returns to the scheduler (11). The scheduler checks for more higher-priority tasks to execute (12) but doesn’t find any and returns. The original interrupt returns to the low-priority task (13), which has been asynchronously preempted all that time. Note that the interrupt return (13) matches the interrupt call (2). Finally, the low-priority task runs to completion (14).

It’s important to point out that conceptually the interrupt handling ends in the SST-specific interrupt exit (8), even though the interrupt stack frame still remains on the stack and the IRET instruction has not yet executed. (The IRET instruction is understood here generically to mean a specific CPU instruction that causes hardware interrupt return.) The interrupt ends because the EOI instruction is issued to the interrupt controller and the interrupts get re-enabled inside the SST scheduler. Before the EOI instruction, the interrupt controller allows only interrupts of higher priority than the currently serviced interrupt. After the EOI instruction and subsequent call

### Synchronous and Asynchronous Preemptions

As a fully preemptive kernel, SST must ensure that at all times the CPU executes the highest-priority task that’s ready to run. Fortunately, only two scenarios can lead to readying a higher-priority task:

A lower-priority task posts an event to a higher-priority task: SST must immediately suspend the execution of the lower-priority task and start the higher-priority task. We call this type of preemption synchronous preemption because it happens synchronously with posting an event to the task’s event queue.

An interrupt posts an event to a higher-priority task than the interrupted task:

- Upon completion of the interrupt service routine (ISR), the SST must start execution of the higher-priority task instead of resuming the lower-priority task. We call this type of preemption asynchronous preemption because it can happen any time interrupts are not explicitly locked.

Figure 2 illustrates the synchronous preemption scenario caused by posting an event from a low-priority task to a high-priority task. Initially a low-priority task is executing (1). At some point during normal execution, the low-priority task posts an event to a high-priority task, thus making the high-priority task ready for execution. Posting the event engages the SST scheduler (2). The scheduler detects that a high-priority task has become ready to run, so the scheduler calls (literally a simple C-function call) the high-priority task (3). Note that because the SST scheduler launches the task on its own thread, the scheduler doesn’t return until after the higher-priority task completes. The high-priority task runs (4), but at some time it too may post an event to other tasks. If it posts to a lower-priority task than itself (5), the SST scheduler will again be invoked (again a simple C call), but will return immediately when it doesn’t find any ready tasks with a higher priority than the current task. When this second call to the scheduler returns, the high-priority task runs to completion (6) and naturally returns to the SST scheduler (7). The SST scheduler checks once more for a higher-priority task to start (8), but it finds none. The SST scheduler returns to the low-priority task, which continues (9).

### Synchronous Preemption by a High-Priority Task

![Synchronous preemption by a high-priority task](image)

**Figure 2**

- **(1)** Task preempted
- **(2)** Task resumes
- **(3)** Function call
- **(4)** Function return
- **(5)** SST scheduler

**A lower-priority task posts an event to a higher-priority task:** SST must immediately suspend the execution of the lower-priority task and start the higher-priority task. We call this type of preemption synchronous preemption because it happens synchronously with posting an event to the task’s event queue.

**An interrupt posts an event to a higher-priority task than the interrupted task:**

Upon completion of the interrupt service routine (ISR), the SST must start execution of the higher-priority task instead of resuming the lower-priority task. We call this type of preemption asynchronous preemption because it can happen any time interrupts are not explicitly locked.

**Figure 2** illustrates the synchronous preemption scenario caused by posting an event from a low-priority task to a high-priority task. Initially a low-priority task is executing (1). At some point during normal execution, the low-priority task posts an event to a high-priority task, thus making the high-priority task ready for execution. Posting the event engages the SST scheduler (2). The scheduler detects that a high-priority task has become ready to run, so the scheduler calls (literally a simple C-function call) the high-priority task (3). Note that because the SST scheduler launches the task on its own thread, the scheduler doesn’t return until after the higher-priority task completes. The high-priority task runs (4), but at some time it too may post an event to other tasks. If it posts to a lower-priority task than itself (5), the SST scheduler will again be invoked (again a simple C call), but will return immediately when it doesn’t find any ready tasks with a higher priority than the current task. When this second call to the scheduler returns, the high-priority task runs to completion (6) and naturally returns to the SST scheduler (7). The SST scheduler checks once more for a higher-priority task to start (8), but it finds none. The SST scheduler returns to the low-priority task, which continues (9).

