

Practical Real-Time Techniques

By Robert Ward

Introduction

Contrary to popular opinion, I believe you can do a great deal with a very simple, homemade scheduler. In this paper I will show you how a few simple tradeoffs will let you create a preemptive, prioritized scheduler in only a few lines of code. This scheduler (SST for Super Simpler Tasker) is ideally suited for machines with limited RAM and program space. It's not a good fit, though for machines like the PIC that have a limited, inaccessible stack. SST adapts well to machines with banked registers, like the Rabbit 2000 and 8051. Because SST is both preemptive and very simple, it offers excellent real-time performance. Finally, the simplicity of the code actually encourages customization and optimization. It's eminently practical to optimize this scheduler to the specifics of the problem and the machine.

More important to me and to this paper, though, SST makes a great vehicle for teaching basic embedded systems real-time techniques and for demonstrating the impact that implementation details can have on total system performance. Unlike in the application world, concurrency in embedded systems (at least in small embedded systems) is seldom about multiple *processes*. Real-time embedded programming is about generating responses in reaction to events. Typically the events arrive via interrupts, and the responses are immediate and short-lived. SST's implementation is ideally suited to this kind of problem. In fact, one could easily argue that SST can barely understand any other kind of problem.

SST is an exercise in design minimalism. At every opportunity, I have opted for simplicity, efficiency, and ease of implementation, rather than generality. My goal is to demonstrate that, even in the most constrained environments, one can still employ a scheduler and still benefit from the structure and predictability that a scheduler can bring to a real time program. The resulting scheduler, though, is only that: a scheduler. It constitutes only the minimum plumbing necessary to dynamically schedule and switch between tasks.

Unlike more general solutions, SST makes no attempt to support any particular computational model. Unlike UNIX processes and other thread models, SST doesn't create separate memory spaces, separate name spaces, or the illusion that each thread runs in a separate process. SST supports task switching. The programmer is responsible for adding scaffolding to create and preserve thread or task instance data and to implement basic concurrency primitives like queues, events, semaphores, and rendezvous. None of this code is large or particularly complex. Some of it depends on idioms unique to SST; most uses only techniques and idioms common to concurrency anywhere. The "disadvantage" is that the programmer must thoroughly understand exactly what they are doing. The "advantage" is that the programmer must thoroughly understand exactly what they are doing -- and can exploit that understanding to tune the system for the problem and machine.

This minimalist approach is not without benefit, though. On most machines, SST requires only these resources:

- Code Space: as little as 300-400 machine instructions. In cases where the application is already tracking event priorities, it may literally add a couple of dozen lines of code.¹
- General purpose RAM: one byte for the scheduler; one byte plus one pointer (or index value) for each task; and one pointer (or index value) for each task in the ready queue. A system with seven tasks

¹ For example, see the The "native" implementation of Miro Samek's Quantum Framework.

running at five levels of priority can require as little as 22 to 25 bytes of general purpose RAM. In some situations the ready queue can be reduced to a single byte.

- **Stack:** One byte, plus one return address, plus enough space to store the complete processor context, for every possible level of preemption. (Plus, of course, the normal stack requirements of the program.)

I have successfully used this scheduler on microcontrollers ranging from the PIC16F877 and 80C51 to an Atmel AT91R40807 (an ARM7tdmi).

In this paper, I'll describe the scheduling behavior and external interface of a base implementation of SST. I'll then use a series of increasingly complex examples to show the SST idioms for various commonly used real-time structures. I'll then detail the base implementation and describe several optimizations and customizations. In addition to examining how implementation details affect the utility of the scheduler, I'll look at how implementation details can make a significant difference in other real-time building blocks. Finally I'll show how to compute meaningful bounds on event sensitivity, response time, and jitter, and how to optimize the scheduler preemption mechanism.

External Characteristics

SST's view of a task and the restrictions it places on task switches are notably different than those of other schedulers. These differences have the advantage of allowing SST to support preemptive scheduling with a single run-time stack. While using the stack for all context information greatly simplifies the scheduler implementation, it creates an environment where task, thread, and priority have different meanings and interact differently than with other schedulers.

SST Task States

SST is a preemptive, prioritized scheduler, but with a twist: in SST, unless a task is preempted or yields, it will run to completion. Moreover, a task may only yield to a higher-priority task, never to some lower priority task. In operating system terms, SST tasks are non-blocking²: i.e., an SST task can never stop and wait for something to happen. More technically, "non-blocking," means that a task cannot be placed in a suspended state while it waits for some internal or external event. Thus, an SST task is always in one of three states:

- Running,
- Preempted by a higher-priority task, or
- Ready.