Obviously, synchronous pre-emptions are not limited to only one level. If the high-priority task were to post an event to a still higher-priority task at point (5) of Figure 2, the high-priority task would be synchronously preempted and the scenario would recursively repeat itself at a higher level of priority.

**Figure 3** illustrates the asynchronous preemption scenario caused by an interrupt. Initially a low-priority task is executing and interrupts are unlocked (1). An asynchronous event interrupts the processor (2). The processor immediately preempts any executing task and starts executing the ISR (interrupt service routine). The ISR executes the SST-specific entry (3), which saves the priority of the interrupted task on the stack and raises the current priority to the ISR level. The ISR performs its work (4) which includes posting an event to the high-priority task (5). Posting an event engages the SST scheduler, but the current priority (that of the ISR) is so high that the scheduler returns immediately, not finding any task with priority higher than an interrupt. The ISR continues (6) and finally executes the SST-specific exit (7). The exit code sends the end-of-interrupt (EOI) instruction to the interrupt controller, restores the saved priority of the interrupted task, and invokes the SST scheduler (8). Now, the scheduler detects that the high-priority task is ready to run, so it enables interrupts and calls the newly ready, high-priority task (9). Note that the SST scheduler doesn’t return. The high-priority task runs to completion (10) unless it also gets interrupted. After completion, the high-priority task naturally returns to the scheduler (11). The scheduler checks for more higher-priority tasks to execute (12) but doesn’t find any and returns. The original interrupt returns to the low-priority task (13), which has been asynchronously preempted all that time. Note that the interrupt return (13) matches the interrupt call (2). Finally, the low-priority task runs to completion (14).

It’s important to point out that conceptually the interrupt handling ends in the SST-specific interrupt exit (8), even though the interrupt stack frame still remains on the stack and the IRET instruction has not yet executed. (The IRET instruction is understood here generically to mean a specific CPU instruction that causes hardware interrupt return.) The interrupt ends because the EOI instruction is issued to the interrupt controller and the interrupts get re-enabled inside the SST scheduler. Before the EOI instruction, the interrupt controller allows only interrupts of higher priority than the currently serviced interrupt. After the EOI instruction and subsequent call...
to the SST scheduler, interrupts get unlocked and the interrupt controller allows all interrupt levels, which is exactly the behaviour expected at the task level.

Consequently, asynchronous preemption is not limited to only one level. The high-priority task runs with interrupts unlocked, shown as segment (10) in Figure 3, so it too can be asynchronously preempted by an interrupt, including the same-level interrupt that launched the task at point (9). If the interrupt posts an event to a still higher-priority task, the high-priority task will also be asynchronously preempted and the scenario will recursively repeat itself at a higher-level of priority.

**SST versus traditional kernels**

By managing all contexts in a single stack, SST can run with significantly less RAM than a typical blocking kernel. Because tasks don’t have separate stacks, no unused private stack space is associated with suspended tasks. SST task switches also tend to incur less execution overhead; a traditional kernel doesn’t distinguish between the synchronous and asynchronous preemptions and charges you a uniformly high price for all context switches. Finally, SST uses much simpler, and smaller task control block (TCB) for each task.

Because of this simplicity, context switches in SST (especially the synchronous preemptions) can involve much less stack space and CPU overhead than under any traditional kernel. But even the asynchronous preemptions in SST end up typically using significantly less stack space and fewer CPU cycles. Here’s why.

Upon an interrupt, a traditional kernel must establish a strictly defined stack frame on each private stack into which it saves all CPU registers. Unlike SST, which can exploit the compiler’s natural register-management strategy, a traditional kernel must be prepared to restore all registers when it resumes the preempted task. Often the construction and destruction of these stack frames must be programmed in assembly, because a traditional kernel cannot leave the context switch magic to the C compiler. In contrast, SST doesn’t really care about any particular stack frame or whether the registers are stored immediately upon the ISR entry or stepwise, as needed. The only relevant aspect is that the CPU state be restored exactly to the status before the interrupt, but it’s irrelevant how this happens. This means that the compiler-generated interrupts that most embedded C compilers support are typically adequate for SST, but are often inadequate for traditional kernels. Not only does SST not require any assembly programming, but in this case the compiler-generated interrupt entry and exit code is often superior to custom assembly, because the C compiler is in a much better position to globally optimise interrupt stack frames for specific ISRs. In this respect, SST allows you to take advantage of the C compiler’s capabilities, something a traditional kernel can’t do.

The last point is perhaps best illustrated by an example. All C compilers for ARM processors, for instance, adhere to the ARM Procedure Call Standard (APCS) that prescribes which registers must be preserved across a C-function call and which can be clobbered. The C-compiler-generated ISR entry saves initially only the registers that might be clobbered in a C function, which is only about half of all ARM registers. The rest of the registers get saved later, inside C functions invoked from the ISR, if and only if such registers are actually used. This example of a context save occurring in several steps is perfectly suited to the SST. In contrast, a traditional kernel must save all ARM registers in one swoop upon ISR entry, and if the assembly ISR “wrapper” calls C functions (which it typically does) many registers are saved again. Needless to say, such policy requires more RAM for each private stack and more CPU cycles (perhaps by factor of two) than SST.

On a side note, we would like to emphasise that all traditional blocking kernels in the industry share the problems we’ve described. We refer here to Labrosse’s book MicroC/OS-II not because MicroC/OS-II has any special deficiency in this regard but because his book is an excellent treatment of traditional kernels and is familiar to many embedded systems programmers.

**Interrupt duration in traditional kernels and SST**

If you have some experience with traditional preemptive kernels, you’ll notice that SST seems to encourage programmers to violate time-honoured advice about what to put in an ISR.

Most of us have been taught that an ISR should be as short as possible and that the main work should always be done at the task level. In SST, however, everything appears to be done at the ISR context and nothing at the task context. It seems to be backwards.

But really, the problem centres on the distinction between an interrupt and a task. SST forces us to revise the naïve understanding of interrupt duration as beginning with saving interrupt context on a stack and ending with restoring the interrupt context followed by IRET because, as it turns out, this definition is problematic even for traditional kernels.

Figure 4 shows the per-task data structures maintained by a traditional preemptive kernel where each task has its own stack and a task control block (TCB) for the execution context of each task. In general, an interrupt handler stores the interrupt context on one task’s stack and restores the context, from another task’s stack. After restoring a task’s context into the CPU registers, the traditional scheduler always issues the IRET instruction. The key point is that the interrupt context remains saved on the preempted task’s stack, so the saved interrupt context outlives the duration of the interrupt handler. Therefore defining the duration of an interrupt from saving the interrupt context to restoring the context is problematic.

The situation is not really that much different under a single-stack kernel, such as SST. An ISR stores the interrupt context on the stack, which happens to be common for all tasks and interrupts. After some processing, the ISR calls the scheduler, which internally unlocks interrupts. If no higher-priority tasks are ready to run, the scheduler exits immediately, in which case the ISR restores the context from the stack and returns to the original task exactly at the point of preemption. Otherwise, the SST scheduler calls a higher-priority task and the interrupt context remains saved on the stack, just as in the traditional kernel.
The point here is that the ISR is defined from the time of storing interrupt context to the time of sending the EOI (End Of Interrupt) command to the interrupt controller followed by invoking the scheduler that internally enables interrupts, not necessarily to the point of restoring the interrupt context. This definition is more precise and universal, because under any kernel the interrupt context remains stored on one stack or another and typically outlives the duration of an interrupt’s processing. Of course, you should strive to keep the ISR code to a minimum, but in SST, you should remember that the final call to the SST scheduler is conceptually not a part of the ISR, but rather it constitutes the task level.

The definition of ISR duration is not purely academic, but has important practical implications. In particular, debugging at the ISR level can be much more challenging than debugging at the task level. Indeed, in SST debugging the ISR to the point of sending the EOI command to the interrupt controller and invoking the SST scheduler is as hard as in any traditional kernel, especially when debugging is accomplished through a ROM monitor. However, debugging an SST task is as easy as debugging the main() task, even though a task is called indirectly from the ISR. This is because the SST scheduler launches each task with interrupts unlocked at the CPU level and with the interrupt controller set to enable all interrupts.