This state model is at least one state smaller than other schedulers, which usually also recognize a state associated with waiting for I/O, among others. In SST, a task can never yield to an equal or lower priority task, thus the only type of suspension possible is preemption in favor of a higher priority task. Thus, "preempted by a higher priority task" is the *only* suspended state in the model.

Running. As in other single-processor systems, there is always exactly one running process. In SST the running process will always have a priority equal to or higher than any preempted or ready process.

Preempted. These tasks were running, but were suspended by an interrupt (or voluntary yield) so that a higher priority task could be started. Preempted tasks cannot resume until all higher priority tasks have completed.

Ready. These are tasks that need to be started. In reality, a ready task is just an entry in a list. As various parts of the system process events, they recognize that a certain response is necessary and request that the

² Like many CS terms, the meaning of non-blocking seems to depend on context -- and may be used inconsistently by different authors. SST can exhibit the kind of "blocking" associated with priority inversions.

scheduler run the associated task. The scheduler then adds the task's ID (usually a pointer) to the list of "ready" tasks. For a task to get executed, some part of the program must register a request with the scheduler by asking to have the task added to the ready list. A particular task can appear more than once in the ready list. The scheduler will run the task once for every time it has been added to the list.

SST Tasks and Priorities

An SST task is just a function that takes no arguments and returns void. The implementation I'll give later makes this code available to the scheduler as a pointer stored in a Task object (see Listing 2 for this trivial class's interface). Every SST task has an associated priority. A task's priority is just an integer (or other scalar) and is assigned statically when the task is created. In certain situations a running task can temporarily increase its priority. In the implementations I'll show, higher values represent higher priorities.

One consequence of the scheduling algorithm is that there can never be two active tasks (by *active* I mean running or preempted) with the same priority. The scheduler will always complete the current task before starting another task of the same or lower priority. It is, however, both allowable and useful to assign the same priority level to different tasks and for the same task to appear multiple times in the ready list.

While interrupt handlers aren't assigned priorities like tasks, I find it useful during design to think of each ISR as a separate task with very high priority. In systems that allow ISRs to nest, each different level of nesting corresponds to yet another level of "super high" priority.

Scheduling

In SST scheduling decisions are only allowed at two junctures: when a task completes and when a task is preempted. A task can be preempted by an interrupt or can preempt itself by voluntarily calling the scheduler. In all cases, the correct operation of the scheduler depends upon the program observing certain conventions. In particular, each time the program adds a task to the list of run requests, it must immediately call the scheduler. Typically, tasks are added to the list of run requests (the "ready list") during ISRs, so almost every ISR will call the scheduler (thereby effecting SST's preemptive behavior.) If a normal task requests a change to the ready list or to its own priority, then it too must call the scheduler.

When called, SST applies a remarkably simple scheduling algorithm:

If a task in the ready list has higher priority than the current task, start the higher priority task;
if not, resume the current task.

Whenever SST starts a new task, it selects the ready task with the highest priority.

Consequences

If the program always calls the scheduler as required by the above conventions, then at every point in time, SST will be running the highest priority ready task. Unless preempted by a higher priority task, this task will run to completion.

Because the scheduler will always run the current task to completion before starting another task of the same priority, tasks with the same priority are automatically serialized and can share data without concern for synchronization. Because of this trait, I find it useful to think of each SST priority level as the root of a thread. Tasks with a given priority level represent the schedulable units of work in that thread. When compared to other environments, an SST task is just a non-blocking segment of a thread.

Tasks with different priority levels must explicitly synchronize their access to shared data.

No task can run unless all eligible (ready or suspended) higher priority tasks have completed. It is helpful to think of lower priority tasks as running in the "gaps" between the executions of higher priority tasks. These gaps, though, are created by tasks terminating, not by some scheduler-imposed time-slicing discipline. The scheduler is not fair; if the queue does not empty periodically, some tasks will never execute. Any task that fails to terminate will block all tasks of equal or lower priority. Except for the lowest priority task, all SST tasks must be transient.

A task is just a normal function. Thus, tasks and their local variables have the same life. When a task ends, all local data disappears. If a task needs access to state information that persists across separate executions, then the task must take responsibility for setting up appropriate global storage. This is admittedly not an approach that scales well, but the scheduler is intended for small environments.

The Scheduler Interface

The interface to SST is quite sparse. Listing 1 shows the complete interface for the scheduler. The most important methods are `Sst::add(Task *)`, which adds a task to the ready list and `Sst::run_next()` which invokes the scheduler, suspending the current task, at least until a new scheduling choice can be made. (The `match_priority_of()` method can be used to implement priority ceiling protocols.) Listing 2 shows the interface for class `Task`. Application code uses only the constructor from the `Task` interface. The remaining methods are of interest primarily to the scheduler and can wait until I detail the scheduler's implementation.

Listing 1 -- The Scheduler Interface

```
class Sst {
public:
    static const int max_queue_len = SST_MAX_Q_LEN;

    Sst(int qsize = max_queue_len); // the constructor

    bool add(Task * item); // add a task to the ready queue
    void run_next(void); // the scheduler/dispatcher entry point
    void match_priority_of(Task * target);

private:
    static Sst_priority_t current_priority;
    Rdq * ready_queue;
};
```

Listing 2 -- The Task Interface

```
class Task {
    friend class Sst;

public:
    Task( Sst_priority_t pri=0, void (* code )( void)=' \0' );
        // assigns a priority and a code body to a task.

    Sst_priority_t get_priority( void ) const; // used by the scheduler

private:
    void (* module )( void );
    Sst_priority_t priority;

    void launch( ) const; // invokes the task's code body
};
```

Using SST

Initialization and Protocols

To use SST, a program usually needs to create these four resources:

- Appropriate interrupt handlers, each with appropriate linkage to the scheduler. I will discuss the necessary conventions later. Interrupt handlers are not absolutely necessary, but the scheduler doesn't offer any meaningful preemption without them.
- A void/void function for each schedulable task. This function needs to be globally visible. I will refer to this as the task's code body.
- A Task object for each schedulable task. This object associates a priority with the code body for each task.
- An instance of the scheduler (i.e., an Sst object).

Generally both the Task objects and the scheduler will need to be defined so that they are globally visible. Once these resources have been created, the program should make ready (i.e., `Sst::add()`) any startup tasks and then invoke the scheduler.

(I will present most of the code in this paper in a subset of C++. I do so, because I think the key ADTs are more clearly described as classes. I use C++, though, only as a better C. I make no use of inheritance, nor polymorphism for two reasons: first, the concurrent relationships in the scheduler are difficult enough to understand; second, the few C++ compilers available for tiny processors seldom offer full, usable support for inheritance and polymorphism. In environments where a good C++ compiler is available, one could greatly improve the scheduler's configurability by using an abstract base class to define the ready queue interface and deriving optimized versions for specific applications.)

Scheduler Implementation

Design Motivation

This scheduler design evolved years ago from a pragmatic need to squeeze a multi-threaded design into an 8748 and from a naïve fascination with three (sometimes accurate) observations:

1. An interrupt service sequence has almost the same structure as an O/S task switch.
2. In a non-blocking system, scheduling decisions are only *necessary* at preemption and at task completion.
3. In a prioritized system, task context can be forced to stack just like function context does.

These observations and the need to preemptively schedule several tasks at three or four different priority levels led me to implement a very small, non-blocking scheduler.

Observation 1: Interrupts are a task switch. This was the most "forcing" of these observations. Consider what happens when an interrupt is serviced:

- The current thread is interrupted,
- The ISR saves the context of the interrupted thread,
- The ISR performs whatever urgent service is required by the interrupting device,
- The original context is restored, and
- The ISR returns control to the original thread.

As Figure 1 shows the ISR sequence accurately mirrors the sequence of events that occur when a (prioritized, preemptive) RTOS preemptively starts a new higher priority task (as opposed to resuming some task). The primary differences are:

- The RTOS always saves all context; sometimes the ISR will save only some context.
- The RTOS may need to manipulate stacks or memory mapping hardware to properly initialize the new task environment; the ISR often runs in the same context as the interrupted routine.

- The RTOS chooses the new task dynamically; the ISR always performs the same task.
- The RTOS probably runs the new task with interrupts enabled; the ISR is easiest to write if interrupts are left disabled throughout.

The similarity is profound. With a slight adjustment in conventions regarding context save and restore and interrupt enabling, the ISR is almost a dynamic task switching mechanism. In most simple environments merely observing such conventions and calling a dispatch routine before restoring context is enough to convert the ISR into a dynamic task switcher (at least for the case of launching a new, higher priority task at a preemptive event.) In fact, the core of the resulting scheduler is a simple dispatch function that is usually called from the middle of an interrupt routine.