**SST implementation**

We first present a minimal stand-alone implementation of SST. The presented code is portable ANSI C, with all CPU and compiler-specific parts clearly separated out. However, the implementation omits some features that might be needed in real-life applications. The main goal at this point is to clearly demonstrate the key concepts while letting you execute the code on any Windows-based PC. We’ve compiled the example with the legacy Turbo C++ 1.01, available as a free download from the Borland Museum.4

The example The SST example demonstrates the multi-tasking and preemption capabilities of SST. This example, shown in **Figure 5**, consists of three tasks and two ISRs. The clock tick ISR produces a “tick event” every 5ms, while the keyboard ISR produces a “key event” every time a key is depressed or released, and at the auto-repeat rate of the keyboard when a key is depressed and held. The two “tick tasks”: tickTaskA() and tickTaskB(), receive the “tick events” and place a letter A or B, respectively, at a random location in the right-hand panel of the screen. The keyboard task kbdTask(), with the priority between tickTaskA() and tickTaskB(), receives the scan codes from the keyboard and sends “colour events” to the tick tasks, which change the colour of the displayed letters in response. Also, the kbdTask() terminates the application when you depress the Esc key.

The left-hand side of the screen in **Figure 5** shows the basic statistics of the running application. The first two columns display the task names and priorities. Note that there are many unused priority levels. The “Calls” column shows the number of calls with 3-digit precision. The last “Preemptions” column shows the number of asynchronous preemptions of a given task or ISR.

The SST example application intentionally uses two independent interrupts (the clock tick and keyboard) to create asynchronous preemptions. To further increase the probability of an interrupt preempting a task, and of an interrupt preempting an interrupt, the code is peppered with calls to a busyDelay() function, which extends the run-to-completion time in a simple counted loop. You can specify the number of iterations through this loop by a command line parameter to the application. You should be careful not to go overboard with this parameter, though, because larger values will produce a task set that is not schedulable and the system will (properly) start losing events.

The following subsections explain the SST code and the application structure. The complete SST source code consists of the header file sst.h located in the include\ directory and the implementation file sst.c located in the source\ directory. The example files are located in the example\ directory, which also contains the Turbo C++ project file to build and debug the application. (NOTE: because the standard keyboard ISR is replaced by the custom one, debugging this application with the Turbo C++ IDE might be difficult.)

**Critical sections in SST**

SST, just like any other kernel, needs to perform certain operations indivisibly. The simplest and most efficient way to protect a section of code from disruptions is to lock interrupts on entry to the section and unlock the interrupts again on exit. Such a section of code is called a critical section.

Processors generally provide instructions to lock/unlock interrupts, and your C compiler must have a mechanism to perform these operations from C. Some compilers allow you to include inline assembly instructions in your C source. Other compilers provide language extensions or at least C-callable functions to lock and unlock interrupts from C.

To hide the actual implementation method chosen, SST provides two macros to lock and unlock interrupts. Here are the macros defined for the Turbo C++ compiler:

```c
#define SST_INT_UNLOCK() \ 
  disable()
#define SST_INT_LOCK() \ 
  enable()
```

In the minimal SST version, we assume the simplest possible critical section: one that unconditionally locks interrupts upon entry and unconditionally unlocks interrupts upon exit. Such simple critical sections should never nest, because interrupts will always be unlocked upon exit from the critical section, regardless of whether they were locked or unlocked before the entry. The SST scheduler is designed to never nest critical sections, but you should be careful when using macros SST_INT_LOCK() and SST_INT_UNLOCK() to protect your own critical sections in the applications. You can avoid this limitation by using smarter
interrupt tickISR() { /*
example application
List 1: SST ISRs from the
From the example application
these macros are used in the
tick ISR and keyboard ISRs
and SST_ISR_EXIT(). The code
in SST macros SST_ISR_ENTRY()
These actions are implemented
actions at each ISR entry and exit.