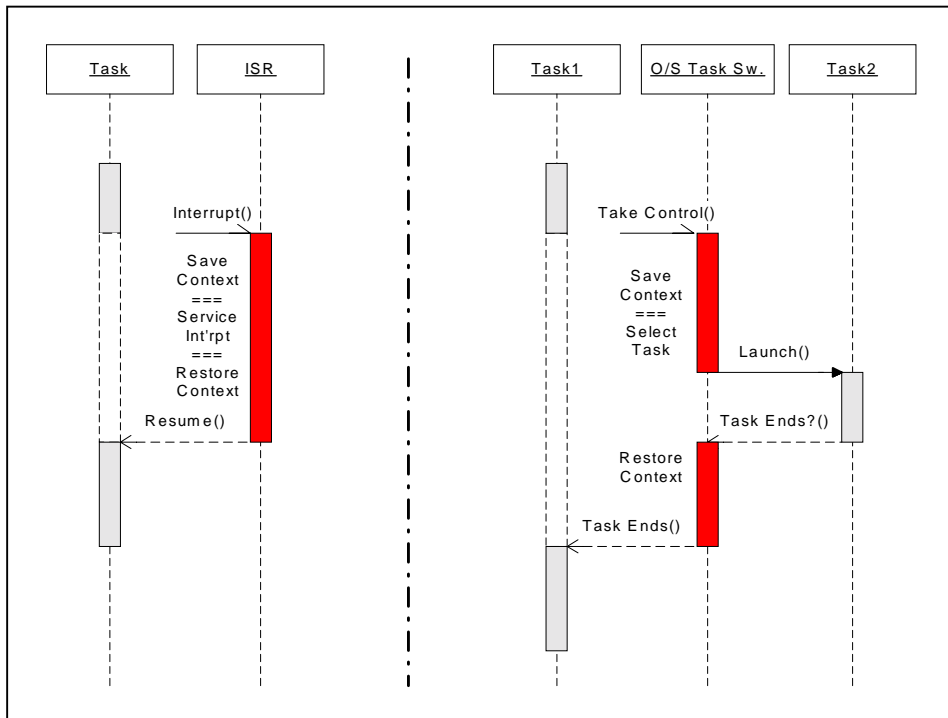


Figure 1 -- Except for the dynamic selection of the service task, an ISR performs the same sequence of tasks as an O/S task switcher.

Observation 2: preemption can be limited to "normal" interrupt times. A prioritized scheduler strives to run the highest priority ready task. Thus, a prioritizing scheduler only needs to perform a task switch when something happens to change the relation between the set of tasks in the ready queue and the currently running task. Because this is a non-blocking scheduler, a task switch can only be necessary if:

- A task has been added to the ready queue, or
- The current task has completed.³

³ Here I'm assuming that task priority is not changed dynamically and that tasks are never removed from the queue (except by being executed). In some situations it makes sense to relax the first restriction to allow priorities to be raised temporarily. While allowing tasks to be arbitrarily removed from the ready queue doesn't cause any scheduling issues, I don't see that it has any useful application that can't easily be addressed in other ways that have less effect on latency.

In the first case, the same code that adds the task to the ready queue should also surrender control to the scheduler (if it has scheduler a higher priority task ... otherwise, there is no need.) If the task was made ready from within an interrupt handler (usually the case because most tasks are triggered by some external input), the scheduler is called as described earlier. If the task is added by code in some normal thread, then that code should also invoke a task switch, either by triggering an interrupt or by calling the dispatch routine directly.⁴ (Often, this can be as simple as disabling interrupts and calling a benign ISR.)

It is important to observe that the preemptive event is *adding the task to the queue*, not receiving an interrupt. If a particular ISR does nothing to change the ready queue, then it does not need to invoke the dispatcher.

If every task is launched by a call from the dispatcher, then control will automatically return to the dispatcher whenever a task completes. Structuring the dispatcher as a loop gives it a chance to revisit the scheduling decision each time one of it's "child" tasks completes.

Observation 3: All context can be saved on the stack. Context is context. To me there's an appealing elegance and uniformity to putting both function call context and thread context into the same stack. The aesthetic issues, though, aren't the most compelling motivators. By putting all context in a single stack, this scheduler can run with far less RAM than the typical blocking scheduler. Because tasks don't have private stacks, there is no unused private stack space associated with suspended tasks. Also, the scheduler needs only one additional data structure -- the ready queue -- to keep track of all task state. Because of this simplicity context switches can involve much less overhead.⁵

This single design decision -- that all context would be kept in a single stack -- is what forces the scheduler to be non-blocking. The dispatcher can never "see" anything but the topmost context in the stack. Thus, the dispatcher can only choose from two alternatives: launch a new task, or resume the task represented by the topmost context saved in the stack.

⁴ This works if the original thread, dispatcher, and all tasks were all compiled to reside in the same context (primarily an issue of address space, here.) If not, then it will be necessary to perform a context save, call the dispatcher, and perform a context restore.

⁵ As always, this depends on the architecture and the quality of implementation. For example, on architectures with several banks of registers, the best option might be to save context by switching banks for certain priorities, and use the stack for all other priorities.

A Reference Implementation

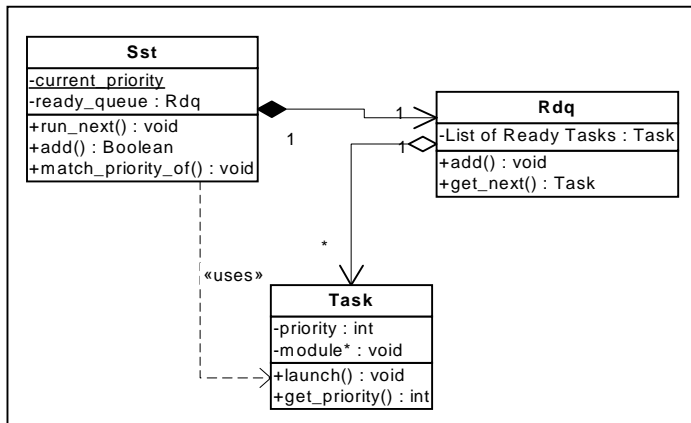


Figure 2 -- The scheduler uses objects from only three classes.

Conceptually the scheduler is constructed from three classes of object: Sst (the dispatcher), Rdq (the ready queue), and Task. Both the dispatcher and the ready queue should be singletons. There should be a distinct instance of Task for each separately schedulable combination of code and priority level. Figure 2 shows the interface details and class relationships. Listing 3 and Listing 5 are the entire implementation for the Sst and Task classes. I'll describe various alternative implementations for the Rdq class later.

Listing 3 -- The Task Class

```

class Task {
private:
    void (* module)( void );
    Sst_priority_t priority;
public:
    Task( Sst_priority_t pri=0, void (* code)( void)=' \0' );
    void launch( ) const;
    Sst_priority_t get_priority( void ) const;
};
Task::Task( Sst_priority_t pri, void (* code)( void ) )
{
    module = code;
    priority = pri;
}

void
Task::launch( ) const
{
    BSP_ENABLE_INTR( );
    (*module)( );
}

Sst_priority_t
Task::get_priority( void ) const
{
    return priority;
}
  
```

Listing 4 -- The Ready Queue Interface

```
class Rdq {
private:
    Task ** queue;
    int q_count;
    Task ** q_ptr; // always points at next available

public:
    Rdq(int max_len);
    ~Rdq();

    bool add( Task * new_task);
    Task * get_next( Sst_priority_t floor );
    Task * peek_next(Sst_priority_t floor ) const;
};
```

Listing 5 -- The Scheduler Class

```
class Sst {
public:
    static const int max_queue_len = SST_MAX_Q_LEN;

    Sst(int qsize = max_queue_len );
    bool add(Task * item);
    void run_next(void);
    void match_priority_of(Task * task);

private:
    static Sst_priority_t current_priority;
    Rdq * ready_queue;
};

Sst_priority_t Sst::current_priority= 0;

Sst::Sst(int qsize){
    current_priority = 0;
    ready_queue = new Rdq (qsize);
}

inline bool
Sst::add(Task * item){
return ready_queue->add(item);
}

void
Sst::run_next(void){

    Task * ready;

    BSP_intr_state_t entry_state;
    Sst_priority_t entry_priority;
```

```

entry_state = BSP_HOLD_INTR(); // disable, but save old state
entry_priority = current_priority; // this instance will hold the processor for
// all priorities higher than this floor
// get a task
while ((ready=ready_queue->get_next(entry_priority)) != '\0') {
    //publish it's priority
    current_priority = ready->get_priority();
    ready->launch(); //launch, the task must enable interrupts
    BSP_DISABLE_INTR(); //to avoid re-entrance issues in scheduler
}

current_priority = entry_priority; // reset to reflect priority of suspended task
BSP_RESTORE_INTR(entry_state); // restore interrupt state
}

void
Sst::match_priority_of(Task * task)
{
    BSP_intr_state_t i_val;

    i_val = BSP_HOLD_INTR();
    current_priority = task->get_priority(); // can avoid locks by
    BSP_RESTORE_INTR(i_val); // using sig_atomic_t for priorities
}

```

Procedurally, the key components in this implementation are `Sst::run_next()` and `Rdq::get_next()`. Listing 5 shows the implementation of the main dispatch routine `Sst::run_next()`. This routine is invoked each time a scheduling decision is to be made. The operation of `run_next()` is much easier to understand if you remember that every running or suspended task was launched by a call from a separate instance of `run_next()`. Thus, there must be a separate suspended instance of `run_next()` waiting for the return of every suspended or running task. `run_next()` not only launches the task, and it also waits around to "catch" the exit when the task completes. Figure 3 shows how the tasks and individual `run_next()` instances nest on the stack after two levels of preemption. Notice that a separate instance of `run_next` is "layered" immediately above each suspended task. This layering is the mechanism that allows the scheduler to recover control of the processor when a task exits.

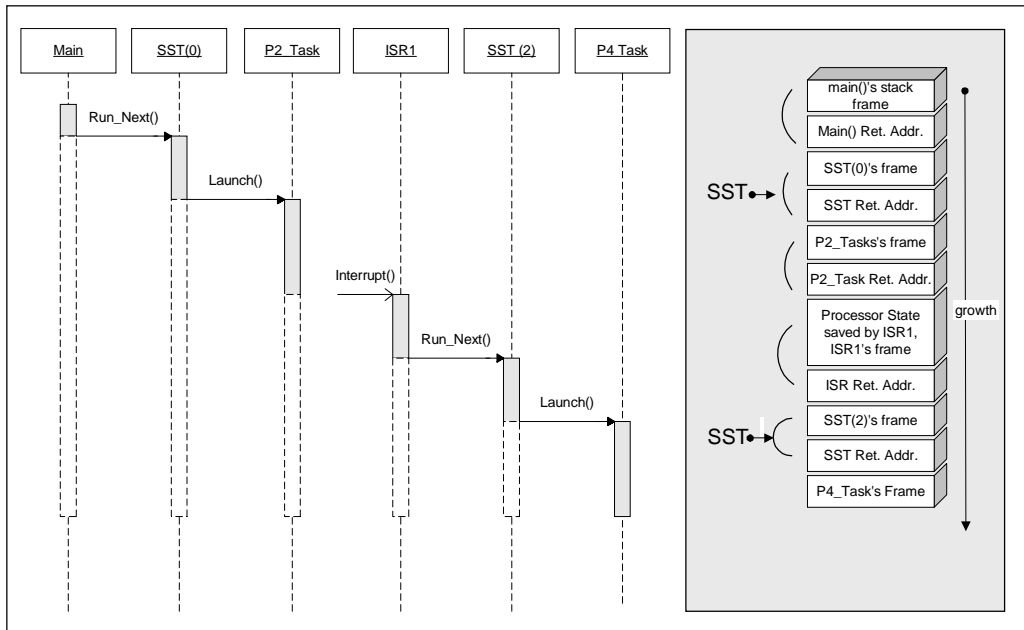


Figure 3 -- This diagram shows how the stack looks after a priority 4 task interrupts a priority 2 task.

At each invocation, `run_next()` will decide whether to launch a higher priority task or resume the suspended task. Thus, each invocation of `run_next()` must know the priority of the most recently suspended task and must make all of its decisions relative to that priority. Each instance of `run_next` makes this information available to subsequent instances of `run_next()` by posting a task's priority level to the global static `SST.current_priority` before launching the task. When first called, each new instance of `run_next()` copies the task priority information from `SST.current_priority` into a local stack variable⁶ `entry_priority`, so that it can "remember" what priority task was executing when it was invoked. The value in `entry_priority` sets a threshold for the particular instance of the dispatcher. Each dispatcher may only launch tasks with priority greater than the value in `entry_priority`.

The `Rdq::get_next(floor)` method retrieves a pointer to the highest priority ready task with priority greater than `floor`. If there are no ready tasks with the required priority, `get_next` returns a null pointer. In addition to returning a pointer to the task, `get_next()` deletes the task from the ready queue.

Method `run_next()` will continue to loop, launching tasks until it has processed all ready tasks with priority higher than the task under it in the runtime stack. Once it has exhausted the list of high-priority ready tasks, `run_next()` simply returns, thereby releasing the suspended task immediately under it.

Code Details

`BSP_XXX` calls are processor-specific macros or inline functions developed as part of the board support package. `BSP_HOLD_INTR()` captures the current interrupt enable state, disables interrupts, and then returns the captured value. `BSP_RESTORE_INTR()` reverses this process, restoring the interrupt enable state from a saved value. (These primitives are necessary if the scheduler can ever be called from within a

⁶ Many compilers for small processors optimize local variable storage and references by mapping locals to various internal RAM or register storage. `run_next()`, however, must be re-entrant, so it is important each instance have its own storage for this variable. Depending on the compiler, you may need to mark `run_next()` reentrant, or explicitly declare this local auto or stack to get the desired results.

critical section.) `BSP_DISABLE_INTR()` and `BSP_ENABLE_INTR()` disable and enable interrupts without regard to any prior state.

The `Sst::add()` method is a pass through definition that simplifies access to the scheduler components. Promoting the `Rdq::add()` method to the `Sst` interface eliminates the need for other components to keep a pointer to the ready queue. I have only given the `Rdq` interface, because I will discuss several alternative implementations later.

Scheduler Optimizations

Optimizing the Dispatch Call

In the examples I've shown so far, the ISR does a full context save and then calls the dispatch routine. While this mechanism is easy to implement and understand, on most machines, it has several disadvantages. First, it requires the code for a context save and restore to be repeated in every interrupt handler. Second, because it always performs a full context save before servicing the interrupt, it adds to interrupt latency. Third, the direct call requires that the stack be large enough to supply every priority level with stack frames for the interrupt, the dispatch call, the task call, and any functions the task invokes.

All of these disadvantages can be addressed by delaying the dispatch call until after the ISR completes. The ISR can effect such a delayed call by manipulating the stack so that when the ISR exits, the CPU will "return" to the dispatch routine. Because the ISR exits before calling the dispatch routine, the scheduler uses one less stack frame per level of preemption. (Compare Figure 5 to Figure 3.) On some machines this pseudo return can be accomplished by pushing the entry point of the dispatch routine just before exiting the ISR, as in Figure 4. You can get nearly identical results by jumping to the dispatch entry at the ISR exit. A "call by return" is usually more compiler-friendly and is more compatible with systems that prioritize interrupts in hardware. On machines where condition flags are easily disturbed (for example where loading a literal address or writing to the stack also affects status flags), the link can be accomplished by "poking" the return address into the stack before the ISR begins its context restore protocol (see the pseudo code in Figure 6). A few processors, like the PIC, place the stack in a separate, inaccessible address space. On these machines, a "jump to dispatch" is the only workable technique for implementing this optimization.

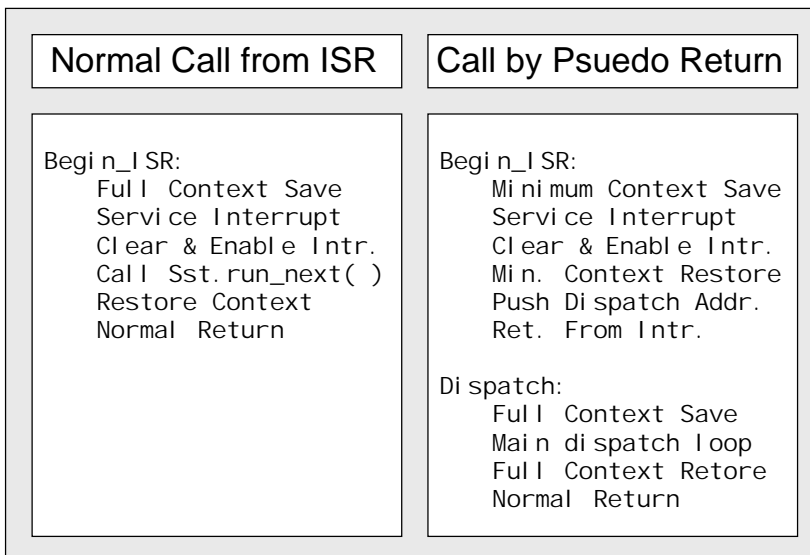


Figure 4 -- On many machines, the ISR can trigger a "call" at exit by pushing a return address before exit.

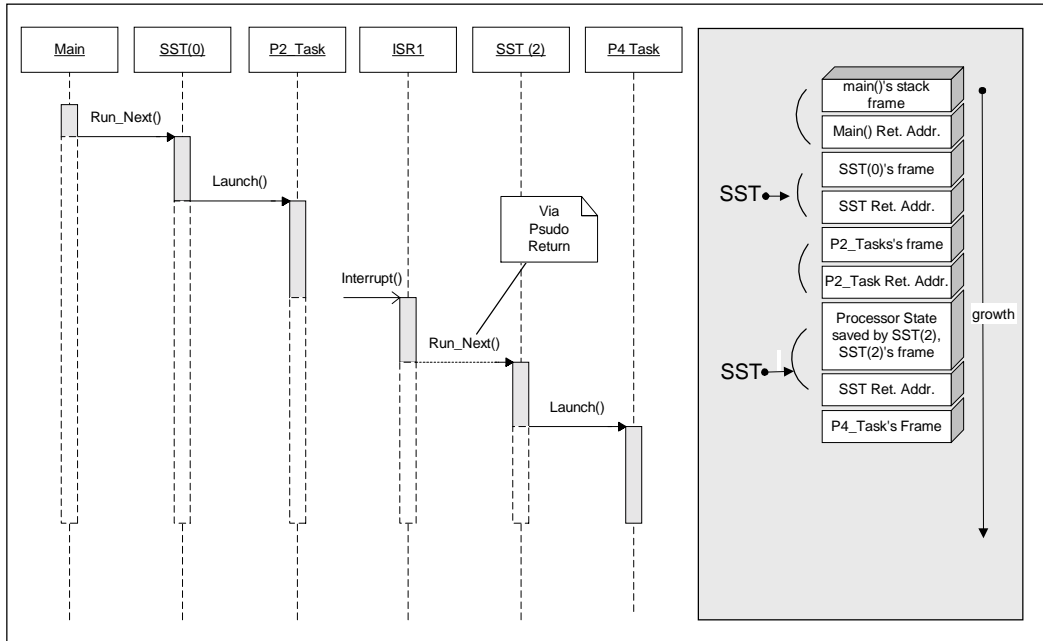


Figure 5 -- Pushing an extra address onto the stack allows the ISR to "return" to the dispatch routine instead of calling it and reduces the number of return addresses kept in the stack.

This delayed call allows one to write much tighter ISRs. Depending on the particular processor with this technique you may be able to achieve the minimum possible delay between interrupt recognition and interrupt service. Delaying the dispatch call also allows all ISRs to share a single copy of the context save and restore code. In most situations, however, you will wind up with two versions of run_next(): one designed to be executed by ISRs, and one (with normal compiler-compatible prolog and epilog, but without context save and restore) that can be called from normal threads.

```

Begin ISR:
    Save status and acc. to dedicated storage
    Push garbage/reserve stack space
    Do minimal context save
    Service Interrupt
    Clear & Enable Intr.
    Poke dispatch entry address into reserved space
    If needed poke "fake" frame
    Min. Context Restore
    Push Dispatch Addr.
    Ret. From Intr.

Dispatch:
    Full Context Save
    Main dispatch loop
    Full Context Restore
    Normal Return
  
```

Figure 6 -- Even on machines with awkward protocols for saving and restoring status, one can usually find a way to implement the delayed "call by return."

Conclusion

SST demonstrates that you only need a few lines of code and a little discipline to bring the structural advantages of a preemptive, prioritized scheduler to almost any environment. While SST's oddities may be unsettling at first, its scheduling policy can easily be exploited to greatly reduce the amount of explicit synchronization needed in the typical reactive program. On balance, SST demands more design effort and real-time design skill than other schedulers, but rewards that investment with cleanly structured programs that can run in the most limited environments.

While SST is different, it isn't *that* different. The insight you gain through mastering task communication and synchronization in SST will *all* transport to other concurrent environments. Regardless of the concurrency model, different rate threads need to be decoupled by queues, separate S/R chains should be separated, locked critical sections introduce jitter, and tasks must be synchronized. At the implementation level, every RTOS or scheduler deals with these issues almost exactly as you do in SST at the application level. In my opinion, knowing what's "under the hood" is always a good thing.