grammer to insert some simple
a simple “super-loop” and SST
One of the biggest advantages
Interrupt processing in SST
One of the biggest advantages of SST is the simple interrupt processing, which is actually not much more complicated with SST than it is in a simple “super-loop” (a.k.a., main+ISRs). Because SST doesn’t rely in any way on the interrupt stack frame layout, with most embedded C compilers, the ISRs can be written entirely in C. One notable difference between a simple “super-loop” and SST ISRs is that SST requires the programmer to insert some simple actions at each ISR entry and exit. These actions are implemented in SST macros SST_ISR_ENTRY() and SST_ISR_EXIT(). The code snippet in Listing 1 shows how these macros are used in the clock tick and keyboard ISRs from the example application defined in the file examplebsp.c.

Listing 1: SST ISRs from the example application
void interrupt tickISR() { /*
every ISR is entered with interrupts locked */
uint8_t pin; /* temporary variable to store the initial
priority */
SST_ISR_ENTRY(pin, TICK_ISR_PRIO);
SST_post(TICK_TASK_A_PRIO, TICK_SIG, 0); /* post
the Tick to Task A */
SST_ISR_EXIT(pin, TICK_ISR_PRIO, TICK_SIG, 0); /* post
the Tick to Task B */
SST_ISR_EXIT(pin, outportb(0x20, 0x20));
} /***********************************
..............*/ void interrupt kbdISR() { /*
every ISR is entered with interrupts locked */
uint8_t pin; /* temporary variable to store the initial
priority */
uint8_t key = inport(0x60); /* get scan code from the
8042 kbd controller */
SST_ISR_EXIT(key, SST_ISR_ENTRY(pin, KBD_ISR_PRIO));
SST_ISR_EXIT(key, outportb(0x20, 0x20));
} /* kHz Timer ISR */

The SST_ISR_ENTRY() macro is invoked with interrupts locked and performs the following three steps: (1) Saves the initial SST priority into the stack variable ‘pin’ (2) Sets the current SST priority to the ISR level (3) Unlocks the interrupts to allow interrupt nesting The SST_ISR_EXIT() macro is invoked with interrupts unlocked and performs the following four steps: (1) Locks the interrupts (2) Writes the EOI command to the interrupt controller (for example, outportb(0x20, 0x20) writes the EOI to the master 8259A PIC) (3) Restores the initial SST priority (4) Calls the SST scheduler, which performs the “asynchronous preemption,” if necessary. The task control blocks Like other real-time kernels, SST keeps track of tasks in an array of data structures called task control blocks (TCBs). Each TCB contains such information as the pointer to the task function, the task mask (calculated as (1 << (priority - 1)), and the event queue associated with the task. The TCB takes only 8 to 10 bytes, depending on the size of the function pointer. Additionally, you need to provide an event queue buffer of the correct size when you create a task. The TCB used here is optimised for a small, fixed number of priority levels and simple events—restrictions that make sense in a classic embedded environment. Neither of these limitations, however, is required by the SST scheduling algorithm.

Posting events to tasks
In this minimal version of SST, events are represented as structures containing two byte-size elements: an event type identifier (for example, the key-press occurrence), and the parameter associated with the occurrence (for example, the scan code of the key). The events are stored in event queues organised as standard ring buffers. SST maintains the status of all event queues in the variable called the SST ready-set. As shown in Figure 6, the SST ready-set SST_readySet_ is just a byte, meaning that this implementation is limited to eight priority levels. Each bit in the SST_readySet_ represents one SST task priority. The bit number ‘n’ in the SST_readySet_ is 1 if the event queue of the task of priority ‘n’+1 is not empty (bits are numbered 0..7). Conversely, bit number ‘m’ in SST_readySet_ is 0 if the event queue of the task of priority ‘m’+1 is empty, or the priority level ‘m’+1 is not used. Listing 3 shows the event posting function SST_post(), which uses a standard ring buffer algorithm (FIFO). If the event
is inserted to an empty queue (1), the corresponding bit in SST_readySet_ is set (2), and the SST scheduler is invoked to check for the "synchronous preemption" (3).

Listing 3: Event posting in SST
uint8_t SST_post(uint8_t prio, SSTSignal sig, SSTParam par) {
  TaskCB *tcb = &l_taskCB[prio - 1];
  SST_INT_LOCK();
  if (tcb->nUsed__ < tcb->end__) {
    tcb->queue__[tcb->head__] = sig;
    tcb->head__ = (uint8_t)0;
    /* wrap the head */
  }
  /* check for synchronous preemption */
  SST_INT_UNLOCK();
  return (uint8_t)0;
}

The SST scheduler
The SST scheduler is a simple C-function SST_schedule_() whose job is to efficiently find the highest-priority task that is ready to run and run it, if its priority is higher than the currently serviced SST priority. To perform this job, the SST scheduler uses the already described SST_readySet_ and the current priority level SST_currPrio_ shown in Figure 6.
......*
void SST_
mutexUnlock(uint8_t org-Prio) {  
SST_INT_LOCK();  
if (orgPrio < SST_currPrio_) {  
SST_currPrio_ = orgPrio;  
/* restore the saved priority to unlock */  
SST_schedule_(); /* the scheduler unlocks the interrupt internally */  
}  
SST_INT_UNLOCK();  
}

Unlike the simple INT_LOCK macros, the SST mutex interface allows locks to be nested, because the original SST priority is preserved on the stack. Listing 6 shows how the mutex is used in the SST example to protect the non-enterant DOS random number generator calls inside the clock-tick tasks tickTaskA() and tickTaskB().

Listing 6: Priority-ceiling mutex used in the SST example to protect the non-enterant random-number generator

void tickTaskA(SSTEvent e) {  
int8_t x, y; /* call to non-reentrant random-number generator */  
uint8_t mutex; /* mutex object preserved on the stack */  
mutex = SST_schedLock(TICK_TASK_B_PRIO); /* mutex lock */  
x = random(34); /* call to non-reentrant random number generator */  
y = random(13); /* call to non-reentrant random number generator */  
SST_schedUnlock(mutex); /* mutex unlock */  
...}

Starting multi-tasking and the SST idle loop

Listing 7 shows the SST_run() function, which is invoked from main() when the application transfers control to SST to start multi-tasking.

Listing 7: Starting SST multi-tasking

void SST_run(void) {  
(1) SST_start();  
/* start ISRs */  
(2) SST_INT_LOCK();  
(3) SST_currPrio_ = (uint8_t)0; /* set the priority for the SST idle loop */  
(4) SST_schedule_(); /* process all events produced so far */  
(5) SST_INT_UNLOCK();  
(6) for (j) { /* the SST idle loop */  
(7) SST_onIdle(); /* invoke the on-idle callback */  
}  
}

The SST_run() function calls the SST_start() callback (1), in which the application can configure and start interrupts (see file examples/bsp.c). After locking interrupts (2), the current SST priority SST_currPrio_ is set to zero, which corresponds to the priority of the idle loop. (The SST priority is statically initialised to 0xFF.) After lowering the priority the SST scheduler is invoked (4) to process any events that might have accumulated during the task initialisation phase. Finally, interrupts are unlocked (5), and SST_run() enters the SST idle loop (6), which runs when the CPU is not processing any of the one-shot SST tasks or interrupts. The idle loop continuously calls the SST_onIdle() callback (7), which the application can use to put the CPU into a power-saving mode.

Where to go from here

The minimal SST implementation presented in this article is remarkably powerful, given that it takes only 421 bytes of code (see Figure 5), 256 bytes of lookup tables, and several bytes of RAM per task (for the TC8 and the event queue buffer) plus, of course, the stack. This might be all you need for smaller projects. When you tackle bigger systems, however, you will inevitably discover shortcomings in this implementation. From our experience, these limitations have little to do with the multi-tasking model and much to do with the infrastructure surrounding the kernel, such as the limited event parameter size, lack of error handling, and the rather primitive event-passing mechanism.

One thing that you’ll surely discover when you go beyond toy applications suitable for publication in an article is that your task functions will grow to contain more and more conditional code to handle various modes of operation. For example, a user interface task for a digital camera must react differently to button-press events when the camera is in the playback mode than to the same button-press events in the picture-taking mode. The best-known method of structuring such code is through a state machine that explicitly captures different modes of the system.

Endnotes: